

C++ 核心指导方针

2022/10/6

编辑：

- [Bjarne Stroustrup](#)
- [Herb Sutter](#)

翻译

- 李一楠 (li_yinan AT 163 DOT com)

本文档是处于持续改进之中的在线文档。

本文档作为开源项目，发布版本为 0.8。

复制，使用，修改，以及创建本项目的衍生物，受到一份 MIT 风格的版权授权。

向本项目作出贡献需要同意一份贡献者授权。详情参见附属的 [LICENSE](#) 文件。

我们将本项目开放给“友好用户”进行使用，复制，修改，以及生产衍生物，并希望能够获得建设性的资源投入。

十分欢迎大家提出意见和改进建议。

随着我们的知识增长，随着语言和可用的程序库的改进，我们计划对这份文档不断进行修改和扩充。

当提出您的意见时，请关注[导言](#)部分，其中概述了我们的目标和所采用的一般方法。

贡献者的列表请参见[这里](#)。

已知问题：

- 仍未对规则集合的完整性、一致性和可强制实施性加以全面的检查。
- 三问号 (???) 用于标记已知的信息缺失。
- 需要更新参考部分；许多前 C++11 的源代码都过于老旧。
- [To-do: 未分类的规则原型](#) 是一份基本上保持最新状态的 to-do 列表。

您可以[阅读本指南的范围和结构的说明](#)，或者直接跳转到：

- [In: 导言](#)
- [P: 理念](#)
- [I: 接口](#)
- [F: 函数](#)
- [C: 类和类层次](#)
- [Enum: 枚举](#)
- [R: 资源管理](#)
- [ES: 表达式和语句](#)
- [Per: 性能](#)
- [CP: 并发与并行](#)
- [E: 错误处理](#)
- [Con: 常量和不可变性](#)
- [T: 模板和泛型编程](#)
- [CPL: C 风格的编程](#)
- [SF: 源文件](#)
- [SL: 标准库](#)

配套章节：

- [A: 架构相关理念](#)
- [NR: 伪规则和错误的看法](#)
- [RF: 参考资料](#)
- [PRO: 剖面配置](#)
- [GSL: 指导方针支持库](#)
- [NL: 命名和代码布局建议](#)
- [FAQ: 常见问题的解答](#)
- [附录 A: 程序库](#)
- [附录 B: 代码的现代化转换](#)
- [附录 C: 相关讨论](#)
- [附录 D: 支持工具](#)
- [词汇表](#)
- [To-do: 未分类的规则原型](#)

您可以查看有关某个具体的语言特性的一些规则：

- 赋值：
[正规类型](#) --
[优先采用初始化](#) --
[复制](#) --
[移动](#) --
[以及其他操作](#) --
[缺省操作](#)
- `class`：
[数据](#) --
[不变式](#) --
[成员](#) --
[辅助函数](#) --
[具体类型](#) --
[构造函数, =, 和析构函数](#) --
[类层次](#) --
[运算符](#)
- `concept`：
[规则](#) --
[泛型编程中](#) --
[模板实参](#) --
[语义](#)
- 构造函数：
[不变式](#) --
[建立不变式](#) --
[throw](#) --
[缺省操作](#) --
[不需要](#) --
[explicit](#) --
[委派](#) --
[virtual](#)
- 派生 `class`：
[何时使用](#) --
[作为接口](#) --
[析构函数](#) --
[复制](#) --

[取值和设值](#) --

[多继承](#) --

[重载](#) --

[分片](#) --

[`dynamic_cast`](#)

- 析构函数：

[以及构造函数](#) --

[何时需要？](#) --

[不可失败](#)

- 异常：

[错误](#) --

[`throw`](#) --

[仅用于错误](#) --

[`noexcept`](#) --

[`最少化 try`](#) --

[无异常如何？](#)

- `for`：

[范围式 `for` 和 `for`](#) --

[`for` 和 `while`](#) --

[`for`-初始化式](#) --

[空循环体](#) --

[循环变量](#) --

[循环变量的类型 ???](#)

- 函数：

[命名](#) --

[单操作](#) --

[不能抛出异常](#) --

[实参](#) --

[实参传递](#) --

[多返回值](#) --

[指针](#) --

[`lambda`](#)

- `inline`：

[小型函数](#) --

[头文件中](#)

- 初始化：

[总是](#) --

[优先采用 `{}`](#) --

[`lambda`](#) --

[类内初始化式](#) --

[类成员](#) --

[工厂函数](#)

- lambda 表达式：

[何时使用](#)

- 运算符：

[约定](#) --

[避免转换运算符](#) --

[与 `lambda`](#)

- `public`, `private`, 和 `protected`:

[信息隐藏](#) --

[一致性](#) --

[protected](#)

- `static_assert`:

[编译时检查](#) --

[和概念](#)

- `struct`:

[用于组织数据](#) --

[没有不变式时使用](#) --

[不能有私有成员](#)

- `template`:

[抽象](#) --

[容器](#) --

[概念](#)

- `unsigned`:

[和 signed](#) --

[位操作](#)

- `virtual`:

[接口](#) --

[非 virtual](#) --

[析构函数](#) --

[不能失败](#)

您可以查看用于表达这些规则的一些设计概念:

- 断言: ???
- 错误: ???
- 异常: 异常保证 (???)
- 故障: ???
- 不变式: ???
- 泄漏: ???
- 程序库: ???
- 前条件: ???
- 后条件: ???
- 资源: ???

概要

本文档是一组有关如何更好使用 C++ 的指导方针的集合。

本文档的目标是帮助人们更有效地使用现代 C++。

所谓“现代 C++”的含义是指有效使用 ISO C++ 标准（目前是 C++20，但几乎所有的推荐也适用于 C++17, C++14 和 C++11）。

换句话说，如果你从现在开始算起，五年后你的代码看起来是怎么样的？十年呢？

这些指导方针所关注的是一些相对高层次的问题，比如接口，资源管理，内存管理，以及并发等等。

这样的规则会对应用的架构，以及程序库的设计都造成影响。

如果遵循这些规则，代码将会是静态类型安全的，没有资源泄露，并且能够捕捉到比当今的代码通常所能捕捉到的多得多的编程逻辑错误。

还能更快速地运行——你不必牺牲程序的正确性。

我们对于如命名约定和缩进风格一类的低层次的问题不那么关注。

当然，对程序员有帮助的任何话题都是可接受的。

我们最初的规则集合强调的是（各种形式的）安全性以及简单性。

它们也许有些过于严格了。

我们预期将会引入更多的例外情况，以便使它们更好地适应现实世界的需要。

我们也需要更多的规则。

您可能会发现，有的规则与您的预期相反，甚至是与您的经验相违背。

其实如果我们没建议您在任何方面改变您的编码风格，那其实就是我们的失败！

请您尝试验证或者证伪这些规则吧！

尤其是，我们十分期望让一些规则能够建立在真实的测量数据上，或者是一些更好的例子之上。

您可能会觉得一些规则很显然，甚至没有什么价值。

但请记住，指导方针的目的之一就在于帮助那些经验不足的，或来自其他背景或使用其他语言的人，能够迅速行动起来。

这里的许多规则有意设计成可以由分析工具提供支持的。

违反规则的代码会打上标记，以引用（或者链接）到相关的规则。

您在开始编码前并不需要记住所有这些规则。

一种看待这些指导方针的方式，是一份恰好可以让人类读懂的针对这些工具的规范文件。

这些规则都是为了逐步引入一个代码库而设计的。

我们计划建立这样的工具，并希望其他人也能提供它们。

十分欢迎大家提出意见和改进建议。

随着我们的知识增长，随着语言和可用的程序库的改进，我们计划对这份文档不断进行修改和扩充。

In: 导言

本文档是一组核心指导方针，针对现代 C++（目前为 C++20 和 C++17），还考虑到了语言将来有希望的增强，以及 ISO 技术规范（TS）。

其目标是帮助 C++ 程序员编写更简单、更高效、更加可维护的代码。

导言概览：

- [In.target: 目标读者](#)
- [In.aims: 目标](#)
- [In.not: 非目标](#)
- [In.force: 强制实施](#)
- [In.struct: 本文档的结构](#)
- [In.sec: 主要章节](#)

In.target: 目标读者

所有 C++ 程序员。其中也包括[考虑采用 C 语言的程序员](#)。

In.aims: 目标

本文档的目标是帮助开发者采用现代 C++（目前是 C++17），并在各个代码库之间达成更加统一的编码风格。

我们并不妄想这些规则中的每一条都能有效地在任何代码库中进行实施。对老旧系统进行升级是很困难的。不过我们确实认为，采纳了一条规则的程序总会比不这样做的程序更加不易出错也更加便于维护。通常，采用规则也会带来更快速或更容易的初始开发活动。

就我们所能说的，这些规则能够使得代码的性能，相对于更传统的技术来说同样好甚至更好；它们都是依照零开销原则设立的——“不使用就没有负担”（“what you don't use, you don't pay for”）或“当恰当地使用抽象机制时，所得的性能至少与使用低级语言构造手工编码的结果一样好”。

我们认为这些规则对新代码来说是理想的，也有很多机会在老代码中实施，并试图尽可能接近并灵活地对这些理想情况进行近似。

请记住：

In.0: 不要慌张！

请花些时间理解一下指南规则对你的程序能够造成的影响。

这些指导方针都是遵循“超集的子集”原则（[Stroustrup05](#)）而设计的。

它们并非仅仅定义了 C++ 的一个可以使用的子集（以获得比如说可靠性，安全性，性能，或者别的什么）。

它们强烈地推崇使用一些简单的“扩展”（[程序库组件](#)），

使得最易出错的 C++ 特性变得不再必须，并且可以（通过这些规则）禁止再使用它们。

这些规则都强调静态类型安全性和资源安全性。

鉴于此，它们强调了进行范围检查，避免对 `nullptr` 解引用，避免悬挂指针，以及（通过 RAII）系统性地使用异常的可能性。

部分地为达成这点，也部分地为了最小化会带来错误的晦涩难懂的代码，这些规则同样强调了简单性，以及将必须的复杂性隐藏于经过充分说明的接口后面。

有许多规则都是约定性质的。

我们认为，那些单纯说“禁止这样！”而又不提供替代方案的规则是不可取的。

但这样的后果就是让一些规则只能以启发式方法，而不是精确地和机械化地进行验证检查。

还有一些规则所表达的是一些一般性的原则。对于这些一般性规则，我们会提供一些更精细和更特定的规则来进行不完全的检查。

这些指导方针所关注的是 C++ 的核心部分及其使用方式。

我们认为大多数的大型团体，特定的应用领域，甚至一些大型项目都会需要更多的规则，也许是更多的限制规则，或是更多的库支持。

例如说，硬实时开发人员通常都无法随意使用自由存储（动态内存），并且在选择程序库上也有许多限制。

我们鼓励各方开发这样的专门规则，以作为我们的核心指导方针的补充。

请构建你自己的基础程序库并使用它，而不要把你的开发层次降低到汇编代码之中。

这些规则的设计使其能够进行[渐进式的采纳](#)。

一些规则的目标是提升各种形式的安全性，而另外一些的目标是减少意外的发生，还有许多则同时兼顾。

目标是避免意外事故的指导方针通常会禁止完全合法的 C++ 用法。

不管怎样，每当存在两种达成效果的方式，其中一种被证实是常见的错误来源，而另外一种并非如此时，我们都会努力引导程序员采纳后者。

In.not: 非目标

我们没打算让这些规则保持精简或正交。

特别地说，一般性规则可以很简单，但却没办法强制实施。

而且要搞清楚一条一般性规则所造成的影响通常是很困难的。

通常更专门的规则都更易于理解清楚，也更易于实施，但如果没有任何那些一般性规则的话，它们不过是一

大堆特殊情况而已。

我们既要提供能够帮到新手的规则，也要提供能够支持专家使用的规则。

其中的一些规则是完全可以强制实行的，而另外的一些则是基于启发式方案的规则。

并不需要像读书一样从头到尾地阅读这些规则。

您可以利用链接来进行浏览。

不过，这些规则的主要预期用途是作为工具的检查目标。

就是说，由工具来查找规则的违反情况，然后工具会返回指向所违反的规则的链接。

而规则之中则提供了其理由，违反规则的潜在后果的例子，以及一些改正建议。

这些指导方针并不是用来替代 C++ 的教程材料的。

如果您需要针对某个经验水平的教程，请参见[参考资料](#)。

本文档并不是一份如何把老旧 C++ 代码转化为更加现代的代码的指南。

而是旨在以一种具体化的方式来阐明对于新代码的设想。

当然，对于进行代码现代化转换，使其恢复活力或者升级的可行方式，可以参考[代码现代化章节](#)。

重要的是，这些规则是允许渐进式采纳的：对大型代码库进行一次性全部转化通常都是不可行的。

这些指导方针并不会对于语言技术的每个细节上都保持完整和精确。

如果需要，请参考 C++ 标准，关于语言定义上的最终文本，其中包括一般性规则的每种例外情况，也包括所有特性。

这些规则不是为了强迫你使用 C++ 的某个阉割子集来编写代码的。

它们尤其着重避免去定义一种像（比如）Java 一样的 C++ 子集。

它们也避免去定义一个单一的所谓“真正的 C++”的语言。

我们重视语言的表达能力和不打折扣的性能。

这些规则并不是价值观中立的。

它们旨在使代码变得相对于现有大多数 C++ 代码来说更简单，并且更加正确和安全，又不会有性能损失。

它们旨在约束对那些完全合法的，但却与错误、虚假的复杂性以及不良性能有关的 C++ 代码的使用。

这些规则并未精炼到人们（或机器）可以无脑实施的程度。

“强制实施”部分试图做到这点，但相对于给出一些精确但却错误的东西来说，

我们更倾向于使得一条规则或者定义略微含糊，并允许不同的解读。

有时候，只有经历时间和经验的凝炼才能带来精确性。

设计（还）并不是数学的某种形式。

这些规则并不完美。

某条规则可能有害，因其可能制止了在特定情形中有益的事物。

某条规则可能有害，因其可能无法制止在特定情形中会导致某种严重错误的事物。

某条规则可能有许多害处，因其含混，有歧义，无法实施，或者对一个问题给出了所有的解决方案。

完全满足“无害”的准则是不可能的。

相对来讲，我们的目标并没那么大野心：“对大多数程序员有最多的好处”；

如果某条规则使你无法工作，你反对它，忽略掉它，但请不要削弱它，除非它已经变得没有意义。

同样，也请给出改进的建议。

In.force: 强制实施

无法强制实施的规则对于大型代码库来说是难以操作的。

所有规则都强制实施，则仅对于一个小的规则集合，或者对于某些特定用户群来说是可行的。

- 但我们需要大量的规则，需要每个人都能使用的规则。
- 不同的人的要求都不一样。
- 人们不想阅读大量的规则。

- 人们也无法记住太多规则。

因此，我们需要建立规则子集以满足各种不同的需要。

- 但任意性地建立子集也会导致混乱。

我们想要的是可以帮助到许多人的指导方针，使代码更加统一，并有力地促进人们将他们的代码现代化。

我们想要促进最佳实践，而不是让人们陷入大量选项之中而增加管理压力。

理想情况是使用全部规则；这会带来极大的好处。

但这样也带来了一些困难之处。

我们试图通过使用工具来解决它们。

每条规则都包括一个**强制实施**小节，列出了进行强制实施的一些建议。

所谓强制实施，可以是通过代码评审，通过静态分析，通过编译器，或者通过运行时检查来进行的。

只要可行，我们都倾向于“机械性的”检查（人类是缓慢的，不精确的，而且很容易疲倦）和静态检查。

只有当不存在其他方案时，我们才偶尔建议进行运行时检查；我们并不想带来所谓“分布式代码爆炸”。

如果适当的话，我们会（在**强制实施**小节中）将规则标以相关的规则组的名字（所谓“剖面配置”）。

一条规则可以属于多个剖面配置，也可以不属于任何剖面配置。

首先，我们有一些对应于常规需求（期望、理想目标）的剖面配置：

- **type**: 消除类型违规（如通过强制转换 (cast)，联合体 (union)，或者变参 (varargs) 把 `T` 重解释为 `U`）
- **bounds**: 消除边界违规（如越过数组范围的访问）
- **lifetime**: 消除泄漏（如未能 `delete` 或者进行多次 `delete`），以及消除对无效对象的访问（如解引用 `nullptr`，或使用悬挂引用）。

这些剖面配置是为工具的使用而准备的，但对人类读者也能有所帮助。

我们不打算把**强制实施**小节中的评述限定在我们了解如何强制实施的方面；其中的一些说明仅仅是一些愿望，它们可能会对一些工具构建者们造成影响。

实现这些规则的工具应当遵循下面的语法以明确抑制一条规则：

```
[[gsl::suppress(tag)]]
```

或可选地带有一条消息（遵循常规的 C++11 标准标注语法）：

```
[[gsl::suppress(tag, justification: "message")]]
```

其中

- `tag` 是包含强制规则的条目的锚定名字（例如，[C.134](#) 的锚定名字为 "Rh-public"），剖面配置的规则组的名字（如 "type"，"bounds"，或 "lifetime"），或者剖面配置中的特定规则（[type.4](#) 或 [bounds.2](#)）
- `"message"` 是字符串字面量

Instruct: 本文档的结构

每条规则（指导方针，建议）可以包含几个部分：

- 规则本身 —— 例如，**不要使用裸 new**
- 一个规则参考编号 —— 例如，[C.7](#)（与类相关的第七条规则）。

因为大章节之间自然是无序的，所以我们用字母来当作规则参考“编号”的第一个部分。

我们在编号之间保留了一些建议，以便当添加或删减规则时尽量减少“断裂”。

- **理由**（原理）——程序员对于他们不理解的规则是难于遵守的
- **示例**——抽象地理解规则是很难的；示例有正面的和负面的
- **替代方案**——针对“请勿……”规则
- **例外**——我们更喜欢简单的一般性规则。但是许多规则都是广泛适用，但并不是普遍适用的，因此必须列出例外情况
- **强制实施**——关于这条规则如何“机械性”地进行检查的建议
- **参见**——指向相关的规则，以及（本文档中或者别处的）进一步讨论
- **注解**——需要说明的一些内容，无法被归类到其他部分
- **探讨**——指向规则主列表之外的更加全面的原理说明和实例

一些规则难于机械地进行检查，但它们都满足一条最小准则，即专家程序员可以不费太多力气就能找出许多违反情况。

我们希望“机械性”工具能够随着时间获得改进，以接近这种专家程序员所能发觉的程度。

而且，我们认为这些规则也会随着时间获得提炼，以变得更明确和易于检查。

规则应当简明，而不是谨慎地列出每种变化和特殊情况。

这些信息应当出现在**替代方案**段落和**探讨**章节中。

如果您不理解或者反对一条规则，请您访问它的**探讨**部分。

如果您觉得一份探讨有缺漏或不完整，请填写一条 [Issue](#)

来解释您的关切，亦或一条相应的问题报告。

各个示例用于演示规则。

- 这些示例并非意图具有产品级的质量，或覆盖所有的教学维度。
例如，许多的例子都是语言技巧，并使用了诸如 `f`, `base`, 和 `x` 这样的名字。
- 我们尝试保证使下文中“好”的示例都遵守《核心指导方针》。
- 注释通常用于演示规则，它们可能是不必要的，或者会干扰“真正的代码”。
- 我们假设读者具有标准程序库的知识。例如我们使用普通的 `vector` 而不是 `std::vector`。

本文档不是语言手册。

它旨在能够对人有所帮助，而不是变得完整，在技术细节上完全准确，或对现存代码的指南。

可以在[参考资料](#)中找到一些推荐的信息来源。

In.sec: 主章节

- [In: 导言](#)
- [P: 理念](#)
- [I: 接口](#)
- [F: 函数](#)
- [C: 类和类层次](#)
- [Enum: 枚举](#)
- [R: 资源管理](#)
- [ES: 表达式和语句](#)
- [Per: 性能](#)
- [CP: 并发与并行](#)
- [E: 错误处理](#)
- [Con: 常量和不可变性](#)
- [T: 模板和泛型编程](#)
- [CPL: C 风格的编程](#)
- [SF: 源文件](#)
- [SL: 标准库](#)

配套章节：

- [A: 架构相关的理念](#)
- [NR: 伪规则和错误的看法](#)
- [RF: 参考资料](#)
- [Pro: 剖面配置](#)
- [GSL: 指导方针支持库](#)
- [NL: 命名和代码布局建议](#)
- [FAQ: 常见问题的解答](#)
- [附录 A: 程序库](#)
- [附录 B: 代码的现代化转换](#)
- [附录 C: 相关讨论](#)
- [附录 D: 支持工具](#)
- [词汇表](#)
- [To-do: 未分类的规则原型](#)

章节之间并非是正交的。

每个章节（比如， "P" 代表“理念”），以及每个子章节（比如， "C.hier" 代表“类层次（OOP）”）都有一个用以简化搜索和引用的缩写。

主章节的缩写也出现在规则编号之中（比如， "C.11" 代表“使具体类型正规化”）。

P: 理念

本章节中的规则都非常具有一般性。

理念性规则概览：

- [P.1: 在代码中直接表达你的想法](#)
- [P.2: 用 ISO 标准 C++ 来编码](#)
- [P.3: 表达你的设计意图](#)
- [P.4: 理想情况下，程序应当是静态类型安全的](#)
- [P.5: 编译期检查优先于运行时检查](#)
- [P.6: 应当使无法在编译期进行的检查能够在运行时实施](#)
- [P.7: 尽早识别运行时错误](#)
- [P.8: 不要泄漏任何资源](#)
- [P.9: 不要浪费时间或空间](#)
- [P.10: 不可变数据优先于可变数据](#)
- [P.11: 把杂乱的构造封装起来，而别让其散布到代码中](#)
- [P.12: 适当采用支持工具](#)
- [P.13: 适当采用支持程序库](#)

通常，理念性的规则都无法机械性地进行检查。

不过，这些理念主题在各个规则中都有体现。

如果没有一个理念基础的话，那些更具体、专门和可检查的规则就是缺少理论根据的了。

P.1: 在代码中直接表达你的想法

理由

编译器是不会去读注释（或设计文档）的，许多程序员也（固执地）不去读它们。

而代码中所表达的东西是带有明确的语义的，并且（原则上）是可以由编译器和其他工具进行检验的。

示例

```
class Date {
public:
    Month month() const; // 好
    int month(); // 不好
    // ...
};
```

`month` 的第一个声明式，显然是要返回一个 `Month`，而且不会修改 `Date` 对象的状态。而第二个版本则需要读者进行猜测，同时带来了更多的出现难于发现 BUG 的可能性。

示例，不好

这个循环是 `std::find` 的一种能力有限的形式：

```
void f(vector<string>& v)
{
    string val;
    cin >> val;
    // ...
    int index = -1; // 不好，而且应该使用 gsl::index
    for (int i = 0; i < v.size(); ++i) {
        if (v[i] == val) {
            index = i;
            break;
        }
    }
    // ...
}
```

示例，好

要清晰得多地表达其设计意图，可以这样：

```
void f(vector<string>& v)
{
    string val;
    cin >> val;
    // ...
    auto p = find(begin(v), end(v), val); // 好多了
    // ...
}
```

用恰当设计的程序库来表达设计意图（要做什么，而不只是怎么做这些事），要远比直接使用语言功能好得多。

C++ 程序员应当熟知标准库的基本知识，并在适当的时候加以利用。

任何程序员都应当熟知其所工作的项目中的基础程序库的基本知识，并适当加以利用。

使用本文档的指导方针的程序员，应当熟知[指导方针支持库](#)，并适当加以利用。

示例

```
change_speed(double s); // bad: s 代表什么?  
// ...  
change_speed(2.3);
```

更好的方案是明确给出这个 double 的含义（新的速度还是对旧速度的增量？）以及所用单位：

```
change_speed(speed s); // 好多了：说明了 s 的含义  
// ...  
change_speed(2.3); // 错误：没有单位  
change_speed(23_m / 10s); // 米每秒
```

确实可以用普通的（没有单位的）`double` 作为增量值，但这样是易于出错的。

如果绝对速度值和增量值都需要的话，我们应当定义一个 `Delta` 类型。

强制实施

通常非常困难。

- 坚持一贯地使用 `const`（检查成员函数是否会修改对象；检查函数是否会修改以指针或引用形式传递的实参）
- 将强制转换标志出来（强制转换阉割了类型系统）
- 检测模仿标准库的代码（困难）

P.2: 用 ISO 标准 C++ 来编码

理由

本文档正是关于用 ISO 标准 C++ 来编码的一组指导方针。

注解

有些环境下是需要使用语言扩展的，例如有关访问系统资源的语言扩展。

这些情况下，应当将对所需语言扩展的使用局部化，并把它们的使用置于非核心的编码指导方针的控制之下。如果可能的话，应当构建一些接口来封装这些语言扩展，以使其能够被关闭，并当针对不支持这些语言扩展的系统时免除它们的编译。

语言扩展通常是没有严密定义的语义的。即便语言扩展很常见，并且在多种编译器上都有实现，它们也可能有略微不一致的行为以及边界情形下的行为，这是缺乏一个严格的标准定义的直接后果。大量使用任何这样的语言扩展，都会对代码的可移植性造成不良影响。

注解

使用合法的 C++ 并不能保证可移植性（不管其正确性）。

应当避免依赖于未定义的行为（例如，[未定义的求值顺序](#)）

并应当关注带有由实现定义的含义的构造（例如，`sizeof(int)`）。

注解

有些环境下是需要对标准 C++ 语言或者程序库的功能特性的使用进行限制的，例如，飞行器控制软件标准要求避免动态内存分配。

这些情况下，应当将对它们的使用（或废弃）置于对本文档针对特定环境所定制的扩充的编码指导方针之下。

强制实施

使用最新版的 C++ 编译器（目前支持 C++20 或 C++17），并打开禁用语言扩展的选项。

P.3: 表达你的设计意图

理由

一些代码如果不（比如通过命名或者代码注释）说明其设计意图的话，是不可能搞清楚代码是否达成其预定目标的。

示例

```
gsl::index i = 0;
while (i < v.size()) {
    // ... 在 v[i] 上做一些事 ...
}
```

这里并未表明其意图是“单纯地”循环访问 `v` 的元素。使用一个索引的实现细节被暴露了出来（因而可能导致被误用），而且 `i` 的存在超出了循环的范围，这也许符合也许违背了设计意图。读者仅从这段代码中是无法了解清楚的。

更好的方式是：

```
for (const auto& x : v) { /* 用 x 的值做一些事 */ }
```

现在，循环机制不明确给出，而且循环的操作针对的是 `const` 元素，以防止发生意外的修改。如果需要进行修改的话，则可以这样：

```
for (auto& x : v) { /* 修改 x */ }
```

`for` 语句的更多细节，请参见 [ES.71](#)。

有时候，使用具名的算法会更好。这个示例使用 Ranges TS 中的 `for_each`，因为它直接表达了意图：

```
for_each(v, [](int x) { /* 用 x 的值做一些事 */ });
for_each(par, v, [](int x) { /* 用 x 的值做一些事 */ });
```

最后一种写法让人明白，我们对按照何种顺序来处理 `v` 的各个元素并不关心。

程序员应当熟悉：

- [指南方针支持库](#)
- [ISO C++ 标准库](#)
- 当前项目所使用的任何基础程序库

注解

其他形式：说明要做什么，而不只是怎么做这些事。

注解

一些语言构造比另一些可以更好地表达设计意图。

示例

如果要用两个 `int` 来代表二维点的坐标值，应当这样：

```
draw_line(int, int, int, int); // 含混的  
draw_line(Point, Point);     // 清晰的
```

强制实施

查找具有更加替代方案的一般模式：

- 简单 `for` 循环 vs. 范围式 `for` 循环
- `f(T*, int)` 接口 vs. `f(span<T>)` 接口
- 循环变量出现在过大的范围内
- 裸的 `new` 和 `delete`
- 带有大量内建类型的形参的函数

在聪敏的人工处理和半自动的程序变换之间存在巨大的空间。

P.4: 理想情况下，程序应当是静态类型安全的

理由

理想情况下，程序应当完全是静态（编译期）类型安全的。

不幸的是，这是不可能的。有问题的领域：

- `union`
- 强制转换
- 数组退化
- 范围错误
- 窄化转换

注解

这些领域是许多严重问题（如程序崩溃和安全性违规）的来源。

我们争取为它们给出替代技术。

强制实施

如果程序各自需要或者条件允许的话，我们可以逐个对这些问题类型分别进行阻止、克制或者检测。
我们总会给出替代方案。

例如：

- `union` - 使用 `variant` (C++17 提供)
- 强制转换 - 尽可能减少其使用；使用模板有助于这点
- 数组退化 - 使用 `span` (来自 GSL)
- 范围错误 - 使用 `span`
- 窄化转换 - 尽可能减少其使用，必须使用时则使用 `narrow` 或者 `narrow_cast` (来自 GSL)

P.5: 编译期检查优先于运行时检查

理由

为了代码清晰性和性能。

对于编译期识别的错误是不需要编写错误处理的。

示例

```
// Int 被用作整数的别名
int bits = 0;           // 请勿如此：可以避免的代码
for (Int i = 1; i; i <= 1)
    ++bits;
if (bits < 32)
    cerr << "Int too small\n";
```

这个例子并没有达成其所要达成的目的（因为溢出是未定义行为），应当被替换为简单的

```
static_assert:
```

```
// Int 被用作整数的别名
static_assert(sizeof(Int) >= 4); // do: 编译时检查
```

或者更好的方式是直接利用类型系统，将 `int` 替换 `int32_t`。

示例

```
void read(int* p, int n); // 读取至多 n 个整数到 *p 之中

int a[100];
read(a, 1000); // 不好，超过末尾了
```

更好的做法是

```
void read(span<int> r); // 读取到整数区域范围 r 之中

int a[100];
read(a); // 好多了：让编译器确定元素数量
```

替代形式：不要把可以在编译期搞定的事推后到运行时进行。

强制实施

- 查找指针参数。
- 查找运行时进行的范围违反检查。

P.6: 应当使无法在编译期进行的检查能够在运行时实施

理由

把难于检测的错误遗留在程序中，总会带来程序崩溃或得到错误的运行结果。

注解

理想情况下我们可以在编译期或者运行时识别所有的错误（它们并非程序员的逻辑错误）。但是要在编译期识别所有的错误是不可能的，而通常也负担不起在运行时识别剩余的全部错误的代价。不过我们编写程序，应当尽量使其在原则上是可以在充足的（分析程序，运行时检查，机器资源，时间等）资源下进行检查的。

示例，不好

```
// 分离编译，可能会被动态加载
extern void f(int* p);

void g(int n)
{
    // 不好的：并未把元素数量传递给 f()
    f(new int[n]);
}
```

此处，关键性的信息（元素数量）被完全掩盖起来，使其无法进行静态分析，而如果 `f()` 属于某个 ABI 的一部分的话，由于无法对这个指针进行“测量插装”，运行时检查也是不可行的。我们确实可以在自由存储中插入有助于检查的信息，但这需要对系统甚至是编译器做出整体改动。这就是一个能让错误检查变得非常困难的设计。

示例，不好

当然可以把元素数量和指针一起进行传递：

```
// 分离编译，可能会被动态加载
extern void f2(int* p, int n);

void g2(int n)
{
    f2(new int[n], n); // 不好的：可能会把错误的元素数量传递给 f()
}
```

把元素数量作为一个参数进行传递，比只传递指针而依靠某种（不明确的）对已知元素个数的约定或者找出元素个数的方式，要好得多，而且是更加常见的做法。但是如上所示，一个简单的错字就可以引入一个严重的错误。`f2()` 的两个参数之间的关联是基于约定的，而并不明确。

而且，这里还隐含假定 `f2()` 应当 `delete` 其参数（要不然就是调用者又犯了另一个错误）。

示例，不好

使用标准库的资源管理指针指向对象时，也不能传递其大小：

```
// 分离编译，可能会被动态加载
// NB：这里假定调用代码是 ABI 兼容的，使用的是
// 兼容的 C++ 编译器和同一个 stdlib 实现
extern void f3(unique_ptr<int[]>, int n);

void g3(int n)
{
    f3(make_unique<int[]>(n), n); // 不好的：把所有权和大小分开进行传递
}
```

示例

我们得把指针和元素数量作为一个对象整体来进行传递：

```
extern void f4(vector<int>&);    // 分离编译，可能会被动态加载
extern void f4(span<int>);        // 分离编译，可能会被动态加载
                                    // NB: 这里假定调用代码是 ABI 兼容的，使用的是
                                    // 兼容的 C++ 编译器和同一个 stdlib 实现

void g3(int n)
{
    vector<int> v(n);
    f4(v);                      // 传递引用，保留所有权
    f4(span<int>{v});          // 传递视图，保留所有权
}
```

这个设计将元素数量作为对象的固有部分，因此不太可能有错误，动态（运行时的）检查即使不总是可承担的，也总是可行的。

示例

如果把所有权和验证所需的全部信息一起传递的话会怎么样呢？

```
vector<int> f5(int n)    // OK: 移动
{
    vector<int> v(n);
    // ... 初始化 v ...
    return v;
}

unique_ptr<int[]> f6(int n)    // 不好的：缺失了 n
{
    auto p = make_unique<int[]>(n);
    // ... 初始化 *p ...
    return p;
}

owner<int*> f7(int n)    // 不好的：缺失了 n 并且我们可能会忘记 delete
{
    owner<int*> p = new int[n];
    // ... 初始化 *p ...
    return p;
}
```

示例

- ???
- 展示传递多态基类的接口是如何避开可能进行的检查的，但它们实际上直到它们需要的类型？还有用字符串当作“自由式”选项的做法

强制实施

- 标示出 (pointer, count) 形式的接口 (这将标示出大量的因为兼容性原因而无法进行修正的实例)
- ???

P.7: 尽早识别运行时错误

理由

避免“神秘的”程序崩溃。

避免能够产生（也许无法识别的）错误结果的程序错误。

示例

```
void increment1(int* p, int n)    // 不好的：易于出错
{
    for (int i = 0; i < n; ++i) ++p[i];
}

void use1(int m)
{
    const int n = 10;
    int a[n] = {};
    // ...
    increment1(a, m);    // 可能是打错字，可能假定有 m <= n
                          // 不过让我们假设 m == 20
    // ...
}
```

我们在 `use1` 里面犯了一个能够导致数据损坏或程序崩溃的小错误。

这个 (pointer, count) 形式的接口让 `increment1()` 没有可以使其防范越界错误的任何现实可行的方式。

如果我们可以检测到越界访问的下标的话，那么这个错误直到对 `p[10]` 进行访问之前都不会被发现。

我们可以提早进行检查来改进这个代码：

```
void increment2(span<int> p)
{
    for (int& x : p) ++x;
}

void use2(int m)
{
    const int n = 10;
    int a[n] = {};
    // ...
    increment2({a, m});    // 可能是打错字，可能假定有 m<=n
    // ...
}
```

现在，就可以在调用点（提早地）检查 `m <= n`，而不是更晚进行了。

如果我们只是打错了字而本想用 `n` 作为边界值的话，代码还可以进一步简化（来消除一处错误的可能性）：

```

void use3(int m)
{
    const int n = 10;
    int a[n] = {};
    // ...
    increment2(a);    // 不需要重复给出 a 的元素数量
    // ...
}

```

示例，不好

不要对同一个值重复进行检查。不要用字符串来传递有结构的数据：

```

Date read_date(istream& is);    // 从 istream 读取日期

Date extract_date(const string& s);    // 从 string 中抽取日期

void user1(const string& date)    // 操作 date
{
    auto d = extract_date(date);
    // ...
}

void user2()
{
    Date d = read_date(cin);
    // ...
    user1(d.to_string());
    // ...
}

```

这个日期被 (`Date` 的构造函数) 验证了两次，并以字符串（无结构的数据）的形式来传递。

示例

过量的检查可能是代价昂贵的。

有些情况下提早检查可能会很无效，因为你可能根本不需要这个值，或者可能仅需要值的一部分，而要比进行整体的检查容易得多。同样来说，不要添加能够改变接口的渐进式行为的验证性检查（例如，不要在平均复杂度为 $O(1)$ 的接口中添加一个 $O(n)$ 的检查）。

```

class Jet {    // 物理规则是: e * e < x * x + y * y + z * z
    float x;
    float y;
    float z;
    float e;
public:
    Jet(float x, float y, float z, float e)
        :x(x), y(y), z(z), e(e)
    {
        // 应不应该在这里检查这些值是物理上有意义的?
    }

    float m() const
    {

```

```

    // 应不应该处理这里的退化情形?
    return sqrt(x * x + y * y + z * z - e * e);
}

???
};

```

喷流 (Jet) 的物理定律 ($e * e < x * x + y * y + z * z$) , 由于可能存在测量误差的缘故并不是不变式。

???

强制实施

- 查找指针和数组：提早且不要重复进行范围检查
- 查找类型转换：消除或标示出窄化转换
- 查找未经检查的来自输入的值。
- 查找被转换成字符串的结构化数据（带有不变式的类的对象）
- ???

P.8: 不要泄漏任何资源

理由

即使是缓慢的资源增长，随着时间推移，也会耗尽这些资源的可用性。

这对于长时间运行的程序来说尤其重要，而且是负责任的编程行为的基础方面。

示例，不好

```

void f(char* name)
{
    FILE* input = fopen(name, "r");
    // ...
    if (something) return;    // 不好的：如果 something == true 的话，将会泄漏一个文件句柄
    // ...
    fclose(input);
}

```

建议采用 [RAII](#)：

```

void f(char* name)
{
    ifstream input {name};
    // ...
    if (something) return;    // OK：没有泄漏
    // ...
}

```

参见：[资源管理相关章节](#)

注解

通俗地说，泄漏就是“有东西没清理干净”。

一种更重要的分类方式是“有东西无法再被清理干净”。

例如，在堆上分配一个对象，然后又丢失了最后一个指向这份分配物的指针。

不应当将这条规则误读为，要求在程序终止时必须把长期存活的对象中的分配物进行回收。

例如，依赖于系统所保证的进程停止时进行的文件关闭和内存回收行为可以简化代码。

然而，依赖于进行隐式清理的抽象机制同样简单，而且通常更加安全。

注解

强制实行[生存期安全性剖面配置](#)可以消除泄漏的发生。

如果和[RAII](#)所提供的资源安全性组合到一起，也可以（通过不产生任何垃圾而）消除对“垃圾收集”的需要。

如果将之和[类型和边界剖面配置](#)组合到一起强制实施的话，你将会得到完全的类型和资源安全性，这是通过使用工具来保证的。

强制实施

- 查找指针：把它们分成非所有者（默认情形）和所有者。

如果可行的话，把所有者替换为标准库的资源封装类（如上例所示）。

或者，也可以把这种所有者用[GSL](#)中的`owner`进行标记。

- 查找裸露的`new`和`delete`

- 查找已知的返回原始指针的资源分配函数（诸如`fopen`，`malloc`，和 `strdup`等）

P.9: 不要浪费时间或空间

理由

你用的语言是 C++。

注解

为达成某个目标（例如开发速度，资源安全性，或者测试的简化等）而正当花费的时间和空间是不会被浪费的。

“力求高效的另一种好处是，这一过程将强迫你更深入地理解问题。”—— Alex Stepanov

示例，不好

```
struct X {
    char ch;
    int i;
    string s;
    char ch2;

    X& operator=(const X& a);
    X(const X&);

};

X waste(const char* p)
{
    if (!p) throw Nullptr_error{};
    int n = strlen(p);
    auto buf = new char[n];
    if (!buf) throw Allocation_error{};
    for (int i = 0; i < n; ++i) buf[i] = p[i];
```

```

// ... 对缓冲区进行操作 ...
X x;
x.ch = 'a';
x.s = string(n);      // 在 x.s 上预留 *p 的空间
for (gs1::index i = 0; i < x.s.size(); ++i) x.s[i] = buf[i]; // 把 buf 复制给
x.s
delete[] buf;
return x;
}

void driver()
{
    X x = waste("Typical argument");
    // ...
}

```

这个确实有些夸张，但我们在产品代码中能够见到这里所犯的每个错误，甚至更糟糕。

注意，`x` 的布局保证会浪费至少 6 个字节，而且很可能更多。

错误的复制操作的定义式废掉了移动语义，使返回操作变得更慢

(请注意这里并不会保证进行返回值优化 (RVO))。

为 `buf` 使用的 `new` 和 `delete` 是多余的；如果确实想要一个局部的字符串的话，我们应当使用局部的 `string`。

还有几个其他的性能 BUG 和无理由的复杂性。

示例，不好

```

void lower(zstring s)
{
    for (int i = 0; i < strlen(s); ++i) s[i] = tolower(s[i]);
}

```

这个其实是一个来自产品代码的例子。

可以看到这里有一句 `i < strlen(s)`。这个表达式在循环的每次重复中都要求值，这意味着每次循环中 `strlen` 都必须走完字符串以确定其长度。我们假定在改动字符串内容过程中 `tolower` 不会影响字符串的长度，因此最好在循环外面缓存长度值，而不是在每次重复中都承担其代价。

注解

单个造成浪费的范例很少是显著的，而一旦它是显著的，通常也可以被高手轻易地清除掉。

但是，代码库中放任地到处散布的浪费情况，则很容易变得显著，而高手们又不像我们期望那样总是有空的。

本条规则（以及其他配套的更加具体的规则）的目的是，将与 C++ 语言的使用有关的大多数浪费情况，在其发生之前就将之清除掉。

在这之后，我们就可以查找与算法和需求有关的浪费情况了，但这超出了我们的指导方针的范畴。

强制实施

许多更加具体的规则都是针对追求简单性并清除无理由浪费的总体目标的。

- 当用户定义的非预置后缀 `operator++` 或 `operator--` 函数的返回值未被使用时进行标记。优先代之以采用前缀形式。（注：使用“用户定义的非预置”是为了减弱噪声。若实践中噪声还是很显著则需要重新审视这条强制措施。）

P.10: 不可变数据优先于可变数据

理由

对常量进行推理要比变量简单得多。
不可变的事物是不可能被意外改变的。
不可变性有时候也带来更好地进行优化的机会。
在常量上不会出现数据竞争。

另见 [Con: 常量和不可变性](#)

P.11: 把杂乱的构造封装起来，而别让其散布到代码中

理由

杂乱的代码更有可能隐藏有 Bug 而且难于编写。
而好的接口使用起来更容易和安全。
杂乱的，底层的代码会混杂出更多这样的代码。

示例

```
int sz = 100;
int* p = (int*) malloc(sizeof(int) * sz);
int count = 0;
// ...
for (;;) {
    // ... 读取一个 int 到 x 中, 如果达到文件尾就退出循环 ...
    // ... 检查 x 有效 ...
    if (count == sz)
        p = (int*) realloc(p, sizeof(int) * sz * 2);
    p[count++] = x;
    // ...
}
```

这段代码是低层的，啰嗦的，而且易错的。
比如说，我们就“忘了”检查内存耗尽情况。
我们可以代之以使用 `vector`：

```
vector<int> v;
v.reserve(100);
// ...
for (int x; cin >> x; ) {
    // ... 检查 x 是有效 ...
    v.push_back(x);
}
```

注解

标准库和 GSL 都是这种理念的例子。
例如，我们并不使用混乱的数组，联合体，强制转换，麻烦的生存期问题，`gsl::owner`，等等，它们用于实现一些关键抽象，诸如 `vector`，`span`，`lock_guard`，以及 `future`，我们使用的一般来说比我们有更多时间和专业能力的人所设计和实现的程序库。
类似地，我们也能够而且应该设计并实现更专门的程序库，而不是将其留给用户（通常是我们自己）

来面对需要重复把低级代码搞正确的挑战。

这是作为指导方针基石的[超集的子集原则](#)的一种变体。

强制实施

- 查找如复杂指针操作和在抽象的实现外面进行强制转换这样的“混乱代码”。

P.12: 适当采用支持工具

理由

许多事情机器都比人做得更好。

对于重复劳动，计算机既不会累也不会厌烦。

相对于重复性的例行任务，我们通常可以做一些更有意义的事情。

示例

运行静态分析工具来验证你的代码是否遵循了你想要遵循的指导方针。

注解

参见

- [静态分析工具](#)
- [并发工具](#)
- [测试工具](#)

还有许多其他种类的工具，诸如源代码仓库和构建工具等等，

但这些超出了本指导方针的范围。

注解

当心不要变得对过于详细定制的或者过于专门的工具链产生依赖。

它们会使得你本来可移植的代码变得不可移植。

P.13: 适当采用支持程序库

理由

使用设计良好，文档全面，并且有良好支持的程序库可以节省时间和工作量；

如果你的大部分工时都必须耗费在实现上的话，

程序库的质量和文档很可能要比你能做到的要好得多。

程序库的成本（时间，工作量和资金等等）可以由大量的用户所分担。

一个被广泛应用的程序库，远比一个独立的应用程序更加能够保持为最新状态，并被移植到新的系统之上。

对于被广泛应用的程序库的相关知识，也可以节省其他或未来的项目中的时间。

因此，如果你的应用领域中存在合适的程序库的话，请使用它。

示例

```
std::sort(begin(v), end(v), std::greater<>());
```

如果你不是排序算法方面的专家而且有大量时间的话，

这样的代码比你为特定的应用所编写的任何代码都更可能正确并且运行得更快。

不使用标准库（或者你的应用所采用的基础程序库）是需要明确理由的，而不是反过来。

注解

默认应当优先使用

- [ISO C++ 标准库](#)
- [指导方针支持库](#)

注解

如果某个重要的领域中不存在设计良好，文档全面，并且有良好支持的程序库的话，可能应当由你来设计并实现它，再进行使用了。

I: 接口

接口是程序中的两个部分之间的契约。严格地规定服务提供者和该服务使用者的预期是必要的。

在代码的组织中，良好的接口（易于理解，促进高效的使用方式，不易出错，支持进行测试，等等）可能是最重要的单个方面了。

接口规则概览：

- [I.1: 使接口明确](#)
- [I.2: 避免非 `const` 全局变量](#)
- [I.3: 避免使用单例](#)
- [I.4: 使接口严格和强类型化](#)
- [I.5: 说明前条件（如果有）](#)
- [I.6: 优先使用 `Requires\(\)` 来表达前条件](#)
- [I.7: 说明后条件](#)
- [I.8: 优先使用 `Ensures\(\)` 来表达后条件](#)
- [I.9: 当接口是模板时，用概念来文档化其参数](#)
- [I.10: 使用异常来表明无法实施所要求的任务](#)
- [I.11: 决不以原始指针 \(`T*`\) 或引用 \(`T&`\) 来传递所有权](#)
- [I.12: 把不能为空的指针声明为 `not_null`](#)
- [I.13: 不要只用一个指针来传递数组](#)
- [I.22: 避免全局对象之间进行复杂的初始化](#)
- [I.23: 保持较少的函数参数数量](#)
- [I.24: 避免可以由同一组实参以不同顺序调用造成不同含义的相邻形参](#)
- [I.25: 优先以空抽象类作为类层次的接口](#)
- [I.26: 当想要跨编译器的 ABI 时，使用一个 C 风格的语言子集](#)
- [I.27: 对于稳定的程序库 ABI，考虑使用 Pimpl 手法](#)
- [I.30: 将有违规则的部分封装](#)

参见

- [F: 函数](#)
- [C.concrete: 具体类型](#)
- [C.hier: 类层次](#)
- [C.over: 函数重载和重载运算符](#)
- [C.con: 容器和其他资源封装类](#)
- [E: 错误处理](#)
- [T: 模板和泛型编程](#)

I.1: 使接口明确

理由

正确性。未在接口中规定的假设很容易被忽视而且难于测试。

示例, 不好

通过全局（命名空间作用域）变量（调用模式）来控制函数的行为，是隐含的，而且潜在会造成困惑。例如：

```
int round(double d)
{
    return (round_up) ? ceil(d) : d;      // 请勿：“不可见的”依赖
}
```

两次调用 `round(7.2)` 的含义可能给出不同的结果，这对于调用者来说是不明显的。

例外

我们有时候会通过环境变量来控制一组操作的细节，比如常规/详细的输出，或者调试/优化版本。使用非局部的控制方式可能带来困惑，但可以只用来控制实现的细节，否则就只有固定的语义了。

示例, 不好

通过非局部变量（比如 `errno`）进行的报告经常被忽略。例如：

```
// 请勿如此：fprintf 的返回值未进行检查
fprintf(connection, "logging: %d %d %d\n", x, y, s);
```

要是连接已经关闭而导致没有产生日志输出的话会怎么样？参见 I.???

替代方案: 抛出异常。异常是无法被忽略的。

其他形式: 避免通过非局部或者隐含的状态来跨越接口传递信息。

注意，非 `const` 的成员函数会通过对对象的状态来向其他成员函数传递信息。

其他形式: 接口应当是函数或者一组函数集合。

函数可以是函数模板，而函数集合可以是类或者类模板。

强制实施

- 【简单】函数不能基于声明于命名空间作用域的变量来作出影响控制流的决定。
- 【简单】函数不能对声明于命名空间作用域的变量进行写入操作。

I.2: 避免非 `const` 全局变量

理由

非 `const` 全局变量能够隐藏依赖关系，并使这些依赖项可能出现无法预测的变动。

示例

```
struct Data {  
    // ... 大量成员 ...  
} data;           // 非 const 数据  
  
void compute()    // 请勿这样做  
{  
    // ... 使用 data ...  
}  
  
void output()     // 请勿这样做  
{  
    // ... 使用 data ...  
}
```

哪个可能会修改 `data` 呢？

警告: 全局对象的初始化并不是完全有序的。

当使用全局对象时，应当用常量为之初始化。

还要注意，即便对于 `const` 对象，也可能发生未定义的初始化顺序。

例外

全局对象通常优于单例。

注解

全局常量是有益的。

注解

针对全局变量的规则同样适用于命名空间作用域的变量。

替代方案: 如果你用全局（或者更一般地说命名空间作用域）数据来避免复制操作的话，请考虑把数据以 `const` 引用的形式进行传递的方案。

另一种方案是把数据定义为某个对象的状态，而把操作定义为其成员函数。

警告: 请关注数据竞争：当一个线程能够访问非局部数据（或以引用传递的数据），而另一个线程执行被调用的函数时，就可能带来数据竞争。

指向可变数据的每个指针或引用都是潜在的数据竞争。

使用全局指针或引用来访问和修改非 `const` 且非局部的数据，并非是比非 `const` 全局变量更好的替代方案，

这是因为它并不能解决隐藏依赖性或潜在竞争条件的问题。

注解

不可变数据是不会带来数据竞争条件的。

参见: 另见 [关于调用函数的规则](#)。

注解

这条规则是“避免”，而不是“不要用”。当然是有（罕见）例外的，比如 `cin`、`cout` 和 `cerr`。

强制实施

【简单】 报告所有在命名空间作用域中声明的非 `const` 变量和全局的指向非 `const` 数据的指针/引用。

I.3: 避免使用单例

理由

单例基本上就是经过伪装的更复杂的全局对象。

示例

```
class Singleton {  
    // ... 大量代码，用于确保只创建一个 singleton,  
    // 进行正确地初始化，等等  
};
```

单例的想法有许多变种。

这也是问题的一方面。

注解

如果不想让全局对象被改变，请将其声明为 `const` 或 `constexpr`。

例外

你可以使用最简单的“单例”形式（简单到通常不被当作单例）来获得首次使用时进行初始化的效果：

```
x& myX()  
{  
    static x my_x {3};  
    return my_x;  
}
```

这是解决初始化顺序相关问题的最有效方案之一。

在多线程环境中，静态对象的初始化并不会引入数据竞争条件
(除非你不小心在其构造函数中访问了某个共享对象)。

注意局部的 `static` 对象初始化并不会蕴含竞争条件。

不过，如果 `x` 的销毁中涉及了需要进行同步的操作的话，我们就得用一个不那么简单的方案。

例如：

```
x& myX()  
{  
    static auto p = new x {3};  
    return *p; // 有可能泄漏  
}
```

这样就必须有人以某种适当的方式来自由地 `delete` 这个对象了。

这是容易出错的，因此除了以下情况外我们并不使用这种技巧：

- `myX` 是在多线程代码中，
- 这个 `x` 对象需要销毁（比如由于它要释放某个资源），而且
- `x` 的析构函数的代码需要进行同步。

如果你和许多人一样把单例定义为只能创建一个对象的类的话，像 `myx` 这样的函数并非单例，而且这种好用的技巧并不算无单例规则的例外。

强制实施

通常非常困难。

- 查找名字中包含 `singleton` 的类。
- 查找只创建一个对象的类（通过对对象计数或者检查其构造函数）。
- 如果某个类 `X` 具有公开的静态函数，并且它包含具有该类 `X` 类型的函数级局部静态变量并返回指向它的指针或者引用，就禁止它。

I.4: 使接口严格和强类型化

理由

类型是最简单和最好的文档，它们有定义明确的含义并因而提高了易读性，并且是在编译期进行检查的。

而且，严格类型化的代码通常也能更好地进行优化。

示例，请勿这样做

考虑：

```
void pass(void* data); // 使用弱的并且缺乏明确性的类型 void* 是有问题的
```

调用者无法确定它允许使用哪些类型，而且因为它并没有指定 `const`，也不确定其数据是否会被改动。注意，任何指针类型都可以隐式转换为 `void*`，因此调用者很容易提供这样的值给它。

被调用方必须以 `static_cast` 将数据强制转换为某个无验证的类型以使用它。这样做易于犯错，而且啰嗦。

应当仅在设计中无法以 C++ 来予以描述的数据的传递时才使用 `const void*`。请考虑使用 `variant` 或指向基类的指针来代替它。

替代方案：通常，利用模板形参可以把 `void*` 排除而改为 `T*` 或者 `T&`。

对于泛型代码，这些个 `T` 可以是一般模板参数或者是概念约束的模板参数。

示例，不好

考虑：

```
draw_rect(100, 200, 100, 500); // 这些数值什么意思？
```

```
draw_rect(p.x, p.y, 10, 20); // 10 和 20 的单位是什么？
```

很明显调用者在描述一个矩形，不明确的是它们都和其哪些部分相关。而且 `int` 可以表示任何形式的信息，包括各种不同单位的值，因此我们必须得猜测这四个 `int` 的含义。前两个很可能代表坐标对偶 `x` 和 `y`，但后两个是什么呢？

注释和参数的名字可以有所帮助，但我们可以直截了当：

```
void draw_rectangle(Point top_left, Point bottom_right);
void draw_rectangle(Point top_left, size height_width);

draw_rectangle(p, Point{10, 20}); // 两个角点
draw_rectangle(p, Size{10, 20}); // 一个角和一对 (height, width)
```

显然，我们是无法利用静态类型系统识别所有的错误的，
例如，假定第一个参数是左上角这一点就依赖于约定（命名或者注释）。

示例，不好

考虑：

```
set_settings(true, false, 42); // 这些数值什么意思？
```

各参数类型及其值并不能表明其所指定的设置项是什么以及它们的值所代表的含义。

下面的设计则更加明确，安全且易读：

```
alarm_settings s{};
s.enabled = true;
s.displayMode = alarm_settings::mode::spinning_light;
s.frequency = alarm_settings::every_10_seconds;
set_settings(s);
```

对于一组布尔值的情况，可以考虑使用某种标记 `enum`；这是一种用于表示一组布尔值的模式。

```
enable_lamp_options(lamp_option::on | lamp_option::animate_state_transitions);
```

示例，不好

下例中，接口中并未明确给出 `time_to_blink` 的含义：按秒还是按毫秒算？

```
void blink_led(int time_to_blink) // 不好 -- 在单位上含糊
{
    // ...
    // 对 time_to_blink 做一些事
    // ...
}

void use()
{
    blink_led(2);
}
```

示例，好

`std::chrono::duration` 类型可以让时间段的单位明确下来。

```

void blink_led(milliseconds time_to_blink) // 好 -- 单位明确
{
    // ...
    // 对 time_to_blink 做一些事
    // ...
}

void use()
{
    blink_led(1500ms);
}

```

这个函数还可以写成使其接受任何时间段单位的形式。

```

template<class rep, class period>
void blink_led(duration<rep, period> time_to_blink) // 好 -- 接受任何单位
{
    // 假设最小的有意义单位是毫秒
    auto milliseconds_to_blink = duration_cast<milliseconds>(time_to_blink);
    // ...
    // 对 milliseconds_to_blink 做一些事
    // ...
}

void use()
{
    blink_led(2s);
    blink_led(1500ms);
}

```

强制实施

- 【简单】报告将 `void*` 用作参数或返回类型的情况
- 【简单】报告使用了多个 `bool` 参数的情况
- 【难于做好】查找使用了过多基础类型的参数的函数。

I.5: 说明前条件 (如果有)

理由

在参数上蕴含着使它们在被调用方中能够恰当使用的约束关系。

示例

考虑：

```
double sqrt(double x);
```

这里 `x` 必须是非负数。类型系统是无法（简洁并自然地）表达这点的，因而我们得用别的方法。例如：

```
double sqrt(double x); // x 必须是非负数
```

一些前条件可以表示为断言。例如：

```
double sqrt(double x) { Expects(x >= 0); /* ... */ }
```

理想情况下，这个 `Expects(x >= 0)` 应当是 `sqrt()` 的接口的一部分，但我们无法轻易做到这点。当前，我们将之放入定义式（函数体）之中。

参考: `Expects()` 在 [GSL](#) 中有说明。

注解

优先使用正式的必要条件说明，比如 `Expects(!p);`。

如果这样不可行，就在注释中使用文字来说明，比如 `// 序列 [p:q) 根据 < 排序。`

注解

许多成员函数都以某个类所保持的不变式作为一项前条件。

这个不变式是由构造函数所建立的，且必须在被从类之外所调用的每个成员函数的退出时重新建立。

我们并不需要对每个成员函数都说明这个不变式。

强制实施

【无法强制实施】

参见: 有关传递指针的规则。???

I.6: 优先使用 `Expects()` 来表达前条件

理由

清晰地表明这个条件是一个前条件，并便于工具的利用。

示例

```
int area(int height, int width)
{
    Expects(height > 0 && width > 0);           // 好
    if (height <= 0 || width <= 0) my_error();    // 隐晦的
    // ...
}
```

注解

前条件是可以用许多方式来说明的，包括代码注释，`if` 语句，以及 `assert()`。

这些方式使其难于与普通代码之间进行区分，难于进行更新，难于利用工具来操作，而且可能具有错误的语义（你真的总是想要在调试模式中止程序而在生产运行中不做任何检查吗？）

注解

前条件应当是接口的一部分，而不是实现的一部分，

但我们至今还没有能够做到这点的语言设施。

一旦语言支持变为可用（例如，参见[契约提案](#)），我们就将会采用前条件，后条件和断言的标准版本。

注解

`Expects()` 还可以用于在算法的中部来检查某个条件。

注解

使用 `unsigned` 并不是回避[确保非负数值](#)问题的好方法。

强制实施

【无法强制实施】要把各种对前条件进行断言的方式都找出来是不可行的。对那些易于识别的（如 `assert()`）实例给出警告的做法，其意义在缺少语言设施的前提下是有问题的。

I.7: 说明后条件

理由

以检测到对返回结果的误解，还可能发现实现中存在错误。

示例，不好

考虑：

```
int area(int height, int width) { return height * width; } // 不好
```

这里，我们（粗心大意地）遗漏了前条件的说明，因此高度和宽度必须是正数这点是不明确的。

我们也遗漏了后条件的说明，因此算法 (`height * width`) 对于大于最大整数的面积来说是错误的这点是不明显的。

可能会有溢出。

应该考虑使用：

```
int area(int height, int width)
{
    auto res = height * width;
    Ensures(res > 0);
    return res;
}
```

示例，不好

考虑一个著名的安全性 BUG：

```
void f() // 有问题的
{
    char buffer[MAX];
    // ...
    memset(buffer, 0, sizeof(buffer));
}
```

由于没有后条件来说明缓冲区应当被清零，优化器可能会将这个看似多余的 `memset()` 调用给清除掉：

```
void f() // 有改进
{
    char buffer[MAX];
    // ...
    memset(buffer, 0, sizeof(buffer));
    Ensures(buffer[0] == 0);
}
```

注解

后条件通常是在说明函数目的的代码注释中非正式地进行说明的；用 `Ensures()` 可以使之更加系统化，更加明显，并且更容易检查。

注解

后条件对于那些无法在所返回的结果中直接体现的东西来说尤其重要，比如要说明所用的数据结构。

示例

考虑一个操作 `Record` 的函数，它使用 `mutex` 来避免数据竞争条件：

```
mutex m;

void manipulate(Record& r)      // 请勿这样做
{
    m.lock();
    // ... 没有 m.unlock() ...
}
```

这里，我们“忘记”说明应当释放 `mutex`，因此我们搞不清楚这里 `mutex` 释放的缺失是一个 BUG 还是一种功能特性。

把后条件说明将使其更加明确：

```
void manipulate(Record& r)      // 后条件：m 在退出后是未锁定的
{
    m.lock();
    // ... 没有 m.unlock() ...
}
```

现在这个 BUG 就明显了（但仅对阅读了代码注释的人类来说）。

更好的做法是使用 [RAII](#) 来在代码中保证后条件（“锁必须进行释放”）的实施：

```
void manipulate(Record& r)      // 最好这样
{
    lock_guard<mutex> _{m};
    // ...
}
```

注解

理想情况下，后条件应当在接口或声明式中说明，让使用者易于见到它们。

只有那些与使用者有关的后条件才应当在接口中说明。

仅与内部状态相关的后条件应当属于定义式或实现。

强制实施

【无法强制实施】这是一条理念性的指导方针，一般情况下进行直接的检查是不可行的。不过许多工具链中都有适用于特定领域的检查器，比如针对锁定持有情况的检查器。

I.8: 优先使用 `Ensures()` 来表达后条件

理由

清晰地表明这个条件是一个后条件，并便于工具的利用。

示例

```
void f()
{
    char buffer[MAX];
    // ...
    memset(buffer, 0, MAX);
    Ensures(buffer[0] == 0);
}
```

注解

后条件是可以用许多方式来说明的，包括代码注释，`if` 语句，以及 `assert()`。

这些方式使其难于与普通代码之间进行区分，难于进行更新，难于利用工具来操作，而且可能具有错误的语义。

替代方案: 如“这个资源必须被释放”这样形式的后条件最好以 [RAII](#) 的方式来表达。

注释

理想情况下，`Ensures` 应当是接口的一部分，但我们无法轻易做到这点。

当前，我们将之放入定义式（函数体）之中。

一旦语言支持变为可用（例如，参见[契约提案](#)），我们就将会采用前条件，后条件和断言的标准版本。

强制实施

【无法强制实施】 要把各种对后条件进行断言的方式都找出来是不可行的。对那些易于识别的（如 `assert()`）实例给出警告的做法，其意义在缺少语言设施的前提下是有问题的。

I.9: 当接口是模板时，用概念来文档化其参数

理由

更严谨地说明接口，并使其在（不远的）将来可以在编译时进行检查。

示例

使用 C++20 风格的必要条件说明。例如：

```
template<typename Iter, typename Val>
    requires input_iterator<Iter> && equality_comparable_with<iter_value_t<Iter>,
Val>
Iter find(Iter first, Iter last, Val v)
{
    // ...
}
```

参见：[泛型编程](#)和[概念](#)。

强制实施

对未被概念所约束（在其声明式之中或者在一个 `requires` 子句中所给出）的并非可变数量的模板形参作出警告。

I.10: 使用异常来表明无法实施所要求的任务

理由

不应该让错误可以被忽略，因为这将导致系统或者一次运算进入未定义的（或者预料之外的）状态。这是错误的一个主要来源。

示例

```
int printf(const char* ...);      // 不好：当输出失败时返回负值

template<class F, class ...Args>
// 好：当无法启动一个新的线程时抛出 system_error
explicit thread(F&& f, Args&&... args);
```

注解

错误是什么？

错误的含义是函数无法达成其所宣称的目标（这包括后条件的建立）。

把错误忽略掉的调用方代码将导致错误的结果，或者未定义的系统状态。

例如，无法连接一个远程服务器本身并不是错误：

这个服务器可以因为各种原因而拒绝连接，因此合乎常理的方式是让其返回一个其调用者必然要检查的结果。

不过，如果无法连接本身就是被当作一种错误的话，这个失败时应当抛出一个异常。

例外

许多传统的接口函数（比如 UNIX 的信号处理器）都使用错误代码（就是 `errno`）来报告其实是状态代码而不是错误的东西。你没有更好的选择只能用它，因此对其调用并不违反本条规则。

替代方案

如果你不能使用异常（比如说由于你的代码全都是老式的原始指针用法，或者由于你有硬实时性的约束），请考虑使用返回一对值的代码风格：

```
int val;
int error_code;
tie(val, error_code) = do_something();
if (error_code) {
    // ... 处理错误或者退出 ...
}
// ... 使用 val ...
```

这种风格不幸地会导致未初始化的变量。

从 C++17 开始，可以使用“结构化绑定”功能特性来从返回值直接对多个变量初始化。

```
auto [val, error_code] = do_something();
if (error_code) {
    // ... 处理错误或者退出 ...
}
// ... 使用 val ...
```

注解

我们并不认为“性能”是一种不使用异常的合理理由。

- 通常，显式的错误检查和处理会消耗掉和异常处理一样多的时间和空间。
- 通常，使用异常的更清晰的代码会带来更好的性能（简化了对程序执行路径的追踪和其优化）。
- 一条对性能关键代码的好规则是，把检查从代码的关键部分中移出去。
- 长期来看，更规整的代码会得到更好的优化。
- 在做出性能相关的声明前一定要小心地进行测量。

参见: [I.5](#) 和 [I.7](#) 有关报告前条件和后条件的违反。

强制实施

- 【无法强制实施】这是一条理念性的指导方针，进行直接的检查是不可行的。
- 查找 `errno`。

I.11: 决不以原始指针 (`T*`) 或引用 (`T&`) 来传递所有权

理由

如果对调用者和被调用方哪一个拥有对象有疑问，那就会造成泄漏或者发生提早的析构。

示例

考虑：

```
X* compute(args)    // 请勿这样做
{
    X* res = new X{};
    // ...
    return res;
}
```

应当由谁来删除返回的这个 `X` 呢？如果 `compute` 返回引用的话这个问题将更难发现。

应该考虑按值来返回结果（如果结果比较大的话就用移动语义）：

```
vector<double> compute(args) // 好的
{
    vector<double> res(10000);
    // ...
    return res;
}
```

替代方案: 用“智能指针”来传递所有权，比如 `unique_ptr` (专有所有权) 和 `shared_ptr` (共享所有权)。

这样做比返回对象自身来说并没有那么简炼，而且通常也不那么高效，因此，仅当需要引用语义时再使用智能指针。

替代方案: 有时候因为 ABI 兼容性的要求或者缺少资源，是无法对老代码进行修改的。

这种情况下，请用[指导方针支持库](#)的 `owner` 来标记拥有对象的指针：

```
owner<X*> compute(args)      // 现在就明确传递了所有权这一点
{
    owner<X*> res = new X{};
    // ...
    return res;
}
```

这告诉了分析工具 `res` 是一个所有者。

就是说，它的值必须被 `delete`，或者被传递给另一个所有者，正如这里的 `return` 所做。

在资源包装类的实现中也同样使用了 `owner`。

注解

以原始指针（或迭代器）的形式传递的对象，都假定是由调用方所有的，因此其生存期也由调用方来处理。换种方式来看：

传递所有权的 API 相对于传递指针的 API 来说比较少见，因此缺省情况就是“不传递所有权”。

参见：[实参传递](#)，[使用智能指针参数](#)，以及[返回值](#)。

强制实施

- 【简单】当对并非 `owner<T>` 的原始指针进行 `delete` 就发出警告。建议使用标准库的资源包装或者使用 `owner<T>`。
- 【简单】当任何代码路径上遗漏了对 `owner` 指针的 `reset` 或者显式的 `delete` 时就发出警告。
- 【简单】当把 `new` 或者返回值为 `owner` 的函数的返回值赋值给原始指针或非 `owner` 的引用时就发出警告。

I.12: 把不能为空的指针声明为 `not_null`

理由

帮助避免对 `nullptr` 解引用的错误。

通过避免多余的 `nullptr` 检查来提高性能。

示例

```
int length(const char* p);           // 不清楚 length(nullptr) 是否有效

length(nullptr);                    // OK?

int length(not_null<const char*> p); // 有改善：可以假定 p 不可能为 nullptr

int length(const char* p);           // 只好假定 p 可以为 nullptr
```

通过在源代码中说明意图，实现者和工具就可以提供更好的诊断能力，比如通过静态分析来找出某些种类的错误，还可以实施优化，比如移除分支和空值测试。

注解

`not_null` 在[指导方针支持库](#)中定义。

注解

指向 `char` 的指针将指向 C 风格的字符串（以零终结的字符的连续串）这一点仍然是潜规则，并且也是混乱和错误的潜在来源。请使用 `cstring` 来代替 `const char*`。

```
// 可以假定 p 不能为 nullptr  
// 可以假定 p 指向以零终结的字符数组  
int length(not_null<zstring> p);
```

注意：`length()` 显然是经过伪装的 `std::strlen()`。

强制实施

- 【简单】【基础】如果有函数在所有控制流路径上访问指针参数之前检查它是否是 `nullptr`，则给出警告称其应当被声明为 `not_null`。
- 【复杂】如果有指针返回值的函数在所有返回路径上都保证其不是 `nullptr`，则给出警告称返回类型应当被声明为 `not_null`。

I.13: 不要只用一个指针来传递数组

理由

(pointer, size) 式的接口是易于出错的。同样，(指向数组的) 普通指针还必须依赖某种约定以使被调用方来确定其大小。

示例

考虑：

```
void copy_n(const T* p, T* q, int n); // 从 [p:p+n) 复制到 [q:q+n)
```

当由 `q` 所指向的数组少于 `n` 个元素会怎么样？此时我们将覆盖一些可能无关的内存。

当由 `p` 所指向的数组少于 `n` 个元素会怎么样？此时我们将读取一些可能无关的内存。

此二者都是未定义的行为，而且可能是非常恶劣的 BUG。

替代方案

考虑使用明确的 `span`：

```
void copy(span<const T> r, span<T> r2); // 将 r 复制给 r2
```

示例，不好

考虑：

```
void draw(Shape* p, int n); // 糟糕的接口；糟糕的代码  
circle arr[10];  
// ...  
draw(arr, 10);
```

把 10 作为参数 n 传递可能是错误的：虽然最常见的约定是假定有 `[0:n]`，但这点并未不是明确的。更糟糕的是，`draw()` 的调用通过编译了：这里有一次从数组到指针的隐式转换（数组退化），然后又进行了从 `circle` 到 `shape` 的另一次隐式转换。`draw()` 是不可能安全地迭代这个数组的：它无法知道元素的大小。

替代方案：使用一个辅助类来确保元素的数量正确，并避免进行危险的隐式转换。例如：

```
void draw2(span<circle>);  
circle arr[10];  
// ...  
draw2(span<circle>(arr)); // 推断出元素的数量  
draw2(arr); // 推断出元素的类型和数组大小  
  
void draw3(span<Shape>);  
draw3(arr); // 错误：无法将 circle[10] 转换为 span<shape>
```

这个 `draw2()` 传递了与 `draw()` 同样数量的信息，但明确指定了它接受的是 `circle` 的范围。参见 `???`。

例外

使用 `zstring` 和 `czstring` 来表示 C 风格的以零终结字符串。

但这样做时，应当使用 `std::string_view` 或 [GSL](#) 中的 `span<char>` 以避免范围错误。

强制实施

- 【简单】〔边界〕对任何依赖于从数组类型向指针类型的隐式转换的表达式给出警告。允许 `zstring/czstring` 指针类型的例外。
- 【简单】〔边界〕对任何指针类型表达式进行且结果为指针类型的值的运算操作给出警告。允许 `zstring/czstring` 指针类型的例外。

I.22: 避免全局对象之间进行复杂的初始化

理由

复杂的初始化可能导致未定义的执行顺序。

示例

```
// file1.c  
  
extern const X x;  
  
const Y y = f(x); // 读取 x; 写入 y  
  
// file2.c  
  
extern const Y y;  
  
const X x = g(y); // 读取 y; 写入 x
```

由于 `x` 和 `y` 是处于不同翻译单元之内的，调用 `f()` 和 `g()` 的顺序就是未定义的；我们可能会访问到还未初始化的 `const` 对象。

这里展示的是，全局（命名空间作用域）对象的初始化顺序难题并不仅限于全局变量而已。

注解

并发代码中的初始化顺序问题是更加难于处理的。
所以通常最好完全避免使用全局（命名空间作用域）的对象。

强制实施

- 标记调用了非 `constexpr` 函数的全局初始化式
- 标记访问了 `extern` 对象的全局初始化式

I.23: 保持较少的函数参数数量

理由

大量参数会带来更大的出现混乱的机会。大量传递参数与其他替代方案相比也通常是代价比较大的。

讨论

两个最常见的使得函数具有过多参数的原因是：

1. 缺乏抽象
缺少一种抽象，使得一个组合值被以一组独立的元素的方式进行传递，而不是以一个单独的保证了不变式的对象来传递。
这不仅使其参数列表变长，而且会导致错误，
因为各个成分值无法再被某种获得保证的不变式进行保护。
2. 违反了“函数单一职责”原则
这个函数试图完成多项任务，它可能应当被重构。

示例

标准库的 `merge()` 函数达到了我们可以自如处理的界限：

```
template<class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result, Compare comp);
```

注意，这属于上面的第一种问题：缺乏抽象。STL 传递的不是范围（抽象），而是一对迭代器（未封装的成分值）。

其中有四个模板参数和六个函数参数。

为简化最常用和最简单的用法，比较器参数可以缺省使用 `<`：

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result);
```

这实际上不会减低其整体复杂性，但它减少了对于许多使用者的表面复杂性。
为了真正地减少参数的数量，我们得把参数归拢到更高层的抽象之中：

```
template<class InputRange1, class InputRange2, class OutputIterator>
OutputIterator merge(InputRange1 r1, InputRange2 r2, OutputIterator result);
```

把参数成“批”进行组合是减少参数数量和增加进行检查的一般性技巧。

或者，我们也可以用标准库概念来定义“三个类型必须可以用于归并”：

```
template<class In1, class In2, class Out>
    requires mergeable<In1, In2, Out>
    Out merge(In1 r1, In2 r2, Out result);
```

示例

安全性剖面配置中建议将以下代码

```
void f(int* some_ints, int some_ints_length); // 不好: C 风格, 不安全
```

替换为

```
void f(gsl::span<int> some_ints); // 好: 安全, 有边界检查
```

这样，使用一种抽象可以获得安全性和健壮性的好处，而且自然地减少了参数的数量。

注解

多少参数算很多？请使用少于四个参数。

有些函数确实最好表现为四个独立的参数，但这样的函数并不多。

替代方案: 使用更好的抽象：把参数归集为由意义的对象，然后（按值或按引用）传递这些对象。

替代方案: 利用默认实参或者重载来让最常见的调用方式可以用比较少的实参来进行。

强制实施

- 当函数声明了两个类型相同的迭代器（也包括指针）而不是一个范围或视图，就给出警告。
- 【无法强制实施】这是一条理念性的指导方针，进行直接的检查是不可行的。

I.24: 避免可以由同一组实参以不同顺序调用造成不同含义的相邻形参

理由

相同类型的相邻参数很容易被不小心互换掉。

示例，不好

考虑：

```
void copy_n(T* p, T* q, int n); // 从 [p:p + n] 复制到 [q:q + n]
```

这是个 K&R C 风格接口的一种恶劣的变种。它导致很容易把“目标”和“来源”参数搞反。

可以在“来源”参数上使用 `const`：

```
void copy_n(const T* p, T* q, int n); // 从 [p:p + n] 复制到 [q:q + n]
```

例外

当参数的顺序不重要时，不会造成问题：

```
int max(int a, int b);
```

替代方案

不要以指针来传递数组，而要传递用来表示一个范围的对象（比如一个 `span`）：

```
void copy_n(span<const T> p, span<T> q); // 从 p 复制到 q
```

替代方案

定义一个 `struct` 来作为参数类型，并依照各个参数来命名它的各字段：

```
struct SystemParams {  
    string config_file;  
    string output_path;  
    seconds timeout;  
};  
void initialize(SystemParams p);
```

这样做带来一种使其调用代码对于以后的读者变得明晰的倾向，因为这种参数在调用点通常都要按名字来进行填充。

注解

只有接口设计者才能胜任处理违反本条指导方针的源头问题。

强制实施策略

【简单】当两个连续的参数具有相同的类型时就给出警告。

我们仍在寻找不这么简单的强制实施方式。

I.25: 优先以空抽象类作为类层次的接口

理由

空的（没有非静态成员数据）抽象类要比带有状态的基类更倾向于保持稳定。

示例，不好

你知道 `shape` 总会冒出来的 :-)

```

class Shape { // 不好：接口类中加载了数据
public:
    Point center() const { return c; }
    virtual void draw() const;
    virtual void rotate(int);
    // ...
private:
    Point c;
    vector<Point> outline;
    Color col;
};

```

这将强制性要求每个派生类都要计算出一个中心点——即使这不容易，而且这个中心点从不会被用到。相似地说，不是每个 `Shape` 都有一个 `color`，而许多 `shape` 也最好别用一个定义成一系列 `Point` 的轮廓来进行表示。使用抽象类要更好：

```

class Shape { // 有改进：Shape 是一个纯接口
public:
    virtual Point center() const = 0; // 纯虚函数
    virtual void draw() const = 0;
    virtual void rotate(int) = 0;
    // ...
    // ... 没有数据成员 ...
    // ...
    virtual ~Shape() = default;
};

```

强制实施

【简单】 当把类 `C` 的指针/引用赋值给 `C` 的某个基类的指针/引用，而这个基类包含数据成员时，就给出警告。

I.26: 当想要跨编译器的 ABI 时，使用一个 C 风格的语言子集

理由

不同的编译器会实现不同的类的二进制布局，异常处理，函数名字，以及其他实现细节。

例外

在一些平台上正有公共的 ABI 兴起，这可以使你从更加苛刻的限制中摆脱出来。

注解

如果你只用一种编译器，你也可以在接口上使用完全的 C++。但当升级到新的编译器版本之后，可能需要进行重新编译。

强制实施

【无法强制实施】 要可靠地识别某个接口是否是构成 ABI 的一部分是很困难的。

I.27: 对于稳定的程序库 ABI, 考虑使用 Pimpl 手法

理由

由于私有数据成员参与类的内存布局, 而私有成员函数参与重载决议, 对这些实现细节的改动都要求使用了这类的所有用户全部重新编译。而持有指向实现的指针 (Pimpl) 的
非常多态的接口类, 则可以将类的用户从其实现的改变隔离开来, 其代价是一层间接。

示例

接口 (widget.h)

```
class widget {
    class impl;
    std::unique_ptr<impl> pimpl;
public:
    void draw(); // 公开 API 转发给实现
    widget(int); // 定义于实现文件中
    ~widget(); // 定义于实现文件中, 其中 impl 将为完整类型
    widget(widget&&); // 定义于实现文件中
    widget(const widget&) = delete;
    widget& operator=(widget&&); // 定义于实现文件中
    widget& operator=(const widget&) = delete;
};
```

实现 (widget.cpp)

```
class widget::impl {
    int n; // private data
public:
    void draw(const widget& w) { /* ... */ }
    impl(int n) : n(n) {}
};
void widget::draw() { pimpl->draw(*this); }
widget::widget(int n) : pimpl{std::make_unique<impl>(n)} {}
widget::widget(widget&&) = default;
widget::~widget() = default;
widget& widget::operator=(widget&&) = default;
```

注解

参见 [GOTW #100](#) 和 [cppreference](#) 有关这个手法相关的权衡和其他实现细节。

强制实施

【无法强制】很难可靠地识别出哪个接口属于 ABI 的一部分。

I.30: 将有违规则的部分封装

理由

维持代码简单且安全。

有时候因为逻辑的或者性能的原因，需要使用难看的，不安全的或者易错的技术。

此时，将它们局部化，而不是使其“感染”接口，可以避免更多的程序员团队必须当心其细节和微妙之处。

如果可能的话，实现复杂度不能通过接口渗透到用户代码之中。

示例

考虑一个程序，其基于某种形式的输入（比如 `main` 的实参）来决定从文件，从命令行，还是从标准输入来获得输入数据。

我们可能会将其写成

```
bool owned;
owner<iostream*> inp;
switch (source) {
    case std_in:      owned = false; inp = &cin;                      break;
    case command_line: owned = true;   inp = new istreamstring{argv[2]}; break;
    case file:         owned = true;   inp = new ifstream{argv[2]};     break;
}
istream& in = *inp;
```

这违反了避免未初始化变量，

避免忽略所有权，

和避免魔法常量等规则。

尤其是，人们必须记得找地方写

```
if (owned) delete inp;
```

我们可以通过使用带有一个特殊的删除器（对 `cin` 不做任何事）的 `unique_ptr` 来处理这个特定的例子，

但这对于新手来说较复杂（他们很容易遇到这种问题），并且这个例子其实是一个更一般的问题的特例：

我们希望将其当做静态的某种属性（此处为所有权），需要在运行时进行偶尔的处理。

一般的，更常见的，且更安全的例子可以被静态处理，因而我们并不希望为它们添加开销和复杂性。然而我们还是不得不处理那些不常见的，较不安全的，而且更为昂贵的情况。

[\[Str15\]](#) 中对这种例子有所探讨。

由此，我们编写这样的类

```

class Istream { [[gs]::suppress(lifetime)]]
public:
    enum Opt { from_line = 1 };
    Istream() { }
    Istream(zstring p) : owned{true}, inp{new ifstream{p}} {}           // 从文件
读取
    Istream(zstring p, Opt) : owned{true}, inp{new istringstream{p}} {} // 从命令
行读取
    ~Istream() { if (owned) delete inp; }
    operator istream& () { return *inp; }
private:
    bool owned = false;
    istream* inp = &cin;
};

```

这样，`istream` 的所有权的动态本质就被封装起来。

大体上，在现实的代码中还是需要针对潜在的错误添加一些检查。

强制实施

- 很难，判断那种违背规则的代码是基本的是很难做到的
- 对允许规则违背的部分跨越接口的规则抑制进行标记

F: 函数

函数指定了一个活动或者一次计算，以将系统从一种一致的状态转移到另一种一致的状态。函数是程序的基础构造块。

应当使函数的名字有意义，说明对其参数的必要条件，并清晰地规定参数和其结果之间的关系。函数的实现本身并不是规格说明。请尝试同时对函数应当做什么和函数应当怎样做来进行思考。

函数在大多数接口中都是最关键的部分，请参考接口的规则。

函数规则概览：

函数定义式的规则：

- [F.1: 把有意义的操作“打包”成为精心命名的函数](#)
- [F.2: 一个函数应当实施单一逻辑操作](#)
- [F.3: 保持函数短小简洁](#)
- [F.4: 如果函数可能必须在编译期进行求值，就将其声明为 `constexpr`](#)
- [F.5: 如果函数非常小，并且是时间敏感的，就将其声明为 `inline`](#)
- [F.6: 如果函数必然不会抛出异常，就将其声明为 `noexcept`](#)
- [F.7: 对于常规用法，应当接受 `T*` 或 `T&` 参数而不是智能指针](#)
- [F.8: 优先采用纯函数](#)
- [F.9: 未使用的形参应当没有名字](#)
- [F.10: 若操作可被重用，则应为其命名](#)
- [F.11: 当需要仅在一处使用的简单函数对象时使用无名 lambda](#)

参数传递表达式的规则：

- [F.15: 优先采用简单的和传统的信息传递方式](#)
- [F.16: 对于“输入 \(in\)”参数，把复制操作廉价的类型按值进行传递，把其他类型按 `const` 引用进行传递](#)
- [F.17: 对于“输入/输出 \(in-out\)”参数，按非 `const` 引用进行传递](#)

- F.18: 对于“将被移动 (will-move-from) ”参数，按 `x&&` 进行传递并对参数 `std::move`
- F.19: 对于“转发 (forward) ”参数，按 `TP&&` 进行传递并只对参数 `std::forward`
- F.20: 对于“输出 (out) ”值，采用返回值优先于输出参数
- F.21: 要返回多个“输出”值，优先返回结构体或元组 (tuple)
- F.60: 当“没有参数”是有效的选项时，采用 `T*` 优先于 `T&`

参数传递语义的规则：

- F.22: 用 `T*` 或 `owner<T*>` 来代表单个对象
- F.23: 用 `not_null<T>` 来表明“空值 (null) ”不是有效的值
- F.24: 用 `span<T>` 或者 `span_p<T>` 来代表一个半开序列
- F.25: 用 `zstring` 或者 `not_null<zstring>` 来代表 C 风格的字符串
- F.26: 当需要指针时，用 `unique_ptr<T>` 来传递所有权
- F.27: 用 `shared_ptr<T>` 来共享所有权

值返回语义的规则：

- F.42: 返回 `T*` 来 (仅仅) 给出一个位置
- F.43: 不要 (直接或间接) 返回指向局部对象的指针或引用
- F.44: 当不想进行复制，且不需要“没有对象被返回”时，返回 `T&`
- F.45: 不要返回 `T&&`
- F.46: `int` 是 `main()` 的返回类型
- F.47: 赋值运算符返回 `T&`
- F.48: 不要用 `std::move(local)`
- F.49: 不要返回 `const T`

其他函数规则：

- F.50: 当函数不适用时 (不能俘获局部变量，或者不能编写局部函数)，就使用 Lambda
- F.51: 如果需要作出选择，采用默认实参应当优先于进行重载
- F.52: 对于局部使用的 (也包括传递给算法的) lambda，优先采用按引用俘获
- F.53: 对于非局部使用的 (包括被返回的，在堆上存储的，或者传递给别的线程的) lambda，避免采用按引用俘获
- F.54: 当俘获了 `this` 时，显式俘获所有的变量 (不使用默认俘获)
- F.55: 不要使用 `va_arg` 参数
- F.56: 避免不必要的条件嵌套

函数和 Lambda 表达式以及函数对象有很强的相似性。

参见：[C.lambdas: 函数对象和 lambda](#)

F.def: 函数的定义式

函数的定义式就是一并指定了函数的实现 (函数体) 的函数声明式。

F.1: 把有意义的操作“打包”成为精心命名的函数

理由

把公共的代码分解出去，将使代码更易于阅读，更可能被重用，并能够对源于复杂代码的错误有所限制。

如果某部分是一个明确指定的活动，就将之从其包围代码中分离出来，并对其进行命名。

示例，请勿这样做

```
void read_and_print(istream& is)    // 读取并打印一个 int
{
    int x;
    if (is >> x)
        cout << "the int is " << x << '\n';
    else
        cerr << "no int on input\n";
}
```

`read_and_print` 的几乎每件事都有问题。

它进行了读取，它（向一个固定 `ostream`）进行了写入，它（向一个固定的 `ostream`）写入了错误消息，它只能处理 `int`。

这里没有可以重用的东西，逻辑上分开的操作被搅拌到了一起，而局部变量在其逻辑上使用完毕之后仍处于作用域中。

作为一个小例子的话还好，但如果输入操作、输出操作和错误处理更加复杂的话，这个纠缠混乱的代码就会变得难于理解了。

注解

如果你编写的一个有些价值的 lambda 可能潜在地被用于多处，那就为它进行命名并将其赋值给一个（通常非局部的）变量。

示例

```
sort(a, b, [](T x, T y) { return x.rank() < y.rank() && x.value() < y.value();
});
```

对 lambda 进行命名，将会把这个表达式进行逻辑上的分解，还会为 lambda 的含义给出有力的提示。

```
auto lessT = [](T x, T y) { return x.rank() < y.rank() && x.value() < y.value();
};

sort(a, b, lessT);
```

对于性能和可维护性来说，最简短的代码并不总是最好的选择。

例外

循环体，包括用作循环体的 lambda，很少需要进行命名。

然而，大型的循环体（比如好多行或者好多页）也是个问题。

规则“[保持函数短小简洁](#)”暗含有“保持循环体短小”。

与此相似，用作回调参数的 lambda 有事后也是有意义的，虽然它们不大可能被重用。

强制实施

- 参见“[保持函数短小简洁](#)”
- 把不同地方所用的同样和非常相似的 lambda 标记出来。

F.2: 一个函数应当实施单一逻辑操作

理由

仅实施单一操作的函数易于理解，测试和重用。

示例

考虑：

```
void read_and_print()    // 不好
{
    int x;
    cin >> x;
    // 检查错误
    cout << x << "\n";
}
```

这是一整块被绑定到一个特定的输入的代码，而且无法为其找到另一种（不同的）用途。作为代替，我们把函数分解为合适的逻辑部分并进行参数化：

```
int read(istream& is)    // 有改进
{
    int x;
    is >> x;
    // 检查错误
    return x;
}

void print(ostream& os, int x)
{
    os << x << "\n";
}
```

这样的话，就可以在需要时进行组合：

```
void read_and_print()
{
    auto x = read(cin);
    print(cout, x);
}
```

如果有需要，我们还可以进一步把 `read()` 和 `print()` 针对数据类型，I/O 机制，以及对错误的反应等等方面进行模板化。例如：

```

auto read = [] (auto& input, auto& value)      // 有改善
{
    input >> value;
    // 检查错误
};

auto print(auto& output, const auto& value)
{
    output << value << "\n";
}

```

强制实施

- 把具有多个“输出”参数的函数当作有问题的。使用返回值来代替，包括以 `tuple` 用作多个返回值。
- 把无法装入编辑器的一屏之内的“大型”函数当作有问题的。考虑把这种函数分解为较小的恰当命名的子操作。
- 把有七个或更多参数的函数当作有问题的。

F.3: 保持函数短小简洁

理由

大型函数难于阅读，更有可能包含复杂的代码，而且更有可能含有其作用域超过最低限度的变量。带有复杂的控制结构的函数更有可能变长，也更有可能隐藏逻辑错误于其中。

示例

考虑：

```

double simple_func(double val, int flag1, int flag2)
// simple_func: 接受一个值并计算所需的 ASIC 值,
// 依赖于两个模式标记。
{
    double intermediate;
    if (flag1 > 0) {
        intermediate = func1(val);
        if (flag2 % 2)
            intermediate = sqrt(intermediate);
    }
    else if (flag1 == -1) {
        intermediate = func1(-val);
        if (flag2 % 2)
            intermediate = sqrt(-intermediate);
        flag1 = -flag1;
    }
    if (abs(flag2) > 10) {
        intermediate = func2(intermediate);
    }
    switch (flag2 / 10) {
        case 1: if (flag1 == -1) return finalize(intermediate, 1.171);
                  break;
        case 2: return finalize(intermediate, 13.1);
        default: break;
    }
}

```

```
    return finalize(intermediate, 0.);  
}
```

这个函数过于复杂了。

要如何判断是否所有的可能性都被正确处理了呢？

当然，它也同样违反了别的规则。

我们可以进行重构：

```
double func1_muon(double val, int flag)  
{  
    // ???  
}  
  
double func1_tau(double val, int flag1, int flag2)  
{  
    // ???  
}  
  
double simple_func(double val, int flag1, int flag2)  
// simple_func: 接受一个值并计算所需的 ASIC 值,  
// 依赖于两个模式标记。  
{  
    if (flag1 > 0)  
        return func1_muon(val, flag2);  
    if (flag1 == -1)  
        // 由 func1_tau 来处理: flag1 = -flag1;  
        return func1_tau(-val, flag1, flag2);  
    return 0.;  
}
```

注解

“无法放入一屏显示”通常是对“太长了”的一种不错的实际定义方式。

一行到五行大小的函数应当被当作是常态。

注解

把大型函数分解成较小的精致的有名字的函数。

小型的简单函数在函数调用的代价比较明显时很容易被内联。

强制实施

- 标记无法“放入一屏”的函数。

一屏有多大？可以试试 60 行，每行 140 个字符；这大致上就是书本页面能够适于阅读的最大值了。

- 标记过于复杂的函数。多复杂算是过于复杂呢？

应当用圈复杂度来度量。可以试试“多于 10 个逻辑路径”。一个简单的开关算作一条路径。

F.4: 如果函数可能必须在编译期进行求值，就将其声明为 `constexpr`

理由

需要用 `constexpr` 来告诉编译器允许对其进行编译期求值。

示例

(不) 著名的阶乘例子：

```
constexpr int fac(int n)
{
    constexpr int max_exp = 17;           // constexpr 使得可以在 Expects 中使用 max_exp
    Expects(0 <= n && n < max_exp);   // 防止犯糊涂和发生溢出
    int x = 1;
    for (int i = 2; i <= n; ++i) x *= i;
    return x;
}
```

这个是 C++14。

对于 C++11，请使用递归形式的 `fac()`。

注解

`constexpr` 并不会保证发生编译期求值；

它只能保证函数可以在当程序员需要或者编译器为优化而决定时，对常量表达式实参进行编译期求值。

```
constexpr int min(int x, int y) { return x < y ? x : y; }

void test(int v)
{
    int m1 = min(-1, 2);           // 可能进行编译期求值
    constexpr int m2 = min(-1, 2);  // 编译期求值
    int m3 = min(-1, v);          // 运行期求值
    constexpr int m4 = min(-1, v); // 错误：无法在编译期求值
}
```

注解

不要试图让所有函数都变成 `constexpr`。

大多数计算都最好在运行时进行。

注解

任何可能最终将依赖于高层次的运行时配置或者业务逻辑的 API，都不应当是 `constexpr` 的。这种定制化是无法由编译期来求值的，并且依赖于这种 API 的任何 `constexpr` 函数也都应当进行重构，或者抛弃掉 `constexpr`。

强制实施

不可能也不必要。

当在要求常量的地方调用了非 `constexpr` 函数时，编译器会报告错误。

F.5: 如果函数非常小，并且是时间敏感的，就将其声明为 `inline`

理由

有些优化器可以不依赖于程序员的提示就能很好地进行内联，但请不要依赖这点。

请测量！至少超过 40 年，我们一直在允诺编译器可以不依赖于人类的提示而做到比人类更好地内联。可是我们还在等。

（显式地，或于类定义体中编写成员函数而隐式地）将其指定为内联能够促进编译器工作得更好。

示例

```
inline string cat(const string& s, const string& s2) { return s + s2; }
```

例外

不要把 `inline` 函数加入需要变得稳定的接口中，除非你十分确定它不会再发生变化。

内联函数是 ABI 的一部分。

注解

`constexpr` 蕴含 `inline`。

注解

在类之中所定义的成员函数默认是 `inline` 的。

例外

函数模板（包括类模板的成员函数 `A<T>::function()` 和成员函数模板 `A::function<T>()`）一般都定义于头文件中，因此是内联的。

强制实施

对超过三条语句，并且本可以声明为非内联的 `inline` 函数（比如类成员函数）标记为 `inline`。

F.6: 如果函数必然不会抛出异常，就将其声明为 `noexcept`

理由

如果不打算抛出异常的话，程序就会认为无法处理这种错误，并且应当尽早终止。把函数声明为 `noexcept` 对优化器有好处，因为其减少了可能的执行路径的数量。它也能使发生故障之后的退出动作有所加速。

示例

给完全以 C 或者其他任何没有异常的语言编写的每个函数都标上 `noexcept`。

C++ 标准库隐含地对 C 标准库中的所有函数做了这件事。

注解

`constexpr` 函数在运行时执行时可能抛出异常，因此可能需要对其中的一些使用有条件 `noexcept`。

示例

对能够抛出异常的函数也可以使用 `noexcept`：

```
vector<string> collect(istream& is) noexcept
{
    vector<string> res;
    for (string s; is >> s;)
        res.push_back(s);
    return res;
}
```

如果 `collect()` 耗光了内存，程序就会崩溃。

除非这个程序特别精心编写成不会耗尽内存，否则这也许正是正确的方式；

`terminate()` 能够产生合适的错误日志信息（但当内存耗尽时是很难做出任何巧妙的事情的）。

注解

当你想决定是否要给函数标上 `noexcept` 时，一定要特别注意你的代码的执行环境，尤其是与抛出异常和内存分配相关的情形。打算成为完全通用的代码（比如像标准库和其他类似的工具代码），应当支持那些可以有意义地处理 `bad_alloc` 异常的执行环境。不过，大多数程序和执行环境都不能有意义地处理内存分配失败，而中止程序则是在这些情况下应对分类失败的最干净和最简单的方式。如果已知应用程序代码无法应对分配失败的话，对于即使确实会进行分配的函数，添加 `noexcept` 也是适当的。

换一种方式来说：在大多数程序中，大多数函数都会抛出异常（比如说，它们可能使用 `new`，调用会抛出异常的函数，或者使用通过抛出异常来报告失败的库函数），因此请勿随意到处散布 `noexcept` 而不考虑清楚是否有异常是可以被处理的。

`noexcept` 对于常用的，底层的函数是最有用处的（并且几乎显然是正确的）。

注解

析构函数，`swap` 函数，移动操作，以及默认构造函数不应当抛出异常。

另请参见 [C.44](#)。

强制实施

- 标记不是 `noexcept`，而又不能抛出异常的函数。
- 标记抛出异常的 `swap`，`move`，析构函数，以及默认构造函数。

F.7: 对于常规用法，应当接受 `T*` 或 `T&` 参数而不是智能指针

理由

智能指针的传递会转移或者共享所有权，因此应当仅在有意要实现所有权语义时才能使用。不操作生存期的函数应当接受原始指针或引用。

使用按智能指针传递方式把函数限制为只能服务于使用智能指针的调用方。

需要一个 `widget` 的函数应当能够接受任何 `widget` 对象，而不只是由某种特定种类的智能指针管理其生存期的那些。

智能指针的传递（比如 `std::shared_ptr`）暗含了一些运行时成本。

示例

```
// 接受任何的 int*
void f(int*);
```



```
// 只能接受你想转移所有权的 int
void g(unique_ptr<int>);
```



```
// 只能接受你想共享所有权的 int
void g(shared_ptr<int>);
```



```
// 不会改变所有权，但要求调用方对其具有特定的所有权。
void h(const unique_ptr<int>&);
```



```
// 接受任何的 int
void h(int&);
```

示例，不好

```
// 被调用方
void f(shared_ptr<widget>& w)
{
    // ...
    use(*w); // w 的唯一使用点 -- 其生存期是完全未被涉及到的
    // ...
}
```



```
// 调用方
shared_ptr<widget> my_widget = /* ... */;
f(my_widget);

widget stack_widget;
f(stack_widget); // 错误
```

示例，好

```
// 被调用方
void f(widget& w)
{
    // ...
    use(w);
    // ...
}
```

```
};

// 调用方
shared_ptr<widget> my_widget = /* ... */;
f(*my_widget);

widget stack_widget;
f(stack_widget); // ok -- 这样就有效了
```

注解

我们可以静态地找出悬挂指针的许多常见情况（参见[生存期安全性剖面配置](#)）。函数实参天然存活于函数调用的生存期，因而具有更少的生存期问题。

强制实施

- 【简单】若函数接受可复制的智能指针类型（即重载了 `operator->` 或 `operator*`），但该函数仅调用了：`operator*`、`operator->` 或 `get()`，则给出警告。
建议代之以 `T*` 或 `T&`。
- 对于智能指针类型（重载了 `operator->` 或 `operator*` 的类型）的参数，若它是可复制/可移动的，但从没有从函数体中被复制/移动出来，且从未对其进行修改，且未将其传递给会修改它其他函数，对之进行标记。这意味着并未使用其所有权语义。
建议代之以 `T*` 或 `T&`。

参见

- [当“无实参”是有效情形时，优先采用 `T*` 而不是 `T&`](#)
- [智能指针规则概述](#)

F.8: 优先采用纯函数

理由

纯函数更容易进行推导，有时候也更易于优化（甚至并行化），有时候还可以进行存储。

示例

```
template<class T>
auto square(T t) { return t * t; }
```

强制实施

不可能进行强制实施。

F.9: 未使用的形参应当没有名字

理由

可读性。

抑制未使用形参的警告消息。

示例

```
widget* find(const set<widget>& s, const widget& w, Hint); // 这里曾经使用过一个提示
```

注解

为解决这个问题，在1980年代早期就引入了允许形参无名的规则。

如果形参是根据条件不被使用的，可以用`[[maybe_unused]]`特性来声明它们。

例如：

```
template <typename value>
value* find(const set<value>& s, const value& v, [[maybe_unused]] Hint h)
{
    if constexpr (sizeof(value) > cachesize)
    {
        // 仅当 value 具有特定大小时才使用提示参数
    }
}
```

强制实施

对有名字的未使用形参进行标记。

F.10: 若操作可被重用，则应为其命名

理由

文档，可读性，重用机会。

示例

```
struct Rec {
    string name;
    string addr;
    int id;           // 唯一标识符
};

bool same(const Rec& a, const Rec& b)
{
    return a.id == b.id;
}

vector<Rec*> find_id(const string& name); // 寻找“name”的所有记录

auto x = find_if(vr.begin(), vr.end(),
    [&](Rec& r) {
        if (r.name.size() != n.size()) return false; // 要比较的名字在 n 里
        for (int i = 0; i < r.name.size(); ++i)
            if (tolower(r.name[i]) != tolower(n[i])) return false;
        return true;
});
```

这里蕴含着一个有用的函数（大小写不敏感的字符串比较），lambda 的参数变大时总会这样。

```
bool compare_insensitive(const string& a, const string& b)
{
    if (a.size() != b.size()) return false;
    for (int i = 0; i < a.size(); ++i) if (tolower(a[i]) != tolower(b[i])) return
false;
    return true;
}

auto x = find_if(vr.begin(), vr.end(),
    [&](Rec& r) { return compare_insensitive(r.name, n); }
);
```

或者可以这样（如果你倾向于避免隐含绑定到 `n` 的名字）：

```
auto cmp_to_n = [&n](const string& a) { return compare_insensitive(a, n); };

auto x = find_if(vr.begin(), vr.end(),
    [] (const Rec& r) { return cmp_to_n(r.name); }
);
```

注解

函数、lambda 或运算符均如此。

例外

- 逻辑上仅在局部使用的 lambda，比如作为 `for_each` 或类似控制流算法的实参。
- 作为初始化的 lambda。

强制实施

- 【困难】标记相似的 lambda
- ???

F.11: 当需要仅在一处使用的简单函数对象时使用无名 lambda

理由

使代码精简，提供比其他方式更好的局部性。

示例

```
auto earlyUsersEnd = std::remove_if(users.begin(), users.end(),
    [] (const User &a) { return a.id > 100; } );
```

例外

为 lambda 命名有助于明晰代码，即便它仅用一次。

强制实施

- 寻找相同或几乎相同的 lambda (以将它们替换为具名的函数或者具名的 lambda)。

F.call: 参数传递

存在各种不同的向函数传递参数和返回值的方式。

F.15: 优先采用简单的和传统的信息传递方式

理由

使用“与众不同和精巧”的技巧会带来意外，其他程序员的理解减慢，并促进 BUG 的发生。

如果你确实想要比常规技巧更好的优化，请进行测量以确保它真的有所提升，并为其写下文档/注释，因为这种提升可能无法移植。

下面的表格总结了以下 F.16-21 的各个指导方针中的建议。

一般性参数传递：

Cheap or impossible to copy (e.g., int, unique_ptr)	Cheap to move (e.g., vector<T>, string) or Moderate cost to move (e.g., array<vector>, BigPOD) or Don't know (e.g., unfamiliar type, template)	Expensive to move (e.g., BigPOD[], array<BigPOD>)
Out	X f()	
In/Out	f(X&)	
In		f(const X&)
In & retain "copy"	f(X)	

*“Cheap” ≈ a handful of hot int copies
“Moderate cost” ≈ memcpy hot/contiguous ~1KB and no allocation*

** or return unique_ptr<X>/make_shared_<X> at the cost of a dynamic allocation*

高级参数传递：

Cheap or impossible to copy (e.g., int, unique_ptr)	Cheap to move (e.g., vector<T>, string) or Moderate cost to move (e.g., array<vector>, BigPOD) or Don't know (e.g., unfamiliar type, template)	Expensive to move (e.g., BigPOD[], array<BigPOD>)
Out	X f()	
In/Out	f(X&)	
In		f(const X&)
In & retain copy	f(X)	f(const X&) + f(X&&) & move **
In & move from		f(X&&) **

** or return unique_ptr<X>/make_shared_<X> at the cost of a dynamic allocation*

*** special cases can also use perfect forwarding (e.g., multiple in+copy params, conversions)*

只有在进行论证必要之后再使用高级技巧，并将其必要性注明在代码注释中。

对于字符序列的传递，参见 [字符串](#)。

例外

使用 `shared_ptr` 等类型来表达共享所有权时，不应遵循指导方针 F.16-21，而应遵循 [R.34](#), [R.35](#), 以及 [R.36](#)。

F.16: 对于“输入 (in) ”参数，把复制操作廉价的类型按值进行传递，把其他类型按 `const` 引用进行传递

理由

既能让调用者了解函数不会修改其参数，也使得参数能够以右值初始化。

何谓“复制操作廉价”依赖于机器的架构，不过只有两三个机器字 (Word) 的类型 (double, 指针, 引用等) 一般最好按值传递。

当可以廉价复制时，没什么比得过进行复制的简单性和安全性，而且对于小型对象（最多两三个机器字）来说，也比按引用传递更快，因为它不需要在函数中进行一次额外的间接访问。

示例

```
void f1(const string& s); // OK: 按 const 引用传递；总是廉价的

void f2(string s); // bad: 可能是昂贵的

void f3(int x); // OK: 无可比拟

void f4(const int& x); // bad: f4() 中的访问带来开销
```

(仅) 对于高级的运用，如果你确实需要为“只当作输入”的参数的按右值传递进行优化的话：

- 如果函数需要无条件地从参数进行移动，那就按 `&&` 来接受参数。参见 [F.18](#)。
- 如果函数需要保留参数的一个副本，那就在按 `const&` 接受参数（对于左值）之外，添加一个按 `&&` 传递参数（对于右值）的重载，并在函数体中将之 `std::move` 到其目标之中。基本上，这个重载是“将被移动 (will-move-from)”；参见 [F.18](#)。
- 在特殊情况中，比如有多个“输入+复制”的参数时，考虑采用完美转发。参见 [F.19](#)。

示例

```
int multiply(int, int); // 仅输入了 int，按值传递

// suffix 仅作输入，但并不如 int 那样廉价，因此按 const& 传递
string& concatenate(string&, const string& suffix);

void sink(unique_ptr<widget>); // 仅作输入，但移动了这个 widget 的所有权
```

避免“为了效率”而按 `T&&` 来传递参数这类的“玄奥技巧”。

关于按 `&&` 传递带来性能好处的大多数传言都是假的或者是脆弱的（不过也请参考 [F.18](#) 和 [F.19](#)）。

注解

可以假定引用都指代了某个有效对象（语言规则）。

“空引用”（正规地说）是不存在的。

如果要表示一个非强制的值，请使用指针，`std::optional`，或者一个用以代表“没有值”的特殊值。

强制实施

- 【简单】〔基础〕当按值传递的参数的大小大于 `2 * sizeof(void*)` 时给出警告。
建议代之以 `const` 的引用。
- 【简单】〔基础〕当按 `const` 引用传递的参数的大小小于或等于 `2 * sizeof(void*)` 时给出警告。建议代之以按值传递。
- 【简单】〔基础〕当按 `const` 引用传递的参数被 `move` 时给出警告。

例外

使用 `shared_ptr` 等类型来表达共享所有权时，应遵循 [R.34](#) 或 [R.36](#)，
取决于函数是否无条件地获取实参的引用。

F.17: 对于“输入/输出 (in-out) ”参数，按非 `const` 引用进行传递

理由

让调用者明了这个对象假定将会被改动。

示例

```
void update(Record& r); // 假定 update 将会写入 r
```

注解

一些用户定义和标准程序库的类型，如 `span<T>` 或迭代器等，
是可廉价复制的，并可按值传递，
这样做时具有可改动 (in-out) 引用语义：

```
void increment_all(span<int> a)
{
    for (auto&& e : a)
        ++e;
}
```

注解

`T&` 参数既可以向函数中传递信息，也可以传递出来。
因此 `T&` 能够作为“输入/输出”参数。这点本身就可能是一种错误的来源：

```
void f(string& s)
{
    s = "New York"; // 不明显的错误
}

void g()
{
    string buffer = ".....";
    f(buffer);
    // ...
}
```

这里，`g()` 的作者提供了一个缓冲区让 `f()` 来填充，但 `f()` 仅仅替换掉了它（以多少比简单的字符复制高一些的成本）。

如果 `g()` 的作者对 `buffer` 的大小作出了错误的假设的话，就会发生糟糕的逻辑错误。

强制实施

- 【中等】【基础】对带有指向非 `const` 的引用参数但又不向其进行写入的函数给出警告。
- 【简单】【基础】当按引用传递的非 `const` 参数被进行 `move` 时给出引用。

F.18: 对于“将被移动 (will-move-from) ”参数，按 `x&&` 进行传递 并对参数 `std::move`

理由

这样做很高效，并且消除了调用点的 BUG：`x&&` 绑定到右值，而要传递左值的话则要求在调用点明确进行 `std::move`。

示例

```
void sink(vector<int>&& v) // 无论参数所拥有的是什么，sink 都获得了其所有权
{
    // 通常这里可能有对 v 的 const 访问
    store_somewhere(std::move(v));
    // 通常这里不再会使用 v 了；它已经被移走
}
```

注意，`std::move(v)` 使得 `store_somewhere()` 可以把 `v` 遗留为被移走的状态。
[这可能很危险](#)。

例外

只能移动并且移动廉价的唯一拥有者类型，比如 `unique_ptr`，也可以按值传递，这样写起来更简单而且效果相同。按值传递确实产生了一次额外的（廉价）移动操作，但我们更加优先于简单性和清晰性。

例如：

```
template<class T>
void sink(std::unique_ptr<T> p)
{
    // 使用 p ... 可能在之后的什么地方 std::move(p)
} // p 被销毁
```

例外

当“将被移动”的形参是 `shared_ptr` 时，应遵循 [R.34](#)，并按值传递 `shared_ptr`。

强制实施

- 对于所有 `x&&` 参数（其中的 `x` 不是模板类型参数的名字），如果函数体中使用它时没有用 `std::move`，就将其标明。
- 标明对已经被移动过的对象的访问。
- 不要有条件地从对象进行移动。

F.19: 对于“转发 (forward) ”参数，按 `TP&&` 进行传递并只对参数

`std::forward`

理由

如果一个对象要被传递给其他代码而并不在本函数中直接使用，我们就想让这个函数对于该参数的 `const` 性质和右值性质来说是中立的。

这种情况下，而且只有这种情况下，才应当让参数为 `TP&&`，其中 `TP` 为模板类型参数——它既忽略了也保持了 `const` 性质和右值性质。因而使用 `TP&&` 的任何代码都隐含地声称它自己并不关心变量的 `const` 性质和右值性质（因为这被忽略了），但它有意把值继续传递给其他确实关心 `const` 性质和右值性质的代码（因为这也是被保持的）。把 `TP&&` 用于参数类型上是安全的，因为从调用方传递来的任何临时对象都会在函数调用期间一直存活。基本上 `TP&&` 类型的参数应当总是在函数体中通过 `std::forward` 来继续传递。

示例

你通常在每个静态控制流路径中恰好进行一次完整的形参（或形参包组，通过 `...`）的转发：

```
template<class F, class... Args>
inline auto invoke(F f, Args&&... args)
{
    return f(forward<Args>(args)...);
}
```

示例

有时候，你可能会在每个静态控制流路径中按每个子对象一次的方式分段转发一个组合形参：

```
template<class PairLike>
inline auto test(PairLike&&... pairlike)
{
    // ...
    f1(some, args, and, forward<PairLike>(pairlike).first);           // 转发
    .first
    f2(and, forward<PairLike>(pairlike).second, in, another, call);   // 转发
    .second
}
```

强制实施

- 对于接受 `TP&&` 参数的函数（其中的 `TP` 不是模板类型参数的名字），如果函数对它做了任何别的事，而不是在每个静态路径中都正好进行一次 `std::forward`，或者在每个静态路径中对其进行多次 `std::forward` 但限定为不同的数据成员均正好进行一次，就将函数进行标明。

F.20: 对于“输出 (out) ”值，采用返回值优先于输出参数

理由

返回值是自我说明的，而 `&` 参数则既可能是输入/输出的也可能是仅输出的，并且倾向于被误用。

适用的情况也包括如标准容器这样的大型对象，它们为性能因素使用了隐式的移动操作，并且避免进行显式的内存管理。

当有多个值要返回时，[使用元组](#)或者类似的多成员类型。

示例

```
// OK: 返回指向具有 x 值的元素的指针
vector<const int*> find_all(const vector<int>&, int x);

// 不好: 把指向具有 x 值的元素的指针放入 out
void find_all(const vector<int>&, vector<const int*>& out, int x);
```

注解

含有许多（每个都廉价移动的）元素的 `struct`，聚合起来则可能是移动操作昂贵的。

例外

- 对于非具体类型，比如继承层次中的类型来说，可以用 `unique_ptr` 或 `shared_ptr` 来返回对象。
- 如果类型的移动操作昂贵（比如 `array<BigPOD>`），就考虑将其分配在自由存储中并返回一个句柄（比如 `unique_ptr`），或者传递一个指代用以填充的非 `const` 目标对象的引用（将其用作输出参数）。
- 对于内部循环中的多次函数调用之间重用自带容量的对象（比如 `std::string` 和 `std::vector`）：[将其按照输入/输出参数处理，并按引用传递。](#)

示例

假设 `Matrix` 带有移动操作（可能它将其元素都保存在一个 `std::vector` 中）：

```
Matrix operator+(const Matrix& a, const Matrix& b)
{
    Matrix res;
    // ... 用二者的和填充 res ...
    return res;
}

Matrix x = m1 + m2;    // 移动构造函数

y = m3 + m3;          // 移动赋值
```

注解

返回值优化无法处理赋值的情况，不过移动赋值却可以。

示例

```
struct Package {           // 特殊情况：移动操作昂贵的对象
    char header[16];
    char load[2024 - 16];
};

Package fill();           // 不好：大型的返回值
void fill(Package&);    // OK

int val();                // OK
void val(int&);         // 不好：val 会不会读取参数？
```

强制实施

- 对于指代非 `const` 的引用参数，如果其被写入之前未进行过读取，而且其类型能够廉价地返回，则标记它们；它们应当是“输入”的返回值。

F.21: 要返回多个“输出”值，优先返回结构体或元组 (tuple)

理由

返回值是自我说明为“仅输出”值的。

注意，C++ 是支持多返回值的，按约定使用的是 `tuple`（包括 `pair`），并可以在调用点使用 `tie` 或结构化绑定（C++17）以带来更多的便利。

优先使用具名的结构体，使其返回值具有语义。不过，没有名字的 `tuple` 在泛型代码中则很有用。

示例

```
// 不好：在代码注释作用说明仅作输出的参数
int f(const string& input, /*output only*/ string& output_data)
{
    // ...
    output_data = something();
    return status;
}

// 好：自我说明的
tuple<int, string> f(const string& input)
{
    // ...
    return {status, something()};
}
```

C++98 的标准库已经使用这种风格了，因为 `pair` 就像一种两个元素的 `tuple` 一样。

例如，给定一个 `set<string> my_set`，请考虑：

```
// C++98
result = my_set.insert("Hello");
if (result.second) do_something_with(result.first);      // 变通方案
```

在 C++11 中我们可以这样写，将结果直接放入现存的局部变量中：

```
Sometype iter;                                // 如果我们还未因为别的目的而使用
Someothertype success;                         // 这些变量，则进行默认初始化

tie(iter, success) = my_set.insert("Hello");    // 普通的返回值
if (success) do_something_with(iter);
```

而在 C++17 中，我们可以使用“结构化绑定”对多个变量进行声明和初始化：

```
if (auto [iter, success] = my_set.insert("Hello"); success)
    do_something_with(iter);
```

例外

有时候需要把对象传递给函数让其操纵它的状态。

这种情况下，按引用传递对象 `T&` 通常是恰当的技巧。

显式传递一个输入/输出参数再让其作为返回值返回出来通常是没必要的。

例如：

```
istream& operator>>(istream& in, string& s); // 与 std::operator>>() 很相似

for (string s; in >> s; ) {
    // 对文本行做些事
}
```

这里，`s` 和 `in` 都用作了输入/输出参数。

`in` 按（非 `const`）引用来传递，以便可以操作其状态。

`s` 的传递是为避免重复进行分配。

通过重用 `s`（按引用传递），我们只需要在为扩展 `s` 的容量时才会分配新的内存。

这种技巧有时候被称为“调用方分配的输出参数”模式，它特别适合于

诸如 `string` 和 `vector` 这样需要进行自由存储分配的类型。

比较一下，如果所有的值都按返回值传递出来的话，得像如下这样做：

```
pair<istream&, string> get_string(istream& in) // 不建议这样做
{
    string s;
    in >> s;
    return {in, move(s)};
}

for (auto p = get_string(cin); p.first; ) {
    // 对 p.second 做些事
}
```

我们觉得这样明显不够简洁，而且性能明显更差。

当严格理解这条规则 (F.21) 时，这些例外并不真的算是例外，因为它依赖于输入/输出参数，而不是规则中提到的单纯的输出参数。

不过我们倾向于进行明确而不是精巧的说明。

注解

许多情况下，返回某种用户定义的某个专门的类型是有好处的。

例如：

```
struct Distance {
    int value;
    int unit = 1; // 1 表示一米
};

Distance d1 = measure(obj1); // 访问 d1.value 和 d1.unit
auto d2 = measure(obj2); // 访问 d2.value 和 d2.unit
auto [value, unit] = measure(obj3); // 访问 value 和 unit;
                                    // 对于了解 measure() 的人来说有点多余
auto [x, y] = measure(obj4); // 请勿如此；这很可能造成混乱
```

只有当返回的值表现的是几个无关实体而不是某个抽象的时候，才应使用过于通用的 `pair` 和 `tuple`。

作为另一个例子，应当使用像 `variant<T, error_code>` 这样的专门的类型，而不使用通用的 `tuple`。

注解

当所要返回的元组是从复制操作昂贵的局部变量进行初始化时，可以用显式 `move` 有效避免复制：

```
pair<LargeObject, LargeObject> f(const string& input)
{
    LargeObject large1 = g(input);
    LargeObject large2 = h(input);
    // ...
    return { move(large1), move(large2) }; // 没有复制
}
```

还可以：

```
pair<LargeObject, LargeObject> f(const string& input)
{
    // ...
    return { g(input), h(input) }; // 没有复制，没有移动
}
```

请注意这与 [ES.56](#) 的 `return move(...)` 反模式是不同的。

强制实施

- 输出参数应当被替换为返回值。
输出参数时由函数写入的，调用了非 `const` 成员函数的，或者将它作为非 `const` 参数继续传递的参数。

F.60: 当“没有参数”是有效的选项时，采用 `T*` 优先于 `T&`

理由

指针 (`T*`) 可能为 `nullptr`，而引用 (`T&`) 则不能，不存在合法的“空引用”。

有时候用 `nullptr` 作为一种代表“没有对象”的方式是有用处的，但若是没有这种情况的话，使用引用的写法更简单，而且可能会产生更好的代码。

示例

```
string zstring_to_string(zstring p) // zstring 就是 char*；这是一个 C 风格的字符串
{
    if (!p) return string{};      // p 可能为 nullptr；别忘了要检查
    return string{p};
}

void print(const vector<int>& r)
{
    // r 指代一个 vector<int>；不需要检查
}
```

注解

构造出一个本质上是 `nullptr` 的引用是可能的，但不是合法的 C++ 代码（比如，`T* p = nullptr;`
`T& r = *p;）。`
这种错误非常罕见。

注解

如果你更喜欢指针写法（`->` 以及 `*` vs. `.`）的话，`not_null<T*>` 可以提供和 `T&` 相同的保证。

强制实施

- Flag ???

F.22: 用 `T*`, `owner<T*>` 或者智能指针来代表一个对象

理由

可读性：这样能够明确普通指针的含义。
带来了显著的工具支持。

注解

在传统的 C 和 C++ 代码中，普通的 `T*` 有各种互相没什么关联的用法，比如：

- 标识单个对象（本函数内不会进行 `delete`）
- 指向分配于自由存储之中的一个对象（随后将会 `delete`）
- 持有 `nullptr` 值
- 标识一个 C 风格字符串（以零结尾的字符数组）
- 标识一个数组，其长度被分开指明
- 标识数组中的一个位置

这样就难于了解代码真正做了什么和打算做什么。
它也会使检查工作和工具支持复杂化。

示例

```
void use(int* p, int n, char* s, int* q)
{
    p[n - 1] = 666; // 不好：不知道 p 是不是指向了 n 个元素；
                      // 应当假定它并非如此，否则应当使用 span<int>
    cout << s;      // 不好：不知道 s 指向的是不是以零结尾的字符数组；
                      // 应当假定它并非如此，否则应当使用 zstring
    delete q;       // 不好：不知道 *q 是不是在自由存储中分配的；
                      // 否则应当使用 owner
}
```

更好的做法

```
void use2(span<int> p, zstring s, owner<int*> q)
{
    p[p.size() - 1] = 666; // OK，会造成范围错误
    cout << s; // OK
    delete q; // OK
}
```

注解

`owner<T*>` 表示所有权, `zstring` 表示 C 风格的字符串。

两者: 应当假定从指向 `T` 的智能指针 (比如 `unique_ptr<T>`) 中获得的 `T*` 是指向单个元素的。

参见: [支持程序库](#)

参见: [请勿将数组作为单个指针来传递](#)

强制实施

- 【简单】〔边界〕对指针类型的表达式的算术操作, 若其结果为指针类型的值, 就给出警告。

F.23: 用 `not_null<T>` 来表明“空值 (null) ”不是有效的值

理由

清晰性。以 `not_null<T>` 为参数的函数很明确地说明, 应当由该函数的调用者来负责进行任何必要的 `nullptr` 检查。

相似地, 以 `not_null<T>` 为返回值的函数很明确地说明, 该函数的调用者无须检查 `nullptr`。

示例

`not_null<T*>` 让读者 (人类或机器) 明了, 在进行解引用前不需要检查 `nullptr`。

而且当进行调试时, 可以对 `owner<T*>` 和 `not_null<T>` 进行植入来进行正确性检查。

考虑:

```
int length(Record* p);
```

当调用 `length(p)` 时, 我应该先检查 `p` 是否为 `nullptr` 吗? 是不是应当由 `length()` 的实现来检查 `p` 是否为 `nullptr`?

```
// 确保 p != nullptr 是调用者的任务
int length(not_null<Record*> p);

// length() 的实现者必须假定可能出现 p == nullptr
int length(Record* p);
```

注解

假定 `not_null<T*>` 不可能是 `nullptr`; 而 `T*` 则可能为 `nullptr`; 二者都可以在内存中表示为 `T*` (因此不会带来运行时开销)。

注解

`not_null` 不仅对内建指针有效。它也能在 `unique_ptr`, `shared_ptr`, 以及其他指针式的类型上使用。

强制实施

- 【简单】当函数中的一个原始指针在未测试 `nullptr` (或等价形式) 之前就被解引用时, 就给出警告。
- 【简单】当函数中的一个原始指针有时候会在测试 `nullptr` (或等价形式) 后进行解引用, 而有时候不会时, 就报错。
- 【简单】当函数中的一个 `not_null` 指针进行了 `nullptr` 测试时, 就给出警告。

F.24: 用 `span<T>` 或者 `span_p<T>` 来代表一个半开序列

理由

非正式和不明确的范围 (range) 是一种错误来源。

示例

```
x* find(span<x> r, const x& v);      // 在 r 中寻找 v

vector<x> vec;
// ...
auto p = find({vec.begin(), vec.end()}, x{}); // 在 vec 中寻找 x{}
```

注解

范围 (Range) 在 C++ 代码中十分常见。典型情况下，它们都是隐含的，且非常难于保证它们能够被正确使用。

特别地，给定一对儿参数 `(p, n)` 来代表数组 `[p:p+n)`，
通常来说不可能确定 `*p` 后面是不是真的存在 `n` 个元素。

`span<T>` 和 `span_p<T>` 两个简单的辅助类，分别用于代表范围 `[p:q)`，以及一个以 `p` 开头并以使谓词为真的第一个元素结尾的范围。

示例

`span` 代表元素的范围，我们应当如何操作范围的各个元素呢？

```
void f(span<int> s)
{
    // 范围的遍历（保证正确进行）
    for (int x : s) cout << x << '\n';

    // C 风格的遍历（可能带有检查）
    for (gs1::index i = 0; i < s.size(); ++i) cout << s[i] << '\n';

    // 随机访问（可能带有检查）
    s[7] = 9;

    // 截取指针（可能带有检查）
    std::sort(&s[0], &s[s.size() / 2]);
}
```

注解

`span<T>` 对象并不拥有其元素，而且很小，可以按值传递。

把一个 `span` 对象作为参数传递的效率完全等同于传递一对儿指针参数或者传递一个指针和一个整数计数值。

参见: [支持程序库](#)

强制实施

【复杂】当对指针参数的访问是以其他整型类型的参数为边界限制时，就给出警告并建议改用 `span`。

F.25: 用 `zstring` 或者 `not_null<zstring>` 来代表 C 风格的字符串

理由

C 风格的字符串非常普遍。它们是按一种约定方式定义的：就是以零结尾的字符数组。

我们必须把 C 风格的字符串从指向单个字符的指针或者指向字符数组的老式的指针当中区分出来。

当不需要零结尾时，请使用 'string_view'。

示例

考虑：

```
int length(const char* p);
```

当调用 `length(p)` 时，我应该先检查 `p` 是否为 `nullptr` 吗？是不是应当由 `length()` 的实现来检查 `p` 是否为 `nullptr`？

```
// length() 的实现者必须假定可能出现 p == nullptr
int length(zstring p);

// it is the caller's job to make sure p != nullptr
int length(not_null<zstring> p);
```

注解

`zstring` 不含有所有权。

参见：[支持程序库](#)

F.26: 当需要指针时，用 `unique_ptr<T>` 来传递所有权

理由

使用 `unique_ptr` 是安全地传递指针的最廉价的方式。

参见：[C.50](#)关于何时从一个工厂中返回 `shared_ptr`。

示例

```
unique_ptr<Shape> get_shape(istream& is) // 从输入流中装配一个形状
{
    auto kind = read_header(is); // 从输入中读取头部并识别下一个形状
    switch (kind) {
        case kCircle:
            return make_unique<Circle>(is);
        case kTriangle:
            return make_unique<Triangle>(is);
        // ...
    }
}
```

注解

当要传递的对象属于某个类层次，且将要通过接口（基类）来使用它时，你需要传递一个指针而不是对象。

强制实施

【简单】当函数返回了局部分配了的原始指针时就给出警告。建议改为使用 `unique_ptr` 或 `shared_ptr`。

F.27: 用 `shared_ptr<T>` 来共享所有权

理由

使用 `std::shared_ptr` 是表示共享所有权的标准方式。其含义是，最后一个拥有者负责删除对象。

示例

```
shared_ptr<const Image> im { read_image(somewhere) };

std::thread t0 {shade, args0, top_left, im};
std::thread t1 {shade, args1, top_right, im};
std::thread t2 {shade, args2, bottom_left, im};
std::thread t3 {shade, args3, bottom_right, im};

// 脱离各线程
// 最后执行完的线程会删除这个图像
```

注解

如果同时不可能超过一个所有者的话，优先采用 `unique_ptr` 而不是 `shared_ptr`。
`shared_ptr` 的作用是共享所有权。

注意，过于普遍的使用 `shared_ptr` 是有成本的（`shared_ptr` 的引用计数上的原子性操作会产生可测量的总体花费）。

替代方案

让单个对象来拥有这个共享对象（比如一个有作用域的对象），并当其所有使用方都完成工作后（最好隐含地）销毁它。

强制实施

【无法强制实施】这种模式过于复杂，无法可靠地进行检测。

F.42: 返回 `T*` 来（仅仅）给出一个位置

理由

指针就是用来干这个的。

使用 `T*` 来传递所有权其实是一种误用。

示例

```
Node* find(Node* t, const string& s) // 在 Node 组成的二叉树中寻找 s
{
    if (!t || t->name == s) return t;
    if ((auto p = find(t->left, s))) return p;
    if ((auto p = find(t->right, s))) return p;
    return nullptr;
}
```

`find` 所返回的指针如果不是 `nullptr` 的话，就指定了一个含有 `s` 的 `Node`。
重要的是，这里面并没有暗含着把所指向的对象的所有权传递给调用者。

注解

迭代器、索引值和引用也可以用来传递位置。

[当不需要使用 `nullptr`，或者当不会改变被指代的对象时](#)，用引用通常比用指针更好。

注解

不要返回指向某个不在调用方的作用域中的东西的指针；参见 [F.43](#)。

参见：有关如何避免悬挂指针的讨论

强制实施

- 标记出施加在普通 `T*` 上的 `delete`, `std::free()` 等等。
只有所有者才能被删除。
- 标记出赋值给普通 `T*` 的 `new`, `malloc()` 等等。
只有所有者才应当负责进行删除。

F.43: 不要（直接或间接）返回指向局部对象的指针或引用

理由

避免由于使用了这种悬挂指针而造成的程序崩溃和数据损坏。

示例, 不好

从函数返回后，其中的局部对象就不再存在了：

```
int* f()
{
    int fx = 9;
    return &fx; // 不好
}

void g(int* p) // 貌似确实是无辜的
{
    int gx;
    cout << "*p == " << *p << '\n';
    *p = 999;
    cout << "gx == " << gx << '\n';
}

void h()
{
```

```
int* p = f();
int z = *p; // 从已经丢弃的栈帧中读取（不好）
g(p); // 把指向已丢弃栈帧的指针传递给函数（不好）
}
```

我在一种流行的实现上得到了以下输出：

```
*p == 999
gx == 999
```

我预期这样的结果是因为，对 `g()` 的调用重用了被 `f()` 的调用所丢弃的栈空间，因此 `*p` 所指代的空间应当会被 `gx` 所占据。

- 请想象一下当 `fx` 和 `gx` 类型不同时会发生什么。
- 请想象一下当 `fx` 或 `gx` 的类型带有不变式时会发生什么。
- 请想象一下当在更大的一组函数之间传递的不止是悬挂指针时会发生什么。
- 请想象一下一个攻击者能够利用悬挂指针干些什么。

幸运的是，大多数（全部？）的当代编译器都可以识别这种简单的情况并给出警告。

注解

这同样适用于引用：

```
int& f()
{
    int x = 7;
    // ...
    return x; // 不好：返回指代即将被销毁的对象的引用
}
```

注解

这条仅适用于非 `static` 的局部变量。

所有的 `static` 变量都是（顾名思义）静态分配的，因此指向它们的指针不可能变为悬挂的。

示例, 不好

并非所有的局部变量指针的泄漏都是那么明显的：

```
int* glob; // 全局变量的不好的方面太多了

template<class T>
void steal(T x)
{
    glob = x(); // 不好
}

void f()
{
    int i = 99;
    steal([&] { return &i; });
}

int main()
```

```
{  
    f();  
    cout << *glob << '\n';  
}
```

我这次成功从 `f` 的调用所丢弃的位置上读到了数据。

存于 `glob` 中的指针可能在很晚才被使用，并可能以无法预测的方式造成各种麻烦。

注解

局部变量的地址的“返回”或者泄漏方式，可能是返回语句，以 `T&` 输出参数，以所返回对象的成员，以所返回数组的元素，还有更多其他方式。

注解

还可以构造出相似的从内部作用域“泄漏”到外部作用域的例子；
对这样的例子应当按照与从函数中泄漏指针的相同方式来处理。

这个问题的一个略有不同的变体是，把指针放入容器使其生存期超过其所指向的对象。

参见: 另一种获得悬挂指针的方式是[指针失效](#)。

这种情况也可以用相似的技术来检测和避免。

强制实施

- 编译器通常可以发现返回局部对象引用，许多情况下也可以发现返回指向局部对象的指针。
- 静态分析可以发现许多常见的确定指针位置的使用模式（因而可以消除掉悬挂指针）

F.44: 当不想进行复制，而“没有对象被返回”不是有效的选项时，返回 `T&`

理由

语言规则保证 `T&` 会指代对象，因此不需要对其测试 `nullptr`。

参见: 所返回的引用决不能蕴含所有权的传递：

[有关如何避免悬挂指针的讨论](#)以及[有关所有权的讨论](#)。

示例

```
class Car  
{  
    array<wheel, 4> w;  
    // ...  
public:  
    wheel& get_wheel(int i) { Expects(i < w.size()); return w[i]; }  
    // ...  
};  
  
void use()  
{  
    Car c;  
    wheel& w0 = c.get_wheel(0); // w0 与 c 的生存期相同  
}
```

强制实施

对不存在可能产生 `nullptr` 的 `return` 表达式的函数进行标记。

F.45: 不要返回 `T&&`

理由

它要求返回对已销毁的临时对象的引用。

`&&` 是吸引临时对象的符号。

示例

返回的右值引用超出了返回的完整表达式结束的范围：

```
auto&& x = max(0, 1); // 到目前为止, 没问题
foo(x); // 未定义的行为
```

这种用法是频繁产生 bug 的根源，经常错误地报告为编译器错误。

函数的实现者应避免为用户设置此类陷阱。

[生存期安全性](#)完全执行时，会捕捉到这个问题。

示例

当临时的引用“向下”传递给被调用对象时，返回右值引用是正常的；

然后，临时对象保证比函数调用生命期更长（参见 [F.18](#) 和 [F.19](#)）。

但是，将这样的引用“向上”传递给更大的调用范围时，不好。

对于通过普通引用或完美传递方式传递参数，并希望返回值的通过函数，使用简单的 `auto` 而不是 `auto &&` 返回推导的类型。

假定 “F” 按值返回：

```
template<class F>
auto&& wrapper(F f)
{
    log_call(typeid(f)); // 或者别的什么测量手段
    return f(); // 不好：返回一个临时对象的引用
}
```

更好的方式：

```
template<class F>
auto wrapper(F f)
{
    log_call(typeid(f)); // 或者别的什么测量手段
    return f(); // 好
}
```

例外

`std::move` 和 `std::forward` 确实会返回 `&&`，但它们只不过是强制转换——只会按惯例在某些表达式上下文中使用，其中指代临时对象的引用只会在该临时对象被销毁之前在同一个表达式中被传递。我们不知道还存在任何别的返回 `&&` 的好例子。

强制实施

对除了 `std::move` 和 `std::forward` 之外的任何把 `&&` 作为返回类型的情况都进行标记。

F.46: `int` 是 `main()` 的返回类型

Reason

这是一条语言规则，但通常被“语言扩展”所违反，因此值得一提。

把 `main`（即程序中的那个全局的 `main`）声明为 `void` 会限制其可移植性。

示例

```
void main() { /* ... */ }; // 不好，不符合 C++  
  
int main()  
{  
    std::cout << "This is the way to do it\n";  
}
```

注解

我们提出这条规则，只是因为这种错误持续存在于大众之间。

注意，虽然其返回类型不是 `void`，但主函数并不需要明确的返回语句。

强制实施

- 编译器应当做到。
- 如果编译器做不到，就让工具把它标记出来。

F.47: 赋值运算符返回 `T&`

理由

运算符重载的惯例（尤其是对于具体类型来说），是让 `operator=(const T&)` 实施赋值之后返回（非 `const`）的 `*this`。这就确保了与标准库类型之间的一致性，并遵从了“像 `int` 一样工作”的原则。

注解

历史上层有过一些建议让赋值运算符返回 `const T&`。

这主要是为了避免 `(a = b) = c` 形式的代码——这种代码其实并不常见到足以成为违反与标准类型之间一致性的理由。

示例

```
class Foo
{
public:
    ...
    Foo& operator=(const Foo& rhs)
    {
        // 复制各个成员。
        ...
        return *this;
    }
};
```

强制实施

应当通过工具对所有赋值运算符的返回类型（和返回值）进行检查来强制实施。

F.48: 不要用 `return std::move(local)`

理由

有了确保进行的副本消除之后，现在在返回语句中明确使用 `std::move` 几乎总是不良的实践。

示例，不好

```
s f()
{
    s result;
    return std::move(result);
}
```

示例，好

```
s f()
{
    s result;
    return result;
}
```

强制实施

应当通过工具对返回语句进行检查来强制实施。

F.49: 不要返回 `const T`

理由

不建议返回 `const` 值。

这种老旧的建议已经过时了；它并不会带来什么价值，而且还会对移动语义造成影响。

示例

```
const vector<int> fct(); // 不好：这个 "const" 带来的麻烦超过其价值

void g(vector<int>& vx)
{
    // ...
    fct() = vx; // 被 "const" 所禁止
    // ...
    vx = fct(); // 昂贵的复制："const" 抑制掉了移动语义
    // ...
}
```

要求对返回值添加 `const` 的理由是可以防止（非常少见的）对临时对象的意外访问。而反对的理由则是它妨碍了（非常常见的）对移动语义的利用。

另见: [F.20, 有关“out”输出值的一般条款](#)

强制实施

- 标记 `const` 返回值。修正方法：移除 `const` 使其变为返回非 `const` 值。

F.50: 当函数不适用时（不能俘获局部变量，或者不能编写局部函数），就使用 Lambda

理由

函数不能俘获局部变量且不能在局部作用域中进行定义；当想要这些能力时，如果可能就应当使用 lambda，不行的就用手写的函数对象。另一方面，lambda 和函数对象是不能重载的；如果想要重载，就优先使用函数（让 lambda 重载的变通方案相当繁复）。如果两种方式都不行的话，就优先写一个函数；应当只使用所需的最简工具。

示例

```
// 编写只会接受 int 或 string 的函数
// -- 重载是很自然的
void f(int);
void f(const string&);

// 编写需要俘获局部状态的函数对象，可以出现于
// 语句或者表达式作用域中 -- Lambda 更自然
vector<work> v = lots_of_work();
for (int tasknum = 0; tasknum < max; ++tasknum) {
    pool.run([=, &v] {
        /*
        ...
        ... 处理 v 的 1 / max，即第 tasknum 个部分
        ...
    });
}
pool.join();
```

例外

泛型的 lambda 可以提供一种更精简的编写函数模板的方式，因此会比较有用，虽然普通的函数模板用稍多一点儿的语法可以做到同样的事情。这种优势在未来一旦所有的函数都获得了 Concept 参数的能力之后就可能会消失。

强制实施

- 有名字的非泛型 lambda (比如 `auto x = [](int i) { /*...*/; };`)，而其并未发生俘获并且出现于全局作用域，对它们给出警告。代之以编写常规的函数。

F.51: 如果需要作出选择，采用默认实参应当优先于进行重载

理由

默认实参本就是为一个单一实现提供替代的接口的。

无法保证一组重载函数全部都实现相同的语义。

使用默认实参可以避免出现代码重复。

注解

当变化来自相同类型的一组参数时，需要在默认实参和重载两种方案之间进行选择。

例如：

```
void print(const string& s, format f = {});
```

相对的则是

```
void print(const string& s); // 使用默认的 format
void print(const string& s, format f);
```

如果要为一组不同类型来实现语义上等价的操作，就不需要进行选择了。例如：

```
void print(const char&);
void print(int);
void print(zstring);
```

参见

[虚函数的默认实参](#Rf-virtual-default-arg)

强制实施

- 如果某个重载集合中的各个重载具有一个共同的形参前缀 (例如 `f(int)`, `f(int, const string&)`, `f(int, const string&, double)`)，则为其给出警告。 (注意：如果这条强制措施实践中产生太多消息，请对此进行复查。)

F.52: 对于局部使用的（也包括传递给算法的）lambda，优先采用按引用俘获

理由

为了效率和正确性，当使用局部的 lambda 时，你基本上总是需要进行按引用俘获。这也包括编写或者调用并行算法的情形，因为它们在返回前会进行联结。

讨论

效率方面的考虑是，大多数的类型都是按引用传递比按值传递更便宜。

正确性方面的考虑是，许多的调用都希望在调用点对原本的对象实施副作用（参见下面的示例）。而按值传递妨碍了这点。

注解

不幸的是，并没有一种简单的方法，能够按 `const` 引用来捕获以获得其局部调用的效率，同时又妨碍其副作用。

示例

此处，一个大型对象（网络消息）被传递给一个迭代算法，而它也许不够高效，或者能够正确复制这个消息（它可能是无法复制的）：

```
std::for_each(begin(sockets), end(sockets), [&message](auto& socket)
{
    socket.send(message);
});
```

示例

下面是一个简单的三阶段并行管线。每个 `stage` 对象封装了一个工作线程和一个队列，有一个用来把任务入队的 `process` 函数，且其析构函数会自动进行阻塞以在线程结束前等待队列变空。

```
void send_packets(buffers& bufs)
{
    stage encryptor([](buffer& b) { encrypt(b); });
    stage compressor([&](buffer& b) { compress(b); encryptor.process(b); });
    stage decorator([&](buffer& b) { decorate(b); compressor.process(b); });
    for (auto& b : bufs) { decorator.process(b); }
} // 自动阻塞以等待管线完成
```

强制实施

对于按引用捕获的 lambda，若其并非局部地用在函数作用域中，或者若其被按引用传递给某个函数，则对其进行标记。（注意：这条规则是一种近似，但确实对按指针传递进行标记，它们更可能被受调方所保存，通过某个参数来向某个堆位置进行写入，返回 lambda，等等。生存期方面的规则也会提供一般性的规则，以针对包括通过 lambda 脱离的指针和引用进行标记。）

F.53: 对于非局部使用的（包括被返回的，在堆上存储的，或者传递给别的线程的）lambda，避免采用按引用俘获

理由

指向局部对象的指针和引用不能超出它们的作用域而存活。按引用捕获的 lambda 恰是另外一种保存指向局部对象的引用的地方，因而当它们（或其副本）存活超出作用域的话，也不应该这样做。

示例，不好

```
int local = 42;

// 需要局部对象的引用。
// 注意，当程序离开作用域时，
// 局部对象不再存在，因此
// process() 的调用将带有未定义行为！
thread_pool.queue_work([&] { process(local); });
```

示例，好

```
int local = 42;
// 需要局部对象的副本。
// 由于为局部变量建立了副本，它将在
// 函数调用的全部时间内可用。
thread_pool.queue_work([=] { process(local); });
```

注解

如果必须捕获非局部指针，则应考虑使用 `unique_ptr`；它会处理生存期和同步问题。

如果必须捕获 `this` 指针，则应考虑使用 `[*this]` 捕获，它会创建整个对象的一个副本。

强制实施

- 【简单】当捕获列表中包含指代局部声明的变量的引用时给出警告。
- 【复杂】当捕获列表中包含指代局部声明的变量的引用，而 lambda 被传递给非 `const` 且非局部的上下文时，进行标记。

F.54: 当俘获了 `this` 时，显式俘获所有的变量（不使用默认俘获）

理由

这是容易混淆的。在成员函数里边写 `[=]` 貌似会按值来俘获，但其实会按引用俘获数据成员，因为它实际上按值俘获了不可见的 `this` 指针。如果你确实要这样做的话，请把 `this` 写明。

示例

```
class My_class {
    int x = 0;
    // ...

    void f()
    {
        int i = 0;
        // ...

        auto lambda = [=] { use(i, x); };    // 不好：“貌似”按复制/按值俘获
        // [&] 在当前的语言规则下的语义是一样的，也会复制 this 指针
        // [=,this] 和 [&,this] 也没好多少，并且也会导致混淆
    }
}
```

```

x = 42;
lambda(); // 调用 use(0, 42);
x = 43;
lambda(); // 调用 use(0, 43);

// ...

auto lambda2 = [i, this] { use(i, x); }; // ok, 最明确并且最不混淆

// ...
}

};


```

注解

这在标准化之中正在进行积极的讨论，而且很可能在未来版本的标准中通过增加一种新的俘获模式或者调整 [=] 的含义而得到结局。当前的话，还是应当明确为好。

强制实施

- 若指定了默认俘获（如 [=] 或 [&]）的 lambda 俘获列表并且还俘获了 this 的情况——无论是如 [&, this] 这样显式，还是通过 [=] 这样的默认俘获而又在函数体中使用了 this——对此进行标识。

F.55: 不要使用 va_arg 参数

理由

从 va_arg 中读取时需要假定确实传递了正确类型的参数。

而向变参传递时则需要假定将会读取正确的类型。

这样是很脆弱的，因为其在语言中无法一般性地强制其安全，因而需要靠程序员的纪律来保证其正确。

示例

```

int sum(...)
{
    // ...
    while /*...*/
        result += va_arg(list, int); // 不好，假定所传递的是 int
    // ...
}

sum(3, 2); // ok
sum(3.14159, 2.71828); // 不好，未定义的行为

template<class ...Args>
auto sum(Args... args) // 好，而且更灵活
{
    return (... + args); // 注意：C++17 的“折叠表达式”
}

sum(3, 2); // ok: 5
sum(3.14159, 2.71828); // ok: ~5.85987

```

替代方案

- 重载
- 变参模板
- `variant` 参数
- `initializer_list` (同质的)

注解

有时候，对于并不涉及实际的参数传递的技巧来说，声明 `...` 形参有其作用，比如当声明“接受任何东西”的函数，以在重载集合中禁止“其他所有东西”，或在模板元程序中表达一种“全覆盖（catchall）”情况时。

强制实施

- 为 `va_list`, `va_start`, 或 `va_arg` 的使用给出诊断。
- 如果 vararg 参数的函数并未提供重载以为该参数位置指定更加特定的类型，则当其传递参数时给出诊断。修正：使用别的函数，或标明 `[[suppress(types)]]`。

F.56: 避免不必要的条件嵌套

理由

浅层嵌套的条件语句使代码容易理解。也会使缩进结构清除明了。

力求将基础代码放在最外层作用域，除非这样做会搞乱缩进。

示例

使用防卫代码块来处理异常情况，并提早返回。

```
// 不好：深层嵌套
void foo() {
    ...
    if (x) {
        computeImportantThings(x);
    }
}

// 不好：还有多余的 else。
void foo() {
    ...
    if (!x) {
        return;
    }
    else {
        computeImportantThings(x);
    }
}

// 好：提早返回，无多余 else
void foo() {
    ...
    if (!x)
        return;

    computeImportantThings(x);
}
```

```
}
```

示例

```
// 不好: 不必要的条件嵌套
void foo() {
    ...
    if (x) {
        if (y) {
            computeImportantThings(x);
        }
    }
}

// 好: 合并条件 + 提早返回
void foo() {
    ...
    if (!(x && y))
        return;

    computeImportantThings(x);
}
```

强制实施

标记多余的 `else`。

对函数体仅为包含一个代码块的条件语句的函数进行标记。

C: 类和类层次

类是一种自定义类型，程序员可以定义它的表示，操作和接口。

类层次用于把相关的类组织到层次化的结构当中。

类的规则概览：

- [C.1: 把相关的数据组织到结构中 \(`struct` 或 `class`\)](#)
- [C.2: 当类具有不变式时使用 `class`；当数据成员可以独立进行变动时使用 `struct`](#)
- [C.3: 用类来表示接口和实现之间的区别](#)
- [C.4: 仅当函数直接访问类的内部表示时才让函数作为其成员](#)
- [C.5: 把辅助函数放在其所支持的类相同的命名空间之中](#)
- [C.7: 不要在同一个语句中同时定义类或枚举并声明该类型的变量](#)
- [C.8: 当有任何非公开成员时使用 `class` 而不是 `struct`](#)
- [C.9: 让成员的暴露最小化](#)

子章节：

- [C.concrete: 具体类型](#)
- [C.ctor: 构造函数, 赋值和析构函数](#)
- [C.con: 容器和其他资源包装](#)
- [C.lambdas: 函数对象和 lambda](#)
- [C.hier: 类层次 \(OOP\)](#)
- [C.over: 重载和运算符重载](#)
- [C.union: 联合体](#)

C.1: 把相关的数据组织到结构中 (`struct` 或 `class`)

理由

易理解性。

如果数据之间（以基本的原因而）相关，应当在代码中体现这点。

示例

```
void draw(int x, int y, int x2, int y2); // 不好：不必要的隐含式的关系
void draw(Point from, Point to); // 好多了
```

注解

没有虚函数的简单的类是不会带来空间或时间开销的。

注解

从语言的角度看，`class` 和 `struct` 的差别只有其成员的默认可见性不同。

强制实施

也许不可能做到。也许对总是一起使用的数据项目进行启发式查找是一种可能方式。

C.2: 当类具有不变式时使用 `class`；当数据成员可以独立进行变动时使用 `struct`

理由

可读性。

易理解性。

`class` 的使用会提醒程序员需要考虑不变式。

这是一种很有用的惯例。

注解

不变式是对象的成员之间的一种逻辑条件，必须由构造函数建立，并由公开成员函数假定成员。

不变式一旦建立（通常是由构造函数），就可以对对象的各个成员函数进行调用了。

不变式既可以非正式地说明（比如在代码注释中），也可以正式地用 `Ensures` 说明。

如果数据成员都可以互相独立地进行改变，则不可能存在不变式。

示例

```
struct Pair { // 成员可以独立地变动
    string name;
    int volume;
};
```

但是：

```
class Date {  
public:  
    // 验证 {yy, mm, dd} 是有效的日期并进行初始化  
    Date(int yy, Month mm, char dd);  
    // ...  
private:  
    int y;  
    Month m;  
    char d;      // day  
};
```

注解

如果一个类中有任何的 `private` 数据的话，其使用者就不可能不通过构造函数而完全初始化一个对象。

因此，类的定义者必然提供构造函数且必须明确其含义。

这就相当于表示该定义者需要定义一种不变式。

参见：

- [把带有私有数据的类定义为 `class`](#)
- [优先将接口部分放在类的开头](#)
- [使成员的暴露最小化](#)
- [避免 `protected` 数据](#)

强制实施

查找所有数据都私有的 `struct` 和带有公开成员的 `class`。

C.3: 用类来表示接口和实现之间的区别

理由

接口和实现之间的明确区别能够提升可读性并简化维护工作。

示例

```
class Date {  
public:  
    Date();  
    // 验证 {yy, mm, dd} 是有效的日期并进行初始化  
    Date(int yy, Month mm, char dd);  
  
    int day() const;  
    Month month() const;  
    // ...  
private:  
    // ... 一些内部表示 ...  
};
```

比如说，我们现在可以改变 `Date` 的表示而不对其使用者造成影响（虽然很可能需要重新编译）。

注解

使用这样的类来表示接口和实现之间的区别当然不是唯一可能的方式。

比如说，我们也可以使用命名空间中的一组自由函数，一个抽象基类，或者一个带有概念的函数模板来表示一个接口。

最重要的一点，在于明确地把接口和其实现“细节”区分开来。

理想地，并且典型地，接口要比其实现稳定得多。

强制实施

???

C.4: 仅当函数直接访问类的内部表示时才让函数作为其成员

理由

比成员函数更少的耦合，减少可能由于改动对象状态而造成问题的函数，减少每当改变内部表示时需要进行修改的函数数量。

示例

```
class Date {  
    // ... 相对较小的接口 ...  
};  
  
// 辅助函数：  
Date next_weekday(Date);  
bool operator==(Date, Date);
```

这些“辅助函数”并不需要直接访问 `Date` 的内部表示。

注解

当 C++ 带来统一函数调用之后，这条规则会更有效。

例外

语言规定 `virtual` 函数为成员函数，而并非所有的 `virtual` 函数都会直接访问数据。特别是，抽象类的成员很少这样做。

注意 [multi methods](#)。

例外

语言规定运算符 `=`, `()`, `[]` 和 `>` 是成员函数。

示例

一个重载集合中的一些成员可以不直接访问 `private` 数据：

```
class Foobar {
public:
    void foo(long x) { /* 操作 private 数据 */ }
    void foo(double x) { foo(std::lround(x)); }
    // ...
private:
    // ...
};
```

例外

类似地，一组函数可以被设计为进行链式调用：

```
x.scale(0.5).rotate(45).set_color(Color::red);
```

典型情况下，这些函数中的一些而并非全部会访问 `private` 数据。

强制实施

- 寻找并不直接访问数据成员的非 `virtual` 成员函数。
麻烦的是由许多并不需要直接访问数据成员的函数也会这么做。
- 忽略 `virtual` 函数。
- 忽略至少包含一个访问了 `private` 成员的函数的重载集合中的函数。
- 忽略返回 `this` 的函数。

C.5: 把辅助函数放在其所支持的类相同的命名空间之中

理由

辅助函数是（由类的作者提供的）并不需要直接访问类的内部表示的函数，它们也被当作是类的可用接口的一部分。

把它们和类放在相同的命名空间中，使它们与类的关系更明显，并允许通过基于参数的查找机制找到它们。

示例

```
namespace Chrono { // 我们在这里放置与时间有关的服务

    class Time { /* ... */ };
    class Date { /* ... */ };

    // 辅助函数:
    bool operator==(Date, Date);
    Date next_weekday(Date);
    // ...
}
```

注解

这点对于[重载运算符](#)来说尤其重要。

强制实施

- 对接受某一个命名空间中的参数类型的全局函数进行标记。

C.7: 不要在同一个语句中同时定义类或枚举并声明该类型的变量

理由

在同一个声明式中混合类型的定义和另一个实体的定义会导致混淆，而且不是必要的。

示例，不好

```
struct Data { /*...*/ } data{ /*...*/ };
```

示例，好

```
struct Data { /*...*/ };
Data data{ /*...*/ };
```

强制实施

- 如果类或者枚举的定义式的 `}` 后面没有跟着 `;` 就标记出来。它缺少了 `;`。

C.8: 当有任何非公开成员时使用 `class` 而不是 `struct`

理由

可读性。

表明有些东西被隐藏或者进行了抽象。

这是一种有用的惯例。

示例，不好

```
struct Date {
    int d, m;

    Date(int i, Month m);
    // ... 许多函数 ...
private:
    int y; // year
};
```

这段代码在 C++ 语言规则方面没有任何问题，

但从设计角度看则几乎全是错误。

私有数据和公开数据相比藏得太远了。

数据在类的声明式中被分到了不同的部分中。

不同部分的数据具有不同的访问性。

所有这些都减弱了可读性，并使维护变得更复杂。

注解

优先将接口部分放在类的开头，[参见 NL.16](#)。

强制实施

对于声明为 `struct` 的类，当其带有 `private` 或 `protected` 成员时就进行标记。

C.9: 让成员的暴露最小化

理由

封装。

信息隐藏。

使发生意外访问的机会最小化。

这会简化维护工作。

示例

```
template<typename T, typename U>
struct pair {
    T a;
    U b;
    // ...
};
```

无论我们在 `//` 部分中干什么，`pair` 的任意用户都可以任意地并且不相关地改动其 `a` 和 `b`。

在大型代码库中，我们无法轻易找出哪段代码对 `pair` 的成员都做了什么。

这也许正是我们想要的，但如果想要在成员之间强加某种关系，就需要使它们为 `private`，并通过构造函数和成员函数来强加这种关系（不变式）。

例如：

```
class Distance {
public:
    // ...
    double meters() const { return magnitude*unit; }
    void set_unit(double u)
    {
        // ... 检查 u 是 10 的倍数 ...
        // ... 适当地改变幅度 ...
        unit = u;
    }
    // ...
private:
    double magnitude;
    double unit;    // 1 为米, 1000 为千米, 0.001 为毫米, 等等
};
```

注解

如果无法轻易确定一组变量的直接用户的集合，那么这个集合的类型或用法也无法被（轻易）改变或改进。

对于 `public` 和 `protected` 数据来说这是常见的情况。

示例

一个类可以向其用户提供两个接口。

一个针对其派生类（`protected`），而另一个针对一般用户（`public`）。

例如，可能允许派生类跳过某种运行时检查，因为其已经确保了正确性：

```
class Foo {
public:
    int bar(int x) { check(x); return do_bar(x); }
    // ...
protected:
    int do_bar(int x); // 在数据上做些操作
    // ...
private:
    // ... 数据 ...
};

class Dir : public Foo {
//...
int mem(int x, int y)
{
    /* ... 做一些事 ... */
    return do_bar(x + y); // OK: 派生类可以略过检查
}
};

void user(Foo& x)
{
    int r1 = x.bar(1);      // OK, 有检查
    int r2 = x.do_bar(2);   // 错误: 可能略过检查
    // ...
}
```

注解

`protected` 数据不是好主意。

注解

优先让 `public` 成员在前，`protected` 成员其次，`private` 成员在后；参见 [NL.16](#)。

强制实施

- 标记 `protected` 数据。
- 标记混合的 `public` 和 `private` 数据。

C.concrete: 具体类型

具体类型的规则概览：

- [C.10: 优先使用具体类型而不是类继承层次](#)
- [C.11: 使具体类型正规化](#)
- [C.12: 不要令可复制或移动类型的数据成员为 `const` 或引用](#)

C.10: 优先使用具体类型而不是类继承层次

理由

具体类型在本质上就比类继承层次中的类型更简单：

它们更易于设计，更易于实现，更易于使用，更易于进行推理，更小，也更快。

使用继承层次是需要一些理由（用例）来支持的。

示例

```
class Point1 {
    int x, y;
    // ... 一些操作 ...
    // ... 没有虚函数 ...
};

class Point2 {
    int x, y;
    // ... 一些操作，其中有些是虚的 ...
    virtual ~Point2();
};

void use()
{
    Point1 p11 {1, 2};    // 在栈上创建一个对象
    Point1 p12 {p11};    // 一个副本

    auto p21 = make_unique<Point2>(1, 2);    // 在自由存储中创建一个对象
    auto p22 = p21->clone();                  // 创建一个副本
    // ...
}
```

当一个类属于某个继承层次时，我们（即使在小例子中不需要，在真实代码中也）必然要通过指针或者引用来操作它的对象。

这意味着更多的内存开销，更多的分配和回收操作，以及更多的用于实施间接操作所带来的运行时开销。

注解

具体类型可以在栈上分配，也可以成为其他类的成员。

注解

对于运行时多态接口来说，使用间接是一项基本要求。

而分配/回收操作的开销则不是（它们只是最常见的情况而已）。

我们可以使用基类来作为有作用域的派生类对象的接口。

当禁止使用动态分配时（比如硬实时）就可以这样做，为某些种类的插件提供一种稳定的接口。

强制实施

???

C.11: 使具体类型正规化

理由

正规类型比不正规的类型更易于理解和进行推导（不正规性会导致理解和使用上花费更多的精力）。

C++ 内建类型都是正规的，标准程序库的一些类型，如 `string`, `vector`, 和 `map` 也是如此。可以定义没有赋值和相等比较的具体类，但它们很罕见（理当如此）。

示例

```
struct Bundle {
    string name;
    vector<Record> vr;
};

bool operator==(const Bundle& a, const Bundle& b)
{
    return a.name == b.name && a.vr == b.vr;
}

Bundle b1 { "my bundle", {r1, r2, r3}};
Bundle b2 = b1;
if (!(b1 == b2)) error("impossible!");
b2.name = "the other bundle";
if (b1 == b2) error("No!");
```

特别是，当具体类型可以复制时，也应当为之提供相等比较运算符，并确保 `a = b` 蕴含 `a == b`。

注解

对于用来与 C 代码共用的结构体，为其定义 `operator==` 是不可行的。

注解

无法进行克隆的资源包装类（例如，包含一个 `mutex` 的 `scoped_lock`）是具体类型，不过通常都无法进行复制（但它们一般都可以被移动），因此它们不是正规类型；但它们倾向于成为“仅可移动类型”。

强制实施

???

C.12: 不要令可复制或移动类型的数据成员为 `const` 或引用

理由

`const` 和引用数据成员在可复制或移动类型中没什么用处，还会由于微妙的原因使这种类型变得至少部分地无法复制/无法移动而很难使用。

示例：不好

```
class bad {
    const int i;      // 不好
    string& s;        // 不好
    // ...
};
```

`const` 和 `&` 数据成员会导致类变成“仅可进行某些复制”——可复制构造但不可复制赋值。

注解

如果需要一个指向某物的成员，就请使用指针（原始的或智能的，而当它不能为 `null` 时使用 `gsl::not_null`）而不是引用。

强制实施

标记具有任意复制或移动操作的类型中的 `const`, `&`, 或者 `&&` 的数据成员。

C.ctor: 构造函数, 赋值, 和析构函数

这些函数控制对象的生存期：创建，复制，移动，以及销毁。

定义构造函数是为了确保以及简化类的初始化过程。

以下被称为默认操作：

- 默认构造函数: `x()`
- 复制构造函数: `x(const x&)`
- 复制赋值: `operator=(const x&)`
- 移动构造函数: `x(x&&)`
- 移动赋值: `operator=(x&&)`
- 析构函数: `~x()`

缺省情况下，编译器会为这些操作中被使用的进行定义，但这些默认定义可以被抑制掉。

默认操作是一组互相关联的操作，它们共同实现了对象的生存期语义。

缺省情况下，C++ 按照值类型的方式来对待各个类，但并非所有的类型都与值类型相符。

默认操作的规则集合：

- [C.20: 只要可能，请避免定义任何的默认操作](#)
- [C.21: 如果定义或者 `=delete` 了任何复制、移动或析构函数，请定义或者 `=delete` 它们全部](#)
- [C.22: 使默认操作之间保持一致](#)

析构函数的规则：

- [C.30: 如果一个类需要在对象销毁时执行明确的操作，请为其定义析构函数](#)
- [C.31: 类所获取的所有资源，必须都在类的析构函数中进行释放](#)
- [C.32: 如果类中带有原始指针 \(`T*`\) 或者引用 \(`T&`\)，请考虑它是否是所有者](#)
- [C.33: 如果类中带有所有权的指针成员，请定义析构函数](#)
- [C.35: 基类的析构函数应当要么是 `public` 和 `virtual`，要么是 `protected` 且非 `virtual`](#)
- [C.36: 析构函数不能失败](#)
- [C.37: 使析构函数 `noexcept`](#)

构造函数的规则：

- [C.40: 如果类具有不变式，请为其定义构造函数](#)

- [C.41: 构造函数应当创建经过完整初始化的对象](#)
- [C.42: 当构造函数无法构造有效对象时，应当抛出异常](#)
- [C.43: 保证可复制类带有默认构造函数](#)
- [C.44: 尽量让默认构造函数简单且不抛出异常](#)
- [C.45: 不要定义仅对数据成员进行初始化的默认构造函数；应当使用成员初始化式](#)
- [C.46: 默认情况下，把单参数的构造函数声明为 `explicit`](#)
- [C.47: 按成员声明的顺序对成员变量进行定义和初始化](#)
- [C.48: 对于常量初始化式来说，优先采用类中的初始化式而不是构造函数中的成员初始化式](#)
- [C.49: 优先进行初始化而不是在构造函数中赋值](#)
- [C.50: 当初始化过程中需要体现“虚函数行为”时，请使用工厂函数](#)
- [C.51: 用委派构造函数来表示类中所有构造函数的共同行为](#)
- [C.52: 使用继承构造函数来把构造函数引入到无须进行其他的明确初始化操作的派生类之中](#)

复制和移动的规则：

- [C.60: 使复制赋值非 `virtual`，接受 `const&` 的参数，并返回非 `const` 的引用](#)
- [C.61: 复制操作应当进行复制](#)
- [C.62: 使复制赋值可以安全进行自赋值](#)
- [C.63: 使移动赋值非 `virtual`，接受 `&&` 的参数，并返回非 `const&`](#)
- [C.64: 移动操作应当进行移动，并使原对象处于有效状态](#)
- [C.65: 使移动赋值可以安全进行自赋值](#)
- [C.66: 使移动操作 `noexcept`](#)
- [C.67: 多态类应当抑制公开的移动/复制操作](#)

其他的默认操作规则：

- [C.80: 当需要明确使用缺省语义时，使用 `=default`](#)
- [C.81: 当需要关闭缺省行为（且不需要替代的行为）时，使用 `=delete`](#)
- [C.82: 不要在构造函数和析构函数中调用虚函数](#)
- [C.83: 考虑为值类型提供 `noexcept` 的 `swap` 函数](#)
- [C.84: `swap` 不能失败](#)
- [C.85: 使 `swap` 函数 `noexcept`](#)
- [C.86: 使 `==` 对操作数的类型对称，并使之 `noexcept`](#)
- [C.87: 请当心基类的 `==`](#)
- [C.89: 使 `hash` 函数 `noexcept`](#)
- [C.90: 依靠构造函数和赋值运算符，不要依靠 `memset` 和 `memcpy`](#)

C.defop: 默认操作

缺省情况下，语言会提供具有预置语义的默认操作。

不过，程序员可以关闭或者替换掉这些缺省实现。

C.20: 只要可能，请避免定义任何的默认操作

理由

这样最简单，而且能提供最清晰的语义。

示例

```
struct Named_map {
public:
    // ... 并未声明任何默认操作 ...
private:
    string name;
    map<int, int> rep;
};

Named_map nm;           // 默认构造
Named_map nm2 {nm};    // 复制构造
```

由于 `std::map` 和 `string` 都带有全部的特殊函数，这里并不需要做别的事情。

注解

这被称为“零之准则（The rule of zero）”。

强制实施

【无法强制实施】虽然无法强制实施，但一个优秀的静态分析器可以检查出一些模式，指出可使之符合本条规则的改进可能性。

例如，一个带有（指针,大小）成员对，同时在析构函数中 `delete` 这个指针的类也许可以被转换为使用一个 `vector`。

C.21: 如果定义或者 `=delete` 了任何复制、移动或析构函数，请定义或者 `=delete` 它们全部

理由

复制、移动和析构的语义互相之间是紧密相关的，一旦需要声明其中一个，麻烦的是其他的也需要予以考虑。

只要声明了复制、移动或析构函数，

即便是声明为 `=default` 或 `=delete`，也将会抑制掉移动构造函数和移动赋值运算符的隐式声明。

而声明移动构造函数或移动赋值运算符，

即便是声明为 `=default` 或 `=delete`，也将会导致隐式生成的复制构造函数或隐式生成的复制赋值运算符被定义为弃置的。

因此，只要声明了它们中的任何一个，就应当将

其他全部都予以声明，以避免出现预期外的效果，比如将所有潜在的移动都变成了更昂贵的复制操作，或者使类变为只能移动的。

示例，不好

```
struct M2 {    // 不好：不完整的复制/移动/析构操作集合
public:
    // ...
    // ... 没有复制和移动操作 ...
    ~M2() { delete[] rep; }

private:
    pair<int, int>* rep; // pair 的以零终止的集合
};
```

```
void use()
{
    M2 x;
    M2 y;
    // ...
    x = y;    // 缺省的赋值
    // ...
}
```

既然对于析构函数需要“特殊关照”（这里是要进行回收操作），隐式定义的复制和移动赋值运算符仍保持正确性的可能是很低的（此处会导致双重删除问题）。

注解

这被称为“五之准则（The rule of five）”。

注解

如果想（于定义了别的函数时）保持缺省实现，请写下 `=default` 以表明对这个函数是特意这样做的。如果不想要一个生成的缺省函数，可以用 `=delete` 来抑制它。

示例，好

如果要声明析构函数仅是为了使其为 `virtual` 的话，可将其定义为预置的。

```
class AbstractBase {
public:
    virtual ~AbstractBase() = default;
    // ...
};
```

为避免发生如 [C.67](#) 所说的切片，使复制和移动操作为受保护的或 `=delete`，并添加 `clone`：

```
class CloneableBase {
public:
    virtual unique_ptr<CloneableBase> clone() const;
    virtual ~CloneableBase() = default;
    CloneableBase() = default;
    CloneableBase(const CloneableBase&) = delete;
    CloneableBase& operator=(const CloneableBase&) = delete;
    CloneableBase& operator=(CloneableBase&&) = delete;
    CloneableBase& operator=(CloneableBase&&) = delete;
    // ... 其他构造函数和函数 ...
};
```

这里仅定义移动操作或者进定义复制操作也可以具有相同效果，但明确说明每个特殊成员的意图，可使其对读者更加易于理解。

注解

编译器会很大程度上强制实施这条规则，并在理想情况下会对任何违反都给出警告。

注解

在带有析构函数的类中，依靠隐式生成的复制操作的做法已经被摒弃。

注解

编写这些函数很容易出错。

注意它们的参数类型：

```
class X {
public:
    // ...
    virtual ~X() = default;           // 析构函数（如果 X 是基类，用 virtual）
    X(const X&) = default;          // 复制构造函数
    X& operator=(const X&) = default; // 复制赋值
    X(X&&) = default;              // 移动构造函数
    X& operator=(X&&) = default; // 移动赋值
};
```

一个小错误（例如拼写错误，遗漏 `const`，使用 `&` 而不是 `&&`，或遗漏一个特殊功能）可能导致错误或警告。

为避免单调乏味和出错的可能性，请尝试遵循[零规则](#)。

强制实施

【简单】 类中应当要么为每个复制/移动/析构函数都提供一个声明（即便是 `=delete`），要么都不这样做。

C.22: 使默认操作之间保持一致

理由

默认操作是一个概念上相配合的集合。它们的语义是相互关联的。

如果复制/移动构造和复制/移动赋值所做的是逻辑上不同的事情的话，这会让使用者感觉诡异。如果构造函数和析构函数并不提供一种对资源管理的统一视角的话，也会让使用者感觉诡异。如果复制和移动操作并不体现出构造函数和析构函数的工作方式的话，同样会让使用者感觉诡异。

示例，不好

```
class Silly { // 不好：复制操作不一致
    class Impl {
        // ...
    };
    shared_ptr<Impl> p;
public:
    Silly(const Silly& a) : p{make_shared<Impl>()} { *p = *a.p; } // 深复制
    Silly& operator=(const Silly& a) { p = a.p; } // 浅复制
    // ...
};
```

这些操作在复制语义上并不统一。这将会导致混乱和出现 BUG。

强制实施

- 【复杂】复制/移动构造函数和对应的复制/移动赋值运算符，应当在相同的解引用层次上向相同的成员变量进行写入。
- 【复杂】在复制/移动构造函数中被写入的任何成员变量，在其他构造函数中也都应当进行初始化。
- 【复杂】如果复制/移动构造函数对某个成员变量进行了深复制，就应当在析构函数中对这个成员变量进行修改。
- 【复杂】如果析构函数修改了某个成员变量，在任何复制/移动构造函数或赋值运算符中就都应当对该成员变量进行写入。

C.dtor: 析构函数

“这个类需要析构函数吗？”是一个出人意料有洞察力的设计问题。

对于大多数类来说，答案是“不需要”，要么是因为类中并没有保持任何资源，要么是因为销毁过程已经被零之准则处理掉了；

就是说，它的成员在销毁之中可以自己照顾自己。

当答案为“需要”时，类的大部分设计应当遵循下列规则（参见五之准则）。

C.30: 如果一个类需要在对象销毁时执行明确的操作，请为其定义析构函数

理由

析构函数是在对象的生存期结束时被隐式执行的。

如果预置的析构函数足堪使用的话，就应当用它。

只有当类需要执行的代码不在其成员的析构函数中时，才需要定义非预置的析构函数。

示例

```
template<typename A>
struct final_action { // 略有简化
    A act;
    final_action(A a) : act{a} {}
    ~final_action() { act(); }
};

template<typename A>
final_action<A> finally(A act) // 推断出动作的类型
{
    return final_action<A>{act};
}

void test()
{
    auto act = finally([] { cout << "Exit test\n"; }); // 设置退出动作
    // ...
    if (something) return; // 动作在这里得到执行
    // ...
} // 动作在这里得到执行
```

`final_action` 的全部目的就是为了在其销毁时执行一段代码（通常是一个 lambda）。

注解

需要自定义析构函数的类大致上有两种：

- 类中具有某个资源，而它并未表示成一个具有析构函数的类，比如 `vector` 或事物类。
- 类的目的主要用于在销毁时执行某个动作，比如一个追踪器，或者 `final_action`。

示例，不好

```
class Foo {    // 不好；使用预置的析构函数
public:
    // ...
    ~Foo() { s = ""; i = 0; vi.clear(); } // 清理
private:
    string s;
    int i;
    vector<int> vi;
};
```

预置的析构函数会做得更好，更高效，而且不会出错。

注解

当需要预置的析构函数，但其生成被抑制（比如由于定义了移动构造函数）时，可以使用 `=default`。

强制实施

查找疑似“隐式的资源”，比如指针和引用等。查找带有析构函数的类，即便其所有数据成员都带有自己的析构函数。

C.31: 类所获取的所有资源，必须都在类的析构函数中进行释放

理由

避免资源泄漏，尤其是错误情形中。

注解

对于以具有完整的默认操作集合的类来表示的资源来说，这些都是会自动发生的。

示例

```
class X {
    ifstream f; // 可能会拥有某个文件
    // ... 没有任何定义或者声明为 =deleted 的默认操作 ...
};
```

`X` 的 `ifstream` 会在其所在 `X` 的销毁时，隐含地关闭任何可能已经被它所打开的文件。

示例，不好

```
class X2 {    // 不好
    FILE* f; // 可能会拥有某个文件
    // ... 没有任何定义或者声明为 =deleted 的默认操作 ...
};
```

`X2` 可能会泄漏文件的句柄。

注解

不过关不掉的 socket 怎么办呢？析构函数、close 以及清理操作[不应当失败](#)。

如果它确实这样的话，就出现了一个不存在真正的好解决方案的问题。

对于新手来说，作为析构函数的编写者，无法了解析构函数是因为什么被调用的，而且不能通过抛出异常来“拒绝执行”。

参见[相关讨论](#)。

让这个问题更加糟糕的，还包括许多的 close/release 操作都是无法重试的。

许多人都曾试图解决这个问题，但仍不存在已知的一般性解决方案。

如果可能的话，可以考虑吧 close/cleanup 的失败看成是基本的设计错误，然后终止程序 (terminate)。

注解

类之中也可以持有指向它并不拥有的对象的指针和引用。

显然这样的对象是不应当在类的析构函数中被 `delete` 的。

例如：

```
Preprocessor pp { /* ... */ };
Parser p { pp, /* ... */ };
Type_checker tc { p, /* ... */ };
```

这里的 `p` 指向 `pp` 但并不拥有它。

强制实施

- 【简单】当类中所有的指针或引用成员变量是所有者（比如通过使用 `gs1::owner` 所断定）时，它们就应当在析构函数中有所引用。
- 【困难】当在所有权上没有明确的说法时，为指针或引用成员变量确定其是否是所有者（比如，检查构造函数的代码）。

C.32: 如果类中带有原始指针 (`T*`) 或者引用 (`T&`)，请考虑它是否是所有者

理由

大量的代码都是和所有权无关的。

示例

```
class legacy_class
{
    foo* m_owning; // 不好：改为 unique_ptr<T> 或 owner<T*>
    bar* m_observer; // OK：不用改
}
```

唯一确定所有权的方式可能就是深入代码中去寻找内存分配了。

注解

所有权在新代码（以及重构的遗留代码）中应当是清晰的：[R.20](#) 对于有所有权指针，[R.3](#) 对于无所有权指针。引用从来不能有所有权，[R.4](#)。

强制实施

查看原始指针成员和引用成员的初始化，看看是否进行了分配操作。

C.33: 如果类中带有所有权的指针成员，请定义析构函数

理由

被拥有的对象，必须在拥有它的对象销毁时进行 `delete`。

示例

指针成员可以表示某种资源。

[不应该这样使用 `T*`](#)，但老代码中这是很常见的。

请把 `T*` 当作一种可能的所有者的嫌疑。

```
template<typename T>
class Smart_ptr {
    T* p; // 不好： *p 的所有权含糊不清
    // ...
public:
    // ... 没有自定义的默认操作 ...
};

void use(Smart_ptr<int> p1)
{
    // 错误： p2.p 泄漏了（当其不为 nullptr 且未被其他代码所拥有时）
    auto p2 = p1;
}
```

注意，当你定义析构函数时，你必须定义或者弃置（`delete`）[所有的默认操作](#)：

```
template<typename T>
class Smart_ptr2 {
    T* p; // 不好： *p 的所有权含糊不清
    // ...
public:
    // ... 没有自定义的复制操作 ...
    ~Smart_ptr2() { delete p; } // p 是所有者！
};

void use(Smart_ptr2<int> p1)
{
    auto p2 = p1; // 错误： 双重删除
}
```

预置的复制操作仅仅把 `p1.p` 复制给了 `p2.p`，这导致对 `p1.p` 进行双重销毁。请明确所有权的处理：

```
template<typename T>
class Smart_ptr3 {
    owner<T*> p; // OK： 明确了 *p 的所有权
    // ...
public:
    // ...
    // ... 复制和移动操作 ...
}
```

```
~Smart_ptr3() { delete p; }

void use(Smart_ptr3<int> p1)
{
    auto p2 = p1; // OK: 未发生双重删除
}
```

注解

通常最简单的处理析构函数的方式，就是把指针换成一个智能指针（比如 `std::unique_ptr`），并让编译器来安排进行恰当的隐式销毁过程。

注解

为什么不直接要求全部带有所有权的指针都是“智能指针”呢？

这样做有时候需要进行不平凡的代码改动，并且可能会对 ABI 造成影响。

强制实施

- 怀疑带有指针数据成员的类。
- 带有 `owner<T>` 的类应当定义其默认操作。

C.35: 基类的析构函数应当要么是 public 和 virtual，要么是 protected 且非 virtual

理由

以防止未定义行为。

若析构函数是 `public`，调用方代码就可以尝试通过基类指针来销毁一个派生类的对象，而如果基类的析构函数是非 `virtual`，则其结果是未定义的。

若析构函数是 `protected`，调用方代码就无法通过基类指针进行销毁，而且这个析构函数不需要是 `virtual`；它应当是 `protected` 而不是 `private`，以便它能够在派生类析构函数中执行。

总之，基类的编写者并不知道什么是当进行销毁时要做的适当操作。

探讨

请参见[这条规则](#)中的探讨段落。

示例，不好

```
struct Base { // 不好：隐含带有 public 的非 virtual 析构函数
    virtual void f();
};

struct D : Base {
    string s {"a resource needing cleanup"};
    ~D() { /* ... do some cleanup ... */ }
    // ...
};

void use()
{
    unique_ptr<Base> p = make_unique<D>();
    // ...
} // p 的销毁调用了 ~Base() 而不是 ~D()，这导致 D::s 的泄漏，也许不止
```

注解

虚函数针对派生类定义了一个接口，使用它并不需要对派生类有所了解。
如果这个接口允许进行销毁，那么它应当安全地做到这点。

注解

析构函数必须是非私有的，否则它会妨碍使用这个类型：

```
class X {
    ~X(); // 私有析构函数
    // ...
};

void use()
{
    X a; // 错误：无法销毁
    auto p = make_unique<X>(); // 错误：无法销毁
}
```

例外

可以构想出一种可能需要受保护虚析构函数的情形：派生类型（且仅限于这种类型）的对象允许通过基类指针来销毁另一个对象（而不是其自身）。不过我们在实际中从未见到过这种情况。

强制实施

- 带有任何虚函数的类的析构函数，应当要么是 `public virtual`，要么是 `protected` 且非 `virtual`。
- 如果某个类公开继承于某个基类，则该基类应当具有要么是 `public virtual`，要么是 `protected` 且非 `virtual` 的析构函数。

C.36: 析构函数不能失败

理由

一般来说当析构函数可能失败时我们不知道怎样写出没有错误的代码。

标准库要求它所处理的所有的类所带有的析构函数都应当不会因抛出异常而退出。

示例

```
class X {
public:
    ~X() noexcept;
    // ...
};

X::~X() noexcept
{
    // ...
    if (cannot_release_a_resource) terminate();
    // ...
}
```

注解

许多人都曾试图针对析构函数中的故障处理设计一种傻瓜式的方案。

但没有人得到过任何一种通用方案。

这确实是真正的实际问题：比如说，怎么处理无法关闭的 socket？

析构函数的编写者无法了解析构函数为什么会被调用，并且不能通过抛出异常来“拒绝执行”。

参见[探讨](#)段落。

让问题更麻烦的是，许多的“关闭/释放”操作还都是不能重试的。

如果确实可行的话，请把“关闭/清理”的失败作为一项基本设计错误并终止（terminate）程序。

注解

把析构函数声明为 `noexcept`。这将确保它要么正常完成执行，要么就终止程序。

注解

如果一个资源无法释放而程序不能失败，请尝试把这个故障用某种方式通知给系统中的其他部分

（也许甚或修改某个全局状态，并希望有人能注意到它并有能力处理这个问题）。

请充分警惕，这种技巧是有专门用途的，并且容易出错。

请考虑“连接关闭不了”的那个例子。

这也许是因为连接的另一端出现了问题，但只有对连接的两端同时负责的代码才能恰当地处理这个问题。

析构函数可以向系统中负责管控的部分发送一个消息（或别的什么），然后认为已经关闭了连接并正常返回。

注解

如果析构函数所用的操作可能会失败的话，它可以捕获这些异常，某些时候仍然可以成功完成执行（例如，换用与抛出异常的清理机制不同的另一种机制）。

强制实施

【简单】如果析构函数可能抛出异常，就应当将其声明为 `noexcept`。

C.37: 使析构函数 `noexcept`

理由

[析构函数不能失败](#)。如果析构函数试图抛出异常来退出，这就是一种设计错误，程序最好终止执行。

注解

当类中的所有成员都带有 `noexcept` 析构函数时，析构函数（无论是自定义的还是编译器生成的）将被隐含地声明为 `noexcept`（这与其函数体中的代码无关）。通过将析构函数明确标记为 `noexcept`，程序员可以防止由于添加或修改类的成员而导致析构函数变为隐含的 `noexcept(false)`。

示例

并非所有析构函数都默认为 `noexcept`；一个抛出异常的成员会影响整个类的层级：

```
struct X {  
    Details x; // 碰巧有一个抛出析构函数  
    // ...  
    ~X() {} // 隐含地 noexcept(false)；也可以抛出异常  
};
```

所以，不确定的话，声明一个析构函数 noexcept.

注解

为什么对所有析构函数声明 noexcept?

因为在许多情况下，特别是简单的情况，会分散混乱。

强制实施

【简单】如果析构函数可能抛出异常，就应当将其声明为 noexcept。

C.ctor: 构造函数

构造函数定义对象如何进行初始化（构造）。

C.40: 如果类具有不变式，请为其定义构造函数

理由

这正是构造函数的用途。

示例

```
class Date { // Date 表示从 1900/1/1 到 2100/12/31 范围中
    // 的一个有效日期
    Date(int dd, int mm, int yy)
        :d{dd}, m{mm}, y{yy}
    {
        if (!is_valid(d, m, y)) throw Bad_date{}; // 不变式的实施
    }
    // ...
private:
    int d, m, y;
};
```

把不变式表达为构造函数上的一个 Ensures 通常是一种好做法。

注解

即便类并没有不变式，也可以用构造函数来简化代码。例如：

```
struct Rec {
    string s;
    int i {0};
    Rec(const string& ss) : s{ss} {}
    Rec(int ii) :i{ii} {}
};

Rec r1 {7};
Rec r2 {"Foo bar"};
```

注解

C++11 的初始化式列表规则免除了对许多构造函数的需求。例如：

```
struct Rec2{
    string s;
    int i;
    Rec2(const string& ss, int ii = 0) :s(ss), i(ii) {} // 多余的
};

Rec2 r1 {"Foo", 7};
Rec2 r2 {"Bar"};
```

`Rec2` 的构造函数是多余的。

同样的，`int` 的默认值最好用[成员初始化式](#)来给出。

参见：[构造有效对象](#)和[构造函数抛出异常](#)。

强制实施

- 对带有自定义的复制操作但没有构造函数的类进行标记（自定义的复制操作是类是否带有不变式的良好指示器）

C.41: 构造函数应当创建经过完整初始化的对象

理由

构造函数为类设立不变式。类的使用者应当能够假定构造完成的对象是可以使用的。

示例，不好

```
class X1 {
    FILE* f; // 在任何其他函数之前应当调用 init()
    // ...
public:
    X1() {}
    void init(); // 初始化 f
    void read(); // 从 f 中读取数据
    // ...
};

void f()
{
    X1 file;
    file.read(); // 程序崩溃或者错误的数据读取!
    // ...
    file.init(); // 太晚了
    // ...
}
```

编译器读不懂代码注释。

例外

如果无法方便地通过构造函数来构造有效的对象的话，请[使用工厂函数](#)。

强制实施

- 【简单】每个构造函数都应当对每个成员变量进行初始化（明确地，通过委派构造函数调用，或者通过默认构造）。
- 【未知】如果构造函数带有 `Ensures` 契约的话，尝试确定它给出的是否是一项后条件。

注解

如果构造函数（为创建有效的对象）获取了某个资源，则这个资源应当由析构函数释放。

这种以构造函数获取资源并以析构函数来释放的惯用法被称为 [RAII](#)（“资源获取即初始化/Resource Acquisition Is Initialization”）。

C.42: 当构造函数无法构造有效对象时，应当抛出异常

理由

留下无效对象不管就是会造成麻烦的做法。

示例

```
class X2 {
    FILE* f;
    // ...
public:
    X2(const string& name)
        :f{fopen(name.c_str(), "r")}{}
    {
        if (!f) throw runtime_error{"could not open" + name};
        // ...
    }

    void read();      // 从 f 中读取数据
    // ...
};

void f()
{
    X2 file {"zeno"}; // 当文件打不开时会抛出异常
    file.read();      // 好的
    // ...
}
```

示例，不好

```
class X3 {      // 不好：构造函数留下了无效的对象
    FILE* f;    // 在任何其他函数之前应当调用 is_valid()
    bool valid;
    // ...
public:
    X3(const string& name)
        :f{fopen(name.c_str(), "r")}, valid{false}{}
```

```

    if (f) valid = true;
    // ...
}

bool is_valid() { return valid; }
void read(); // 从 f 中读取数据
// ...
};

void f()
{
    x3 file {"Heraclides"};
    file.read(); // 程序崩溃或错误的数据读取!
    // ...
    if (file.is_valid()) {
        file.read();
        // ...
    }
    else {
        // ... 处理错误 ...
    }
    // ...
}

```

注解

对于变量的定义式（比如在栈上，或者作为其他对象的成员），不存在可以返回错误代码的明确函数调用。

留下无效的对象并依赖使用者能够一贯地在使用之前检查 `is_valid()` 函数是啰嗦的，易错的，并且是低效的做法。

例外

有些领域，比如像飞行器控制这样的硬实时系统中，（在没有其他工具支持下）异常处理在计时方面不具有足够的可预测性。

这样的话就必须使用 `is_valid()` 技巧。这种情况下，可以一贯并即刻地检查 `is_valid()` 来模拟 [RAII](#)。

替代方案

如果你觉得想要使用某种“构造函数之后初始化”或者“两阶段初始化”手法，请试着避免这样做。
如果你确实要如此的话，请参考[工厂函数](#)。

注解

人们使用 `init()` 函数而不是在构造函数中进行初始化的一种原因是为了避免代码重复。

[委派构造函数和默认成员初始化式](#)可以更好地做到这点。

另一种原因是为了把初始化推迟到需要对象的位置；它的解决方法通常为“[直到变量可以正确进行初始化的位置再声明变量](#)”。

强制实施

???

C.43: 保证可复制类带有默认构造函数

理由

就是说，确保当具体类可复制时它也满足“半正规”类型的其他规定。

许多的语言和程序库设施都依赖于默认构造函数来初始化其各个元素，比如 `T a[10]` 和 `std::vector<T> v(10)`。

对于同时是可复制的类型来说，默认构造函数通常会简化定义一个适当的[移动遗留状态](#)的任务。

示例

```
class Date { // 不好：缺少默认构造函数
public:
    Date(int dd, int mm, int yyyy);
    // ...
};

vector<Date> vd1(1000);    // 需要默认的 Date
vector<Date> vd2(1000, Date{7, Month::October, 1885});    // 替代方式
```

仅当没有用户声明的构造函数时，默认构造函数才会自动生成，因此上面的例子中的 `vector vd1` 是无法进行初始化的。

缺乏默认值会导致用户感觉奇怪，并且使其使用变复杂，因此如果可以合理定义的话就应当定义默认值。

`Date` 可以推动我们考虑：

“天然的”默认日期是不存在的（大爆炸对大多数人来说在时间上太过久远了），因此这并非是毫无意义的例子。

`{0, 0, 0}` 在大多数历法系统中都不是有效的日期，因此选用它可能会引入某种如同浮点的 `NaN` 这样的东西。

不过，大多数现实的 `Date` 类都有某个“首日”（比如很常见的 `1970/1/1`），因此以它为默认日期通常很容易做到。

```
class Date {
public:
    Date(int dd, int mm, int yyyy);
    Date() = default; // [参见](#Rc-default)
    // ...
private:
    int dd {1};
    int mm {1};
    int yyyy {1970};
    // ...
};

vector<Date> vd1(1000);
```

注解

所有成员都带有默认构造函数的类，隐含得到一个默认构造函数：

```
struct X {  
    string s;  
    vector<int> v;  
};  
  
X x; // 意为 x{{}, {}}; 即空字符串和空 vector
```

需要注意的是，内建类型并不会进行正确的默认构造：

```
struct X {  
    string s;  
    int i;  
};  
  
void f()  
{  
    X x; // x.s 被初始化为空字符串；x.i 未初始化  
  
    cout << x.s << ' ' << x.i << '\n';  
    ++x.i;  
}
```

静态分配的内建类型对象被默认初始化为 0，但局部的内建变量并非如此。

请注意你的编译期也许默认初始化了局部内建变量，而它在优化构建中并不会这样做。

因此，上例这样的代码也许恰好可以工作，但这其实依赖于未定义的行为。

假定你确实需要初始化的话，可以使用明确的默认初始化：

```
struct X {  
    string s;  
    int i {}; // 默认初始化（为 0）  
};
```

注解

缺乏合理的默认构造的类，通常也都不是可以复制的，因此它们并不受本条指导方针所限。

例如，基类不能进行复制，且因而并不需要一个默认构造函数：

```
// Shape 是个抽象基类，而不是可复制类型  
// 它可以有也可以没有默认构造函数。  
struct Shape {  
    virtual void draw() = 0;  
    virtual void rotate(int) = 0;  
    // =delete 复制/移动函数  
    // ...  
};
```

必须在构造过程中获取由调用方提供的资源的类，通常无法提供默认构造函数，但它们并不受本条指导方针所限，因为这样的类通常也不是可复制的：

```
// std::lock_guard 不是可复制类型。  
// 它没有默认构造函数。  
lock_guard g {mx}; // 护卫 mutex mx  
lock_guard g2; // 错误：不护卫任何东西
```

带有必须由其成员函数或者其用户进行特殊处理的“特殊状态”的类，会带来额外的工作量，（而且很可能有更多的错误）。这样的类型不管其是否可以复制，都可以以这个特殊状态作为其默认构造的值：

```
// std::ofstream 不是可复制类型。  
// 它刚好有一个默认构造函数，  
// 并带来一种特殊的“未打开”状态。  
ofstream out {"Foobar"};  
// ...  
out << log(time, transaction);
```

一些类似的可复制的具有特殊状态的类型，比如具有特殊状态“==nullptr”的可复制的智能指针，也应该以该特殊状态作为其默认构造的值。

不过，为有意义的状态提供默认构造函数（比如 `std::string` 的 `""` 和 `std::vector` 的 `{}`），也是推荐的做法。

强制实施

- 对于可用 `=` 进行复制的类，若没有默认构造函数则对其进行标记。
- 对于可用 `==` 进行比较但不可复制的类进行标记。

C.44: 尽量让默认构造函数简单且不抛出异常

理由

如果可以设置一个“默认”值同时又不会涉及可能失败的操作的话，就可以简化错误处理以及对移动操作的推理。

示例，有问题的

```
template<typename T>  
// elem 指向以 new 分配的 space-elem 个元素  
class Vector0 {  
public:  
    Vector0() :Vector0{0} {}  
    Vector0(int n) :elem{new T[n]}, space{elem + n}, last{elem} {}  
    // ...  
private:  
    own<T*> elem;  
    T* space;  
    T* last;  
};
```

这段代码很不错而且通用，不过在发生错误之后把一个 `Vector0` 进行置空会涉及一次分配，而它是可能失败的。

而且把默认的 `Vector` 表示为 `{new T[0], 0, 0}` 也比较浪费。

比如说，`Vector0<int> v[100]` 会耗费 100 次分配操作。

示例

```
template<typename T>
// elem 为 nullptr, 否则 elem 指向以 new 分配的 space-elem 个元素
class Vector1 {
public:
    // 设置表示为 {nullptr, nullptr, nullptr}; 不会抛出异常
    Vector1() noexcept {}
    Vector1(int n) : elem{new T[n]}, space{elem + n}, last{elem} {}
    // ...
private:
    own<T*> elem {};
    T* space {};
    T* last {};
};
```

表示为 `{nullptr, nullptr, nullptr}` 的 `Vector1{}` 很廉价，但这是一种特殊情况并且隐含了运行时检查。

在检测到错误后可以很容易地把 `Vector1` 置空。

强制实施

- 标记会抛出的默认构造函数

C.45: 不要定义仅对数据成员进行初始化的默认构造函数；应当使用成员初始化式

理由

使用类内部的成员初始化式，编译器可以据此生成函数。由编译器生成的函数可能更高效。

示例，不好

```
class X1 { // 不好：未使用成员初始化式
    string s;
    int i;
public:
    X1() : s{"default"}, i{1} {}
    // ...
};
```

示例

```
class X2 {
    string s {"default"};
    int i {1};
public:
    // 使用编译期生成的默认构造函数
    // ...
};
```

强制实施

【简单】默认构造函数应当不只是用常量初始化成员变量。

C.46: 默认情况下，把单参数的构造函数声明为 `explicit`

理由

用以避免意外的类型转换。

示例，不好

```
class String {  
public:  
    String(int); // 不好  
    // ...  
};  
  
String s = 10; // 意外：大小为 10 的字符串
```

例外

如果确实想要从构造函数参数类型隐式转换为类类型的话，就不使用 `explicit`：

```
class Complex {  
public:  
    Complex(double d); // OK：希望进行从 d 向 {d, 0} 的转换  
    // ...  
};  
  
Complex z = 10.7; // 无意外的转换
```

参见：[有关隐式转换的讨论](#)

注解

不应当将复制和移动构造函数作为 `explicit` 的，因为它们并不进行转换。显式的复制/移动构造函数会把按值传递和返回变麻烦。

强制实施

【简单】单参数的构造函数应当声明为 `explicit`。有益的单参数非 `explicit` 构造函数在大多数代码库中都是很少见的。对没在“已确认列表”中列出的每个违规都要给出警告。

C.47: 按成员声明的顺序对成员变量进行定义和初始化

理由

以尽量避免混淆和错误。该顺序正是初始化的发生顺序（而这与成员初始化式的顺序无关）。

示例，不好

```

class Foo {
    int m1;
    int m2;
public:
    Foo(int x) :m2{x}, m1{++x} { } // 不好：有误导性的初始化式顺序
    // ...
};

Foo x(1); // 意外：x.m1 == x.m2 == 2

```

强制实施

【简单】成员初始化式的列表中应当以成员声明的相同顺序列出各个成员。

参见: [讨论](#)

C.48: 对于常量初始化式来说，优先采用类中的初始化式而不是构造函数中的成员初始化式

理由

明确所有构造函数都将使用相同的值。避免重复。避免可维护性问题。这样做会产生最简短最高效的代码。

示例，不好

```

class X { // 不好
    int i;
    string s;
    int j;
public:
    X() :i{666}, s{"qqq"} { } // j 未初始化
    X(int ii) :i{ii} {} // s 为 "" 而 j 未初始化
    // ...
};

```

维护者如何能看出 `j` 是否是故意未初始化的（尽管这可能是个糟糕的想法），而且是不是故意要使 `s` 的默认值在一种情况下为 `""` 而另一种情况下为 `qqq` 呢（几乎可以肯定是个 Bug）？这里 `j` 的问题（忘记对成员初始化）通常会出现在向现存类中添加新成员的时候。

示例

```

class X2 {
    int i {666};
    string s {"qqq"};
    int j {0};
public:
    X2() = default; // 所有成员都初始化为默认值
    X2(int ii) :i{ii} {} // s 和 j 被初始化为默认值
    // ...
};

```

替代方案: 也可以用构造函数的默认实参来获得一部分的好处，而且这在比较老的代码中也并不少见。不过这种方式不够直白，会导致需要传递较多的参数，并且当有多个构造函数时也会造成重复：

```

class X3 { // 不好：不明确，参数传递开销
    int i;
    string s;
    int j;
public:
    X3(int ii = 666, const string& ss = "qqq", int jj = 0)
        :i{ii}, s{ss}, j{jj} {} // 所有成员都初始化为默认值
    // ...
};

```

强制实施

- 【简单】每个构造函数都应该对所有成员变量进行初始化（明确进行，通过委派构造函数调用，或者通过默认构造）。
- 【简单】构造函数的默认实参的出现表明类内部的初始化式可能更合适。

C.49: 优先进行初始化而不是在构造函数中赋值

理由

初始化语法明确指出所进行的是初始化而不是赋值，它更加精炼和高效。这样也避免了“未设值前就使用”的错误。

示例，好

```

class A { // 好
    string s1;
public:
    A(czstring p) : s1{p} {} // 好：直接构造（这里明确指名了 C 风格字符串）
    // ...
};

```

示例，不好

```

class B { // 不好
    string s1;
public:
    B(const char* p) { s1 = p; } // 不好：执行默认构造函数之后进行赋值
    // ...
};

class C { // 恶劣，非常不好
    int* p;
public:
    C() { cout << *p; p = new int{10}; } // 意外，初始化前就被使用了
    // ...
};

```

示例，更好的做法

可以使用 C++17 的 `std::string_view` 或 `gs1::span<char>` 代替这些 `const char*` 来作为[一种表示函数实参的更通用的方式](#)：

```

class D { // 好
    string s1;
public:
    D(string_view v) : s1{v} {} // 好：直接构造
    // ...
};

```

C.50: 当初始化过程中需要体现“虚函数行为”时，请使用工厂函数

理由

当基类对象的状态必须依赖于对象的派生部分的状态时，需要使用虚函数（或等价手段），并最小化造成误用和不完全构造的对象的机会窗口。

注解

工厂的返回类型默认情况下通常应当为 `unique_ptr`；如果某些用法需要共享，则调用方可以将这个 `unique_ptr move` 到一个 `shared_ptr` 中。但是，如果工厂的作者已知其所返回的对象的所有用法都是共享使用的话，就可返回 `shared_ptr`，并在函数体中使用 `make_shared` 以节省一次分配。

示例，不好

```

class B {
public:
    B()
    {
        /* ... */
        f(); // 不好：C.82：不要在构造函数和析构函数中调用虚函数
        /* ... */
    }

    virtual void f() = 0;
};

```

示例

```

class B {
protected:
    class Token {};

public:
    explicit B(Token) { /* ... */ } // 创建不完全初始化的对象
    virtual void f() = 0;

    template<class T>
    static shared_ptr<T> create() // 创建共享对象的接口
    {
        auto p = make_shared<T>(typename T::Token{});
        p->post_initialize();
        return p;
    }

protected:
    virtual void post_initialize() // 构造之后立即调用
};

```

```

{ /* ... */ f(); /* ... */ } // 好：虚函数分派是安全的
};

class D : public B { // 某个派生类
protected:
    class Token {};

public:
    explicit D(Token) : B( B::Token{} ) {}
    void f() override { /* ... */ };

protected:
    template<class T>
    friend shared_ptr<T> B::create();
};

shared_ptr<D> p = D::create<D>(); // 创建一个 D 的对象

```

`make_shared` 要求公开的构造函数。构造函数通过要求一个受保护的 `Token` 而无法再被公开调用，从而避免不完全构造的对象泄漏出去。

通过提供工厂函数 `create()`，（在自由存储上）构造对象变得简便。

注解

根据惯例，工厂方法在自由存储上进行分配，而不是在运行栈或者某个外围对象之内进行。

参见: [讨论](#)

C.51: 用委派构造函数来表示类中所有构造函数的共同行为

理由

以避免代码重复和意外出现的差异。

示例，不好

```

class Date { // 不好：有重复
    int d;
    Month m;
    int y;
public:
    Date(int dd, Month mm, year yy)
        :d{dd}, m{mm}, y{yy}
        { if (!valid(d, m, y)) throw Bad_date{}; }

    Date(int dd, Month mm)
        :d{dd}, m{mm} y{current_year()}
        { if (!valid(d, m, y)) throw Bad_date{}; }
    // ...
};

```

写这些共同行为很啰嗦，而且可能意外出现不一致。

示例

```
class Date2 {
    int d;
    Month m;
    int y;
public:
    Date2(int dd, Month mm, year yy)
        :d{dd}, m{mm} y{yy}
    { if (!valid(d, m, y)) throw Bad_date{}; }

    Date2(int dd, Month mm)
        :Date2{dd, mm, current_year()} {}
    // ...
};
```

参见: 当“重复行为”是简单的初始化时, 考虑使用[类内部的成员初始化式](#)。

强制实施

【中等】查找相似的构造函数体。

C.52: 使用继承构造函数来把构造函数引入到无须进行其他的明确初始化操作的派生类之中

理由

当派生类需要这些构造函数时, 重新实现它们既啰嗦又容易出错。

示例

`std::vector` 有许多棘手的构造函数, 如果我想要创建自己的 `vector` 的话, 我并不想重新实现它们:

```
class Rec {
    // ... 数据, 以及许多不错的构造函数 ...
};

class Oper : public Rec {
    using Rec::Rec;
    // ... 没有数据成员 ...
    // ... 许多不错的工具函数 ...
};
```

示例, 不好

```
struct Rec2 : public Rec {
    int x;
    using Rec::Rec;
};

Rec2 r {"foo", 7};
int val = r.x;    // 未初始化
```

强制实施

确保派生类的每个成员都被初始化。

C.copy: 复制和移动

具体类型一般都应当是可以复制的，而类层次中的接口则不应如此。

资源包装可以复制也可以不能复制。

我们可以基于逻辑因素，也可以为性能原因而将类型定义为可移动的。

C.60: 使复制赋值非 `virtual`，接受 `const&` 的参数，并返回非 `const` 的引用

理由

这样做简单且高效。如果想对右值进行优化，则可以提供一个接受 `&&` 的重载（参见 F.18）。

示例

```
class Foo {
public:
    Foo& operator=(const Foo& x)
    {
        // 好：不需要检查自赋值的情况（除非为性能考虑）
        auto tmp = x;
        swap(tmp); // 参见 C.83
        return *this;
    }
    // ...
};

Foo a;
Foo b;
Foo f();

a = b;      // 用左值赋值：复制
a = f();    // 用右值赋值：可能进行移动
```

注解

`swap` 实现技巧可以提供[强保证](#)。

示例

如果不产生临时副本能够得到明显好得多的性能的话应当怎么办呢？考虑一个简单的 `Vector` 类，其所使用的领域中常常要对大型的、大小相同的 `vector` 进行赋值。这种情况下，`swap` 实现技巧中所蕴含的元素复制操作将导致运行成本按数量级增长。

```
template<typename T>
class vector {
public:
    Vector& operator=(const Vector&);
    // ...
private:
    T* elem;
```

```

    int sz;
};

vector& Vector::operator=(const Vector& a)
{
    if (a.sz > sz) {
        // ... 使用 swap 技巧, 没有更好的方式了 ...
        return *this;
    }
    // ... 从 *a.elem 复制 sz 个元素给 elem ...
    if (a.sz < sz) {
        // ... 销毁 *this 中过剩的元素并调整大小 ...
    }
    return *this;
}

```

直接向目标元素中进行写入的话, 我们得到的是基本保证而不是 `swap` 技巧所提供的强保证。还要当心 自赋值。

替代方案: 如果你想要 `virtual` 的赋值运算符, 并了解为何这样做很有问题的话, 请不要使其为 `operator=`。请使用一个命名函数, 如 `virtual void assign(const Foo&)`。

参见[复制构造函数 vs. `clone\(\)`](#)。

强制实施

- 【简单】赋值运算符不能为 `virtual`。有怪兽出没!
- 【简单】赋值运算符应当返回 `T&` 以支持调用链, 不要改为如 `const T&` 等类型, 这样会影响可组合性以及把对象放入容器的能力。
- 【中等】赋值运算符应当 (隐式或者显式) 调用所有的基类和成员的赋值运算符。
检查析构函数以分辨类型具有指针语义还是值语义。

C.61: 复制操作应当进行复制

理由

这正是一般假定所具有的语义。执行 `x = y` 之后, 应当有 `x == y`。

进行复制之后, `x` 和 `y` 可以是各自独立的对象 (值语义, 非指针的内建类型和标准库类型的工作方式), 也可以代表某个共享的对象 (指针语义, 就是指针的工作方式)。

示例

```

class X {    // OK: 值语义
public:
    X();
    X(const X&);      // 复制 X
    void modify();     // 改变 X 的值
    // ...
    ~X() { delete[] p; }

private:
    T* p;
    int sz;
};

bool operator==(const X& a, const X& b)
{

```

```

        return a.sz == b.sz && equal(a.p, a.p + a.sz, b.p, b.p + b.sz);
    }

X::X(const X& a)
    :p{new T[a.sz]}, sz{a.sz}
{
    copy(a.p, a.p + sz, p);
}

X x;
X y = x;
if (x != y) throw Bad{};
x.modify();
if (x == y) throw Bad{}; // 假定具有值语义

```

示例

```

class X2 { // OK: 指针语义
public:
    X2();
    X2(const X2&) = default; // 浅拷贝
    ~X2() = default;
    void modify();           // 改变所指向的值
    // ...
private:
    T* p;
    int sz;
};

bool operator==(const X2& a, const X2& b)
{
    return a.sz == b.sz && a.p == b.p;
}

X2 x;
X2 y = x;
if (x != y) throw Bad{};
x.modify();
if (x != y) throw Bad{}; // 假定具有指针语义

```

注解

应当优先采用值语义，除非你要构建某种“智能指针”。值语义是最容易进行推理的，而且也是被标准库设施所期望的。

强制实施

【无法强制实施】

C.62: 使复制赋值可以安全进行自赋值

理由

如果 `x=x` 会改变 `x` 的值的话，会让人惊异，并导致发生严重的错误（通常会含有资源泄漏）。

示例

标准库的容器类都能优雅且高效地处理自赋值：

```
std::vector<int> v = {3, 1, 4, 1, 5, 9};  
v = v;  
// v 的值仍然是 {3, 1, 4, 1, 5, 9}
```

注解

从可以处理自赋值的成员所生成的默认复制操作是能够正确处理自赋值的。

```
struct Bar {  
    vector<pair<int, int>> v;  
    map<string, int> m;  
    string s;  
};  
  
Bar b;  
// ...  
b = b; // 正确而且高效
```

注解

可以通过明确检测自赋值来处理自赋值的情况，不过通常不进行这种检测会变得更快并且更优雅（比如说，利用 [swap](#)）。

```
class Foo {  
    string s;  
    int i;  
public:  
    Foo& operator=(const Foo& a);  
    // ...  
};  
  
Foo& Foo::operator=(const Foo& a) // OK, 但增加了成本  
{  
    if (this == &a) return *this;  
    s = a.s;  
    i = a.i;  
    return *this;  
}
```

这显然是安全的，也貌似高效。

不过，如果一百万次赋值才会做一次自赋值会怎么样呢？

这样就有大约一百万次多余的测试（不过由于基本上每次的答案都相同，计算机的分支预测电路也基本上每次都会猜对）。

考虑：

```
Foo& Foo::operator=(const Foo& a)    // 更简单，而且可能也更好
{
    s = a.s;
    i = a.i;
    return *this;
}
```

`std::string` 的自赋值是安全的，`int` 也是如此。所有的成本都将花在（罕见的）自赋值情况中。

强制实施

【简单】赋值运算符不应当包含 `if (this == &a) return *this;` 这样的代码模式 ???

C.63: 使移动赋值非 `virtual`，接受 `&&` 的参数，并返回非 `const&`

理由

这样简单而且高效。

参见: [针对复制赋值的规则](#)。

强制实施

和针对[复制赋值](#)所做的相同。

- 【简单】赋值运算符不能为 `virtual`。有怪兽出没！
- 【简单】赋值运算符应当返回 `T&` 以支持调用链，不要改为如 `const T&` 等类型，这样会影响可组合性以及把对象放入容器的能力。
- 【中等】移动赋值运算符应当（隐式或者显式）调用所有的基类和成员的移动赋值运算符。

C.64: 移动操作应当进行移动，并使原对象处于有效状态

理由

这正是一般假定所具有的语义。

执行 `y=std::move(x)` 之后，`y` 的值应当为 `x` 曾经的值，而 `x` 应当处于有效状态。

示例

```
class X {    // OK: 值语义
public:
    X();
    X(X&& a) noexcept;    // 移动 X
    X& operator=(X&& a) noexcept; // 移动赋值 X
    void modify();        // 改变 X 的值
    // ...
    ~X() { delete[] p; }
private:
    T* p;
    int sz;
};

X::X(X&& a) noexcept
    : p{a.p}, sz{a.sz} // 窃取其表示
{
```

```

    a.p = nullptr;      // 设其为“空”
    a.sz = 0;
}

void use()
{
    X x{};
    // ...
    X y = std::move(x);
    x = X{};    // OK
} // OK: x 可以销毁

```

注解

理想情况下，被移走的对象应当为类型的默认值。

请确保体现这点，除非有非常好的理由不这样做。

然而，并非所有类型都有默认值，而有些类型建立默认值则是昂贵操作。

标准所要求的仅仅是被移走的对象应当可以被销毁。

我们通常也可以轻易且廉价地做得更好一些：标准库假定它可以向被移走的对象进行赋值。

请保证总是让被移走的对象处于某种（需要明确的）有效状态。

注解

请让 `x = std::move(y); y = z;` 按照惯例约定的语义工作，除非有某个十分强大的理由不这样做。

强制实施

【无法强制实施】 检查移动操作中对成员的赋值。如果有默认构造函数的话，则把这些赋值和默认构造函数中的初始化之间进行比较。

C.65: 使移动赋值可以安全进行自赋值

理由

如果 `x = x` 会改变 `x` 的值的话，会让人惊异，并导致发生严重的错误（通常会含有资源泄漏）。不过，通常不会有人写出能够变成移动操作的自赋值代码，但它确实是会发生的。不管怎样，`std::swap` 就是利用移动操作来实现的，因此如果你不小心写了 `swap(a, b)` 而 `a` 和 `b` 指代相同的对象的话，未能处理自移动情况将是一种严重而且微妙的错误。

示例

```

class Foo {
    string s;
    int i;
public:
    Foo& operator=(Foo&& a);
    // ...
};

Foo& Foo::operator=(Foo&& a) noexcept // OK, 但增加了成本
{
    if (this == &a) return *this; // 这行是多余的
    s = std::move(a.s);
    i = a.i;
    return *this;
}

```

[自赋值](#)中反对 `if (this == &a) return *this;` 测试的“每一百万次有一次”的论点，在自移动的情况下更加适当。

注解

并不存在已知的通用方法，以在移动赋值中避免进行 `if (this == &a) return *this;` 测试，又能使其得到正确的结果（亦即，执行 `x = x` 之后不改变 `x` 的值）。

注解

ISO 标准中对标准库容器类仅仅保证了“有效但未指明”的状态。貌似这样做在差不多十年的实验性和产品级代码的使用中并未造成什么问题。如果你找到了反例的话，请联系各位编辑。本条规则更多的是提醒小心并强调完全的安全性。

示例

下面是一种不进行测试而移动一个指针的方法（请想象这段代码来自某个移动赋值的实现）：

```
// 从 other.ptr 移动到 this->ptr
T* temp = other.ptr;
other.ptr = nullptr;
delete ptr; // 在自移动情况中, this->ptr 也为 null; delete 是空操作
ptr = temp; // 在自移动情况中, 恢复了原 ptr
```

强制实施

- 【中级】在自赋值的情况下，移动赋值运算符不应当使持有已经被 `delete` 或设为 `nullptr` 的指针成员。
- 【无法强制实施】查看标准库容器类型（包括 `string`）的使用方式，在普通（非性命攸关）使用中将它们当作是安全的。

C.66: 使移动操作 noexcept

理由

能够抛出异常的移动操作将违反大多数人的合理假设。

不会抛出异常的移动操作可以更高效地被标准库和语言设施所利用。

示例

```
template<typename T>
class Vector {
public:
    Vector(Vector&& a) noexcept : elem{a.elem}, sz{a.sz} { a.sz = 0; a.elem =
        nullptr; }
    Vector& operator=(Vector&& a) noexcept { elem = a.elem; sz = a.sz; a.sz = 0;
        a.elem = nullptr; }
    // ...
private:
    T* elem;
    int sz;
};
```

这些操作不会抛出异常。

示例，不好

```
template<typename T>
class Vector2 {
public:
    Vector2(Vector2&& a) { *this = a; } // 直接利用复制操作
    Vector2& operator=(Vector2&& a) { *this = a; } // 直接利用复制操作
    // ...
private:
    T* elem;
    int sz;
};
```

`Vector2` 不仅低效，而且由于向量的复制需要分配内存而使其可能抛出异常。

强制实施

【简单】移动操作应当被标为 `noexcept`。

C.67: 多态类应当抑制公开的移动/复制操作

理由

多态类是定义或继承了至少一个虚函数的类。它很可能要被用作其他具有多态行为的派生类的基类。如果不小心将其按值传递了，如果它带有隐式生成的复制构造函数和赋值的话，它就面临发生切片的风险：只会复制派生类对象的基类部分，但将损坏其多态行为。

如果类中没有数据，则使其复制/移动函数 `=delete`。否则，使它们为受保护的。

示例，不好

```
class B { // 不好：多态基类并未抑制复制操作
public:
    virtual char m() { return 'B'; }
    // ... 没有提供复制操作，使用预置实现 ...
};

class D : public B {
public:
    char m() override { return 'D'; }
    // ...
};

void f(B& b)
{
    auto b2 = b; // 啊呀，对象切片了；b2.m() 将返回 'B'
}

D d;
f(d);
```

示例

```
class B { // 好: 多态类抑制了复制操作
public:
    B() = default;
    B(const B&) = delete;
    B& operator=(const B&) = delete;
    virtual char m() { return 'B'; }
    // ...
};

class D : public B {
public:
    char m() override { return 'D'; }
    // ...
};

void f(B& b)
{
    auto b2 = b; // ok, 编译器能够检测到不恰当的复制并给出警告
}

D d;
f(d);
```

注解

当需要创建多态对象的深拷贝副本时，应当使用 `clone()` 函数：参见 [C.130](#)。

例外

表示异常对象的类应当既是多态的，也可以进行复制构造。

强制实施

- 对带有公开的复制操作的多态类进行标记。
- 对多态类对象的赋值操作进行标记。

C.other: 默认操作的其他规则

除了语言为之提供默认实现的操作之外，还有一些操作也是非常基础的，需要对它们的定义给出专门的规则：
比较，`swap`，以及`hash`。

C.80: 当需要明确使用缺省语义时，使用 `=default`

理由

编译器能更正确地实现缺省语义，你所实现的这些函数也不会比编译器更好。

示例

```

class Tracer {
    string message;
public:
    Tracer(const string& m) : message{m} { cerr << "entering " << message << '\n'; }
    ~Tracer() { cerr << "exiting " << message << '\n'; }

    Tracer(const Tracer&) = default;
    Tracer& operator=(const Tracer&) = default;
    Tracer(Tracer&&) = default;
    Tracer& operator=(Tracer&&) = default;
};

```

由于定义了析构函数，所以也得定义它的复制和移动操作。最佳且最简单的做法就是 `= default`。

示例，不好

```

class Tracer2 {
    string message;
public:
    Tracer2(const string& m) : message{m} { cerr << "entering " << message << '\n'; }
    ~Tracer2() { cerr << "exiting " << message << '\n'; }

    Tracer2(const Tracer2& a) : message{a.message} {}
    Tracer2& operator=(const Tracer2& a) { message = a.message; return *this; }
    Tracer2(Tracer2&& a) : message{a.message} {}
    Tracer2& operator=(Tracer2&& a) { message = a.message; return *this; }
};

```

把复制和移动操作的函数体写明的做法，既啰嗦又乏味，而且易于出错。编译器则能干得更好。

强制实施

【中级】 特殊操作的函数体不应当和编译器生成的版本具有同样的访问性和语义，因为这样做是多余的。

C.81: 当需要关闭缺省行为（且不需要替代的行为）时，使用 `=delete`

理由

少数情况下是不需要提供默认操作的。

示例

```

class Immortal {
public:
    ~Immortal() = delete; // 不允许进行销毁
    // ...
};

void use()
{
    Immortal ugh; // 错误: ugh 无法销毁
    Immortal* p = new Immortal{};
    delete p; // 错误: 无法销毁 *p
}

```

示例

`unique_ptr` 可以移动但不能复制。为达成这点，其复制操作是被弃置的。为了避免发生复制，需要将其从左值进行复制的操作定义为 `=delete`：

```

template<class T, class D = default_delete<T>> class unique_ptr {
public:
    // ...
    constexpr unique_ptr() noexcept;
    explicit unique_ptr(pointer p) noexcept;
    // ...
    unique_ptr(unique_ptr&& u) noexcept; // 移动构造函数
    // ...
    unique_ptr(const unique_ptr&) = delete; // 关闭从左值进行的复制
    // ...
};

unique_ptr<int> make(); // 创建“某个对象”并以移动方式返回

void f()
{
    unique_ptr<int> pi {};
    auto pi2 {pi}; // 错误: 不存在从左值进行的移动构造函数
    auto pi3 {make()}; // OK, 进行移动: make() 的结果为右值
}

```

注意，弃置的函数应当是公开的。

强制实施

消除一个默认操作，是（应当是）基于类所要达成的语义考虑的。应当对这样的类保持怀疑，但可以维护一个“确认列表”，由人工断言其语义是正确的。

C.82: 不要在构造函数和析构函数中调用虚函数

理由

其中所调用的函数其实是目前所构造的对象的函数，而不是可能在派生类中覆盖它的函数。

这可能是最容易混淆的。

更糟的是，从构造函数或析构函数中直接或间接调用未被实现的纯虚函数的话，还会导致未定义的行为。

示例，不好

```
class Base {
public:
    virtual void f() = 0;      // 未实现
    virtual void g();         // 有 Base 版本的实现
    virtual void h();         // 有 Base 版本的实现
    virtual ~Base();          // 有 Base 版本的实现
};

class Derived : public Base {
public:
    void g() override;      // 提供 Derived 版本的实现
    void h() final;          // 提供 Derived 版本的实现

    Derived()
    {
        // 不好：试图调用未经事先的虚函数
        f();

        // 不好：想要调用 derived::g，但并未发生虚函数分派
        g();

        // 好：明确说明想要调用的就是写明的版本
        Derived::g();
    }

    // ok，不需要进行限定，h 为 final
    h();
}
};
```

注意，调用一个明确限定的函数时，即便函数是 `virtual` 的，也不会发生虚函数调用。

参见 [工厂函数](#)，以了解如何获得调用派生类函数的效果又不会引发未定义行为。

注解

其实在构造函数和析构函数中调用虚函数并不存在固有的错误。

这种调用的语义是类型安全的。

然而，经验表明这种调用很少真正需要，易于让维护者混淆，而且当被新手使用之后还会成为一种错误来源。

强制实施

- 标记构造函数和析构函数中对虚函数的调用。

C.83: 考虑为值类型提供 `noexcept` 的 `swap` 函数

理由

`swap` 对于实现许多惯用法都很有用，其范围包括从平滑地进行对象移动，到轻易实现提供了受保证的提交功能的赋值操作以允许编写具有强异常安全性的调用代码。考虑利用 `swap` 来基于复制构造实现复制赋值操作。另见 [析构函数，回收，以及 swap 不允许失败](#)。

示例，好

```
class Foo {
public:
    void swap(Foo& rhs) noexcept
    {
        m1.swap(rhs.m1);
        std::swap(m2, rhs.m2);
    }
private:
    Bar m1;
    int m2;
};
```

为调用者方便起见，可以在类型所在的相同命名空间中提供一个非成员的 `swap` 函数。

```
void swap(Foo& a, Foo& b)
{
    a.swap(b);
}
```

强制实施

- 可非平凡复制的类型应当提供成员 `swap` 或自由 `swap` 函数的重载。
- 【简单】当类带有 `swap` 成员函数时，它应当被声明为 `noexcept`。

C.84: `swap` 函数不能失败

理由

`swap` 广泛地以假定永不失败的方式被使用，而且如果存在可能失败的 `swap` 函数的话，程序也很难编写为可以正确工作。如果元素类型的 `swap` 会失败的话，标准库的容器和算法也无法正确工作。

示例，不好

```
void swap(My_vector& x, My_vector& y)
{
    auto tmp = x;    // 复制各元素
    x = y;
    y = tmp;
}
```

这样做不仅很慢，而且如果为 `tmp` 中的元素进行了内存分配的话，这个 `swap` 也可能抛出异常，并导致使用它的 STL 算法的失败。

强制实施

- 【简单】当类带有 `swap` 成员函数时，它应当被声明为 `noexcept`。

C.85: 使 `swap` 函数 `noexcept`

理由

`swap` 不能失败。

如果 `swap` 试图用异常来退出的话，这就是严重的设计错误，程序最好理解终止 `terminate`。

强制实施

【简单】当类带有 `swap` 成员函数时，它应当被声明为 `noexcept`。

C.86: 使 `==` 对操作数的类型对称，并使之 `noexcept`

理由

不对称的操作数是出人意料的，而且当可能发生类型转换时也是一种错误来源。

`==` 是一项基础操作，程序员应当能够随意使用而不担心失败。

示例

```
struct X {
    string name;
    int number;
};

bool operator==(const X& a, const X& b) noexcept {
    return a.name == b.name && a.number == b.number;
}
```

示例，不好

```
class B {
    string name;
    int number;
    bool operator==(const B& a) const {
        return name == a.name && number == a.number;
    }
    // ...
};
```

`B` 的比较函数接受其第二个操作数上的类型转换，但第一个操作数不可以。

注解

如果类带有比如 `double` 的 `NaN` 这样的故障状态的话，就诱惑人们让与故障状态之间的比较抛出异常。

其替代方案是让两个故障状态的比较相等，而任何有效状态和故障状态的比较都不相等。

注解

本条规则适用于所有的常规比较运算符：`!=`, `<`, `<=`, `>`, 以及 `>=`。

强制实施

- 对两个参数类型不同的 `operator==()` 进行标记；其他比较运算符也是如此：`!=`, `<`, `<=`, `>`, 和 `>=`。
- 对成员 `operator==()` 进行标记；其他比较运算符也是如此：`!=`, `<`, `<=`, `>`, 和 `>=`。

C.87: 请当心基类的 `==`

理由

为类层次编写一个傻瓜式的并且有用处的 `==` 是相当困难的。

示例，不好

```
class B {  
    string name;  
    int number;  
public:  
    virtual bool operator==(const B& a) const  
{  
    return name == a.name && number == a.number;  
}  
// ...  
};
```

`B` 的比较函数接受对其第二个操作数的类型转换，但第一个则并非如此。

```
class D : public B {  
    char character;  
public:  
    virtual bool operator==(const D& a) const  
{  
    return B::operator==(a) && character == a.character;  
}  
// ...  
};  
  
B b = ...  
D d = ...  
b == d;      // 比较 name 和 number，但忽略了 d 的 character  
d == b;      // 比较 name 和 number，但忽略了 b 的 character  
D d2;  
d == d2;     // 比较 name、number 和 character  
B& b2 = d2;  
b2 == d;     // 比较 name 和 number，但忽略了 d2 和 d 的 character
```

显然有许多使 `==` 在类层次中可以工作的方式，但不成熟的方案是无法适应范围扩展的。

注解

本条规则适用于所有的常规比较运算符：`!=`, `<`, `<=`, `>`, `>=`, 以及 `<=>`。

强制实施

- 对虚的 `operator==()` 进行标记；其他比较运算符也是如此：`!=`, `<`, `<=`, `>`, `>=`, 以及 `<=>`。

C.89: 使 `hash` 函数 `noexcept`

理由

哈希容器的使用者都会间接地使用 `hash`, 并且不会预期简单的访问操作也会抛出异常。这是标准库的一条要求。

示例, 不好

```
template<>
struct hash<My_type> { // 非常不好的 hash 特化
    using result_type = size_t;
    using argument_type = My_type;

    size_t operator()(const My_type & x) const
    {
        size_t xs = x.s.size();
        if (xs < 4) throw Bad_My_type{}; // "没有人期待西班牙宗教裁判所!"
        return hash<size_t>()(x.s.size()) ^ trim(x.s);
    }
};

int main()
{
    unordered_map<My_type, int> m;
    My_type mt{ "asdfg" };
    m[mt] = 7;
    cout << m[My_type{ "asdfg" }] << '\n';
}
```

如果你必须定义 `hash` 的特化的话, 请尝试单纯地用 `^` (异或 xor) 把标准库的 `hash` 特化进行组合。这样做对于非专业人士来说往往会更好。

强制实施

- 标记可能抛出异常的 `hash`。

C.90: 依靠构造函数和赋值运算符, 不要依靠 `memset` 和 `memcpy`

理由

构造某个类型的实例的标准 C++ 机制是调用其构造函数。如指导方针 C.41 所述：构造函数应当创建一个已完全初始化的对象。不应当需要进行如用 `memcpy` 来进行的额外初始化。

为适当地做出一个类的副本并维持类型的不变式, 类型将提供复制构造函数和/或复制赋值运算符。使用 `memcpy` 来复制一个非可平凡复制的类型具有未定义的行为。这经常会导致切片, 或者数据损坏。

示例，好

```
struct base {
    virtual void update() = 0;
    std::shared_ptr<int> sp;
};

struct derived : public base {
    void update() override {}
};
```

示例，不好

```
void init(derived& a)
{
    memset(&a, 0, sizeof(derived));
}
```

这样做类型不安全并且会覆写掉虚表。

示例，不好

```
void copy(derived& a, derived& b)
{
    memcpy(&a, &b, sizeof(derived));
}
```

这样做同样类型不安全并且会覆写掉虚表。

强制实施

- 对将非可平凡复制类型传递给 `memset` 或 `memcpy` 进行标记。

C.con: 容器和其他资源包装类

容器是一种持有某个类型的对象序列的对象；`std::vector` 就是一种典型的容器。

资源包装类是拥有某个资源的类；`std::vector` 是一种典型的资源包装类；它的资源就是其元素的序列。

容器规则概览

- [C.100: 定义容器的时候要遵循 STL](#)
- [C.101: 为容器提供值语义](#)
- [C.102: 为容器提供移动操作](#)
- [C.103: 为容器提供一个初始化式列表构造函数](#)
- [C.104: 为容器提供一个将之置空的默认构造函数](#)
- ???
- [C.109: 当资源包装类具有指针语义时，应提供 `*` 和 `>`](#)

参见: [资源](#)

C.100: 定义容器的时候要遵循 STL

理由

大多数 C++ 程序员都熟悉 STL 容器，而且它们具有本质上十分健全的设计。

注解

当然也存在其他本质上健全的设计风格，有时候也存在不遵循标准程序库的设计风格的各种理由，但在没有非常坚实的理由的情况下，让实现者和用户都遵循标准，既简单又容易。

尤其是，`std::vector` 和 `std::map` 都提供了相当简单的模型。

示例

```
// 简化版本（比如没有分配器）：

template<typename T>
class Sorted_vector {
    using value_type = T;
    // ... 各迭代器类型 ...

    Sorted_vector() = default;
    Sorted_vector(initializer_list<T>);      // 初始化式列表构造函数：进行排序并存储
    Sorted_vector(const Sorted_vector&) = default;
    Sorted_vector(Sorted_vector&&) = default;
    Sorted_vector& operator=(const Sorted_vector&) = default; // 复制赋值
    Sorted_vector& operator=(Sorted_vector&&) = default;       // 移动赋值
    ~Sorted_vector() = default;

    Sorted_vector(const std::vector<T>& v); // 存储并排序
    Sorted_vector(std::vector<T>&& v);        // 排序并“窃取表示”

    const T& operator[](int i) const { return rep[i]; }
    // 不提供非 const 的直接访问，以维持顺序

    void push_back(const T&); // 在正确位置插入（不一定在末尾）
    void push_back(T&&);   // 在正确位置插入（不一定在末尾）

    // ... cbegin(), cend() ...

private:
    std::vector<T> rep; // 用一个 std::vector 来持有各元素
};

template<typename T> bool operator==(const Sorted_vector<T>&, const
Sorted_vector<T>&);
template<typename T> bool operator!=(const Sorted_vector<T>&, const
Sorted_vector<T>&);
// ...
```

这段代码遵循 STL 风格但并不完整。

这种做法并不少见。

仅仅为特定的容器提供足以使其有意义的功能即可。

这里的关键在于，定义（对特定容器来说有意义的）符合约定的构造、赋值、析构函数和各迭代器

并提供它们符合约定的语义。
在此基础上，可以根据需要对这个容器进行扩展。
这里添加了来自 `std::vector` 的一些特殊构造函数。

强制实施

???

C.101: 为容器提供值语义

理由

常规对象的理解和推理要比非常规对象更加简单。
使人感觉熟悉。

注解

如果有意义的话，要使容器满足 `Regular` (概念)。
尤其是，确保对象与自身的副本比较时相等。

示例

```
void f(const Sorted_vector<string>& v)
{
    Sorted_vector<string> v2 {v};
    if (v != v2)
        cout << "Behavior against reason and logic.\n";
    // ...
}
```

强制实施

???

C.102: 为容器提供移动操作

理由

容器都有变大的趋势；没有移动构造函数和复制构造函数的对象
进行到处移动可以很昂贵，因而趋使人们转而传递指向它的指针，
从而带来资源管理方面的问题。

示例

```
Sorted_vector<int> read_sorted(istream& is)
{
    vector<int> v;
    cin >> v;    // 假定存在向量的读取操作
    Sorted_vector<int> sv = v; // 进行排序
    return sv;
}
```

用户可以合理地假设返回一个标准程序库风格的容器是廉价的。

强制实施

???

C.103: 为容器提供一个初始化式列表构造函数

理由

人们期望能够以一组值来初始化一个容器。

使人感觉熟悉。

示例

```
Sorted_vector<int> sv {1, 3, -1, 7, 0, 0}; // sorted_vector 按需对其元素进行排序
```

强制实施

???

C.104: 为容器提供一个将之置空的默认构造函数

理由

使其满足 Regular。

示例

```
vector<Sorted_sequence<string>> vs(100); // 100 个 sorted_sequence, 值均为 ""
```

强制实施

???

C.109: 当资源包装类具有指针语义时，应提供 * 和 ->

理由

这正是对指针所预期的行为，

使人感觉熟悉。

示例

```
???
```

强制实施

???

C.lambdas: 函数对象和 lambda

函数对象是提供了重载的 () 的对象，因此可以进行调用。

Lambda 表达式（通常通俗地简称为“lambda”）是一种产生函数对象的写法。

函数对象应当可廉价地复制（因此可以按值传递）。

概要：

- [F.10: 若操作可被重用，则应为其命名](#)

- F.11: 当需要仅在一处使用的简单函数对象时使用无名 lambda
- F.50: 无法用函数达成（捕捉局部变量，或者编写局部函数）时，应使用 lambda
- F.52: 在将被局部范围内使用（包括将之传递给算法）的 lambda 中优先按引用捕捉
- F.53: 在不被局部范围内使用（包括存储在堆上，或传递给其他线程）的 lambda 中避免按引用捕捉
- ES.28: 针对复杂的初始化（尤其是 `const` 变量）使用 lambda

C.hier: 类层次 (OOP)

构建类层次（仅）用于表达一组按层次组织的概念。

基类通常都表现为接口。

类层次有两种主要的用法，它们通常被叫做实现继承和接口继承。

类层次规则概览：

- C.120: 类层次（仅）用于表达具有天然层次化结构的概念
- C.121: 如果基类被用作接口的话，应使其成为纯抽象类
- C.122: 当需要完全区分接口和实现时，应当用抽象类作为接口

类层次的设计规则概览：

- C.126: 抽象类通常并不需要用户编写的构造函数
- C.127: 带有虚函数的类应当带有虚的或受保护的析构函数
- C.128: 虚函数应当指明 `virtual`、`override`、`final` 三者之一
- C.129: 当设计类层次时，应区分实现继承和接口继承
- C.130: 多态类的深拷贝；优先采用虚函数 `clone` 来替代公开复制构造/赋值
- C.131: 避免无价值的取值和设值函数
- C.132: 请勿无理由地使函数 `virtual`
- C.133: 避免 `protected` 数据
- C.134: 确保所有非 `const` 数据成员有相同的访问级别
- C.135: 用多继承来表达多个不同的接口
- C.136: 用多继承来表达一些实现特性的合并
- C.137: 用 `virtual` 基类以避免过于通用的基类
- C.138: 用 `using` 来为派生类和其基类建立重载集合
- C.139: 对类运用 `final` 应当保守
- C.140: 不要在虚函数和其覆盖函数上提供不同的默认参数

对类层次中的对象进行访问的规则概览：

- C.145: 通过指针和引用来访问多态对象
- C.146: 当无法避免在类层次上进行导航时应使用 `dynamic_cast`
- C.147: 当查找所需类的失败被当做一种错误时，应当对引用类型使用 `dynamic_cast`
- C.148: 当查找所需类的失败被当做一种有效的可能情况时，应当对指针类型使用 `dynamic_cast`
- C.149: 用 `unique_ptr` 或 `shared_ptr` 来避免忘记对以 `new` 所创建的对象进行 `delete` 的情况
- C.150: 用 `make_unique()` 来构建由 `unique_ptr` 所拥有的对象
- C.151: 用 `make_shared()` 来构建由 `shared_ptr` 所拥有的对象
- C.152: 禁止把指向派生类对象的数组的指针赋值给指向基类的指针
- C.153: 优先采用虚函数而不是强制转换

C.120: 使用类层次来表达具有天然层次化结构的概念

理由

直接在代码中表达想法可以简化理解和维护工作。应当保证各个基类所表达的想法与全部派生类型精确匹配，并且确实找不到比使用继承所带来的紧耦合方式更好的表达方式。

当单纯使用数据成员就能搞定时请不要使用继承。这种情况通常意味着派生类型需要覆盖某个基类虚函数或者需要访问某个受保护成员。

示例

```
class DrawableUIElement {
public:
    virtual void render() const = 0;
    // ...
};

class AbstractButton : public DrawableUIElement {
public:
    virtual void onClick() = 0;
    // ...
};

class PushButton : public AbstractButton {
    void render() const override;
    void onClick() override;
    // ...
};

class Checkbox : public AbstractButton {
// ...
};
```

示例，不好

请不要把非层次化的领域概念表示成类层次。

```
template<typename T>
class Container {
public:
    // 列表操作:
    virtual T& get() = 0;
    virtual void put(T&) = 0;
    virtual void insert(Position) = 0;
    // ...
    // 向量操作:
    virtual T& operator[](int) = 0;
    virtual void sort() = 0;
    // ...
    // 树操作:
    virtual void balance() = 0;
    // ...
};
```

大多数派生类都无法恰当实现这个接口所要求的大多数函数。
因而这个基类成为了一个实现负担。
此外，`Container` 的使用者无法依靠成员函数来相当高效地确实实施某个有意义的操作；
它可能会抛出某个异常。
因此使用者只得诉诸于运行时检查，并且
放弃使用这个（过于）一般化的接口，代之以某种运行时类型查询（比如 `dynamic_cast`）所确定的接
口。

强制实施

- 寻找带有许多不干别的只会抛出异常的成员的类。
- 对非公用基类 `B` 的每次使用进行标记，其中派生类 `D` 并未覆盖 `B` 的某个虚函数，或访问某个受
保护成员，而 `B` 并非以下情况：为空，为 `D` 的模板参数或参数包组，或者为以 `D` 所特化的类模
板。

C.121: 如果基类被用作接口的话，应使其成为纯抽象类

理由

不包含数据的类更加稳定（更不脆弱易变）。
接口通常都应当全部由公开的纯虚函数和一个预置的或为空的虚析构函数组成。

示例

```
class My_interface {  
public:  
    // ... 只有一个纯虚函数 ...  
    virtual ~My_interface() {} // 或者 =default  
};
```

示例，不好

```
class Goof {  
public:  
    // ... 只有一个纯虚函数 ...  
    // 没有虚析构函数  
};  
  
class Derived : public Goof {  
    string s;  
    // ...  
};  
  
void use()  
{  
    unique_ptr<Goof> p {new Derived{"here we go"};}  
    f(p.get()); // 通过 Goof 接口使用 Derived  
    g(p.get()); // 通过 Goof 接口使用 Derived  
} // 泄漏
```

`Derived` 是通过其 `Goof` 接口而被 `delete` 的，而它的 `string` 则泄漏了。
为 `Goof` 提供虚析构函数就能使其正常工作。

强制实施

- 对任何含有数据成员同时带有并非从基类继承的可被覆盖（非 `final`）的虚函数的类给出警告。

C.122: 当需要完全区分接口和实现时，应当用抽象类作为接口

理由

诸如在 ABI (连接) 边界这种地方。

示例

```
struct Device {
    virtual ~Device() = default;
    virtual void write(span<const char> outbuf) = 0;
    virtual void read(span<char> inbuf) = 0;
};

class D1 : public Device {
    // ... 数据 ...

    void write(span<const char> outbuf) override;
    void read(span<char> inbuf) override;
};

class D2 : public Device {
    // ... 不同的数据 ...

    void write(span<const char> outbuf) override;
    void read(span<char> inbuf) override;
};
```

使用者可以通过由 `Device` 所提供的接口来互换地使用 `D1` 和 `D2`。

而且，只要其访问一直是通过 `Device` 进行的话，也可以以与老版本二进制不兼容的方式来更新 `D1` 和 `D2`。

强制实施

???

C.hierclass: 类层次的设计:

C.126: 抽象类通常并不需要用户编写的构造函数

理由

通常抽象类并没有任何需要由构造函数来初始化的对象。

示例

```
class Shape {
public:
    // 抽象基类中不需要用户编写的构造函数
    virtual Point center() const = 0;      // 纯虚
    virtual void move(Point to) = 0;
```

```

    // ... 其他纯虚函数 ...
    virtual ~Shape() {} // 析构函数
};

class Circle : public Shape {
public:
    Circle(Point p, int rad); // 派生类中的构造函数
    Point center() const override { return x; }
};

```

例外

- 有任务的基类构造函数，比如把对象注册到什么地方的时候，可能是需要构造函数的。
- 在极端少见的情况下，你可能发觉让抽象类来包含一点所有派生类都会共享的数据是有意义的（比如说，使用情况统计数据，调试信息等）；这样的类倾向于带有构造函数。但应当警醒的是：这样的类也倾向于要求进行虚继承。

强制实施

对带有构造函数的抽象类进行标记。

C.127: 带有虚函数的类应当带有虚的或受保护的析构函数

理由

带有虚函数的类通常是通过指向基类的指针来使用的。一般来说，最后一个使用者必须在基类指针上执行 `delete`，这常常是通过基类智能指针来做到的，因而析构函数应当为 `public` 和 `virtual`。而不那么常见的情况是当并不打算支持通过基类指针来删除时，这时析构函数应当为 `protected` 和非 `virtual`；参见 [C.35](#)。

示例，不好

```

struct B {
    virtual int f() = 0;
    // ... 没有用户编写的析构函数，缺省为 public 非 virtual ...
};

// 不好：继承于没有虚析构函数的类
struct D : B {
    string s {"default"};
    // ...
};

void use()
{
    unique_ptr<B> p = make_unique<D>();
    // ...
} // 未定义行为。可能仅仅调用了 B::~B 而字符串则被泄漏了

```

注解

有些人不遵守本条规则，因为他们打算仅通过 `shared_ptr` 来使用这些类：`std::shared_ptr p = std::make_shared<D>(args)`；这种情况下，由共享指针来负责删除对象，因此并不会发生不适当的基类 `delete` 所导致的泄漏。坚持一贯这样做的人可能会得到假阳性的警告，但这条规则其实很重要——当通过 `make_unique` 分配对象时会如何呢？这样的话是不安全的，除非 `B` 的作者保证它不会被误用，

比如可以让所有的构造函数为私有的并提供一个工厂函数，来强制保证分配都是通过 `make_shared` 进行。

强制实施

- 带有任何虚函数的类的析构函数应当要么是 `public` 和 `virtual`，要么是 `protected` 和非 `virtual` 的。
- 把对带有虚函数但没有虚析构函数的类的 `delete` 标记出来。

C.128: 虚函数应当指明 `virtual`、`override`、`final` 三者之一

理由

可读性。

检测错误。

明确写下的 `virtual`、`override` 或 `final` 是自说明的，并使编译器可以检查到基类和派生类之间的类型和/或名字的不匹配。不过写出超过一个则不仅多余而且是潜在的错误来源。

可以遵循简单明确的含义：

- `virtual` 刚好仅仅表明“这是一个新的虚函数”。
- `override` 刚好仅仅表明“这是一个非最终覆盖函数”。
- `final` 刚好仅仅表明“这是一个最终覆盖函数”。

示例，不好

```
struct B {  
    void f1(int);  
    virtual void f2(int) const;  
    virtual void f3(int);  
    // ...  
};  
  
struct D : B {  
    void f1(int);           // 不好（希望会有警告）：D::f1() 隐藏了 B::f1()  
    void f2(int) const;     // 不好（但惯用且合法）：没有明确 override  
    void f3(double);        // 不好（希望会有警告）：D::f3() 隐藏了 B::f3()  
    // ...  
};
```

示例，好

```
struct Better : B {  
    void f1(int) override;      // 错误（被发现）：Better::f1() 隐藏了 B::f1()  
    void f2(int) const override;  
    void f3(double) override;   // 错误（被发现）：Better::f3() 隐藏了 B::f3()  
    // ...  
};
```

讨论

我们希望消除两种特定类型的错误：

- **隐式虚函数**: 程序员有意使函数隐含为虚函数，而它确实如此（但代码的读者搞不清楚这点）；或者，程序员有意使函数隐含为虚函数，但它并非如此（例如，由于微妙的参数列表不匹配所导致）；或者，程序员并非有意使函数为虚函数，但它却成为虚函数（由于它刚好与基类中的某个虚函数具有相同的签名）
- **隐式覆盖**: 程序员有意使函数隐式地成为覆盖函数，而它确实如此（但代码的读者搞不清楚这点）；或者，程序员有意使函数隐式地成为覆盖函数，但它并非如此（例如，由于微妙的参数列表不匹配）；或者，程序员并非有意使函数成为覆盖函数，但它却成为覆盖函数（由于它刚好与基类中的某个虚函数具有相同的签名 -- 注意无论这个函数是否被显式声明为虚函数都会发生这个问题，因为程序员的意图既可能就是要创建一个新的虚函数也可能要创建一个新的非虚函数）

注意：对于定义为 `final` 的类来说，是否在一个虚函数上标记 `override` 或 `final` 是无所谓的。

注意：对函数使用 `final` 要保守。它不一定会带来优化，但会排除进一步的覆盖。

强制实施

- 比较基类和派生类中的虚函数的名字，并对并未进行覆盖的相同名字的使用进行标记。
- 对既没有 `override` 也没有 `final` 的覆盖函数进行标记。
- 对函数声明中使用 `virtual`、`override` 和 `final` 中超过一个的情况进行标记。

C.129: 当设计类层次时，应区分实现继承和接口继承

理由

接口中的实现细节会使接口变得脆弱；

就是说，当实现被改变时其用户不得不进行重新编译。

基类中的数据增加了基类实现的复杂性，而且会导致代码的重复。

注解

定义：

- 接口继承，是使用继承来把用户和实现进行分离，特别是允许添加和修改派生类而不影响基类的用户。
- 实现继承，是使用继承来简化新设施的实现，通过将有用的操作提供给相关的新操作的实现者（有时候称作“差异式编程”）。

纯粹的接口类只是一组纯虚函数；参见 [I.25](#)。

在早期的 OOP 时代（比如 80 和 90 年代），实现继承和接口继承通常是混在一起的，而不良习惯则很难改掉。

即便是现在，这种混合在旧代码和老式的教学材料中也不少见。

对两种继承进行区分的重要性随着以下情形而增长：

- 类层次的大小（比如几十个派生类），
- 类层次的使用时期（比如几十年），以及
- 使用这个类层次的独立团体的数量
(比如，可能对分发和更新某个基类造成困难)

示例，不好

```
class Shape {    // 不好，混合了接口和实现
public:
    Shape();
    Shape(Point ce = {0, 0}, Color co = none): cent{ce}, col{co} { /* ... */}

    Point center() const { return cent; }
    Color color() const { return col; }

    virtual void rotate(int) = 0;
    virtual void move(Point p) { cent = p; redraw(); }

    virtual void redraw();

    // ...
private:
    Point cent;
    Color col;
};

class Circle : public Shape {
public:
    Circle(Point c, int r) : Shape{c}, rad{r} { /* ... */ }

    // ...
private:
    int rad;
};

class Triangle : public Shape {
public:
    Triangle(Point p1, Point p2, Point p3); // 计算中心点
    // ...
};
```

问题：

- 随着类层次的增长和向 `Shape` 添加更多的数据，构造函数会越发难于编写和维护。
- 为什么要计算 `Triangle` 的中心点？我们也许从不用它。
- 向 `Shape` 添加新的数据成员（比如绘制风格或者画布）
将导致所有派生于 `Shape` 的类和所有使用 `shape` 的代码都需要进行复审，可能需要修改，而且很可能需要重新编译。

`Shape::move()` 的实现就是实现继承的一个例子：

我们为所有派生类一次性定义 `move()`。

在这种基类成员函数实现中的代码越多，在基类中放入的共享数据越多，就能获得越多的好处——而类层次则越不稳定。

示例

这个 `shape` 类层次可以用接口继承重新编写：

```
class Shape { // 纯接口
public:
    virtual Point center() const = 0;
    virtual Color color() const = 0;

    virtual void rotate(int) = 0;
    virtual void move(Point p) = 0;

    virtual void redraw() = 0;

    // ...
};
```

注意纯接口很少会有构造函数：没什么需要构造的。

```
class Circle : public Shape {
public:
    Circle(Point c, int r, Color c) : cent{c}, rad{r}, col{c} { /* ... */ }

    Point center() const override { return cent; }
    Color color() const override { return col; }

    // ...
private:
    Point cent;
    int rad;
    Color col;
};
```

接口不再那么脆弱了，但成员函数的实现需要做更多工作。

比如说，每个派生于 `shape` 的类都得实现 `center`。

示例，双类层次

如何才能同时获得接口继承的稳定类层次的好处和实现继承的实现重用的好处呢？

一种流行的技巧是双类层次。

有许多实现双类层次的方式；这里，我们使用一种多重继承形式。

首先规划一个接口类的层次：

```
class Shape { // 纯接口
public:
    virtual Point center() const = 0;
    virtual Color color() const = 0;

    virtual void rotate(int) = 0;
    virtual void move(Point p) = 0;

    virtual void redraw() = 0;

    // ...
```

```

};

class circle : public virtual shape { // 纯接口
public:
    virtual int radius() = 0;
    // ...
};

```

为使这个接口有用处，我们必须提供其实现类（我们这里用相同的名字，但放入 `Impl` 命名空间）：

```

class Impl::Shape : public virtual ::shape { // 实现
public:
    // 构造函数，析构函数
    // ...
    Point center() const override { /* ... */ }
    Color color() const override { /* ... */ }

    void rotate(int) override { /* ... */ }
    void move(Point p) override { /* ... */ }

    void redraw() override { /* ... */ }

    // ...
};

```

现在 `shape` 是一个贫乏的具有一个实现的类的例子，
但还请谅解，因为这只是用来展现一种针对更复杂的类层次的技巧的简单例子。

```

class Impl::Circle : public virtual ::Circle, public Impl::Shape { // 实现
public:
    // 构造函数，析构函数

    int radius() override { /* ... */ }
    // ...
};

```

我们可以通过添加一个 `Smiley` 类来扩展它 (:-))：

```

class Smiley : public virtual Circle { // 纯接口
public:
    // ...
};

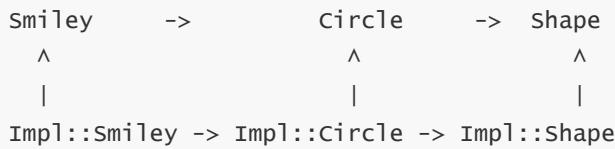
class Impl::Smiley : public virtual ::Smiley, public Impl::Circle { // 实现
public:
    // 构造函数，析构函数
    // ...
};

```

这里有两个类层次：

- 接口：Smiley -> Circle -> Shape
- 实现：Impl::Smiley -> Impl::Circle -> Impl::Shape

由于每个实现都同时派生于其接口和其实现基类，我们因此获得了一个晶格（DAG）：



我们曾经说过，这只是用来构造双类层次的一种方式。

可以直接使用实现类层次，而不用通过抽象接口来进行。

```
void work_with_shape(Shape&);

int user()
{
    Impl::Smiley my_smiley{ /* args */ }; // 创建具体的形状
    // ...
    my_smiley.some_member(); // 直接使用实现类
    // ...
    work_with_shape(my_smiley); // 通过抽象接口使用实现
    // ...
}
```

这种做法在实现类带有并未由抽象接口提供的成员时，或者当直接使用某个成员具有优化机会（比如当实现成员函数为 `final`）时，比较有用。

注解

分离接口和实现的另一个（相关的）技巧是 [Pimpl](#)。

注解

在提供公共的功能时，我们通常需要在作为（有实现的）基类函数和（在某个实现命名空间中的）自由函数之间进行选择。

基类能够提供更简短的写法，以及更容易访问（基类中的）共享数据，但所付出的是其功能将仅能被这个类层次的用户所使用。

强制实施

- 若派生类向基类转换的基类同时具有数据和虚函数，则对其进行标记（但排除在派生类成员中对基类成员的调用）。
- ???

C.130: 多态类的深拷贝；优先采用虚函数 `clone` 来替代公开复制构造/赋值

理由

由于切片的问题，不鼓励多态类的复制操作，参见 [C.67](#)。如果确实需要复制语义的话，应当进行深复制：提供一个虚 `clone` 函数，它复制的是真正的最终派生类型，并返回指向新对象的具有所有权的指针，而且在派生类中它返回的也是派生类型（利用协变返回类型）。

示例

```
class B {
public:
    B() = default;
    virtual ~B() = default;
    virtual gsl::owner<B*> clone() const = 0;
protected:
    B(const B&) = default;
    B& operator=(const B&) = default;
    B(B&&) = default;
    B& operator=(B&&) = default;
    // ...
};

class D : public B {
public:
    gsl::owner<D*> clone() override
    {
        return new D{*this};
    }
};
```

通常来说，推荐使用智能指针来表示所有权（参见 [R.20](#)）。不过根据语言规则，协变返回类型不能是智能指针：当 `B::clone` 返回 `unique_ptr` 时 `D::clone` 不能返回 `unique_ptr<D>`。因此，你得在所有覆盖函数中统一都返回 `unique_ptr`，或者也可以使用[指导方针支持库](#)中的 `owner<>` 工具类。

C.131: 避免无价值的取值和设值函数

理由

无价值的取值和设值函数没有提供语义价值；让数据项自己 `public` 是一样的。

示例

```
class Point { // 不好: 嘴嗦
    int x;
    int y;
public:
    Point(int xx, int yy) : x{xx}, y{yy} { }
    int get_x() const { return x; }
    void set_x(int xx) { x = xx; }
    int get_y() const { return y; }
    void set_y(int yy) { y = yy; }
    // 没有有行为的成员函数
};
```

应当考虑把这个类变为 `struct`——就是一组没有行为的变量，全部都是公开数据而没有成员函数。

```
struct Point {  
    int x {0};  
    int y {0};  
};
```

注意，我们可以为成员变量提供默认初始化式：[C.49: 优先进行初始化而不是在构造函数中赋值](#).

注解

这条规则的关键在于取值和设值函数的语义是否是平凡的。虽然并非是对“平凡”的完整定义，但我们考虑在取值/设值函数，以及当使用公开数据成员之间除了语法上的差别之外是否存在什么差别。非平凡的语义的例子可能有：维护类的不变式，或者在某种内部类型和接口类型之间进行的转换。

强制实施

对大量仅提供单纯的成员访问而没有其他语义的 `get` 和 `set` 成员函数进行标记。

C.132: 请勿无理由地使函数 `virtual`

理由

多余的 `virtual` 会增加运行时间和对象代码的大小。

虚函数可以被覆盖，因此派生类中可能因此发生错误。

虚函数保证会在模板化的层次中造成代码重复。

示例，不好

```
template<class T>  
class vector {  
public:  
    // ...  
    virtual int size() const { return sz; }    // 不好：派生类能得到什么好处?  
private:  
    T* elem;      // 元素  
    int sz;        // 元素的数量  
};
```

这种类型的“向量”并非是为了让人用作基类的。

强制实施

- 对带有虚函数但没有派生类的类进行标记。
- 对所有成员函数都为虚函数并带有实现的类进行标记。

C.133: 避免 `protected` 数据

理由

`protected` 数据是复杂性和错误的一种来源。

`protected` 数据会把不变式的陈述搞复杂。

`protected` 数据天生违反了避免把数据放入基类的指导原则，而这样通常还会导致不得不采用虚继承。

示例，不好

```
class Shape {  
public:  
    // ... 接口函数 ...  
protected:  
    // 为派生类所使用的数据：  
    Color fill_color;  
    Color edge_color;  
    Style st;  
};
```

这样，由每个所定义的 `shape` 来保证对受保护数据正确进行操作。

这样做一度很流行，但同样是维护性问题的一种主要来源。

在大型的类层次中，很难保持对受保护数据的一致性使用，因为代码可能有很多，分散于大量的类之中。

能够触碰这些数据的类的集合是开放的：任何人都可以派生一个新的类并开始操作这些受保护数据。

检查这些类的完整集合通常是不可能做到的，因此对类的表示进行任何改动都是不可行的。

这些受保护数据上并没有强加的不变式；它们更像是一组全局变量。

受保护数据在大块代码中事实上成为了全局变量。

注解

受保护数据经常看起来倾向于允许通过派生来进行任意的改进。

而通常你得到的是肆无忌惮的改动和各种错误。

应当[优先采用 `private` 数据](#)并提供良好定义并强加的不变式。

或者通常更好的做法是，[不要在用作接口的任何类中存放数据](#)。

注解

受保护的成员函数则没有问题。

强制实施

对带有 `protected` 数据的类进行标记。

C.134: 确保所有非 `const` 数据成员有相同的访问级别

理由

防止出现会导致错误的逻辑混乱。

当非 `const` 数据成员的访问级别不同时，这个类型就会在它应当做什么上出现混乱。

这个类型是用来维持某个不变式的类型，还是仅仅集合了一组值而已？

探讨

其核心问题是：哪段代码应当负责为变量维护有意义/正确的值？

确切地说存在两种数据成员：

- A: 不参与对象的不变式的数据成员。这些成员的任何值的互相组合都是有效的。
- B: 参与对象不变式的数据成员。并非每种值组合都是有意义的（否则就没有不变式了）。因此所有具有对这些变量的写访问权限的代码都应当了解这个不变式，了解其语义，并了解（而且积极实现并加强）用以维持值的正确性的规则。

A 类别中的数据成员就应当是 `public` (或者很少情况下, 当你只想在派生类中见到它们时为 `protected`)。不需要对它们进行封装。系统中的所有代码都可以见到并操控它们。

B 类别中的数据成员应当为 `private` 或 `const`。这是因为封装很重要。让它们非 `private` 且非 `const` 可能意味着对象无法控制自身的状态: 这个类以外的无限量的代码可能都需要了解这个不变式, 并精确地参与它的维护工作——当这些数据成员都是 `public` 时, 这可能包括使用这个对象的所有调用方代码; 而当它们为 `protected` 时, 这可能包括当前以及未来的派生类的所有代码。这会导致易碎的且紧耦合的代码, 而且很快将会成为维护的噩梦。任何代码如果把这些数据成员设值为无效的或者预料之外的值的组合, 都可能搞坏对象以及随后对象的所有使用。

大多数的类要么都是 A, 要么都是 B:

- 全 `public`: 如果编写的是聚集一组变量而没有在这些变量之间维护不变式的话, 所有这些变量都应当为 `public`。
[依照惯例, 应当把这样的类声明为 `struct` 而不是 `class`](#)
- 全 `private`: 如果编写的类型维护了某个不变式, 则所有的非 `const` 变量都应当是 `private`——应当对它进行封装。

例外

偶尔会出现混合了 A 和 B 的类, 通常是为了方便调试。一个被封装的对象可能包含如非 `const` 调试信息的某种东西, 它并不是不变式的一部分, 因此属于 A 类别——其实它并非对象的值的一部分, 也不是这个对象的有意义的可观察状态。这种情况下, A 类别的部分应当按 A 的方式对待 (使之 `public`, 或罕见情况下当它们只应对派生类可见时, 使之为 `protected`), 而 B 类别的部分应当仍按 B 的方式对待 (`private` 或 `const`)。

强制实施

对包含具有不同访问级别的非 `const` 数据成员的类给出标记。

C.135: 用多继承来表达多个不同的接口

理由

并非所有的类都应当支持全部的接口, 而且并非所有的调用方都应当处理所有的操作。
尤其应当把巨大的接口拆解为可以被某个给定派生类所支持的不同的行为“方面”。

示例

```
class iostream : public istream, public ostream { // 充分简化  
    // ...  
};
```

`istream` 提供了输入操作的接口; `ostream` 提供了输出操作的接口。

`iostream` 提供了 `istream` 和 `ostream` 接口的并集, 以及在单个流上同时允许二者所需的同步操作。

注解

这是非常常见的继承的用法, 因为需要同一个实现提供多个不同接口是很常见的, 而通常把这种接口组织到一个单根层次中都是不容易的或者不自然的。

注解

这样的接口通常都是抽象类。

强制实施

???

C.136: 用多继承来表达一些实现特性的合并

理由

一些形式的混元（Mixin）带有状态，并通常会有对其状态提供的操作。

如果这些操作是虚的，就必须进行继承，而如果不是虚的，使用继承也可以避免例行代码和转发函数。

示例

```
class iostream : public istream, public ostream { // 充分简化  
    // ...  
};
```

`istream` 提供了输入操作的接口（以及一些数据）；`ostream` 提供了输出操作的接口（以及一些数据）。

`iostream` 提供了 `istream` 和 `ostream` 接口的并集，以及在单个流上同时允许二者所需的同步操作。

注解

这是一种相对少见的用法，因为实现通常都可以被组织到一个单根层次之中。

示例

有时候，“实现特性”更像是“混元”，决定实现的行为，并向其中注入成员以使该实现提供其所要求的策略。

相关的例子可以参考 `std::enable_shared_from_this`

或者 boost.intrusive 中的各种基类（例如 `list_base_hook` 和 `intrusive_ref_counter`）。

强制实施

???

C.137: 用 `virtual` 基类以避免过于通用的基类

理由

允许共享的数据和接口的分离。

避免将所有共享数据都被放到一个终极基类之中。

示例

```
struct Interface {  
    virtual void f();  
    virtual int g();  
    // ... 没有数据 ...  
};  
  
class Utility { // 带有数据
```

```

    void utility1();
    virtual void utility2(); // 定制点
public:
    int x;
    int y;
};

class Derive1 : public Interface, virtual protected Utility {
    // 覆盖了 Interface 的函数
    // 可能覆盖 Utility 的虚函数
    // ...
};

class Derive2 : public Interface, virtual protected Utility {
    // 覆盖了 Interface 的函数
    // 可能覆盖 Utility 的虚函数
    // ...
};

```

如果许多派生类都共享了显著的“实现细节”，弄一个 `Utility` 出来就是有意义的。

注解

很明显，这个例子过于“理论化”，但确实很难找到一个小型的现实例子出来。

`Interface` 是一个 [接口层次](#)的根，

而 `Utility` 则是一个 [实现层次](#)的根。

[一个稍微更现实的例子](#)，有一些解释。

注解

将层次结构线性化通常是更好的方案。

强制实施

对接口和实现混合的层次进行标记。

C.138: 用 `using` 来为派生类和其基类建立重载集合

理由

没有 `using` 声明的话，派生类的成员函数将会隐藏全部其所继承的重载集合。

示例，不好

```

#include <iostream>
class B {
public:
    virtual int f(int i) { std::cout << "f(int): "; return i; }
    virtual double f(double d) { std::cout << "f(double): "; return d; }
    virtual ~B() = default;
};
class D: public B {
public:
    int f(int i) override { std::cout << "f(int): "; return i + 1; }
};
int main()
{

```

```
D d;
std::cout << d.f(2) << '\n'; // 打印 "f(int): 3"
std::cout << d.f(2.3) << '\n'; // 打印 "f(double): 3"
}
```

示例，好

```
class D: public B {
public:
    int f(int i) override { std::cout << "f(int): "; return i + 1; }
    using B::f; // 展露了 f(double)
};
```

注解

这个问题对虚的和非虚的成员函数都有影响。

对于可变基类，C++17 引入了一种 using 声明的可变形式：

```
template<class... Ts>
struct Overloader : Ts... {
    using Ts::operator()...; // 展露了每个基类中的 operator()
};
```

强制实施

诊断名字隐藏情况

C.139: 对类运用 `final` 应当保守

理由

用 `final` 类来把类层次封闭很少是由于逻辑上的原因而必须的，并可能破坏掉类层次的可扩展性。

示例，不好

```
class Widget { /* ... */ };

// 没有人会想要改进 My_Widget (你可能这么觉得)
class My_Widget final : public Widget { /* ... */ };

class My_improved_Widget : public My_Widget { /* ... */ }; // 错误：办不到了
```

注解

并非每个类都要作为基类的。

大多数标准库的类都是这样（比如，`std::vector` 和 `std::string` 都并非为了派生而设计）。

这条规则是关于在准备作为某个类层次的接口的带有虚函数的类中有关 `final` 的使用的。

注解

用 `final` 来把单个的虚函数封印则是易错的，因为在定义和覆盖一组函数时，`final` 是很容易被忽视的。

幸运的是，编译器能够捕捉到这种错误：你无法在派生类中重新声明/重新打开一个 `final` 成员。

注解

有关 `final` 带来的性能提升的断言是需要证实的。

非常常见的是，这种断言都是基于推测或者在其他语言上的经验而来的。

有一些例子中的 `final` 对于逻辑和性能因素来说可能都是重要的。

一个例子是编译器和语言分析工具中的性能关键的 AST 层次结构。

其中并非每年都会添加新的派生类，而且只有程序库的实现者会做这种事。

不过，误用（或者至少曾经的误用）的情况远远比这常见。

强制实施

标记出所有在类上使用的 `final`。

C.140: 不要在虚函数和其覆盖函数上提供不同的默认参数

理由

这会造成混乱：覆盖函数是不会继承默认参数的。

示例，不好

```
class Base {
public:
    virtual int multiply(int value, int factor = 2) = 0;
    virtual ~Base() = default;
};

class Derived : public Base {
public:
    int multiply(int value, int factor = 10) override;
};

Derived d;
Base& b = d;

b.multiply(10); // 这两次调用将会调用相同的函数，但分别
d.multiply(10); // 使用不同的默认实参，因此获得不同的结果
```

强制实施

对虚函数默认参数中，如果其在基类和派生类的声明中是不同的，则对其进行标记。

C.hier-access: 对类层次中的对象进行访问

C.145: 通过指针和引用来访问多态对象

理由

如果类带有虚函数的话，你（通常）并不知道你所使用的函数是由哪个类提供的。

示例

```
struct B { int a; virtual int f(); virtual ~B() = default };
struct D : B { int b; int f() override; };

void use(B b)
{
    D d;
    B b2 = d;    // 切片
    B b3 = b;
}

void use2()
{
    D d;
    use(d);    // 切片
}
```

两个 `d` 都发生了切片。

例外

你可以安全地访问处于自身定义的作用域之内的具名多态对象，这并不会将之切片。

```
void use3()
{
    D d;
    d.f();    // OK
}
```

另见

[多态类应当抑制复制操作](#)

强制实施

对所有切片都进行标记。

C.146: 当无法避免在类层次上进行导航时应使用 `dynamic_cast`

理由

`dynamic_cast` 是运行时检查。

示例

```
struct B {    // 接口
    virtual void f();
    virtual void g();
    virtual ~B();
};

struct D : B {    // 更宽的接口
    void f() override;
    virtual void h();
};
```

```

void user(B* pb)
{
    if (D* pd = dynamic_cast<D*>(pb)) {
        // ... 使用 D 的接口 ...
    }
    else {
        // ... 通过 B 的接口做事 ...
    }
}

```

使用其他的强制转换可能会违反类型安全，导致程序中所访问的某个真实类型为 `x` 的变量，被当做具有某个无关的类型 `z` 而进行访问：

```

void user2(B* pb)    // 不好
{
    D* pd = static_cast<D*>(pb);    // I know that pb really points to a D; trust
    me
    // ... 使用 D 的接口 ...
}

void user3(B* pb)    // 不安全
{
    if (some_condition) {
        D* pd = static_cast<D*>(pb);    // I know that pb really points to a D;
        trust me
        // ... 使用 D 的接口 ...
    }
    else {
        // ... 通过 B 的接口做事 ...
    }
}

void f()
{
    B b;
    user(&b);    // OK
    user2(&b);   // 糟糕的错误
    user3(&b);   // OK, *如果*程序员已经正确检查了 some_condition 的话
}

```

注解

和其他强制转换一样，`dynamic_cast` 被过度使用了。

[优先使用虚函数而不是强制转换。](#)

如果可行（无须运行时决议）并且相当便利的话，优先使用[静态多态](#)而不是继承层次的导航。

注解

一些人会在 `typeid` 更合适的时候使用 `dynamic_cast`；

`dynamic_cast` 是一个通用的“是一个”操作，用以发现对象上的最佳接口，

而 `typeid` 是“报告对象的精确类型”操作，用以发现对象的真实类型。

后者本质上就是更简单的操作，因而应当更快一些。

后者 (`typeid`) 是可以在需要时进行手工模仿的（比如说，当工作在（因为某种原因）禁止使用 RTTI 的系统上），
而前者 (`dynamic_cast`) 要正确地实现则要困难得多。

考虑：

```
struct B {
    const char* name {"B"};
    // 若 pb1->id() == pb2->id() 则 *pb1 与 *pb2 类型相同
    virtual const char* id() const { return name; }
    // ...
};

struct D : B {
    const char* name {"D"};
    const char* id() const override { return name; }
    // ...
};

void use()
{
    B* pb1 = new B;
    B* pb2 = new D;

    cout << pb1->id(); // "B"
    cout << pb2->id(); // "D"
```

```
if (pb1->id() == "D") {           // 貌似没问题
    D* pd = static_cast<D*>(pb1);
    // ...
}
// ...
}
```

`pb2->id == "D"` 的结果实际上是由实现定义的。

我们加上这个是为了警告自造的 RTTI 中的危险之处。

这个代码可能可以多年正常工作，但只在不会对字符串字面量进行唯一化的新机器，新编译器，或者新的连接器上失败。

当实现你自己的 RTTI 时，请当心这一点。

例外

如果你所用的实现提供的 `dynamic_cast` 确实很慢的话，你可能只得使用一种替代方案了。

不过，所有无法静态决议的替代方案都涉及显式的强制转换（通常为 `static_cast`），而且易于出错。

你基本上都会创建自己的特殊目的 `dynamic_cast`。

因此，首先应当确定你的 `dynamic_cast` 确实和你想的一样慢（其实有相当多的并不被支持的流言），而且你使用的 `dynamic_cast` 确实是性能十分关键的。

我们的观点是，当前的实现中的 `dynamic_cast` 并非很慢。

比如说，在合适的条件下，是可以以[快速常量时间](#)来实施 `dynamic_cast` 的。

但是，兼容性问题使得作出改变很难，虽然大家都同意对优化付出的努力是值得的。

在很罕见的情况下，如果你已经测量出 `dynamic_cast` 的开销确实有影响，你也有其他的方式来静态地保证向下转换的成功（比如说小心地应用 CRTP 时），而且其中并不涉及虚继承的话，可以考虑战术性地使用 `static_cast` 并带上显著的代码注释和免责声明，概述这个段落，而且由于类型系统无法验证其正确性而在维护中需要人工的关切。即便是这样，以我们的经验来说，这种“我知道我在干什么”的情况仍然是一种已知的 BUG 来源。

例外

考虑：

```
template<typename B>
class Dx : B {
    // ...
};
```

强制实施

- 对所有用 `static_cast` 来进行向下转换进行标记，其中也包括实施 `static_cast` 的 C 风格的强制转换。
- 本条规则属于[类型安全性剖面配置](#)。

C.147: 当查找所需类的失败被当做一种错误时，应当对引用类型使用 `dynamic_cast`

理由

对引用进行的强制转换，所表达的意图是最终会获得一个有效对象，因此这种强制转换必须成功。如果无法成功的话，`dynamic_cast` 将会抛出异常。

示例

```
std::string f(Base& b)
{
    return dynamic_cast<Derived&>(b).to_string();
}
```

强制实施

???

C.148: 当查找所需类的失败被当做一种有效的可能情况时，应当对指针类型使用 `dynamic_cast`

理由

`dynamic_cast` 转换允许测试指针是否指向某个其类层次中包含给定类的多态对象。由于其找不到类时仅会返回空值，因而可以在运行时予以测试。这允许编写的代码可以基于其结果而选择不同的代码路径。

与此相对，[C.147](#) 中失败即是错误，而且不能用于条件执行。

示例

下面的例子的 `Shape_owner` 的 `add` 函数获取构造的 `shape` 对象的所有权。这些对象也根据其几何特性被存储到了不同视图中。

这个例子中的 `shape` 并不继承于 `Geometric_attributes`，而是其各个子类继承。

```
void add(Shape * const item)
{
    // 总是获得其所有权
    owned_shapes.emplace_back(item);

    // 检查 Geometric_attribute 并将该形状加入到（零个/一个/某些/全部）视图中

    if (auto even = dynamic_cast<Even_sided*>(item))
    {
        view_of_evens.emplace_back(even);
    }

    if (auto trisym = dynamic_cast<Trilaterally_symmetrical*>(item))
    {
        view_of_trisyms.emplace_back(trisym);
    }
}
```

注解

找不到所需的类时 `dynamic_cast` 将返回空值，而解引用空指针将导致未定义的行为。
因此 `dynamic_cast` 的结果应当总是当做可能包含空值并进行测试。

强制实施

- 【复杂】除非在指针类型 `dynamic_cast` 的结果上进行了空值测试，否则就对该指针的解引用给出警告。

C.149: 用 `unique_ptr` 或 `shared_ptr` 来避免忘记对以 `new` 所创建的对象进行 `delete` 的情况

理由

避免资源泄漏。

示例

```
void use(int i)
{
    auto p = new int {7};           // 不好：用 new 来初始化局部指针
    auto q = make_unique<int>(9);   // ok：保证了为 9 所分配的内存会被回收
    if (0 < i) return;             // 可能会返回并泄漏
    delete p;                     // 太晚了
}
```

强制实施

- 标记用 `new` 的结果来对裸指针所进行的初始化。
- 标记对局部变量的 `delete`。

C.150: 用 `make_unique()` 来构建由 `unique_ptr` 所拥有的对象

参见 [R.23](#)。

C.151: 用 `make_shared()` 来构建由 `shared_ptr` 所拥有的对象

参见 [R.22](#)。

C.152: 禁止把指向派生类对象的数组的指针赋值给指向基类的指针

理由

对所得到的基类指针进行下标操作，将导致无效的对象访问并且可能造成内存损坏。

示例

```
struct B { int x; };
struct D : B { int y; };

void use(B*);

D a[] = {{1, 2}, {3, 4}, {5, 6}};
B* p = a;      // 不好：a 退化为 &a[0]，并被转换为 B*
p[1].x = 7;   // 覆盖了 a[0].y

use(a);        // 不好：a 退化为 &a[0]，并被转换为 B*
```

强制实施

- 对任何的数组退化和基类向派生类转换之间的组合进行标记。
- 应当将数组作为 `span` 而不是指针来进行传递，而且不能让数组的名字在放入 `span` 之前经手派生类向基类的转换。

C.153: 优先采用虚函数而不是强制转换

理由

虚函数调用安全，而强制转换易错。

虚函数调用达到全派生函数，而强制转换可能达到某个中间类而得到错误的结果（尤其是当类层次在维护中被修改之后）。

示例

```
???
```

强制实施

参见 [C.146](#) 和 ???

C.over: 重载和运算符重载

可以对普通函数，函数模板，以及运算符进行重载。
不能重载函数对象。

重载规则概览：

- [C.160: 定义运算符应当主要用于模仿传统用法](#)
- [C.161: 对于对称的运算符应当采用非成员函数](#)
- [C.162: 重载的操作之间应当大体上是等价的](#)
- [C.163: 应当仅对大体上等价的操作进行重载](#)
- [C.164: 避免隐式转换运算符](#)
- [C.165: 为定制点采用 `using`](#)
- [C.166: 一元 `&` 的重载只能作为某个智能指针或引用系统的一部分而提供](#)
- [C.167: 应当为带有传统含义的操作提供运算符](#)
- [C.168: 应当在操作数所在的命名空间中定义重载运算符](#)
- [C.170: 当想要重载 `lambda` 时，应当使用泛型 `lambda`](#)

C.160: 定义运算符应当主要用于模仿传统用法

理由

最小化意外情况。

示例

```
class X {  
public:  
    // ...  
    X& operator=(const X&); // 定义赋值的成员函数  
    friend bool operator==(const X&, const X&); // == 需要访问其内部表示  
    // 执行 a = b 之后将有 a == b  
    // ...  
};
```

这里维持了传统的语义：[副本之间相等](#)。

示例，不好

```
X operator+(X a, X b) { return a.v - b.v; } // 不好：让 + 执行减法
```

注解

非成员运算符应当要么是友元，要么定义于[其操作数所在的命名空间中](#)。
[二元运算符应当等价地对待其两个操作数](#)。

强制实施

也许是不可能的。

C.161: 对于对称的运算符应当采用非成员函数

理由

如果使用的是成员函数的话，就需要两个才行。

如果为（比如）`==` 采用的不是非成员函数的话，`a == b` 和 `b == a` 就会存在微妙的差别。

示例

```
bool operator==(Point a, Point b) { return a.x == b.x && a.y == b.y; }
```

强制实施

标记成员运算符函数。

C.162: 重载的操作之间应当大体上是等价的

理由

让逻辑上互相等价的操作对不同的参数类型使用不同的名字会带来混乱，导致在函数名字中编码类型信息，并妨碍泛型编程。

示例

考虑：

```
void print(int a);
void print(int a, int base);
void print(const string&);
```

这三个函数都对其参数进行（适当的）打印。相对而言：

```
void print_int(int a);
void print_based(int a, int base);
void print_string(const string&);
```

这三个函数也都对其参数进行（适当的）打印。在名字上附加仅仅增添了啰嗦程度，而且妨碍了泛型代码。

强制实施

???

C.163: 应当仅对大体上等价的操作进行重载

理由

让逻辑上不同的函数使用相同的名字会带来混乱，并导致在泛型编程时发生错误。

示例

考虑：

```
void open_gate(Gate& g); // 把车库出口通道的障碍移除  
void fopen(const char* name, const char* mode); // 打开文件
```

这两个操作本质上就是不同的（而且没有关联），因此让它们的名字相区别是正确的。相对而言：

```
void open(Gate& g); // 把车库出口通道的障碍移除  
void open(const char* name, const char* mode = "r"); // 打开文件
```

这两个操作仍旧本质不同（且没有关联），但它们的名字缩减成了（共同的）最小词，并带来了发生混乱的机会。

幸运的是，类型系统能够识别出许多这种错误。

注解

对于一般性的和流行的名字，比如 `open`、`move`、`+` 和 `==` 等等，应当特别小心。

强制实施

???

C.164: 避免隐式转换运算符

理由

隐式类型转换可以很基础（比如 `double` 向 `int`），但经常会导致意外（比如 `string` 到 C 风格的字符串）。

注解

优先采用明确命名的转换，除非给出了一条很重要的需求。

我们所谓“很重要的需求”的意思是，它在其应用领域中是基本的（比如整数向复数的转换），并且经常需要用到。不要仅仅为了微小的便利就（通过转换运算符或非 `explicit` 构造函数）引入隐式的类型转换。

示例

```
struct S1 {  
    string s;  
    // ...  
    operator char*() { return s.data(); } // 不好，可能会引起意外  
}  
  
struct S2 {  
    string s;  
    // ...  
    explicit operator char*() { return s.data(); }  
};  
  
void f(S1 s1, S2 s2)  
{  
    char* x1 = s1; // 可以，但在许多情况下会引起意外  
    char* x2 = s2; // 错误，这通常是一件好事
```

```
char* x3 = static_cast<char*>(s2); // 我们可以明确（在你的头脑里）  
}
```

可能在任意的难以发现的上下文里发生令人惊讶且可能具有破坏性的隐式转换，例如，

```
s1 ff();  
  
char* g()  
{  
    return ff();  
}
```

由`ff()`返回的字符串在返回它的指针之前被销毁。

强制实施

标记所有的非显式转换运算符。

C.165: 为定制点采用 `using`

理由

以便找到在一个独立的命名空间中所定义的函数对象或函数，它们对一个一般性函数进行了“定制”。

示例

考虑`swap`。这是一个通用的（标准库）函数，其定义可以在任何类型上工作。

不过，通常需要为特定的类型定义专门的`swap()`。

比如说，通用的`swap()`会对互相交换的两个`vector`的元素进行复制，而一个好的专门实现则完全不会复制任何元素。

```
namespace N {  
    My_type x { /* ... */ };  
    void swap(x&, x&); // 针对 N::x 所优化的 swap  
    // ...  
}  
  
void f1(N::x& a, N::x& b)  
{  
    std::swap(a, b); // 可能不是我们所要的：调用 std::swap()  
}
```

`f1()`中的`std::swap()`严格做了我们所要求的工作：它调用了命名空间`std`中的`swap()`。

不幸的是，这可能并非我们想要的。

怎样让其考虑`N::x`呢？

```
void f2(N::x& a, N::x& b)  
{  
    swap(a, b); // 调用了 N::swap  
}
```

但这也许并不是我们在泛型代码中所要的。

通常如果专门的函数存在的话我们就想用它，否则的话我们则需要通用的函数。

这是通过在函数的查找中包含通用函数而达成的：

```

void f3(N::x& a, N::x& b)
{
    using std::swap; // 使得 std::swap 可用
    swap(a, b); // 如果存在 N::swap 则调用之, 否则为 std::swap
}

```

强制实施

不大可能, 除非是如 `swap` 这样的已知的定制点。

问题在于无限定和带限定的名字查找都有其作用。

C.166: 一元 & 的重载只能作为某个智能指针或引用系统的一部分而提供

理由

& 运算符在 C++ 中很基本。

C++ 语义中的很多部分都假定了其默认的含义。

示例

```

class Ptr { // 一种智能指针
    Ptr(X* pp) : p(pp) { /* 检查 */ }
    X* operator->() { /* 检查 */ return p; }
    X operator[](int i);
    X operator*();
private:
    T* p;
};

class X {
    Ptr operator&() { return Ptr{this}; }
    // ...
};

```

注解

如果你要“折腾”运算符 & 的话, 请保证其定义和 `->`, `[]`, `*` 和 `.` 在结果类型上具有匹配的含义。

注意, 运算符 `.` 现在是无法重载的, 因此不可能做出一个完美的系统。

我们打算修正这一点: [Operator Dot \(R2\)](#)。

注意 `std::addressof()` 总会产生一个内建指针。

强制实施

比较微妙。如果用户定义了 & 而并未同时为其结果类型定义 `->`, 则进行警告。

C.167: 应当为带有传统含义的操作提供运算符

理由

可读性。约定。可重用性。支持泛型代码。

示例

```
void cout_my_class(const My_class& c) // 含糊，并非传统约定，非泛型
{
    std::cout << /* 此处为类成员 */;
}

std::ostream& operator<<(std::ostream& os, const my_class& c) // OK
{
    return os << /* 此处为类成员 */;
}
```

对于其自身而言，`cout_my_class` 也许是没问题的，但它无法用在（或组合到）依赖于 `<<` 的输出用法约定代码之中：

```
My_class var { /* ... */ };
// ...
cout << "var = " << var << '\n';
```

注解

大多数运算符都有很强烈和活跃的含义约定用法，比如

- 比较：`==`, `!=`, `<`, `<=`, `>`, `>=`, 以及 `<=>`
- 算术操作：`+`, `-`, `*`, `/`, 以及 `%`
- 访问操作：`.`, `->`, 一元 `*`, 以及 `[]`
- 赋值：`=`

请勿定义违反约定的用法，请勿为它们发明自己的名字。

强制实施

比较棘手。需要洞悉其语义。

C.168: 应当在操作数所在的命名空间中定义重载运算符

理由

可读性。

通过 ADL 寻找运算符的能力。

避免不同命名空间中的定义之间不一致。

示例

```
struct S { };
S operator+(S, S); // OK: 和 S 在相同的命名空间，甚至紧跟着 S
S s;

S r = s + s;
```

示例

```
namespace N {  
    struct S { };  
    S operator+(S, S); // OK: 和 S 在相同的命名空间，甚至紧跟着 S  
}  
  
N::S s;  
  
S r = s + s; // 通过 ADL 找到了 N::operator+()
```

示例，不好

```
struct S { };  
S s;  
  
namespace N {  
    bool operator!(S a) { return true; }  
    bool not_s = !s;  
}  
  
namespace M {  
    bool operator!(S a) { return false; }  
    bool not_s = !s;  
}
```

这里的 `!s` 的含义在 `N` 和 `M` 中是不同的。

这可能是最易混淆的一点。

移除 `namespace M` 的定义之后，混乱就被一种犯错的机会所代替。

注解

当为定义于不同命名空间的两个类型定义一个二元运算符时，无法遵循这条规则。

例如：

```
Vec::Vector operator*(const Vec::Vector&, const Mat::Matrix&);
```

也许最好避免这种情形。

参见

这是这条规则的一种特殊情况：[辅助函数应当定义于它们的类相同的命名空间之中](#)。

强制实施

- 对并非处于其操作数的命名空间中的运算符的定义进行标记。

C.170: 当想要重载 lambda 时，应当使用泛型 lambda

理由

无法通过定义两个带有相同名字的不同 lambda 来进行重载。

示例

```
void f(int);
void f(double);
auto f = [](char);    // 错误：无法重载变量和函数

auto g = [](int) { /* ... */ };
auto g = [](double) { /* ... */ }; // 错误：无法重载变量

auto h = [](auto) { /* ... */ }; // OK
```

强制实施

编译期会查明对 lambda 进行重载的企图。

C.union: 联合体

`union` 是一种 `struct`，其所有成员都开始于相同的地址，因而它同时只能持有一个成员。

`union` 并不会跟踪其所存储的是哪个成员，因此必须由程序员来保证其正确；

这本质上就是易错的，但有一些弥补的办法。

由一个 `union` 加上一个用于指出其当前持有哪个成员的指示符构成的类型被称为 **带标记联合体** (*tagged union*)，**区分性联合体** (*discriminated union*)，或者**变异体** (*variant*)。

联合体规则概览：

- [C.180: 采用 `union` 用以节省内存](#)
- [C.181: 避免“裸” `union`](#)
- [C.182: 利用匿名 `union` 实现带标记联合体](#)
- [C.183: 不要将 `union` 用于类型双关](#)
- ???

C.180: 采用 `union` 用以节省内存

理由

`union` 可以让一块内存在不同的时间用于不同类型的数据。

因此，当我们有几个不可能同时使用的对象时，可以用它来节省内存。

示例

```
union value {
    int x;
    double d;
};

value v = { 123 }; // 现在 v 持有一个 int
cout << v.x << '\n'; // 写下 123
v.d = 987.654; // 现在 v 持有一个 double
cout << v.d << '\n'; // 写下 987.654
```

但请你留意这个警告：[避免“裸” `union`](#)。

示例

```
// 短字符串优化

constexpr size_t buffer_size = 16; // 比指针的大小稍大

class Immutable_string {
public:
    Immutable_string(const char* str) :
        size(strlen(str))
    {
        if (size < buffer_size)
            strcpy_s(string_buffer, buffer_size, str);
        else {
            string_ptr = new char[size + 1];
            strcpy_s(string_ptr, size + 1, str);
        }
    }

    ~Immutable_string()
    {
        if (size >= buffer_size)
            delete[] string_ptr;
    }

    const char* get_str() const
    {
        return (size < buffer_size) ? string_buffer : string_ptr;
    }

private:
    // 当字符串足够短时，可以以其自己保存字符串
    // 而不是指向字符串的指针。
    union {
        char* string_ptr;
        char string_buffer[buffer_size];
    };

    const size_t size;
};
```

强制实施

???

C.181: 避免“裸” union

理由

裸联合体 (*naked union*) 是没有相关的指示其持有哪个成员 (如果有) 的指示器的联合体，程序员必须保持对它的跟踪。

裸联合体是类型错误的一种来源。

Example, bad

```
union value {  
    int x;  
    double d;  
};  
  
value v;  
v.d = 987.654; // v 持有一个 double
```

至此为止还都不错，但我们可能会轻易地误用这个 `union`：

```
cout << v.x << '\n'; // 不好，未定义的行为：v 持有一个 double，但我们将之当做一个 int  
来读取
```

注意这个类型错误的发生并没有任何明确的强制转换。

当我们测试程序时，其所打印的最后一个值是 `1683627180`，这是 `987.654` 的位模式的整数值。

我们这里遇到的是一个“不可见”的类型错误，它刚好给出的结果轻易可能被看作是无辜清白的。

对于“不可见”来说，下面的代码不会产生任何输出：

```
v.x = 123;  
cout << v.d << '\n'; // 不好：未定义的行为
```

替代方案

将它们和一个类型字段一起包装到类之中。

可以使用 C++17 的 `variant` 类型（在 `<variant>` 中可以找到）：

```
variant<int, double> v;  
v = 123; // v 持有一个 int  
int x = get<int>(v);  
v = 123.456; // v 持有一个 double  
w = get<double>(v);
```

强制实施

???

C.182: 利用匿名 union 实现带标记联合体

理由

设计良好的带标记联合体是类型安全的。

匿名联合体可以简化这种带有 (tag, union) 对的类的定义。

示例

这个例子基本上是从 TC++PL4 pp216-218 借鉴而来的。

你可以查看原文以获得其介绍。

这段代码多少有点复杂。

处理一个带有用户定义的赋值和析构函数的类型是比较麻烦的。

在标准中包含 `variant` 的原因之一就是避免程序员不得不编写这样的代码。

```
class value { // 以一个联合体来表现两个候选表示
private:
    enum class Tag { number, text };
    Tag type; // 区分

    union { // 表示（注意这是匿名联合体）
        int i;
        string s; // string 带有默认构造函数，复制操作，和析构函数
    };
public:
    struct Bad_entry { }; // 用作异常

    ~value();
    value& operator=(const value&); // 因为 string 变体的缘故而需要这个
    value(const value&);
    // ...
    int number() const;
    string text() const;

    void set_number(int n);
    void set_text(const string&);
    // ...
};

int value::number() const
{
    if (type != Tag::number) throw Bad_entry{};
    return i;
}

string value::text() const
{
    if (type != Tag::text) throw Bad_entry{};
    return s;
}

void value::set_number(int n)
{
    if (type == Tag::text) {
        s.~string(); // 显式销毁 string
        type = Tag::number;
    }
    i = n;
}

void value::set_text(const string& ss)
{
    if (type == Tag::text)
        s = ss;
    else {
        new(&s) string{ss}; // 放置式 new: 显式构造 string
        type = Tag::text;
    }
}
```

```

value& value::operator=(const value& e) // 因为 string 变体的缘故而需要这个
{
    if (type == Tag::text && e.type == Tag::text) {
        s = e.s; // 常规的 string 赋值
        return *this;
    }

    if (type == Tag::text) s.~string(); // 显式销毁

    switch (e.type) {
    case Tag::number:
        i = e.i;
        break;
    case Tag::text:
        new(&s) string(e.s); // 放置式 new: 显式构造
    }

    type = e.type;
    return *this;
}

value::~value()
{
    if (type == Tag::text) s.~string(); // 显式销毁
}

```

强制实施

???

C.183: 不要将 `union` 用于类型双关

理由

读取一个 `union` 曾写入的成员不同类型的成员是未定义的行为。
这种双关是不可见的，或者至少比具名强制转换更难于找出。
使用 `union` 的类型双关是一种错误来源。

示例，不好

```

union Pun {
    int x;
    unsigned char c[sizeof(int)];
};

```

`Pun` 的想法是要能够查看 `int` 的字符表示。

```

void bad(Pun& u)
{
    u.x = 'x';
    cout << u.c[0] << '\n'; // 未定义行为
}

```

如果你想要查看 `int` 的字节的话，应使用（具名）强制转换：

```
void if_you_must_pun(int& x)
{
    auto p = reinterpret_cast<std::byte*>(&x);
    cout << p[0] << '\n';      // OK: 好多了
    // ...
}
```

对从对象的声明类型向 `char*`, `unsigned char*`, 或 `std::byte*` 进行 `reinterpret_cast` 的结果进行访问是有定义的行为。（不建议使用 `reinterpret_cast`，但至少我们可以发觉发生了某种微妙的事情。）

注解

不幸的是, `union` 经常被用于类型双关。

我们认为“它有时候能够按预期工作”并不是一种令人信服的理由。

C++17 引入了一个独立类型 `std::byte` 以支持在原始对象表示上进行的操作。在这些操作中应当使用这个类型代替 `unsigned char` 或 `char`。

强制实施

???

Enum: 枚举

枚举用于定义整数值的集合，并用于为这种值集定义类型。有两种类型的枚举，“普通”的 `enum` 和 `class enum`。

枚举规则概览：

- [Enum.1: 优先采用枚举而不是宏](#)
- [Enum.2: 采用枚举来表示相关的具名常量的集合](#)
- [Enum.3: 优先采用 `enum class` 而不是“普通”`enum`](#)
- [Enum.4: 针对安全和简单的用法来为枚举定义操作](#)
- [Enum.5: 请勿为枚举符采用 `ALL_CAPS` 命名方式](#)
- [Enum.6: 避免使用无名枚举](#)
- [Enum.7: 仅在必要时才为枚举指定其底层类型](#)
- [Enum.8: 仅在必要时才指定枚举符的值](#)

Enum.1: 优先采用 `enum` 而不是宏

理由

宏不遵守作用域和类型规则。而且，宏的名字在预处理中就被移除，因而通常不会出现在如调试器这样的工具中。

示例

首先是一些不好的老代码：

```
// webcolors.h (第三方头文件)
#define RED    0xFF0000
#define GREEN  0x00FF00
#define BLUE   0x0000FF

// productinfo.h
// 以下则基于颜色定义了产品的子类型
#define RED    0
#define PURPLE 1
#define BLUE   2

int webby = BLUE; // webby == 2; 可能不是我们所想要的
```

代之以 `enum`：

```
enum class Web_color { red = 0xFF0000, green = 0x00FF00, blue = 0x0000FF };
enum class Product_info { red = 0, purple = 1, blue = 2 };

int webby = blue; // 错误：应当明确
Web_color webby = Web_color::blue;
```

我们用 `enum class` 来避免名字冲突。

强制实施

标记定义整数值的宏。

Enum.2: 采用枚举来表示相关的具名常量的集合

理由

枚举展示其枚举符之间是有关联的，且可以用作具名类型。

示例

```
enum class Web_color { red = 0xFF0000, green = 0x00FF00, blue = 0x0000FF };
```

注解

对枚举的 `switch` 是很常见的，编译器可以对不平常的 `case` 标签进行警告。例如：

```
enum class Product_info { red = 0, purple = 1, blue = 2 };

void print(Product_info inf)
{
    switch (inf) {
        case Product_info::red: cout << "red"; break;
        case Product_info::purple: cout << "purple"; break;
    }
}
```

这种漏掉一个的 `switch` 语句通常是添加枚举符并缺少测试的结果。

强制实施

- 当 `switch` 语句的 `case` 标签并未覆盖枚举的全部枚举符时，对其进行标记。
- 当 `switch` 语句的 `case` 覆盖了枚举的几个枚举符，但没有 `default` 时，对其进行标记。

Enum.3: 优先采用 `class enum` 而不是“普通”`enum`

理由

最小化意外情况：传统的 `enum` 太容易转换为 `int` 了。

示例

```
void Print_color(int color);

enum Web_color { red = 0xFF0000, green = 0x00FF00, blue = 0x0000FF };
enum Product_info { red = 0, purple = 1, blue = 2 };

Web_color webby = Web_color::blue;

// 显然至少有一个调用是有问题的。
Print_color(webby);
Print_color(Product_info::blue);
```

代之以 `enum class`：

```
void Print_color(int color);

enum class Web_color { red = 0xFF0000, green = 0x00FF00, blue = 0x0000FF };
enum class Product_info { red = 0, purple = 1, blue = 2 };

Web_color webby = Web_color::blue;
Print_color(webby); // 错误：无法转换 Web_color 为 int。
Print_color(Product_info::red); // 错误：无法转换 Product_info 为 int。
```

强制实施

【简单】对所有非 `class enum` 定义进行警告。

Enum.4: 针对安全和简单的用法来为枚举定义操作

理由

便于使用并避免犯错。

示例

```
enum Day { mon, tue, wed, thu, fri, sat, sun };

Day& operator++(Day& d)
{
    return d = (d == Day::sun) ? Day::mon : static_cast<Day>(static_cast<int>(d)+1);
}

Day today = Day::sat;
Day tomorrow = ++today;
```

这里使用 `static_cast` 有点不好，但

```
Day& operator++(Day& d)
{
    return d = (d == Day::sun) ? Day::mon : Day(++d);      // 错误
}
```

是无限递归，而且不用强制转换而使用一个针对所有情况的 `switch` 太冗长了。

强制实施

对重复出现的强制转换回枚举的表达式进行标记。

Enum.5: 请勿为枚举符采用 ALL_CAPS 命名方式

理由

避免和宏之间发生冲突

示例，不好

```
// webcolors.h (第三方头文件)
#define RED    0xFF0000
#define GREEN  0x00FF00
#define BLUE   0x0000FF

// productinfo.h
// 以下则基于颜色定义了产品的子类型

enum class Product_info { RED, PURPLE, BLUE };    // 语法错误
```

强制实施

标记 ALL_CAPS 风格的枚举符。

Enum.6: 避免使用无名枚举

理由

如果无法对枚举命名的话，它的值之间就是没有关联的。

示例，不好

```
enum { red = 0xFF0000, scale = 4, is_signed = 1 };
```

这种代码在出现指定整数常量的其他方便方式之前并不少见。

替代方案

代之以使用 `constexpr` 值。例如：

```
constexpr int red = 0xFF0000;
constexpr short scale = 4;
constexpr bool is_signed = true;
```

强制实施

对无名枚举进行标记。

Enum.7: 仅在必要时才为枚举指定其底层类型

理由

缺省情况的读写都是最简单的。

`int` 是缺省的整数类型。

`int` 是和 C 的 `enum` 相兼容的。

示例

```
enum class Direction : char { n, s, e, w,
                               ne, nw, se, sw }; // 底层类型可以节省空间

enum class Web_color : int32_t { red    = 0xFF0000,
                                 green   = 0x00FF00,
                                 blue    = 0x0000FF }; // 底层类型是多余的
```

注解

对 `enum` 或 `enum class` 前向声明时有必要指定底层类型：

```
enum Flags : char;

void f(Flags);

// ....

enum Flags : char { /* ... */ };
```

或者用以确保该类型的值具有指定的位精度：

```
enum Bitboard : uint64_t { /* ... */ };
```

强制实施

????

Enum.8: 仅在必要时才指定枚举符的值

理由

这是最简单的。

避免了枚举符值发生重复。

缺省情况会提供一组连续的值，并有利于 `switch` 语句的实现。

示例

```
enum class Col1 { red, yellow, blue };
enum class Col2 { red = 1, yellow = 2, blue = 2 }; // 打错字
enum class Month { jan = 1, feb, mar, apr, may, jun,
                  jul, august, sep, oct, nov, dec }; // 传统是从 1 开始
enum class Base_flag { dec = 1, oct = dec << 1, hex = dec << 2 }; // 位的集合
```

为了和传统的值相匹配（比如 `Month`），以及当连续的值不合要求

（比如像 `Base_flag` 一样分配不同的位），是需要指定值的。

强制实施

- 标记重复的枚举值
- 对明确指定的全部连续的枚举符的值进行标记。

R: 资源管理

本章节中包含于资源相关的各项规则。

资源，就是任何必须进行获取，并（显式或隐式）进行释放的东西，比如内存、文件句柄、Socket 和锁等等。

其必须进行释放的原因在于它们是短缺的，因而即便是采用延迟释放也许也是有害的。

基本的目标是要确保不会泄漏任何资源，而且不会持有不在需要的任何资源。

负责释放某个资源的实体被称作是其所有者。

少数情况下，发生泄漏是可接受的甚至是理想的：

如果所编写的程序只是基于输入来产生输出，而其所需的内存正比于输入的大小，那么最理想的（性能和开发便利性）策略有时候恰是不要删除任何东西。

如果有足够的内存来处理最大输入的话，让其泄漏即可，但如果并非如此，应当保证给出一条恰当的错误消息。

这里，我们将忽略这样的情况。

- 资源管理规则概览：
 - [R.1: 利用资源句柄和 RAI \(资源获取即初始化\) 来自动管理资源](#)
 - [R.2: 接口中的原生指针 \(仅\) 代表个体对象](#)
 - [R.3: 原生指针 \(`T*`\) 没有所有权](#)
 - [R.4: 原生引用 \(`T&`\) 没有所有权](#)
 - [R.5: 优先采用有作用域的对象，避免不必要的堆分配](#)
 - [R.6: 避免非 `const` 的全局变量](#)
- 分配和回收规则概览：
 - [R.10: 避免 `malloc\(\)` 和 `free\(\)`](#)

- [R.11: 避免显式调用 `new` 和 `delete`](#)
- [R.12: 显式资源分配的结果应当立即交给一个管理对象](#)
- [R.13: 单个表达式语句中至多进行一次显式资源分配](#)
- [R.14: 避免使用 `\[\]` 形参, 优先使用 `span`](#)
- [R.15: 总是同时重载相匹配的分配、回收函数对](#)
- 智能指针规则概览:
 - [R.20: 用 `unique_ptr` 或 `shared_ptr` 表示所有权](#)
 - [R.21: 优先采用 `unique_ptr` 而不是 `shared_ptr`, 除非需要共享所有权](#)
 - [R.22: 使用 `make_shared\(\)` 创建 `shared_ptr`](#)
 - [R.23: 使用 `make_unique\(\)` 创建 `unique_ptr`](#)
 - [R.24: 使用 `std::weak_ptr` 来打断 `shared_ptr` 的循环引用](#)
 - [R.30: 以智能指针为参数, 仅用于明确表达生存期语义](#)
 - [R.31: 非 `std` 的智能指针, 应当遵循 `std` 的行为模式](#)
 - [R.32: `unique_ptr<widget>` 参数用以表达函数假定获得 `widget` 的所有权](#)
 - [R.33: `unique_ptr<widget>&` 参数用以表达函数对该 `widget` 重新置位](#)
 - [R.34: `shared_ptr<widget>` 参数用以表达函数是所有者的一份子](#)
 - [R.35: `shared_ptr<widget>&` 参数用以表达函数可能会对共享的指针重新置位](#)
 - [R.36: `const shared_ptr<widget>&` 参数用以表达它可能将保留一个对对象的引用 ???](#)
 - [R.37: 不要把来自某个智能指针别名的指针或引用传递出去](#)

R.1: 利用资源句柄和 RAI (资源获取即初始化) 来自动管理资源

理由

避免资源泄漏和人工资源管理的复杂性。

C++ 语言确保的构造函数/析构函数对称性, 反映了资源的获取/释放函数对 (比如 `fopen / fclose`, `lock / unlock`, 以及 `new / delete` 等) 的对称性本质。

每当需要处理某个需要成对儿的获取/释放函数调用的资源时, 应当将资源封装到保证这种配对调用的对象之中——在构造函数中获取资源, 并在其析构函数中释放它。

示例, 不好

考虑:

```
void send(X* x, string_view destination)
{
    auto port = open_port(destination);
    my_mutex.lock();
    // ...
    send(port, x);
    // ...
    my_mutex.unlock();
    close_port(port);
    delete x;
}
```

这段代码中, 你必须记得在所有路径中调用 `unlock`、`close_port` 和 `delete`, 并且每个都恰好调用一次。

而且, 一旦上面标有 `...` 的任何代码抛出了异常, `x` 就会泄漏, 而 `my_mutex` 则保持锁定。

示例

考虑：

```
void send(unique_ptr<x> x, string_view destination) // x 拥有这个 x
{
    Port port{destination}; // port 拥有这个 PortHandle
    lock_guard<mutex> guard{my_mutex}; // guard 拥有这个锁
    // ...
    send(port, x);
    // ...
} // 自动解锁 my_mutex 并删除 x 中的指针
```

现在所有的资源清理都是自动进行的，不管是否发生了异常，所有路径中都会执行一次。额外的好处是，该函数现在明确声称它将接过指针的所有权。

`Port` 又是什么呢？是一个封装资源的便利句柄：

```
class Port {
    PortHandle port;
public:
    Port(string_view destination) : port{open_port(destination)} { }
    ~Port() { close_port(port); }
    operator PortHandle() { return port; }

    // port 句柄通常是不能克隆的，因此根据需要关闭了复制和赋值
    Port(const Port&) = delete;
    Port& operator=(const Port&) = delete;
};
```

注解

一旦发现一个“表现不良”的资源并未以带有析构函数的类来表示，就用一个类来包装它，或者使用 [finally](#)。

参见: [RAII](#)

R.2: 接口中的原生指针（仅）代表个体对象

理由

最好用某个容器类型（比如 `vector`，拥有数据），或者用 `span`（不拥有数据）来表示数组。这些容器和视图都带有足够的信息来进行范围检查。

示例，不好

```
void f(int* p, int n) // n 为 p[] 中的元素数量
{
    // ...
    p[2] = 7; // 不好：对原生指针采用下标
    // ...
}
```

编译期不会读注释，而如果不读其他代码的话你也无法指导 `p` 是否真的指向了 `n` 个元素。应当代之以 `span`。

示例

```
void g(int* p, int fmt)    // 用格式 fmt 打印 *p
{
    // ... 只使用 *p 和 p[0] ...
}
```

例外

C 风格的字符串是以单个指向以零结尾的字符序列的指针来传递的。

为了表明对这种约定的依赖，应当使用 `zstring` 而不是 `char*`。

注解

当前许多的单元素指针的用法其实都应当用引用。

不过，如果 `nullptr` 是可能的值的话，也许引用就不是合理的替代方案了。

强制实施

- 对并非来自容器、视图或迭代器的指针进行的指针算术（包括 `++`）进行标记。
这条规则对比较老的代码库实施时，可能会产生巨量的误报。
- 对把数组名被传递为单纯的指针进行标记。

R.3: 原生指针 (`T*`) 没有所有权

理由

对此（C++ 标准中和大多数代码中都）没有异议，大多数原生指针都是无所有权的。

我们希望将所有权的指针标示出来，以使得可以可靠和高效地删除由所有权指针所指向的对象。

示例

```
void f()
{
    int* p1 = new int{7};           // 不好：原生指针拥有了所有权
    auto p2 = make_unique<int>(7); // OK: int 被一个唯一指针所拥有
    // ...
}
```

`unique_ptr` 保证对它的对象进行删除（即便是发生异常时也如此），以此保护不发生泄漏。而 `T*` 做不到这点。

示例

```
template<typename T>
class X {
public:
    T* p;    // 不好：不清楚 p 是不是带有所有权
    T* q;    // 不好：不清楚 q 是不是带有所有权
    // ...
};
```

可以通过明确所有权来修正这个问题：

```
template<typename T>
class X2 {
public:
    owner<T*> p; // OK: p 具有所有权
    T* q; // OK: q 没有所有权
    // ...
};
```

例外

最主要的例外就是遗留代码，尤其是那些必须维持可以用 C 编译或者通过 ABI 来建立 C 和 C 风格的 C++ 之间的接口的代码。

亿万行的代码都违反本条规则而让 `T*` 具有所有权的现实是无法被忽略的。

我们由衷希望看到程序变换工具把这些 20 岁以上的“遗留”代码转换成光鲜的现代代码，

我们鼓励这种工具的开发、部署和使用，

我们希望这里的各项指导方针能够有助于这种工具的开发，

而且我们也在这一领域的研发工作中持续作出贡献。

但是，这是需要时间的：“遗留代码”的产生比我们能翻新的老代码还要快，因此这将会花费许多年的时间。

这些代码是无法被全部重写的（即便假定有良好的代码转换软件），尤其不会很快发生。

这个问题是不能简单通过把所有有所有权的指针都转换成 `unique_ptr` 和 `shared_ptr` 来（大规模）解决的，

这部分是因为我们确实需要在基础的资源句柄的实现中一起使用有所有权的“原生指针”和简单的指针。

例如，常见的 `vector` 实现中都有一个有所有权的指针和两个没有所有权的指针。

许多 ABI（以及基本上全部的面向 C 的接口代码）都使用 `T*`，其中不少都是有所有权的。

一些接口是无法简单地用 `owner` 来标记的，因为它们需要维持可以作为 C 来编译

，（这可能是少见的恰当的使用宏的场合，它仅在 C++ 模式中扩展为 `owner`）。

注解

`owner<T*>` 并没有超出 `T*` 的默认语义。使用它可以不改动任何使用方代码，也不会影响 ABI。

它只不过是一项针对程序员和分析工具的提示。

比如说，当 `owner<T*>` 是某个类的成员时，这个类最好提供一个析构函数来 `delete` 它。

示例，不好

返回（原生）指针的做法向调用方暴露了在生存期管理上的不确定性；就是说，谁应该删除其所指向的对象呢？

```
Gadget* make_gadget(int n)
{
    auto p = new Gadget{n};
    // ...
    return p;
}

void caller(int n)
{
    auto p = make_gadget(n); // 要记得 delete p
    // ...
    delete p;
}
```

除了遭受[资源泄漏](#)的问题外，这也带来了一组假性的分配和回收操作，而这其实是不必要的。如果 Gadget 可以廉价地从函数转移出来（就是说，它很小，或者具有高效的移动操作）的话，直接“按值”返回即可（参见[输出返回值](#)）：

```
Gadget make_gadget(int n)
{
    Gadget g{n};
    // ...
    return g;
}
```

注解

这条规则适用于工厂函数。

注解

如果指针语义是必须的（比如说，因为返回类型需要指代类层次中的基类（或接口）），则可以返回“智能指针”。

强制实施

- 【简单】对在并非 `owner<T>` 的原生指针上进行的 `delete` 给出警告。
- 【中等】对一个 `owner<T>` 指针，当并非每个代码路径中都要么进行 `reset` 要么明确 `delete`，则给出警告。
- 【简单】当 `new` 的返回值被赋值给原生指针时，给出警告。
- 【简单】当函数所返回的对象是在函数中所分配的，并且它具有移动构造函数时，给出警告。
建议代之以按值返回。

R.4: 原生引用 (`T&`) 没有所有权

理由

对此（C++ 标准中和大多数代码中都）没有异议，大多数原生引用都是无所有权的。

我们希望将所有者都标示出来，以使得可以可靠和高效地删除由有所有权指针所指向的对象。

示例

```
void f()
{
    int& r = *new int{7}; // 不好：原生的具有所有权的引用
    // ...
    delete &r;           // 不好：违反了有关删除原生指针的规则
}
```

参见：[原生指针的规则](#)

强制实施

参见[原生指针的规则](#)

R.5: 优先采用有作用域的对象，避免不必要的堆分配

理由

有作用域的对象是局部对象、全局对象，或者成员。

它们也意味着在其所在作用域或者所在对象之外无须花费单独的分配和回收成本。

有作用域对象的成员自身也是有作用域的，有作用域对象的构造函数和析构函数负责管理其成员的生存期。

示例

下面的例子效率不佳（因为无必要的分配和回收），在 `...` 中抛出异常和返回也是脆弱的（导致发生泄漏），而且也比较啰嗦：

```
void f(int n)
{
    auto p = new Gadget{n};
    // ...
    delete p;
}
```

可以用局部变量来代替它：

```
void f(int n)
{
    Gadget g{n};
    // ...
}
```

强制实施

- 【中级】如果分配了某个对象，又在函数内的所有路径中都进行了回收，则给出警告。建议它应当被代之以一个局部的栈对象。
- 【简单】当局部的 `unique_pointer` 或 `shared_pointer` 在其生存期结束前未被移动、复制、赋值或 `reset`，则给出警告。

例外：不对指向无界数组的局部 `unique_pointer` 产生这种警告。（见下文）

例外

创建指向堆分配缓冲区的局部 `const unique_ptr<T[]>` 是没问题的，这是一种有效的表现有作用域的动态数组的方法。

示例

一个局部 `const unique_ptr<T[]>` 变量的有效用例：

```

int get_median_value(const std::list<int>& integers)
{
    const auto size = integers.size();

    // OK: declaring a local unique_ptr<T[]>.
    const auto local_buffer = std::make_unique_for_overwrite<int[]>(size);

    std::copy_n(begin(integers), size, local_buffer.get());
    std::nth_element(local_buffer.get(), local_buffer.get() + size/2,
                     local_buffer.get() + size);

    return local_buffer[size/2];
}

```

R.6: 避免非 `const` 的全局变量

参见 [I.2](#)

R.alloc: 分配与回收

R.10: 避免 `malloc()` 和 `free()`

理由

`malloc()` 和 `free()` 并不支持构造和销毁，而且无法同 `new` 和 `delete` 之间进行混用。

示例

```

class Record {
    int id;
    string name;
    // ...
};

void use()
{
    // p1 可能是 nullptr
    // *p1 并未初始化；尤其是，
    // 其中的 string 也还不是个 string，而是一片和 string 大小相同的字节而已
    Record* p1 = static_cast<Record*>(malloc(sizeof(Record)));

    auto p2 = new Record;

    // 如果没有抛出异常的话，*p2 就经过了默认初始化
    auto p3 = new(nothrow) Record;
    // p3 可能为 nullptr；否则 *p3 就经过了默认初始化

    // ...

    delete p1;      // 错误：不能 delete 由 malloc() 分配的对象
    free(p2);      // 错误：不能 free() 由 new 分配的对象
}

```

在某些实现中，`delete` 和 `free()` 可能可以工作，或者也可能引发运行时的错误。

例外

有些应用程序或者代码段是不能接受异常的。
它们的最佳例子就是那些性命攸关的硬实时代码。
但要注意的是，许多对异常的禁止其实是基于某种（不良的）迷信，
或者来源于对没有进行系统性的资源管理的老式代码库的关注（很不幸，但这经常是必须的）。
在这些情况下，应当考虑使用 `nothrow` 版本的 `new`。

强制实施

对 `malloc` 和 `free` 的使用进行标记。

R.11: 避免显式调用 `new` 和 `delete`

理由

由 `new` 所返回的指针应当属于一个资源句柄（它将调用 `delete`）。
若由 `new` 所返回的指针被赋值给普通的裸指针，那么这个对象就可能会泄漏。

注解

大型程序中，裸露的 `delete`（即出现于应用程序代码中，而不是专门进行资源管理的代码中）
很可能是一个 BUG：如果已经有了 N 个 `delete` 的话，怎么确定我们需要的不是 N+1 或者 N-1 个呢？
这种 BUG 可能会潜伏起来：它也许只会在维护过程中才暴露出来。
如果出现了裸露的 `new`，那就可能在别的什么地方需要一个裸露的 `delete`，因而可能也是一个
BUG。

强制实施

【简单】对任何 `new` 和 `delete` 的显式使用都给出警告。建议代之以 `make_unique`。

R.12: 显式资源分配的结果应当立即交给一个管理对象

理由

如果不这样做的话，当发生异常或者返回时就可能造成泄露。

示例，不好

```
void func(const string& name)
{
    FILE* f = fopen(name, "r");           // 打开文件
    vector<char> buf(1024);
    auto _ = finally([f] { fclose(f); }); // 记得要关闭文件
    // ...
}
```

`buf` 的分配可能会失败，并导致文件句柄的泄漏。

示例

```
void func(const string& name)
{
    ifstream f{name}; // 打开文件
    vector<char> buf(1024);
    // ...
}
```

对文件句柄（在 `ifstream` 中）的使用是简单、高效而且安全的。

强制实施

- 将用于初始化指针的显式分配标记出来。（问题：我们能识别出多少直接资源分配呢？）

R.13: 单个表达式语句中至多进行一次显式资源分配

理由

如果在一条语句中进行两次显式资源分配的话就可能发生资源泄漏，这是因为许多的子表达式（包括函数参数）的求值顺序都是未指明的。

示例

```
void fun(shared_ptr<Widget> sp1, shared_ptr<Widget> sp2);
```

可以这样调用 `fun`：

```
// 不好：可能会泄漏
fun(shared_ptr<Widget>(new Widget(a, b)), shared_ptr<Widget>(new Widget(c, d)));
```

这是异常不安全的，因为编译器可能会把两个用以创建函数的两个参数的表达式重新排序。

特别是，编译器是可以交错执行这两个表达式的：

它可能会首先为两个对象都（通过调用 `operator new`）进行内存分配，然后再试图调用二者的 `Widget` 构造函数。

一旦其中一个构造函数调用抛出了异常，那么另一个对象的内存可能永远不会被释放了！

这个微妙的问题其实有一种简单的解决方案：永远不要在一条表达式语句中进行多于一次的显式资源分配。

例如：

```
shared_ptr<Widget> sp1(new Widget(a, b)); // 好多了，但不太干净
fun(sp1, new Widget(c, d));
```

最佳的方案是使用返回所有者对象的工厂函数，而完全避免显式的分配：

```
fun(make_shared<Widget>(a, b), make_shared<Widget>(c, d)); // 最佳
```

如果还没有，请自己编写一个工厂包装。

强制实施

- 将包含多次显式资源分配的表达式标记出来。（问题：我们能识别出多少直接资源分配呢？）

R.14: 避免使用 `[]` 形参，优先使用 `span`

理由

数组会退化为指针，因而丢失其大小信息，并留下了发生范围错误的机会。

使用 `span` 来保留大小信息。

示例

```
void f(int[]);           // 不建议的做法

void f(int*);           // 对多个对象不建议的做法
                        // （指针应当指向单个对象，不要进行下标运算）

void f(gsl::span<int>); // 好，建议的做法
```

强制实施

标记出 `[]` 参数。代之以使用 `span`。

R.15: 总是同时重载相匹配的分配、回收函数对

理由

不然的话就出现不匹配的操作，并导致混乱。

示例

```
class X {
    // ...
    void* operator new(size_t s);
    void operator delete(void*);
    // ...
};
```

注解

如果想要无法进行回收的内存的话，可以将回收操作 `=delete`。

请勿留下它而不进行声明。

强制实施

标记出不完全的操作对。

R.smart: 智能指针

R.20: 用 `unique_ptr` 或 `shared_ptr` 表示所有权

理由

它们可以避免资源泄漏。

示例

考虑：

```
void f()
{
    X* p1 { new X };           // 不好, p1 会泄漏
    auto p4 = make_unique<X>(); // 好, 唯一所有权
    auto p5 = make_shared<X>(); // 好, 共享所有权
}
```

这里（只有）初始化 `p1` 的对象将会泄漏。

强制实施

【简单】如果 `new` 的返回值被赋值给了原生指针，就给出警告。

【简单】如果返回带所有权原始指针的函数的结果被赋值给了原生指针，就给出警告。

R.21: 优先采用 `unique_ptr` 而不是 `shared_ptr`，除非需要共享所有权

理由

`unique_ptr` 概念上要更简单且更可预测（知道它何时会销毁），而且更快（不需要暗中维护引用计数）。

示例，不好

这里并不需要维护一个引用计数。

```
void f()
{
    shared_ptr<Base> base = make_shared<Derived>();
    // 局部范围内使用 base，并未进行复制--引用计数不会超过 1
} // 销毁 base
```

示例

这样更加高效：

```
void f()
{
    unique_ptr<Base> base = make_unique<Derived>();
    // 局部范围内使用 base
} // 销毁 base
```

强制实施

【简单】如果函数所使用的 `shared_pointer` 的对象是函数之内所分配的，而且既不会将这个 `shared_pointer` 返回，也不会将其传递给其他接受 `shared_pointer&` 的函数的话，就给出警告。建议代之以 `unique_ptr`。

R.22: 使用 `make_shared()` 创建 `shared_ptr`

理由

`make_shared` 为构造提供了更精炼的语句。

它也提供了一个机会，通过把 `shared_ptr` 的使用计数和对象相邻放置，来消除为引用计数进行独立的内存分配操作。

示例

考虑：

```
shared_ptr<x> p1 { new x{2} }; // 不好  
auto p = make_shared<x>(2);    // 好
```

`make_shared()` 版本仅提到一次 `x`，因而它通常比显式的 `new` 方式要更简短（而且更快）。

强制实施

【简单】如果 `shared_ptr` 从 `new` 的结果而不是 `make_shared` 进行构造，就给出警告。

R.23: 使用 `make_unique()` 创建 `unique_ptr`

理由

`make_unique` 为构造提供了更精炼的语句。

它也保证了复杂表达式中的异常安全性。

示例

```
unique_ptr<Foo> p {new Foo{7}};      // OK: 不过有重复  
auto q = make_unique<Foo>(7);        // 有改善：并未重复 Foo
```

强制实施

【简单】如果 `unique_ptr` 从 `new` 的结果而不是 `make_unique` 进行构造，就给出警告。

R.24: 使用 `std::weak_ptr` 来打断 `shared_ptr` 的循环引用

理由

`shared_ptr` 是基于引用计数的，而带有循环的结构中的引用计数不可能变为零，因此我们需要一种机制

来打破带有循环的结构。

示例

```
#include <memory>

class bar;

class foo {
public:
    explicit foo(const std::shared_ptr<bar>& forward_reference)
        : forward_reference_(forward_reference)
    { }
private:
    std::shared_ptr<bar> forward_reference_;
};

class bar {
public:
    explicit bar(const std::weak_ptr<foo>& back_reference)
        : back_reference_(back_reference)
    { }
    void do_something()
    {
        if (auto shared_back_reference = back_reference_.lock()) {
            // 使用 *shared_back_reference
        }
    }
private:
    std::weak_ptr<foo> back_reference_;
};
}
```

注解

??? (HS: 许多人都说要“打破循环引用”，不过我觉得“临时性共享所有权”可能更关键。)

???(BS: 打破循环引用是必须达成的目标；临时性共享所有权则是达成的方式。

你也可以仅仅使用另一个 `shared_ptr` 来得到“临时性共享所有权”。)

强制实施

??? 可能无法做到。如果能够静态地检测出循环引用的话，我们就不需要 `weak_ptr` 了。

R.30: 以智能指针为参数，仅用于明确表达生存期语义

参见 [F.7](#)。

R.31: 非 `std` 的智能指针，应当遵循 `std` 的行为模式

理由

下面段落中的规则同样适用于第三方和自定义的其他种类的智能指针，而且对于诊断引发导致了性能和正确性问题的一般性的智能指针错误来说也是非常有帮助的。

你将会期望你所使用的所有智能指针都遵循这些规则。

任何重载了一元 `*` 和 `->` 的类型（无论主模板还是特化）都被当成是智能指针：

- 如果它可以复制，则将其当做一种具有引用计数的 `shared_ptr`。
- 如果它不能复制，则将其当做一种唯一的 `unique_ptr`。

示例，不好

```
// 使用 Boost 的 intrusive_ptr
#include <boost/intrusive_ptr.hpp>
void f(boost::intrusive_ptr<widget> p) // 根据 'sharedptrparam' 规则是错误的
{
    p->foo();
}

// 使用 Microsoft 的 CComPtr
#include <atlbase.h>
void f(CComPtr<widget> p) // 根据 'sharedptrparam' 规则是错误的
{
    p->foo();
}
```

上面两段根据 [sharedptrparam 指导方针](#)来说都是错误的：

`p` 是一个 `shared_pointer`，但其共享性质完全没有被用到，而对其进行按值传递则是一种暗含的劣化；

这两个函数应当仅当它们需要参与 `widget` 的生存期管理时才接受智能指针。否则当可以为 `nullptr` 时它们就应当接受 `widget*`，否则，理想情况下，函数应当接受 `widget&`。

这些智能指针都符合 `shared_pointer` 的概念，因此这些强制实施指导方针的规则可以直接应用，并使得这种一般性的劣化情况暴露出来。

R.32: `unique_ptr<widget>` 参数用以表达函数假定获得 `widget` 的所有权

理由

以这种方式使用 `unique_ptr` 同时说明并强制施加了函数调用时的所有权转移。

示例

```
void sink(unique_ptr<widget>); // 获得这个 widget 的所有权

void uses(widget*); // 仅仅使用了这个 widget
```

示例，不好

```
void thinko(const unique_ptr<widget>&); // 通常不是你想要的
```

强制实施

- 【简单】如果函数以左值引用接受 `unique_pointer<T>` 参数，但并未在至少一个代码路径中向其赋值或者对其调用 `reset()`，则给出警告。建议代之以接受 `T*` 或 `T&`。
- 【简单】〔基础〕如果函数以 `const` 引用接受 `unique_pointer<T>` 参数，则给出警告。建议代之以接受 `const T*` 或 `const T&`。

R.33: `unique_ptr<widget>&` 参数用以表达函数对该 `widget` 重新置位

示例

以这种方式使用 `unique_ptr` 同时说明并强制施加了函数调用时的重新置位语义。

注解

所谓“重新置位 (Reseat) ”的含义是“让指针或智能指针指代某个不同的对象”。

示例

```
void reseat(unique_ptr<widget>&); // “将要”或“可能”重新置位指针
```

示例，不好

```
void thinko(const unique_ptr<widget>&); // 通常不是你想要的
```

强制实施

- 【简单】如果函数以左值引用接受 `Unique_pointer<T>` 参数，但并未在至少一个代码路径中向其赋值或者对其调用 `reset()`，则给出警告。建议代之以接受 `T*` 或 `T&`。
- 【简单】〔基础〕如果函数以 `const` 引用接受 `Unique_pointer<T>` 参数，则给出警告。建议代之以接受 `const T*` 或 `const T&`。

R.34: 用 `shared_ptr<widget>` 参数表达共享所有权

理由

这样做明确了函数的所有权共享语义。

示例，好

```
class WidgetUser
{
public:
    // WidgetUser 将会共享这个 widget 的所有权
    explicit WidgetUser(std::shared_ptr<Widget> w) noexcept:
        m_widget{std::move(w)} {}

    // ...
private:
    std::shared_ptr<Widget> m_widget;
};
```

强制实施

- 【简单】如果函数以左值引用接受 `shared_pointer<T>` 参数，但并未在至少一个代码路径中向其赋值或者对其调用 `reset()`，则给出警告。建议代之以接受 `T*` 或 `T&`。
- 【简单】〔基础〕如果函数按值或者以 `const` 引用接受 `Shared_pointer<T>` 参数，但并未在至少一个代码路径中将其复制或移动给另一个 `shared_pointer`，则给出警告。建议代之以接受 `T*` 或 `T&`。
- 【简单】〔基础〕如果函数以右值引用接受 `shared_pointer<T>` 参数，则给出警告。建议代之以按值传递。

R.35: `shared_ptr<widget>&` 参数用以表达函数可能会对共享的指针重新置位

理由

这样做明确了函数的重新置位语义。

注解

所谓“重新置位 (Reseat) ”的含义是“让引用或智能指针指代某个不同的对象”。

示例，好

```
void Changewidget(std::shared_ptr<widget>& w)
{
    // 这将会改变调用方的 widget
    w = std::make_shared<widget>(widget{});
}
```

强制实施

- 【简单】如果函数以左值引用接受 `shared_pointer<T>` 参数，但并未在至少一个代码路径中向其赋值或者对其调用 `reset()`，则给出警告。建议代之以接受 `T*` 或 `T&`。
- 【简单】〔基础〕如果函数按值或者以 `const` 引用接受 `shared_pointer<T>` 参数，但并未在至少一个代码路径中将其复制或移动给另一个 `shared_pointer`，则给出警告。建议代之以接受 `T*` 或 `T&`。
- 【简单】〔基础〕如果函数以右值引用接受 `shared_pointer<T>` 参数，则给出警告。建议代之以按值传递。

R.36: `const shared_ptr<widget>&` 参数用以表达它可能将保留一个对对象的引用 ???

理由

这样做明确了函数的 ??? 语义。

示例，好

```
void share(shared_ptr<widget>);           // 共享--“将会”保持一个引用计数

void reseat(shared_ptr<widget>&);          // “可能”重新置位指针

void may_share(const shared_ptr<widget>&); // “可能”保持一个引用计数
```

强制实施

- 【简单】如果函数以左值引用接受 `shared_pointer<T>` 参数，但并未在至少一个代码路径中向其赋值或者对其调用 `reset()`，则给出警告。建议代之以接受 `T*` 或 `T&`。
- 【简单】〔基础〕如果函数按值或者以 `const` 引用接受 `shared_pointer<T>` 参数，但并未在至少一个代码路径中将其复制或移动给另一个 `shared_pointer`，则给出警告。建议代之以接受 `T*` 或 `T&`。
- 【简单】〔基础〕如果函数以右值引用接受 `shared_pointer<T>` 参数，则给出警告。建议代之以按值传递。

R.37: 不要把来自某个智能指针别名的指针或引用传递出去

理由

违反这条规则，是导致引用计数的丢失和出现悬挂指针的首要原因。

函数应当优先向其调用链中传递原生指针和引用。

在调用树的顶层，原生指针或引用是从用以保持对象存活的智能指针中获得的。

我们需要确保这个智能指针不会在调用树的下面被疏忽地进行重置或者重新赋值。

注解

为了做到这点，有时候需要获得智能指针的一个局部副本，它可以确保在函数及其调用树的执行期间维持对象存活。

示例

考虑下面的代码：

```
// 全局（静态或堆）对象，或者有别名的局部对象 ...
shared_ptr<widget> g_p = ...;

void f(widget& w)
{
    g();
    use(w); // A
}

void g()
{
    g_p = ...; // 噢，如果这就是这个 widget 的最后一个 shared_ptr 的话，这会销毁这个
    widget
}
```

下面的代码是不应该通过代码评审的：

```
void my_code()
{
    // 不好：传递的是从非局部的智能指针中获得的指针或引用
    //           而这可能会在 f 或其调用的函数中的某处被不经意地重置掉
    f(*g_p);

    // 不好：原因相同，只不过将其作为“this”指针传递
    g_p->func();
}
```

修正很简单——获取该指针的一个局部副本，为调用树“保持一个引用计数”：

```

void my_code()
{
    // 很廉价：一次增量就搞定了整个函数以及下面的所有调用树
    auto pin = g_p;

    // 好：传递的是从局部的无别名智能指针中获得的指针或引用
    f(*pin);

    // 好：原因相同
    pin->func();
}

```

强制实施

- 【简单】如果从非局部或局部但潜在具有别名的智能指针变量（`unique_pointer` 或 `shared_pointer`）中所获取的指针或引用，被用于进行函数调用，则给出警告。如果智能指针是一个 `shared_pointer`，则建议代之以获取该智能指针的一个局部副本并从中获取指针或引用。

ES: 表达式和语句

表达式和语句是用以表达动作和计算的最底层也是最直接的方式。局部作用域中的声明也是语句。

有关命名、注释和缩进的规则，参见 [NL: 命名与代码布局](#)。

一般规则：

- [ES.1: 优先采用标准库而不是其他的库或者“手工自制代码”](#)
- [ES.2: 优先采用适当的抽象而不是直接使用语言功能特性](#)
- [ES.3: 避免重复 \(DRY\)，避免冗余代码](#)

声明的规则：

- [ES.5: 保持作用域尽量小](#)
- [ES.6: 在 for 语句的初始化式和条件中声明名字以限制其作用域](#)
- [ES.7: 保持常用的和局部的名字尽量简短，而让非常用的和非局部的名字较长](#)
- [ES.8: 避免使用看起来相似的名字](#)
- [ES.9: 避免 ALL_CAPS 式的名字](#)
- [ES.10: 每条声明中（仅）声明一个名字](#)
- [ES.11: 使用 `auto` 来避免类型名字的多余重复](#)
- [ES.12: 不要在嵌套作用域中重用名字](#)
- [ES.20: 坚持为对象进行初始化](#)
- [ES.21: 不要在确实需要使用变量（或常量）之前就引入它](#)
- [ES.22: 要等到获得了用以初始化变量的值之后才声明变量](#)
- [ES.23: 优先使用 `{}` 初始化式语法](#)
- [ES.24: 用 `unique_ptr<T>` 来保存指针](#)
- [ES.25: 应当将对象声明为 `const` 或 `constexpr`，除非后面需要修改其值](#)
- [ES.26: 不要用一个变量来达成两个不相关的目的](#)
- [ES.27: 使用 `std::array` 或 `stack_array` 作为栈上的数组](#)
- [ES.28: 为复杂的初始化（尤其是 `const` 变量）使用 `lambda`](#)
- [ES.30: 不要用宏来操纵程序文本](#)
- [ES.31: 不要用宏来作为常量或“函数”](#)
- [ES.32: 对所有的宏名采用 `ALL_CAPS` 命名方式](#)
- [ES.33: 如果必须使用宏的话，请为之提供唯一的名字](#)

- [ES.34: 不要定义 \(C 风格的\) 变参函数](#)

表达式的规则:

- [ES.40: 避免复杂的表达式](#)
- [ES.41: 对运算符优先级不保证时应使用括号](#)
- [ES.42: 保持单纯直接的指针使用方式](#)
- [ES.43: 避免带有未定义的求值顺序的表达式](#)
- [ES.44: 不要对函数参数求值顺序有依赖](#)
- [ES.45: 避免“魔法常量”，采用符号化常量](#)
- [ES.46: 避免窄化转换](#)
- [ES.47: 使用 `nullptr` 而不是 `0` 或 `NULL`](#)
- [ES.48: 避免强制转换](#)
- [ES.49: 当必须使用强制转换时，使用具名的强制转换](#)
- [ES.50: 不要强制掉 `const`](#)
- [ES.55: 避免发生对范围检查的需要](#)
- [ES.56: 仅在确实需要明确移动某个对象到别的作用域时才使用 `std::move\(\)`](#)
- [ES.60: 避免在资源管理函数之外使用 `new` 和 `delete`](#)
- [ES.61: 用 `delete\[\]` 删除数组，用 `delete` 删除非数组对象](#)
- [ES.62: 不要在不同的数组之间进行指针比较](#)
- [ES.63: 不要产生切片](#)
- [ES.64: 使用 `T{e}` 写法来进行构造](#)
- [ES.65: 不要解引用无效指针](#)

语句的规则:

- [ES.70: 面临选择时，优先采用 `switch` 语句而不是 `if` 语句](#)
- [ES.71: 面临选择时，优先采用范围式 `for` 语句而不是普通 `for` 语句](#)
- [ES.72: 当存在显然的循环变量时，优先采用 `for` 语句而不是 `while` 语句](#)
- [ES.73: 当没有显然的循环变量时，优先采用 `while` 语句而不是 `for` 语句](#)
- [ES.74: 优先在 `for` 语句的初始化部分中声明循环变量](#)
- [ES.75: 避免使用 `do` 语句](#)
- [ES.76: 避免 `goto`](#)
- [ES.77: 尽量减少循环中使用的 `break` 和 `continue`](#)
- [ES.78: 不要依靠 `switch` 语句中的隐含直落行为](#)
- [ES.79: `default` \(仅\) 用于处理一般情况](#)
- [ES.84: 不要试图声明没有名字的局部变量](#)
- [ES.85: 让空语句显著可见](#)
- [ES.86: 避免在原生的 `for` 循环中修改循环控制变量](#)
- [ES.87: 请勿在条件下添加多余的 `==` 或 `!=`](#)

算术规则:

- [ES.100: 不要进行有符号和无符号混合运算](#)
- [ES.101: 使用无符号类型进行位操作](#)
- [ES.102: 使用有符号类型进行算术运算](#)
- [ES.103: 避免上溢出](#)
- [ES.104: 避免下溢出](#)
- [ES.105: 避免除整数零](#)
- [ES.106: 不要试图用 `unsigned` 来防止负数值](#)
- [ES.107: 不要对下标使用 `unsigned`，优先使用 `gsl::index`](#)

ES.1: 优先采用标准库而不是其他的库或者“手工自制代码”

理由

使用程序库的代码要远比直接使用语言功能特性的代码易于编写，更为简短，更倾向于更高的抽象层次，而且程序库代码假定已经经过测试。

ISO C++ 标准库是最广为了解而且经过最好测试的程序库之一。

它是任何 C++ 实现的一部分。

示例

```
auto sum = accumulate(begin(a), end(a), 0.0); // 好
```

`accumulate` 的范围版本就更好了：

```
auto sum = accumulate(v, 0.0); // 更好
```

但请不要手工编写众所周知的算法：

```
int max = v.size(); // 不好：啰嗦，目的不清晰
double sum = 0.0;
for (int i = 0; i < max; ++i)
    sum = sum + v[i];
```

例外

标准库的很大部分都依赖于动态分配（自由存储）。这些部分，尤其是容器但并不包括算法，并不适用于某些硬实时和嵌入式的应用的。在这些情况下，请考虑提供或使用类似的设施，比如说某个标准库风格的采用池分配器的容器实现。

强制实施

不容易。??? 寻找混乱的循环，嵌套的循环，长函数，函数调用的缺失，缺乏使用内建类型。圈复杂度？

ES.2: 优先采用适当的抽象而不是直接使用语言功能特性

理由

“适当的抽象”（比如程序库或者类），更加贴近应用的概念而不是语言概念，将带来更简洁的代码，而且更容易进行测试。

示例

```
vector<string> read1(istream& is) // 好
{
    vector<string> res;
    for (string s; is >> s;)
        res.push_back(s);
    return res;
}
```

与之近乎等价的更传统且更低层的代码，更长、更混乱、更难编写正确，而且很可能更慢：

```

char** read2(istream& is, int maxelem, int maxstring, int* nread) // 不好: 嘟嗦而且不完整
{
    auto res = new char*[maxelem];
    int elemcount = 0;
    while (is && elemcount < maxelem) {
        auto s = new char[maxstring];
        is.read(s, maxstring);
        res[elemcount++] = s;
    }
    *nread = elemcount;
    return res;
}

```

一旦添加了溢出检查和错误处理，代码就变得相当混乱了，而且还有个要记得 `delete` 其所返回的指针以及数组所包含的 C 风格字符串的问题。

强制实施

不容易。??? 寻找混乱的循环，嵌套的循环，长函数，函数调用的缺失，缺乏使用内建类型。圈复杂度？

ES.3: 避免重复 (DRY) , 避免冗余代码

重复或者冗余的代码会干扰编码意图，导致对逻辑的理解变得困难，并使维护变得困难，以及一些其他问题。它经常出现于拷贝粘贴式编程中。

只要合适就使用标准算法，而不是编写自己的实现。

另请参见: [SL.1](#), [ES.11](#)

示例

```

void func(bool flag) // 不好, 重复代码
{
    if (flag) {
        x();
        y();
    }
    else {
        x();
        z();
    }
}

void func(bool flag) // 更好, 没有重复代码
{
    x();

    if (flag)
        y();
    else
        z();
}

```

强制实施

- 采用静态分析器。它至少会找出一些重复的代码构造。
- 代码评审

ES.dcl: 声明

声明也是语句。一条声明向一个作用域中引入一个名字，并可能导致对一个具名对象进行构造。

ES.5: 保持作用域尽量小

理由

可读性。最小化资源持有率。避免值的意外误用。

其他形式: 不要把名字在不必要的大作用域中进行声明。

示例

```
void use()
{
    int i;      // 不好：循环之后并不需要访问 i
    for (i = 0; i < 20; ++i) { /* ... */ }
    // 此处不存在对 i 的有意使用
    for (int i = 0; i < 20; ++i) { /* ... */ } // 好：i 局部于 for 循环

    if (auto pc = dynamic_cast<Circle*>(ps)) { // 好：pc 局部于 if 语句
        // ... 处理 Circle ...
    }
    else {
        // ... 处理错误 ...
    }
}
```

示例，不好

```
void use(const string& name)
{
    string fn = name + ".txt";
    ifstream is {fn};
    Record r;
    is >> r;
    // ... 200 行代码，都不存在使用 fn 或 is 的意图 ...
}
```

大多数测量都会称这个函数太长了，但其关键点是 `fn` 所使用的资源和 `is` 所持有的文件句柄所持有的时间比其所需长太多了，而且可能在函数后面意外地出现对 `is` 和 `fn` 的使用。
这种情况下，把读取操作重构出来可能是一个好主意：

```
Record load_record(const string& name)
{
    string fn = name + ".txt";
    ifstream is {fn};
    Record r;
    is >> r;
```

```
    return r;
}

void use(const string& name)
{
    Record r = load_record(name);
    // ... 200 行代码 ...
}
```

强制实施

- 对声明于循环之外，且在循环之后不再使用的循环变量进行标记。
- 当诸如文件句柄和锁这样的昂贵资源超过 N 行未被使用时进行标记（对某个适当的 N）。

ES.6: 在 for 语句的初始化式和条件中声明名字以限制其作用域

理由

可读性。

将循环变量的可见性限制到循环范围内。

避免在循环之后将循环变量用于其他目的。

最小化资源持有率。

理由

```
void use()
{
    for (string s; cin >> s;)
        v.push_back(s);

    for (int i = 0; i < 20; ++i) {    // 好: i 局部于 for 循环
        // ...
    }

    if (auto pc = dynamic_cast<Circle*>(ps)) {    // 好: pc 局部于 if 语句
        // ... 处理 Circle ...
    }
    else {
        // ... 处理错误 ...
    }
}
```

示例，请勿如此

```
int j;                                // 不好: j 在循环之外可见
for (j = 0; j < 100; ++j) {
    // ...
}
// j 在此处仍可见但并不需要
```

另请参见: [不要用一个变量来达成两个不相关的目的](#)

强制实施

- 若在 `for` 语句中修改的变量是在循环之外声明的，但并未在循环之外使用，则给出警告。
- 【困难】对声明与循环之前，且在循环之后用于某个无关用途的循环变量进行标记。

讨论：限制循环变量的作用域到循环体还相当有助于代码优化。认识到引入的变量仅在循环体中可以访问，

允许进行诸如外提 (hoisting)，强度消减，循环不变式代码外提等等优化。

C++17 和 C++20 示例

注：C++17 和 C++20 还增加了 `if`、`switch` 和范围式 `for` 的初始化式语句。以下代码要求支持 C++17 和 C++20。

```
map<int, string> mymap;

if (auto result = mymap.insert(value); result.second) {
    // 本代码块中，插入成功且 result 有效
    use(result.first); // ok
    // ...
} // result 在此处销毁
```

C++17 和 C++20 强制实施（当使用 C++17 或 C++20 编译器时）

- 选择/循环变量，若其在选择或循环体之前声明而在其之后不再使用，则对其进行标记
- 【困难】选择/循环变量，若其在选择或循环体之前声明而在其之后用于某个无关目的，则对其进行标记

ES.7: 保持常用的和局部的名字尽量简短，而让非常用的和非局部的名字较长

理由

可读性。减低在无关的非局部名字之间发生冲突的机会。

示例

符合管理的简短的局部名字能够增强可读性：

```
template<typename T> // 好
void print(ostream& os, const vector<T>& v)
{
    for (gs1::index i = 0; i < v.size(); ++i)
        os << v[i] << '\n';
}
```

索引根据惯例称为 `i`，而这个泛型函数中不存在有关这个 `vector` 的含义的提示，因此和其他名字一样，`v` 也没问题。与之相比，

```
template<typename Element_type> // 不好：啰嗦，难于阅读
void print(ostream& target_stream, const vector<Element_type>& current_vector)
{
    for (gsl::index current_element_index = 0;
        current_element_index < current_vector.size();
        ++current_element_index
    )
        target_stream << current_vector[current_element_index] << '\n';
}
```

当然，这是一个讽刺，但我们还见过更糟糕的。

示例

不合惯例而简短的非局部名字则会搞乱代码：

```
void use1(const string& s)
{
    // ...
    tt(s); // 不好：tt() 是什么？
    // ...
}
```

更好的做法是，为非局部实体提供可读的名字：

```
void use1(const string& s)
{
    // ...
    trim_tail(s); // 好多了
    // ...
}
```

这样的话，有可能读者指导 `trim_tail` 是什么意思，而且读者可以找一下它并回忆起来。

示例，不好

大型函数的参数的名字实际上可当作是非局部的，因此应当有意义：

```
void complicated_algorithm(vector<Record>& vr, const vector<int>& vi, map<string,
int>& out)
// 根据 vi 中的索引，从 vr 中读取事件（并标记所用的 Record），
// 向 out 中放置（名字，索引）对
{
    // ... 500 行的代码，使用 vr, vi, 和 out ...
}
```

我们建议保持函数短小，但这条规则并不受到普遍坚持，而命名应当能反映这一点。

强制实施

检查局部和非局部的名字的长度。同时考虑函数的长度。

ES.8: 避免使用看起来相似的名字

理由

代码清晰性和可读性。太过相似的名字会拖慢理解过程并增加出错的可能性。

示例, 不好

```
if (readable(i1 + l1 + o1 + o1 + o0 + o1 + o1 + i0 + l0)) surprise();
```

示例, 不好

不要在同一个作用域中用和类型相同的名字声明一个非类型实体。这样做将消除为区分它们所需的关键字 `struct` 或 `enum` 等。这同样消除了一种错误来源, 因为 `struct x` 在对 `x` 的查找失败时会隐含地声明新的 `x`。

```
struct foo { int n; };
struct foo foo();           // 不好, foo 在作用域中已经是一个类型了
struct foo x = foo();      // 需要进行区分
```

例外

很古老的头文件中可能会用在相同作用域中用同一个名字同时声明非类型实体和类型。

强制实施

- 用一个已知的易混淆字母和数字组合的列表来对名字进行检查。
- 当变量、函数或枚举符的声明隐藏了在相同作用域中所声明的类或枚举时, 给出警告。

ES.9: 避免 ALL_CAPS 式的名字

理由

这样的名字通常是用于宏的。因此, `ALL_CAPS` 这样的名字可能遭受意外的宏替换。

示例

```
// 某个头文件中:
#define NE !=

// 某个别的头文件中的别处:
enum Coord { N, NE, NW, S, SE, SW, E, W };

// 某个糟糕程序员的 .cpp 中的第三处:
switch (direction) {
case N:
    // ...
case NE:
    // ...
// ...
}
```

注解

不要仅仅因为常量曾经是宏，就使用 ALL_CAPS 作为常量。

强制实施

对所有的 ALL CAPS 进行标记。对于老代码，则接受宏名字的 ALL CAPS 而标记所有的非 ALL-CAPS 的宏名字。

ES.10: 每条声明中（仅）声明一个名字

理由

每行一条声明的做法增加可读性并可避免与 C++ 的文法相关的错误。这样做也为更具描述性的行尾注释留下了空间。

示例，不好

```
char *p, c, a[7], *pp[7], **aa[10]; // 讨厌!
```

例外

函数声明中可以包含多个函数参数声明。

例外

结构化绑定 (C++17) 就是专门设计用于引入多个变量的：

```
auto [iter, inserted] = m.insert_or_assign(k, val);
if (inserted) { /* 已插入新条目 */ }
```

示例

```
template<class InputIterator, class Predicate>
bool any_of(InputIterator first, InputIterator last, Predicate pred);
```

用 concept 则更佳：

```
bool any_of(input_iterator auto first, input_iterator auto last, predicate auto pred);
```

示例

```
double scalbn(double x, int n); // OK: x * pow(FLOAT_RADIX, n); FLOAT_RADIX 通常为 2
```

或者：

```
double scalbn(          // 有改善: x * pow(FLOAT_RADIX, n); FLOAT_RADIX 通常为 2
    double x,           // 基数
    int n               // 指数
);
```

或者：

```
// 有改善: base * powFLT_RADIX, exponent); FLT_RADIX 通常为 2
double scalbn(double base, int exponent);
```

示例

```
int a = 10, b = 11, c = 12, d, e = 14, f = 15;
```

在较长的声明符列表中，很容易忽视某个未能初始化的变量。

强制实施

对具有多个声明符的变量或常量的声明式（比如 `int* p, q;`）进行标记。

ES.11: 使用 `auto` 来避免类型名字的多余重复

理由

- 单纯的重复既麻烦又易错。
- 当使用 `auto` 时，所声明的实体的名字是处于声明的固定位置的，这增加了可读性。
- 函数模板声明的返回类型可能是某个成员类型。

示例

考虑：

```
auto p = v.begin();           // vector<DataRecord>::iterator
auto z1 = v[3];               // 产生 DataRecord 的副本
auto& z2 = v[3];             // 避免复制
const auto& z3 = v[3];       // const 并避免复制
auto h = t.future();
auto q = make_unique<int[]>(s);
auto f = [](int x) { return x + 10; };
```

以上都避免了写下冗长又难记的类型，它们是编译器已知的，但程序员则可能会搞错。

示例

```
template<class T>
auto Container<T>::first() -> Iterator; // Container<T>::Iterator
```

例外

当使用初始化式列表，而所需要的确切类型是已知的，同时某个初始化式可能需要转换时，应当避免使用 `auto`。

示例

```
auto lst = { 1, 2, 3 };    // lst 是一个 initializer_list
auto x{1};    // x 是一个 int (C++17; 在 C++11 中则为 initializer_list)
```

注解

C++20 的情况是，我们可以（而且应该）使用概念来更加明确地说明所推断的类型：

```
// ...
forward_iterator auto p = algo(x, y, z);
```

示例 (C++17)

```
std::set<int> values;
// ...
auto [ position, newly_inserted ] = values.insert(5); // 展开 std::pair 的成员
```

强制实施

对声明中多余的类型名字进行标记。

ES.12: 不要在嵌套作用域中重用名字

理由

这样很容易把所用的是哪个变量搞混。
会造成维护问题。

示例，不好

```
int d = 0;
// ...
if (cond) {
    // ...
    d = 9;
    // ...
}
else {
    // ...
    int d = 7;
    // ...
    d = value_to_be_returned;
    // ...
}

return d;
```

这是个大型的 `if` 语句，很容易忽视在内部作用域中所引入的新的 `d`。

这是一种已知的 BUG 来源。

这种在内部作用域中的名字重用有时候被称为“遮蔽”。

注解

当函数变得过大和过于复杂时，遮蔽是一种主要的问题。

示例

语言不允许在函数的最外层块中遮蔽函数参数：

```
void f(int x)
{
    int x = 4; // 错误：重用函数参数的名字

    if (x) {
        int x = 7; // 允许，但不好
        // ...
    }
}
```

示例，不好

把成员名重用为局部变量也会造成问题：

```
struct S {
    int m;
    void f(int x);
};

void S::f(int x)
{
    m = 7; // 对成员赋值
    if (x) {
        int m = 9;
        // ...
        m = 99; // 对局部变量赋值
        // ...
    }
}
```

例外

我们经常在派生类中重用基类中的函数名：

```
struct B {
    void f(int);
};

struct D : B {
    void f(double);
    using B::f;
};
```

这样做是易错的。

例如，要是忘了 using 声明式的话，`d.f(1)` 的调用就不会找到 `int` 版本的 `f`。

??? 我们需要为类层次中的遮蔽/隐藏给出专门的规则吗？

强制实施

- 对嵌套局部作用域中的名字重用进行标记。
- 对成员函数中将成员名重用为局部变量进行标记。
- 对把全局名字重用为局部变量或成员的名字进行标记。
- 对在派生类中重用（除函数名之外的）基类成员名进行标记。

ES.20: 坚持为对象进行初始化

理由

避免发生“设值前使用”的错误及其所关联的未定义行为。

避免由复杂的初始化的理解所带来的问题。

简化重构。

示例

```
void use(int arg)
{
    int i; // 不好：未初始化的变量
    // ...
    i = 7; // 初始化 i
}
```

错了，`i = 7` 并不是 `i` 的初始化；它是向其赋值。而且 `i` 也可能在 `...` 的部分中被读取。更好的做法是：

```
void use(int arg) // OK
{
    int i = 7; // OK：初始化
    string s; // OK：默认初始化
    // ...
}
```

注释

我们有意让总是进行初始化规则比对象在使用前必须设值的语言规则更强。

后者是较为宽松的规则，虽然能够识别出技术上的 BUG，不过：

- 它会导致较不可读的代码，
- 它鼓励人们在比所需更大的作用域中声明名字，
- 它会导致较难于阅读的代码，
- 它会因为鼓励复杂的代码而导致出现逻辑 BUG，
- 它会妨碍进行重构。

而总是进行初始化规则则是以提升可维护性为目标的一条风格规则，同样也是保护避免出现“设值前使用”错误的规则。

示例

这个例子经常被当成是用来展示需要更宽松的初始化规则的例子。

```
widget i;      // "widget" 是一个初始化操作昂贵的类型，可能是一种大型 POD
widget j;

if (cond) { // 不好： i 和 j 进行了“延迟”初始化
    i = f1();
    j = f2();
}
else {
    i = f3();
    j = f4();
}
```

这段代码是无法简单重写为用初始化式来对 `i` 和 `j` 进行初始化的。

注意，对于带有默认构造函数的类型来说，试图延后初始化只会导致变为一次默认初始化之后跟着一次赋值的做法。

这种例子的一种更加流行的理由是“效率”，不过可以检查出是否出现“设置前使用”错误的编译器，同样可以消除任何多余的双重初始化。

假定 `i` 和 `j` 之间存在某种逻辑关联，则这种关联可能应当在函数中予以表达：

```
pair<widget, widget> make_related_widgets(bool x)
{
    return (x) ? {f1(), f2()} : {f3(), f4()};
}

auto [i, j] = make_related_widgets(cond); // C++17
```

如果除此之外 `make_related_widgets` 函数是多余的，

可以使用 lambda [ES.28](#) 来消除之：

```
auto [i, j] = [x] { return (x) ? pair{f1(), f2()} : pair{f3(), f4()} }(); // C++17
```

用一个值代表 `uninitialized` 只是一种问题的症状，而不是一种解决方案：

```
widget i = uninit; // 不好
widget j = uninit;

// ...
use(i);           // 可能发生设值前使用
// ...

if (cond) {       // 不好： i 和 j 进行了“延迟”初始化
    i = f1();
    j = f2();
}
else {
    i = f3();
    j = f4();
}
```

这样的话编译器甚至无法再简单地检测出“设值前使用”。而且我们也在 widget 的状态空间中引入了复杂性：哪些操作对 `uninit` 的 widget 是有效的，哪些不是？

注解

几十年来，精明的程序员中都流行进行复杂的初始化。

这样做也是一种错误和复杂性的主要来源。

而许多这样的错误都是在最初实现之后的多年之后的维护过程中所引入的。

示例

本条规则涵盖成员变量。

```
class X {
public:
    X(int i, int ci) : m2{i}, cm2{ci} {}
    // ...

private:
    int m1 = 7;
    int m2;
    int m3;

    const int cm1 = 7;
    const int cm2;
    const int cm3;
};
```

编译器能够标记 `cm3` 为未初始化（因其为 `const`），但它无法发觉 `m3` 缺少初始化。

通常来说，以很少不恰当的成员初始化来消除错误，要比缺乏初始化更有价值，

而且优化器是可以消除冗余的初始化的（比如紧跟在赋值之前的初始化）。

例外

当声明一个即将从输入进行初始化的对象时，其初始化就可能导致发生双重初始化。

不过，应当注意这也可能造成输入之后留下未初始化的数据——而这已经是一种错误和安全攻击的重大来源：

```
constexpr int max = 8 * 1024;
int buf[max];           // OK, 但是可疑: 未初始化
f.read(buf, max);
```

由于数组和 `std::array` 的有所限制的初始化规则，它们提供了对于需要这种例外的大多数有意义的例子。

某些情况下，这个数组进行初始化的成本可能是显著的。

但是，这样的例子确实倾向于留下可访问到的未初始化变量，因而应当严肃对待它们。

```
constexpr int max = 8 * 1024;
int buf[max] = {};      // 某些情况下更好
f.read(buf, max);
```

如果可行的话，应当用某个已知不会溢出的库函数。例如：

```
string s;    // s 默认初始化为 ""
cin >> s;    // s 进行扩充以持有字符串
```

不要把用于输入操作的简单变量作为本条规则的例外：

```
int i;    // 不好
// ...
cin >> i;
```

在并不罕见的情况下，当输入目标和输入操作分开（其实不应该）时，就带来了发生“设值前使用”的可能性。

```
int i2 = 0;    // 更好，假设 0 是 i2 可接受的值
// ...
cin >> i2;
```

优秀的优化器应当能够识别输入操作并消除这种多余的操作。

注解

有时候，可以用 lambda 作为初始化式以避免未初始化变量：

```
error_code ec;
value v = [&] {
    auto p = get_value();    // get_value() 返回 pair<error_code, value>
    ec = p.first;
    return p.second;
}();
```

还可以是：

```
value v = [] {
    auto p = get_value();    // get_value() 返回 pair<error_code, value>
    if (p.first) throw Bad_value{p.first};
    return p.second;
}();
```

参见: [ES.28](#)

强制实施

- 标记出每个未初始化的变量。
不要对具有默认构造函数的自定义类型的变量进行标记。
- 检查未初始化的缓冲区是否在声明后立即进行了写入。
将未初始化变量作为一个非 `const` 的引用参数进行传递可以被假定为向变量进行的写入。

ES.21: 不要在确实需要使用变量（或常量）之前就引入它

理由

可读性。限制变量可以被使用的范围。

示例

```
int x = 7;  
// ... 这里没有对 x 的使用 ...  
++x;
```

强制实施

对离其首次使用很远的声明进行标记。

ES.22: 要等到获得了用以初始化变量的值之后才声明变量

理由

可读性。限制变量可以被使用的范围。避免“设值前使用”的风险。初始化通常比赋值更加高效。

示例，不好

```
string s;  
// ... 此处没有 s 的使用 ...  
s = "what a waste";
```

示例，不好

```
SomeLargeType var; // 很难读的驼峰变量  
  
if (cond) // 某个不简单的条件  
    Set(&var);  
else if (cond2 || !cond3) {  
    var = Set2(3.14);  
}  
else {  
    var = 0;  
    for (auto& e : something)  
        var += e;  
}  
  
// 使用 var; 可以仅通过控制流而静态地保证这并不会过早进行
```

如果 `SomeLargeType` 的默认初始化并非过于昂贵的话就没什么问题。

不过，程序员可能十分想知道是否所有的穿过这个条件迷宫的路径都已经被覆盖到了。

如果没有的话，就存在一个“设值前使用”的 BUG。这是维护工作的一个陷阱。

对于具有必要复杂性的初始化式，也包括 `const` 变量的初始化式，应当考虑使用 `lambda` 来表达它；参见 [ES.28](#)。

强制实施

- 如果具有默认初始化的声明在其首次被读取前就进行赋值，则对其进行标记。
- 对于任何在未初始化变量之后且在其使用之前进行的复杂计算进行标记。

ES.23: 优先使用 `{}` 初始化式语法

理由

优先使用 `{}`。`{}` 初始化的规则比其他形式的初始化更简单，更通用，更少歧义，而且更安全。

仅当你确定不存在窄化转换时才可使用 `=`。对于内建算术类型，`=` 仅和 `auto` 一起使用。

避免 `()` 初始化，它会导致解析中的歧义。

示例

```
int x {f(99)};  
int y = x;  
vector<int> v = {1, 2, 3, 4, 5, 6};
```

例外

对于容器来说，存在用 `{...}` 给出元素列表而用 `(...)` 给出大小的传统做法：

```
vector<int> v1(10);      // vector 有 10 个具有默认值 0 的元素  
vector<int> v2{10};      // vector 有 1 个值为 10 的元素  
  
vector<int> v3(1, 2);    // vector 有 1 个值为 2 的元素  
vector<int> v4{1, 2};    // vector 有 2 个值为 1 和 2 的元素
```

注解

`{}` 初始化式不允许进行窄化转换（这点通常都很不错），并允许使用显式构造函数（这没有问题，我们的意图就是初始化一个新变量）。

示例

```
int x {7.9};    // 错误：发生窄化  
int y = 7.9;    // OK：y 变为 7. 希望编译器给出了警告消息  
int z = gsl::narrow_cast<int>(7.9); // OK：这个正是你想要的
```

注解

`{}` 初始化可以用于几乎所有的初始化；而其他的初始化则不行：

```
auto p = new vector<int> {1, 2, 3, 4, 5};    // 初始化 vector  
D::D(int a, int b) :m{a, b} {} // 成员初始化式（比如说 m 可能是 pair）  
// ...  
};  
X var {}; // 初始化 var 为空  
struct S {  
    int m {7}; // 成员的默认初始化  
    // ...  
};
```

由于这个原因，以 `{}` 进行初始化通常被称为“统一初始化”，
(但很可惜存在少数不符合规则的例外)。

注解

对以 `auto` 声明的变量用单个值进行的初始化，比如 `{v}`，直到 C++17 之前都还具有令人意外的含义。

C++17 的规则多少会少些意外：

```
auto x1 {7};           // x1 是一个值为 7 的 int
auto x2 = {7};          // x2 是一个具有一个元素 7 的 initializer_list<int>

auto x11 {7, 8};        // 错误：两个初始化式
auto x22 = {7, 8};      // x22 是一个具有元素 7 和 8 的 initializer_list<int>
```

如果确实需要一个 `initializer_list<T>` 的话，可以使用 `={...}`：

```
auto fib10 = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55};    // fib10 是一个列表
```

注解

`={}` 进行的是复制初始化，而 `{}` 则进行直接初始化。

与在复制初始化和直接初始化自身之间存在的区别类似的是，这里也可能带来一些意外情况。

`{}` 可以接受 `explicit` 构造函数；而 `={}` 则不能。例如：

```
struct z { explicit z() {} };

z z1{};      // OK: 直接初始化，使用的是 explicit 构造函数
z z2 = {};// 错误：复制初始化，不能使用 explicit 构造函数
```

除非特别要求禁止使用显式构造函数，否则都应当使用普通的 `{}` 初始化。

示例

```
template<typename T>
void f()
{
    T x1(1);    // T 以 1 进行初始化
    T x0();     // 不好：函数声明（一般都是一个错误）

    T y1 {1};   // T 以 1 进行初始化
    T y0 {} ;   // 默认初始化 T
    // ...
}
```

参见：[讨论](#)

强制实施

- 当使用 `=` 初始化算术类型并发生窄化转换时予以标记。
- 当使用 `()` 初始化语法但实际上声明式时予以标记。（许多编译器已经可就此给出警告。）

ES.24: 用 `unique_ptr<T>` 来保存指针

理由

使用 `std::unique_ptr` 是避免泄漏的最简单方法。它是可靠的，它利用类型系统完成验证所有权安全性的大部分工作，它增加可读性，而且它没有或几乎没有运行时成本。

示例

```
void use(bool leak)
{
    auto p1 = make_unique<int>(7);    // OK
    int* p2 = new int{7};             // 不好：可能泄漏
    // ... 未对 p2 赋值 ...
    if (leak) return;
    // ... 未对 p2 赋值 ...
    vector<int> v(7);
    v.at(7) = 0;                   // 抛出异常
    delete p2;                     // 避免泄漏已太晚了
    // ...
}
```

当 `leak == true` 时，`p2` 所指向的对象就会泄漏，而 `p1` 所指向的对象则不会。当 `at()` 抛出异常时也是同样的情况。两种情况下，都没能到达 `delete p2` 语句。

强制实施

寻找作为这些函数的目标的原生指针：`new`，`malloc()`，或者可能返回这类指针的函数。

ES.25: 应当将对象声明为 `const` 或 `constexpr`，除非后面需要修改其值

理由

这样的话你就不会误改掉这个值。而且这种方式可能会给编译器的带来优化机会。

示例

```
void f(int n)
{
    const int bufmax = 2 * n + 2; // 好：无法意外改掉 bufmax
    int xmax = n;               // 可疑：xmax 是不是会改掉？
    // ...
}
```

强制实施

查看变量是不是真的被改动过，若并非如此就进行标记。
不幸的是，也许不可能检测出某个非 `const` 是不是有意要改动，还是仅仅是没被改动而已。

ES.26: 不要用一个变量来达成两个不相关的目的

理由

可读性和安全性。

示例，不好

```
void use()
{
    int i;
    for (i = 0; i < 20; ++i) { /* ... */ }
    for (i = 0; i < 200; ++i) { /* ... */ } // 不好: i 重复使用了
}
```

+##### 注解

也许你想把一个缓冲区当做暂存器来重复使用以作为一种优化措施，但即便如此也请尽可能限定该变量的作用域，还要当心不要导致由于遗留在重用的缓冲区中的数据而引发的 BUG，这是安全性 BUG 的一种常见来源。

```
void write_to_file()
{
    std::string buffer;           // 以避免每次循环重复中的重新分配
    for (auto& o : objects) {
        // 第一部分工作。
        generate_first_string(buffer, o);
        write_to_file(buffer);

        // 第二部分工作。
        generate_second_string(buffer, o);
        write_to_file(buffer);

        // 等等...
    }
}
```

强制实施

标记被重复使用的变量。

ES.27: 使用 `std::array` 或 `stack_array` 作为栈上的数组

理由

它们是可读的，而且不会隐式转换为指针。
它们不会和内建数组的非标准扩展相混淆。

示例，不好

```
const int n = 7;
int m = 9;

void f()
{
    int a1[n];
    int a2[m]; // 错误：并非 ISO C++
    // ...
}
```

注解

`a1` 的定义是合法的 C++ 而且一直都是。

存在大量的这类代码。

不过它是易错的，尤其当它的界并非局部时更是如此。

而且它也是一种“流行”的错误来源（缓冲区溢出，数组退化而成的指针，等等）。

而 `a2` 的定义符合 C 但不符合 C++，而且被认为存在安全性风险。

示例

```
const int n = 7;
int m = 9;

void f()
{
    array<int, n> a1;
    stack_array<int> a2(m);
    // ...
}
```

强制实施

- 对具有非常量界的数组（C 风格的 VLA）作出标记。
- 对具有非局部的常量界的数组作出标记。

ES.28: 为复杂的初始化（尤其是 `const` 变量）使用 lambda

理由

它可以很好地封装局部的初始化，包括对仅为初始化所需的临时变量进行清理，而且避免了创建不必要的非局部而且无法重用的函数。它对于应当为 `const` 的变量也可以工作，不过必须先进行一些初始化。

示例，不好

```
widget x; // 应当为 const，不过：
for (auto i = 2; i <= N; ++i) {           // 这是由 x 的
    x += some_obj.do_something_with(i); // 初始化所需的
}                                         // 一段任意长的代码
// 自此开始，x 应当为 const，不过我们无法在这种风格的代码中做到这点
```

示例，好

```
const widget x = [&] {
    widget val;                                // 假定 widget 具有默认构造函数
    for (auto i = 2; i <= N; ++i) {             // 这是由 x 的
        val += some_obj.do_something_with(i);   // 初始化所需的
    }                                              // 一段任意长的代码
    return val;
}();
```

如果可能的话，应当将条件缩减成一个后续的简单集合（比如一个 `enum`），并避免把选择和初始化相互混合起来。

强制实施

很难。最多是某种启发式方案。查找跟随某个未初始化变量之后的循环中向其赋值。

ES.30: 不要用宏来操纵程序文本

理由

宏是 BUG 的一个主要来源。

宏不遵守常规的作用域和类型规则。

宏保证会让人读到的东西和编译器见到的东西不一样。

宏使得工具的建造复杂化。

示例，不好

```
#define Case break; case /* 不好 */
```

这个貌似无害的宏会把某个大写的 `C` 替换为小写的 `c` 导致一个严重的控制流错误。

注解

这条规则并不禁止在 `#ifdef` 等部分中使用用于“配置控制”的宏。

将来，模块可能会消除配置控制中对宏的需求。

注解

此规则也意味着不鼓励使用 `#` 进行字符串化和使用 `##` 进行连接。

照例，宏有一些“无害”的用途，但即使这些也会给工具带来麻烦，

例如自动完成器、静态分析器和调试器。

通常，使用花式宏的欲望是过于复杂的设计的标志。

另外，`#` 和 `##` 促进了宏的定义和使用：

```
#define CAT(a, b) a ## b
#define STRINGIFY(a) #a

void f(int x, int y)
{
    string CAT(x, y) = "asdf";    // 不好：工具难以处理（也很丑陋）
    string sx2 = STRINGIFY(x);
    // ...
}
```

有使用宏进行低级字符串操作的变通方法。例如：

```
string s = "asdf" "lkjh";    // 普通的字符串文字连接

enum E { a, b };

template<int x>
constexpr const char* stringify()
{
    switch (x) {
        case a: return "a";
        case b: return "b";
    }
}

void f(int x, int y)
{
    string sx = stringify<x>();
    // ...
}
```

这不像定义宏那样方便，但是易于使用、零开销，并且是类型化的和作用域化的。

将来，静态反射可能会消除对程序文本操作的预处理器的最终需求。

强制实施

见到并非仅用于源代码控制（比如 `#ifdef`）的宏时应当大声尖叫。

ES.31: 不要用宏来作为常量或“函数”

理由

宏是 BUG 的一个主要来源。

宏不遵守常规的作用域和类型规则。

宏不遵守常规的参数传递规则。

宏保证会让人读到的东西和编译器见到的东西不一样。

宏使得工具的建造复杂化。

示例，不好

```
#define PI 3.14
#define SQUARE(a, b) (a * b)
```

即便我们并未在 `SQUARE` 中留下这个众所周知的 BUG，也存在多种表现好得多的替代方式；比如：

```
constexpr double pi = 3.14;
template<typename T> T square(T a, T b) { return a * b; }
```

强制实施

见到并非仅用于源代码控制（比如 `#ifdef`）的宏时应当大声尖叫。

ES.32: 对所有的宏名采用 ALL_CAPS 命名方式

理由

遵循约定。可读性。区分宏。

示例

```
#define forever for (;;) /* 非常不好 */  
  
#define FOREVER for (;;) /* 仍然很邪恶，但至少对人来说是可见的 */
```

强制实施

见到小写的宏时应当大声尖叫。

ES.33: 如果必须使用宏的话，请为之提供唯一的名字

理由

宏并不遵守作用域规则。

示例

```
#define MYCHAR /* 不好，最终将会和别人的 MYCHAR 相冲突 */  
  
#define ZCORP_CHAR /* 还是不好，但冲突的机会较小 */
```

注解

如果可能就应当避免使用宏：[ES.30](#), [ES.31](#), 以及 [ES.32](#)。

然而，存在亿万行的代码中包含宏，以及一种使用并过度使用宏的长期传统。

如果你被迫使用宏的话，请使用长名字，而且应当带有唯一前缀（比如你的组织机构的名字）以减少冲突的可能性。

强制实施

对较短的宏名给出警告。

ES.34: 不要定义（C 风格的）变参函数

理由

它并非类型安全。

而且需要杂乱的满是强制转换和宏的代码才能正确工作。

示例

```
#include <cstdarg>  
  
// "severity" 后面跟着以零终结的 char* 列表；将 C 风格字符串写入 cerr  
void error(int severity ...)  
{
```

```

va_list ap; // 一个持有参数的神奇类型
va_start(ap, severity); // 参数启动: "severity" 是 error() 的第一个参数

for (;;) {
    // 将下一个变量看作 char*: 没有检查: 经过伪装的强制转换
    char* p = va_arg(ap, char*);
    if (!p) break;
    cerr << p << ' ';
}

va_end(ap); // 参数清理 (不能忘了这个)

cerr << '\n';
if (severity) exit(severity);
}

void use()
{
    error(7, "this", "is", "an", "error", nullptr);
    error(7); // 崩溃
    error(7, "this", "is", "an", "error"); // 崩溃
    const char* is = "is";
    string an = "an";
    error(7, "this", "is", an, "error"); // 崩溃
}

```

替代方案: 重载。模板。变参模板。

```

#include <iostream>

void error(int severity)
{
    std::cerr << '\n';
    std::exit(severity);
}

template<typename T, typename... Ts>
constexpr void error(int severity, T head, Ts... tail)
{
    std::cerr << head;
    error(severity, tail...);
}

void use()
{
    error(7); // 不会崩溃!
    error(5, "this", "is", "not", "an", "error"); // 不会崩溃!

    std::string an = "an";
    error(7, "this", "is", "not", an, "error"); // 不会崩溃!

    error(5, "oh", "no", nullptr); // 编译器报错! 不需要 nullptr.
}

```

注解

这基本上就是 `printf` 的实现方式。

强制实施

- 对 C 风格的变参函数的定义作出标记。
- 对 `#include <cstdarg>` 和 `#include <stdarg.h>` 作出标记。

ES.expr: 表达式

表达式对值进行操作。

ES.40: 避免复杂的表达式

理由

复杂的表达式是易错的。

示例

```
// 不好：在子表达式中藏有赋值
while ((c = getc()) != -1)

// 不好：在一个子表达式中对两个非局部变量进行了赋值
while ((cin >> c1, cin >> c2), c1 == c2)

// 有改善，但可能仍然过于复杂
for (char c1, c2; cin >> c1 >> c2 && c1 == c2;)

// OK：若 i 和 j 并非别名
int x = ++i + ++j;

// OK：若 i != j 且 i != k
v[i] = v[j] + v[k];

// 不好：子表达式中“隐藏”了多个赋值
x = a + (b = f()) + (c = g()) * 7;

// 不好：依赖于经常被误解的优先级规则
x = a & b + c * d && e ^ f == 7;

// 不好：未定义行为
x = x++ + x++ + ++x;
```

这些表达式中有几个是无条件不好的（比如说依赖于未定义行为）。其他的只不过过于复杂和不常见，即便是优秀的程序员匆忙中也可能会误解或者忽略其中的某个问题。

注解

C++17 收紧了有关求值顺序的规则

（除了赋值中从右向左，以及函数实参数求值顺序未指明外均为从左向右，[参见 ES.43](#)），但这并不影响复杂表达式很容易引起混乱的事实。

注解

程序员应当了解并运用表达式的基本规则。

示例

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

auto t1 = k * y;          // 不好：不必要的啰嗦
x = t1 + z;

if (0 <= x && x < max) // OK

auto t1 = 0 <= x;         // 不好：不必要的啰嗦
auto t2 = x < max;
if (t1 && t2)            // ...
```

强制实施

很麻烦。多复杂的表达式才能被当成复杂的？把计算写成每条语句一个操作同样是让人混乱的。需要考虑的有：

- 副作用：（对于某种非局部性定义，）多个非局部变量上发生副作用是值得怀疑的，尤其是当这些副作用是在不同的子表达式中时
- 向别名变量的写入
- 超过 N 个运算符（N 应当为多少？）
- 依赖于微妙的优先级规则
- 使用了未定义行为（我们是否应当识别所有的未定义行为？）
- 实现定义的行为？
- ???

ES.41: 对运算符优先级不保准时应使用括号

理由

避免错误。可读性。不是每个人都能记住运算符表格。

示例

```
const unsigned int flag = 2;
unsigned int a = flag;

if (a & flag != 0) // 不好：含义为 a&(flag != 0)
```

注意：我们建议程序员了解算术运算和逻辑运算的优先级表，但应当考虑当按位逻辑运算和其他运算符混合使用时需要采用括号。

```
if ((a & flag) != 0) // OK：按预期工作
```

注解

你应当了解足够的知识以避免在这样的情况下需要括号：

```
if (a < 0 || a <= max) {  
    // ...  
}
```

强制实施

- 当按位逻辑运算符符合其他运算符组合时进行标记。
- 当赋值运算符不是最左边的运算符时进行标记。
- ???

ES.42: 保持单纯直接的指针使用方式

理由

复杂的指针操作是一种重大的错误来源。

注解

代之以使用 `gsl::span`。

指针[只应当指代单个对象](#)。

指针算术是脆弱而易错的，是许多许多糟糕的 BUG 和安全漏洞的来源。

`span` 是一种用于访问数组对象的带有边界检查的安全类型。

以常量为下标来访问已知边界的数组，编译器可以进行验证。

示例，不好

```
void f(int* p, int count)  
{  
    if (count < 2) return;  
  
    int* q = p + 1;      // 不好  
  
    ptrdiff_t d;  
    int n;  
    d = (p - &n);      // OK  
    d = (q - p);       // OK  
  
    int n = *p++;       // 不好  
  
    if (count < 6) return;  
  
    p[4] = 1;           // 不好  
    p[count - 1] = 2;   // 不好  
    use(&p[0], 3);     // 不好  
}
```

示例，好

```
void f(span<int> a) // 好多了：函数声明中使用了 span
{
    if (a.size() < 2) return;

    int n = a[0];          // OK

    span<int> q = a.subspan(1); // OK

    if (a.size() < 6) return;

    a[4] = 1;              // OK

    a[a.size() - 1] = 2;   // OK

    use(a.data(), 3);    // OK
}
```

注解

用变量做下标，对于工具和人类来说都是很难将其验证为安全的。

`span` 是一种用于访问数组对象的带有运行时边界检查的安全类型。

`at()` 是可以保证单次访问进行边界检查的另一种替代方案。

如果需要用迭代器来访问数组的话，应使用构造于数组之上的 `span` 所提供的迭代器。

示例，不好

```
void f(array<int, 10> a, int pos)
{
    a[pos / 2] = 1; // 不好
    a[pos - 1] = 2; // 不好
    a[-1] = 3;      // 不好（但易于被工具查出） - 没有替代方案，请勿这样做
    a[10] = 4;      // 不好（但易于被工具查出） - 没有替代方案，请勿这样做
}
```

示例，好

使用 `span`：

```
void f1(span<int, 10> a, int pos) // A1: 将参数类型改为使用 span
{
    a[pos / 2] = 1; // OK
    a[pos - 1] = 2; // OK
}

void f2(array<int, 10> arr, int pos) // A2: 增加局部的 span 并使用之
{
    span<int> a = {arr.data(), pos};
    a[pos / 2] = 1; // OK
    a[pos - 1] = 2; // OK
}
```

使用 `at()`：

```
void f3(array<int, 10> a, int pos) // 替代方案 B: 用 at() 进行访问
{
    at(a, pos / 2) = 1; // OK
    at(a, pos - 1) = 2; // OK
}
```

示例，不好

```
void f()
{
    int arr[COUNT];
    for (int i = 0; i < COUNT; ++i)
        arr[i] = i; // 不好，不能使用非常量索引
}
```

示例，好

使用 `span`:

```
void f1()
{
    int arr[COUNT];
    span<int> av = arr;
    for (int i = 0; i < COUNT; ++i)
        av[i] = i;
}
```

使用 `span` 和基于范围的 `for`:

```
void f1a()
{
    int arr[COUNT];
    span<int, COUNT> av = arr;
    int i = 0;
    for (auto& e : av)
        e = i++;
}
```

使用 `at()` 进行访问:

```
void f2()
{
    int arr[COUNT];
    int i = 0;
    for (int i = 0; i < COUNT; ++i)
        at(arr, i) = i;
}
```

使用基于范围的 `for`:

```
void f3()
{
    int arr[COUNT];
    for (auto& e : arr)
        e = i++;
}
```

注解

工具可以提供重写能力，以将涉及动态索引表达式的数组访问替换为使用 `at()` 进行访问：

```
static int a[10];

void f(int i, int j)
{
    a[i + j] = 12;           // 不好，可以重写为 ...
    at(a, i + j) = 12;      // OK - 带有边界检查
}
```

示例

把数组转变为指针（语言基本上总会这样做），移除了进行检查的机会，因此应当予以避免

```
void g(int* p);

void f()
{
    int a[5];
    g(a);           // 不好：是要传递一个数组吗？
    g(&a[0]);      // OK：传递单个对象
}
```

如果要传递数组的话，应该这样：

```
void g(int* p, size_t length); // 老的（危险）代码

void g1(span<int> av); // 好多了：改动了 g()。

void f()
{
    int a[5];
    span<int> av = a;

    g(av.data(), av.size()); // OK，如果没有其他选择的话
    g1(a);                  // OK - 这里没有退化，而是使用了隐式的 span 构造函数
}
```

强制实施

- 对任何在指针类型的表达式上进行的产生指针类型的值的算术运算进行标记。
- 对任何数组类型的表达式或变量（无论是静态数组还是 `std::array`）上进行索引的表达式，若其索引不是值为从 0 到数组上界之内的编译期常量表达式，则进行标记。
- 对任何可能依赖于从数组类型向指针类型的隐式转换的表达式进行标记。

本条规则属于[边界安全性剖面配置](#)。

ES.43: 避免带有未定义的求值顺序的表达式

理由

你没办法搞清楚这种代码会做什么。可移植性。

即便它做到了对你可能有意义的事情，它在别的编译器（比如你的编译器的下个版本）或者不同的优化设置中也可能会做出不同的事情。

注解

C++17 收紧了有关求值顺序的规则：

除了赋值中从右向左，以及函数实参数求值顺序未指明外均为从左向右。

不过，要记住你的代码可能是由 C++17 之前的编译器进行编译的（比如通过复制粘贴），请勿自作聪明。

示例

```
v[i] = ++i; // 其结果是未定义的
```

一条不错经验法则是，你不应当在一个表达式中两次读取你所写入的值。

强制实施

可以由优秀的分析器检测出来。

ES.44: 不要对函数参数求值顺序有依赖

理由

因为这种顺序是未定义的。

注解

C++17 收紧了有关求值顺序的规则，但函数实参数求值顺序仍然是未指明的。

示例

```
int i = 0;
f(++i, ++i);
```

在 C++17 之前，其行为是未定义的。因此其行为可能是任何事（比如 `f(2, 2)`）。

自 C++17 起，中这段代码没有未定义行为，但仍未指定是哪个实参被首先求值。这个调用会是 `f(0, 1)` 或 `f(1, 0)`，但你不知道是哪个。

示例

重载运算符可能导致求值顺序问题：

```
f1() ->m(f2());           // m(f1(), f2())
cout << f1() << f2();    // operator<<(operator<<(cout, f1()), f2())
```

在 C++17 中，这些例子将按预期工作（自左向右），而赋值则按自右向左求值（`=` 正是自右向左绑定的）

```
f1() = f2();      // C++14 中为未定义行为; C++17 中 f2() 在 f1() 之前求值
```

强制实施

可以由优秀的分析器检测出来。

ES.45: 避免“魔法常量”，采用符号化常量

理由

表达式中内嵌的无名的常量很容易被忽略，而且经常难于理解：

示例

```
for (int m = 1; m <= 12; ++m)    // 请勿如此：魔法常量 12
    cout << month[m] << '\n';
```

不是所有人都知道一年中有 12 个月份，号码是 1 到 12。更好的做法是：

```
// 月份索引值为 1..12
constexpr int first_month = 1;
constexpr int last_month = 12;

for (int m = first_month; m <= last_month; ++m)    // 好多了
    cout << month[m] << '\n';
```

更好的做法是，不要暴露常量：

```
for (auto m : month)
    cout << m << '\n';
```

强制实施

标记代码中的字面量。让 `0`, `1`, `nullptr`, `\n`, `""`, 以及某个确认列表中的其他字面量通过检查。

ES.46: 避免丢失数据（窄化、截断）的算术转换

理由

窄化转换会销毁信息，通常是不期望发生的。

示例，不好

关键的例子就是基本的窄化：

```
double d = 7.9;
int i = d;      // 不好: 窄化: i 变为了 7
i = (int) d;   // 不好: 我们打算声称这样的做法仍然不够明确

void f(int x, long y, double d)
{
    char c1 = x;    // 不好: 窄化
    char c2 = y;    // 不好: 窄化
    char c3 = d;    // 不好: 窄化
}
```

注解

指导方针支持库提供了一个 `narrow_cast` 操作，用以指名发生窄化是可接受的，以及一个 `narrow` (“窄化判定”) 当窄化将会损失合法值时将会抛出一个异常：

```
i = gsl::narrow_cast<int>(d);    // OK (明确需要): 窄化: i 变为了 7
i = gsl::narrow<int>(d);        // OK: 抛出 narrowing_error
```

其中还包含了一些含有损失的算术强制转换，比如从负的浮点类型到无符号整型类型的强制转换：

```
double d = -7.9;
unsigned u = 0;

u = d;                      // 不好: 发生窄化
u = gsl::narrow_cast<unsigned>(d); // OK (明确需要): u 变为了 4294967289
u = gsl::narrow<unsigned>(d);    // OK: 抛出 narrowing_error
```

注解

这条规则不适用于[按语境转换为 `bool`](#) 的情形：

```
if (ptr) do_something(*ptr);    // OK: ptr 被用作条件
bool b = ptr;                  // 不好: 发生窄化
```

强制实施

优良的分析器可以检测到所有的窄化转换。不过，对所有的窄化转换都进行标记将带来大量的误报。建议的做法是：

- 标记出所有的浮点向整数转换（可能只有 `float -> char` 和 `double -> int`。这里有问题！需要数据支持）。
- 标记出所有的 `long -> char`（我怀疑 `int -> char` 非常常见。这里有问题！需要数据支持）。
- 在函数参数上发生的窄化转换特别值得怀疑。

ES.47: 使用 `nullptr` 而不是 0 或 `NULL`

理由

可读性。最小化意外：`nullptr` 不可能和 `int` 混淆。

`nullptr` 还有一个严格定义的（非常严格）类型，且因此可以在类型推断可能在 `NULL` 或 `0` 上犯错的场合中仍能正常工作。

示例

考虑：

```
void f(int);
void f(char*);
f(0);           // 调用 f(int)
f(nullptr);    // 调用 f(char*)
```

强制实施

对用作指针的 `0` 和 `NULL` 进行标记。可以用简单的程序变换来达成这种变换。

ES.48: 避免强制转换

理由

强制转换是众所周知的错误来源，它们使得一些优化措施变得不可靠。

示例，不好

```
double d = 2;
auto p = (long*)&d;
auto q = (long long*)&d;
cout << d << ' ' << *p << ' ' << *q << '\n';
```

你觉得这段代码会打印出什么呢？结果最好由实现定义。我得到的是

```
2 0 4611686018427387904
```

加上这些

```
*q = 666;
cout << d << ' ' << *p << ' ' << *q << '\n';
```

得到的是

```
3.29048e-321 666 666
```

奇怪吗？我很庆幸程序没有崩溃掉。

注解

写下强制转换的程序员通常认为他们知道所做的是什么事情，

或者写出强制转换能让程序“更易读”。

而实际上，他们这样经常会禁止掉使用值的一些一般规则。

如果存在正确的函数的话，重载决议和模板实例化通常都能挑选出正确的函数。

如果没有的话，则可能本应如此，而不应该进行某种局部的修补（强制转换）。

注解

强制转换在系统编程语言中是必要的。例如，否则我们怎么才能把设备寄存器的地址放入一个指针呢？然而，强制转换却被严重过度使用了，而且也是一种主要的错误来源。

当你觉得需要进行大量强制转换时，可能存在一个基本的设计问题。

[类型剖面配置](#) 禁止使用 `reinterpret_cast` 和 C 风格强制转换。

不要以强制转换为 `(void)` 来忽略 `[[nodiscard]]` 返回值。

当你有意要丢弃这种返回值时，应当首先深入思考这是不是确实是个好主意（通常，这个函数或者使用了 `[[nodiscard]]` 的返回类型的作者，当初确实是有充分理由的）。

要是你仍然觉得这样做是合适的，而且你的代码评审者也同意的话，使用 `std::ignore =` 来关闭这个警告，这既简单，可移植，也易于 grep。

替代方案

强制转换被广泛（误）用了。现代 C++ 已经提供了一些规则和语言构造，消除了许多语境中对强制转换的需求，比如

- 使用模板
- 使用 `std::variant`
- 借助良好定义的，安全的，指针类型之间的隐式转换
- 使用 `std::ignore =` 来忽略 `[[nodiscard]]` 值

强制实施

- 对包括向 `void` 在内的所有 C 风格强制转换进行标记。
- 对使用 `Type(value)` 的函数风格强制转换进行标记。应代之以使用不会发生窄化的 `Type{value}`。（参见 [ES.64](#)。）
- 对指针类型之间的[同一强制转换](#)，若其中的源类型和目标类型相同(#Pro-type-identitycast)则进行标记。
- 对可以作为[隐式转换](#)的显示指针强制转换进行标记。

ES.49: 当必须使用强制转换时，使用具名的强制转换

理由

可读性。避免错误。

具名的强制转换比 C 风格或函数风格的强制转换更加特殊，允许编译器捕捉到某些错误。

具名的强制转换包括：

- `static_cast`
- `const_cast`
- `reinterpret_cast`
- `dynamic_cast`
- `std::move` // `move(x)` 是指代 `x` 的右值引用
- `std::forward` // `forward<T>(x)` 是指代 `x` 的左值或右值引用（取决于 `T`）
- `gsl::narrow_cast` // `narrow_cast<T>(x)` 就是 `static_cast<T>(x)`
- `gsl::narrow` // `narrow<T>(x)` 在当 `static_cast<T>(x) == x` 时即为 `static_cast<T>(x)` 否则会抛出 `narrowing_error`

示例

```
class B { /* ... */ };
class D { /* ... */ };

template<typename D> D* upcast(B* pb)
{
    D* pd0 = pb;                                // 错误: 不存在从 B* 向 D* 的隐式转换
    D* pd1 = (D*)pb;                            // 合法, 但干了什么?
    D* pd2 = static_cast<D*>(pb);            // 错误: D 并非派生于 B
    D* pd3 = reinterpret_cast<D*>(pb);        // OK: 你自己负责!
    D* pd4 = dynamic_cast<D*>(pb);           // OK: 返回 nullptr
    // ...
}
```

这个例子是从真实世界的 BUG 合成的，其中 `D` 曾经派生于 `B`，但某个人重构了继承层次。
C 风格的强制转换很危险，因为它可以进行任何种类的转换，使我们丧失了今后受保护不犯错的机会。

注解

当在类型之间进行没有信息丢失的转换时（比如从 `float` 到 `double` 或者从 `int32` 到 `int64`），可以代之以使用花括号初始化。

```
double d {some_float};
int64_t i {some_int32};
```

这样做明确了有意进行类型转换，而且同样避免了发生可能导致精度损失的结果的类型转换。（比如说，试图用这种风格来从 `double` 初始化 `float` 会导致编译错误。）

注解

`reinterpret_cast` 可以很基础，但其基础用法（如将机器地址转化为指针）并不是类型安全的：

```
auto p = reinterpret_cast<Device_register>(0x800); // 天生危险
```

强制实施

- 对包括向 `void` 在内的所有 C 风格强制转换进行标记。
- 对使用 `Type(value)` 的函数风格强制转换进行标记。应代之以使用不会发生窄化的 `Type{value}`。（参见 [ES.64](#)。）
- [类型剖面配置](#)禁用了 `reinterpret_cast`。
- [类型剖面配置](#)对于在算术类型之间使用 `static_cast` 时给出警告。

ES.50: 不要强制掉 `const`

理由

这是在 `const` 上说谎。
若变量确实声明为 `const`，修改它将导致未定义的行为。

示例，不好

```
void f(const int& x)
{
    const_cast<int&>(x) = 42;    // 不好
}

static int i = 0;
static const int j = 0;

f(i); // 暗藏的副作用
f(j); // 未定义的行为
```

示例

有时候，你可能倾向于借助 `const_cast` 来避免代码重复，比如两个访问函数仅在是否 `const` 上有区别而实现相似的情况。例如：

```
class Bar;

class Foo {
public:
    // 不好，逻辑重复
    Bar& get_bar()
    {
        /* 获取 my_bar 的非 const 引用前后的复杂逻辑 */
    }

    const Bar& get_bar() const
    {
        /* 获取 my_bar 的 const 引用前后的相同的复杂逻辑 */
    }
private:
    Bar my_bar;
};
```

应当改为共享实现。通常可以直让非 `const` 函数来调用 `const` 函数。不过当逻辑复杂的时候这可能会导致下面这样的模式，仍然需要借助于 `const_cast`：

```
class Foo {
public:
    // 不大好，非 const 函数调用 const 版本但借助于 const_cast
    Bar& get_bar()
    {
        return const_cast<Bar&>(static_cast<const Foo&>(*this).get_bar());
    }

    const Bar& get_bar() const
    {
        /* 获取 my_bar 的 const 引用前后的复杂逻辑 */
    }
private:
    Bar my_bar;
};
```

虽然这个模式如果恰当应用的话是安全的（因为调用方必然以一个非 `const` 对象来开始），但这并不理想，因为其安全性无法作为检查工具的规则而自动强制实施。

换种方式，可以优先将公共代码放入一个公共辅助函数中，并将之作为模板以使其推断 `const`。这完全不会用到 `const_cast`：

```
class Foo {  
public: // 好  
    Bar& get_bar() { return get_bar_impl(*this); }  
    const Bar& get_bar() const { return get_bar_impl(*this); }  
private:  
    Bar my_bar;  
  
    template<class T> // 好，推断出 T 是 const 还是非 const  
    static auto& get_bar_impl(T& t)  
    { /* 获取 my_bar 的可能为 const 的引用前后的复杂逻辑 */ }  
};
```

注意：不要在模板中编写大型的非待决代码，因为它将导致代码爆炸。例如，进一步改进是当 `get_bar_impl` 的全部或部分代码是非待决代码时将之重构移出到一个公共的非模板函数中去，这可能会使代码大小显著变小。

例外

当调用 `const` 不正确的函数时，你可能需要强制掉 `const`。

应当优先将这种函数包装到内联的 `const` 正确的包装函数中，以将强制转换封装到一处中。

示例

有时候，“强制掉 `const`”是为了允许对本来无法改动的对象中的某种临时性的信息进行更新操作。其例子包括进行缓存，备忘，以及预先计算等。

这样的例子，通常可以通过使用 `mutable` 或者通过一层间接进行处理，而同使用 `const_cast` 一样甚或比之更好。

考虑为昂贵操作将之前所计算的结果保留下来：

```
int compute(int x); // 为 x 计算一个值；假设这是昂贵的  
  
class Cache { // 为 int->int 操作实现一种高速缓存的某个类型  
public:  
    pair<bool, int> find(int x) const; // 有针对 x 的值吗?  
    void set(int x, int v); // 使 y 成为针对 x 的值  
    // ...  
private:  
    // ...  
};  
  
class X {  
public:  
    int get_val(int x)  
    {  
        auto p = cache.find(x);  
        if (p.first) return p.second;  
        int val = compute(x);  
        cache.set(x, val); // 插入针对 x 的值  
    }
```

```
    return val;
}
// ...
private:
    Cache cache;
};
```

这里的 `get_val()` 逻辑上是个常量，因此我们想使其成为 `const` 成员。为此我们仍然需要改动 `cache`，因此人们有时候会求助于 `const_cast`：

```
class X { // 基于强制转换的可疑的方案
public:
    int get_val(int x) const
    {
        auto p = cache.find(x);
        if (p.first) return p.second;
        int val = compute(x);
        const_cast<Cache*>(cache).set(x, val); // 很难看
        return val;
    }
    // ...
private:
    Cache cache;
};
```

幸运的是，有一种更好的方案：

将 `cache` 称为即便对于 `const` 对象来说也是可改变的：

```
class X { // 更好的方案
public:
    int get_val(int x) const
    {
        auto p = cache.find(x);
        if (p.first) return p.second;
        int val = compute(x);
        cache.set(x, val);
        return val;
    }
    // ...
private:
    mutable Cache cache;
};
```

另一种替代方案是存储指向 `cache` 的指针：

```
class X { // OK, 但有点麻烦的方案
public:
    int get_val(int x) const
    {
        auto p = cache->find(x);
        if (p.first) return p.second;
        int val = compute(x);
        cache->set(x, val);
```

```
    return val;
}
// ...
private:
    unique_ptr<Cache> cache;
};
```

这个方案最灵活，但需要显式进行 `*cache` 的构造和销毁

(最可能发生于 `x` 的构造函数和析构函数中)。

无论采用哪种形式，在多线程代码中都需要保护对 `cache` 的数据竞争，可能需要使用一个 `std::mutex`。

强制实施

- 标记 `const_cast`。
- 本条规则属于[类型安全性剖面配置](#)。

ES.55: 避免发生对范围检查的需要

理由

无法溢出的构造时不会溢出的（而且通常运行得更快）：

示例

```
for (auto& x : v)      // 打印 v 的所有元素
    cout << x << '\n';

auto p = find(v, x);   // 在 v 中寻找 x
```

强制实施

查找显式的范围检查，并启发式地给出替代方案建议。

ES.56: 仅在确实需要明确移动某个对象到别的作用域时才使用

`std::move()`

理由

我们用移动而不是复制，以避免发生重复并提升性能。

一次移动通常会遗留一个空对象 ([C.64](#))，这可能令人意外甚至很危险，因此我们试图避免从左值进行移动（它们可能随后会被访问到）。

注解

当来源是右值（比如 `return` 的值或者函数的结果）时就会隐式地进行移动，因此请不要在这些情况下明确写下 `move` 而无意义地使代码复杂化。可以代之以编写简短的返回值的函数，这样的话无论是函数的返回还是调用方的返回值接收，都会很自然地得到优化。

一般来说，遵循本文档中的指导方针（包括不要让变量的作用域无必要地变大，编写返回值的简短函数，返回局部变量等），有助于消除大多数对显式使用 `std::move` 的需要。

显式的 `move` 需要用于把某个对象明确移动到另一个作用域，尤其是将其传递给某个“接收器”函数，以及移动操作自身（移动构造函数，移动赋值运算符）和交换（`swap`）操作的实现之中。

示例，不好

```
void sink(x&& x); // sink 接收 x 的所有权

void user()
{
    x x;
    // 错误：无法将作者绑定到右值引用
    sink(x);
    // OK: sink 接收了 x 的内容，x 随即必须假定为空
    sink(std::move(x));

    // ...

    // 可能是个错误
    use(x);
}
```

通常来说，`std::move()` 都用做某个 `&&` 形参的实参。

而这点之后，应当假定对象已经被移走（参见 [C.64](#)），而直到首次向它设置某个新值之前，请勿再次读取它的状态。

```
void f()
{
    string s1 = "supercalifragilisticexpialidocious";

    string s2 = s1; // ok, 接收了一个副本
    assert(s1 == "supercalifragilisticexpialidocious"); // ok

    // 不好，如果你打算保留 s1 的值的话
    string s3 = move(s1);

    // 不好，assert 很可能会失败，s1 很可能被改动了
    assert(s1 == "supercalifragilisticexpialidocious");
}
```

示例

```
void sink(unique_ptr<widget> p); // 将 p 的所有权传递给 sink()

void f()
{
    auto w = make_unique<widget>();
    // ...
    sink(std::move(w)); // ok, 交给 sink()
    // ...
    sink(w); // 错误：unique_ptr 经过严格设计，你无法复制它
}
```

注解

`std::move()` 经过伪装的向 `&&` 的强制转换；其自身并不会移动任何东西，但会把具名的对象标记为可被移动的候选者。

语言中已经了解了对象可以被移动的一般情况，尤其是从函数返回时，因此请不要用多余的 `std::move()` 使代码复杂化。

绝不要仅仅因为听说过“这样更加高效”就使用 `std::move()`。

通常来说，请不要相信那些没有支持数据的有关“效率”的断言。（??？）

通常来说，请不要无理由地使代码复杂化。（??）

绝不要在 `const` 对象上 `std::move()`，它只会暗中将其转变成一个副本（参见 [Meyers15](#) 的条款 23）。

示例，不好

```
vector<int> make_vector()
{
    vector<int> result;
    // ... 加载 result 的数据
    return std::move(result);           // 不好；直接写 "return result;" 即可
}
```

绝不要写 `return move(local_variable);`，这是因为语言已经知道这个变量是移动的候选了。

在这段代码中用 `move` 并不会带来帮助，而且可能实际上是有害的，因为它创建了局部变量的一个额外引用别名，而在某些编译器中这回对 RVO（返回值优化）造成影响。

示例，不好

```
vector<int> v = std::move(make_vector()); // 不好；这个 std::move 完全是多余的
```

绝不在返回值上使用 `move`，如 `x = move(f());`，其中的 `f` 按值返回。

语言已经知道返回值是临时对象而且可以被移动。

示例

```
void mover(x&& x)
{
    call_something(std::move(x));           // ok
    call_something(std::forward<x>(x));    // 不好，请勿对右值引用 std::forward
    call_something(x);                     // 可疑 为什么不用 std::move?
}

template<class T>
void forwarder(T&& t)
{
    call_something(std::move(t));           // 不好，请勿对转发引用 std::move
    call_something(std::forward<T>(t));    // ok
    call_something(t);                    // 可疑，为什么不用 std::forward?
}
```

强制实施

- 对于 `std::move(x)` 的使用，当 `x` 是右值，或者语言已经将其当做右值，这包括 `return std::move(local_variable);` 以及在按值返回的函数上的 `std::move(f())`，进行标记。
- 当没有接受 `const s&` 的函数重载来处理左值时，对接受 `s&&` 参数的函数进行标记。
- 若实参经过 `std::move` 传递给形参则进行标记，除非形参的类型为右值引用 `x&&`，或者类型是只能移动的而该形参为按值传递。
- 当对转发引用 (`T&&` 其中 `T` 为模板参数类型) 使用 `std::move` 时进行标记。应当代之以使用 `std::forward`。
- 当对并非 `const` 右值引用的变量使用 `std::move` 时进行标记。（这是前一条规则的更一般的情况，以覆盖非转发的情况。）
- 当对右值引用 (`x&&` 其中 `x` 为非模板形参类型) 使用 `std::forward` 时进行标记。应当代之以使用 `std::move`。
- 当对并非转发引用使用 `std::forward` 时进行标记。（这是前一条规则的更一般的情况，以覆盖非移动的情况。）
- 如果对象潜在地被移动走之后的下一个操作是 `const` 操作的话，则进行标记；首先应当交错进行一个非 `const` 操作，最好是赋值，以首先对对象的值进行重置。

ES.60: 避免在资源管理函数之外使用 `new` 和 `delete`

理由

应用程序代码中的直接资源管理既易错又麻烦。

注解

通常也被称为“禁止裸 `new!`”规则。

示例，不好

```
void f(int n)
{
    auto p = new X[n]; // n 个默认构造的 X
    // ...
    delete[] p;
}
```

... 部分中的代码可能导致 `delete` 永远不会发生。

参见: [R: 资源管理](#)

强制实施

对裸的 `new` 和裸的 `delete` 进行标记。

ES.61: 用 `delete[]` 删除数组，用 `delete` 删除非数组对象

理由

这正是语言的要求，而且所犯的错误将导致资源释放的错误以及内存破坏。

示例，不好

```
void f(int n)
{
    auto p = new X[n]; // n 个默认初始化的 X
    // ...
    delete p; // 错误：仅仅删除了对象 p，而并未删除数组 p[]
}
```

注解

这个例子不仅像上一个例子一样违反了[禁止裸 new 规则](#)，它还有更多的问题。

强制实施

- 如果 `new` 和 `delete` 在同一个作用域中的话，就可以标记出现错误。
- 如果 `new` 和 `delete` 出现在构造函数/析构函数对之中的话，就可以标记出现错误。

ES.62: 不要在不同的数组之间进行指针比较

理由

这样做的结果是未定义的。

示例，不好

```
void f()
{
    int a1[7];
    int a2[9];
    if (&a1[5] < &a2[7]) {} // 不好：未定义
    if (0 < &a1[5] - &a2[7]) {} // 不好：未定义
}
```

注解

这个例子中有许多问题。

强制实施

???

ES.63: 不要产生切片

理由

切片——亦即使用赋值或初始化而只对对象的一部分进行复制——通常会导致错误，这是因为对象总是被当成是一个整体。

在罕见的进行蓄意的切片的代码中，其代码会让人意外。

示例

```
class Shape { /* ... */ };
class Circle : public Shape { /* ... */ Point c; int r; };

Circle c {{0, 0}, 42};
Shape s {c}; // 仅复制构造了 Circle 中的 Shape 部分
```

```

s = c;           // 仅复制赋值了 Circle 中的 Shape 部分

void assign(const Shape& src, Shape& dest)
{
    dest = src;
}
Circle c2 {{1, 1}, 43};
assign(c, c2); // 噢, 传递的并不是整个状态
assert(c == c2); // 如果提供复制操作, 就也得提供比较操作,
                  // 但这里很可能返回 false

```

这样的结果是无意义的, 因为不会把中心和半径从 `c` 复制给 `s`。

针对这个的第一条防线是[将基类 Shape 定义为不允许这样做](#)。

替代方案

如果确实需要切片的话, 应当为之定义一个明确的操作。

这会避免读者产生混乱。

例如:

```

class Smiley : public Circle {
public:
    Circle copy_circle();
    // ...
};

Smiley sm { /* ... */ };

```

`Circle c1 {sm};` // 理想情况下由 Circle 的定义所禁止

`Circle c2 {sm.copy_circle()};`

强制实施

针对切片给出警告。

ES.64: 使用 `T{e}` 写法来进行构造

理由

对象构造语法 `T{e}` 明确了所需进行的构造。

对象构造语法 `T{e}` 不允许发生窄化。

`T{e}` 是唯一安全且通用的由表达式 `e` 构造一个 `T` 类型的值的表达式。

强制转换的写法 `T(e)` 和 `(T)e` 既不安全也不通用。

示例

对于内建类型, 构造的写法保护了不发生窄化和重解释

```

void use(char ch, int i, double d, char* p, long long lng)
{
    int x1 = int{ch};      // OK, 但多余
    int x2 = int{d};      // 错误: double->int 窄化; 如果需要的话应使用强制转换
    int x3 = int{p};      // 错误: 指针->int; 如果确实需要的话应使用 reinterpret_cast
    int x4 = int{lng};   // 错误: long long->int 窄化; 如果需要的话应使用强制转换
}

```

```

int y1 = int(ch);      // OK, 但多余
int y2 = int(d);      // 不好: double->int 窄化; 如果需要的话应使用强制转换
int y3 = int(p);      // 不好: 指针->int; 如果确实需要的话应使用 reinterpret_cast
int y4 = int(lng);    // 不好: long long->int 窄化; 如果需要的话应使用强制转换

int z1 = (int)ch;     // OK, 但多余
int z2 = (int)d;      // 不好: double->int 窄化; 如果需要的话应使用强制转换
int z3 = (int)p;      // 不好: 指针->int; 如果确实需要的话应使用 reinterpret_cast
int z4 = (int)lng;    // 不好: long long->int 窄化; 如果需要的话应使用强制转换
}

```

整数和指针之间的转换，在使用 `T(e)` 和 `(T)e` 时是由实现定义的，而且在不同整数和指针大小的平台之间不可移植。

注解

[避免强制转换](#) (显式类型转换)，如果必须要做的话[优先采用具名强制转换](#)。

注解

当没有歧义时，可以不写 `T{e}` 中的 `T`。

```

complex<double> f(complex<double>);

auto z = f({2*pi,1});

```

注解

对象构造语法是最通用的[初始化式语法](#)。

例外

`std::vector` 和其他的容器是在 `{}` 作为对象构造语法之前定义的。

考虑：

```

vector<string> vs {10};                                // 十个空字符串
vector<int> vi1 {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; // 十个元素 1..10
vector<int> vi2 {10};                                 // 一个值为 10 的元素

```

如何得到包含十个默认初始化的 `int` 的 `vector`？

```
vector<int> v3(10); // 十个值为 0 的元素
```

使用 `(10)` 而不是 `{10}` 作为元素数量是一种约定（源于 1980 年代早期），很难改变，但仍然是一个设计错误：对于元素类型与元素数量可能发生混淆的容器，必须解决其中的歧义。

约定的方案是将 `{10}` 解释为单个元素的列表，而用 `(10)` 来指定大小。

不应当在新代码中重复这个错误。

可以定义一个类型来表示元素的数量：

```
struct Count { int n; };

template<typename T>
class vector {
public:
    vector(Count n); // n 个默认构造的元素
    vector(initializer_list<T> init); // init.size() 个元素
    // ...
};

vector<int> v1{10};
vector<int> v2{Count{10}};
vector<Count> v3{Count{10}}; // 这里仍有一个很小的问题
```

剩下的主要问题就是为 `Count` 找个合适的名字了。

强制实施

标记 C 风格的 `(T)e` 和函数式风格的 `T(e)` 强制转换。

ES.65: 不要解引用无效指针

理由

解引用如 `nullptr` 这样的无效指针是未定义的行为，通常会导致程序立刻崩溃，产生错误结果，或者内存被损坏。

注解

本条规则是显然并且广为知晓的语言规则，但其可能难于遵守。

这需要良好的编码风格，程序库支持，以及静态分析来消除违反情况而不耗费大量开销。

这正是 [C++ 的类型和资源安全性模型](#) 中所讨论的主要部分。

参见：

- 使用 [RAII](#) 以避免生存期问题。
- 使用 [unique_ptr](#) 以避免生存期问题。
- 使用 [shared_ptr](#) 以避免生存期问题。
- 当不可能出现 `nullptr` 时应使用[引用](#)。
- 使用 [not_null](#) 以尽早捕捉到预期外的 `nullptr`。
- 使用[边界剖面配置](#)以避免范围错误。

示例

```

void f()
{
    int x = 0;
    int* p = &x;

    if (condition()) {
        int y = 0;
        p = &y;
    } // p 失效

    *p = 42;           // 不好，若走了上面的分支则 p 无效
}

```

为解决这个问题，要么应当扩展指针打算指代的这个对象的生存期，要么应当缩短指针的生存期（将解引用移动到所指代的对象生存期结束之前进行）。

```

void f1()
{
    int x = 0;
    int* p = &x;

    int y = 0;
    if (condition()) {
        p = &y;
    }

    *p = 42;           // OK, p 可指向 x 或 y，而它们都仍在作用域中
}

```

不幸的是，大多数无效指针问题都更加难于定位且更加难于解决。

示例

```

void f(int* p)
{
    int x = *p; // 不好：如何确定 p 是否有效？
}

```

有大量的这种代码存在。

它们大多数都能工作（经过了大量的测试），但其各自是很难确定 `p` 是否可能为 `nullptr` 的。后果就是，这同样是错误的一大来源。

有许多方案试图解决这个潜在问题：

```

void f1(int* p) // 处理 nullptr
{
    if (!p) {
        // 处理 nullptr (分配, 返回, 抛出, 使 p 指向什么, 等等
    }
    int x = *p;
}

```

测试 `nullptr` 的做法有两个潜在的问题：

- 当遇到 `nullptr` 时应当做什么并不总是明确的
- 其测试可能多余并且相对比较昂贵
- 这个测试是为了保护某种违例还是所需逻辑的一部分并不明显

```
void f2(int* p) // 声称 p 不应当为 nullptr
{
    Assert(p);
    int x = *p;
}
```

这样，仅当打开了断言检查时才会有所耗费，而且会向编译器/分析器提供有用的信息。当 C++ 出现契约的直接支持后，还可以做的更好：

```
void f3(int* p) // 声称 p 不应当为 nullptr
[[expects: p]]
{
    int x = *p;
}
```

或者，还可以使用 `gs1::not_null` 来保证 `p` 不为 `nullptr`。

```
void f(not_null<int*> p)
{
    int x = *p;
}
```

这些只是关于 `nullptr` 的处理办法。

要知道还有其他出现无效指针的方式。

示例

```
void f(int* p) // 老代码，没使用 owner
{
    delete p;
}

void g() // 老代码：使用了裸 new
{
    auto q = new int{7};
    f(q);
    int x = *q; // 不好：解引用了无效指针
}
```

示例

```
void f()
{
    vector<int> v(10);
    int* p = &v[5];
    v.push_back(99); // 可能重新分配 v 中的元素
    int x = *p; // 不好：解引用了潜在的无效指针
}
```

强制实施

本条规则属于[生存期安全性剖面配置](#)

- 当对指向已经超出作用域的对象的指针进行解引用时进行标记
- 当对可能已经通过赋值 `nullptr` 而无效的指针进行解引用时进行标记
- 当对可能已经因 `delete` 而无效的指针进行解引用时进行标记
- 当对指向可能已经失效的容器元素的指针进行解引用时进行标记

ES stmt: 语句

语句控制了控制的流向（除了函数调用和异常抛出，它们是表达式）。

ES.70: 面临选择时，优先采用 `switch` 语句而不是 `if` 语句

理由

- 可读性。
- 效率：`switch` 与常量进行比较，且通常比一个 `if-then-else` 链中的一系列测试获得更好的优化。
- `switch` 可以启用某种启发式的一致性检查。例如，是否某个 `enum` 的所有值都被覆盖了？如果没有的话，是否存在 `default`？

示例

```
void use(int n)
{
    switch (n) { // 好
        case 0:
            // ...
            break;
        case 7:
            // ...
            break;
        default:
            // ...
            break;
    }
}
```

要好于：

```
void use2(int n)
{
    if (n == 0) // 不好：以 if-then-else 链和一组常量进行比较
        // ...
    else if (n == 7)
        // ...
}
```

强制实施

对以 `if-then-else` 链条（仅）和常量进行比较的情况进行标记。

ES.71: 面临选择时，优先采用范围式 `for` 语句而不是普通 `for` 语句

理由

可读性。避免错误。效率。

示例

```
for (gsl::index i = 0; i < v.size(); ++i)    // 不好
    cout << v[i] << '\n';

for (auto p = v.begin(); p != v.end(); ++p)    // 不好
    cout << *p << '\n';

for (auto& x : v)    // OK
    cout << x << '\n';

for (gsl::index i = 1; i < v.size(); ++i) // 接触了两个元素：无法作为范围式的 for
    cout << v[i] + v[i - 1] << '\n';

for (gsl::index i = 0; i < v.size(); ++i) // 可能具有副作用：无法作为范围式的 for
    cout << f(v, &v[i]) << '\n';

for (gsl::index i = 0; i < v.size(); ++i) { // 循环体中混入了循环变量：无法作为范围式
for
    if (i % 2 != 0)
        cout << v[i] << '\n'; // 输出奇数元素
}
```

人类或优良的静态分析器也许可以确定，其实在 `f(v, &v[i])` 中的 `v` 的上并不真的存在副作用，因此这个循环可以被重写。

在循环体中“混入循环变量”的情况通常是最好的进行避免的。

注解

不要在范围式 `for` 循环中使用昂贵的循环变量副本：

```
for (string s : vs) // ...
```

这将会对 `vs` 中的每个元素复制给 `s`。这样好一点：

```
for (string& s : vs) // ...
```

更好的做法是，当循环变量不会被修改或复制时：

```
for (const string& s : vs) // ...
```

强制实施

查看循环，如果一个传统的循环仅会查看序列中的各个元素，而且其对这些元素所做的事中没有发生副作用，则将该循环重写为范围式的 `for` 循环。

ES.72: 当存在显然的循环变量时，优先采用 `for` 语句而不是 `while` 语句

理由

可读性：循环的全部逻辑都“直观可见”。循环变量的作用域是有限的。

示例

```
for (gsl::index i = 0; i < vec.size(); i++) {  
    // 干活  
}
```

示例，不好

```
int i = 0;  
while (i < vec.size()) {  
    // 干活  
    i++;  
}
```

强制实施

???

ES.73: 当没有显然的循环变量时，优先采用 `while` 语句而不是 `for` 语句

理由

可读性。

示例

```
int events = 0;  
for (; wait_for_event(); ++events) { // 不好，含糊  
    // ...  
}
```

这个“事件循环”会误导人，计数器 `events` 跟循环条件 (`wait_for_event()`) 并没有任何关系。
更好的做法是

```
int events = 0;  
while (wait_for_event()) { // 更好  
    ++events;  
    // ...  
}
```

强制实施

对和 `for` 的条件不相关的 `for` 初始化式和 `for` 增量部分进行标记。

ES.74: 优先在 `for` 语句的初始化部分中声明循环变量

参见 [ES.6](#)

ES.75: 避免使用 `do` 语句

理由

可读性，避免错误。

其终止条件处于尾部（而这可能会被忽略），且其条件不会在第一时间进行检查。

示例

```
int x;
do {
    cin >> x;
    // ...
} while (x < 0);
```

注解

确实有一些天才的例子中，`do` 语句是更简洁的方案，但有问题的更多。

强制实施

标记 `do` 语句。

ES.76: 避免 `goto`

理由

可读性，避免错误。存在对于人类更好的控制结构；`goto` 是用于机器生成的代码的。

例外

跳出嵌套循环。

这种情况下应当总是向前跳出。

```
for (int i = 0; i < imax; ++i)
    for (int j = 0; j < jmax; ++j) {
        if (a[i][j] > elem_max) goto finished;
        // ...
    }
finished:
// ...
```

示例，不好

有相当数量的代码采用 C 风格的 `goto-exit` 惯用法：

```
void f()
{
    // ...
    goto exit;
    // ...
    goto exit;
    // ...
exit:
    // ... 公共的清理代码 ...
}
```

这是对析构函数的一种专门模仿。

应当将资源声明为带有清理的析构函数的包装类。

如果你由于某种原因无法用析构函数来处理所使用的各个变量的清理工作，

请考虑用 `gsl::finally()` 作为 `goto exit` 的一种简洁且更加可靠的替代方案。

强制实施

- 标记 `goto`。更好的做法是标记出除了从嵌套内层循环中跳出到紧跟一组嵌套循环之后的语句的 `goto` 以外的所有 `goto`。

ES.77: 尽量减少循环中使用的 `break` 和 `continue`

理由

在不平凡的循环体中，容易忽略掉 `break` 或 `continue`。

循环中的 `break` 和 `switch` 语句中的 `break` 的含义有很大的区别，

(而且循环中可以有 `switch` 语句，`switch` 的 `case` 中也可以有循环)。

示例

```
switch(x) {
    case 1 :
        while /* 某种条件 */ {
            // ...
            break;
        } // 哟！打算 break switch 还是 break while？
    case 2 :
        // ...
        break;
}
```

替代方案

通常，需要 `break` 的循环都是作为一个函数（算法）的良好候选者，其 `break` 将会变为 `return`。

```
// 原始代码: break 内部循环
void use1()
{
    std::vector<T> vec = {/* 初始化为一些值 */};
    T value;
    for (const T item : vec) {
        if /* 某种条件 */ {
            value = item;
        }
    }
}
```

```

        break;
    }
}

/* 然后对 value 做些事 */

}

// 这样更好：创建一个函数使其从循环中返回
T search(const std::vector<T> &vec)
{
    for (const T &item : vec) {
        if /* 某种条件 */) return item;
    }
    return T(); // 默认值
}

void use2()
{
    std::vector<T> vec = {/* 初始化为一些值 */};
    T value = search(vec);
    /* 然后对 value 做些事 */
}

```

通常，使用 `continue` 的循环都可以等价且同样简洁地用 `if` 语句来表达。

```

for (int item : vec) { // 不好
    if (item%2 == 0) continue;
    if (item == 5) continue;
    if (item > 10) continue;
    /* 对 item 做些事 */
}

for (int item : vec) { // 好
    if (item%2 != 0 && item != 5 && item <= 10) {
        /* 对 item 做些事 */
    }
}

```

注解

如果你确实要打断一个循环，使用 `break` 通常比使用诸如[修改循环变量](#)或 `goto` 等其他方案更好：

强制实施

???

ES.78: 不要依靠 `switch` 语句中的隐含直落行为

理由

总是以 `break` 来结束非空的 `case`。意外地遗漏 `break` 是一种相当常见的 BUG。
蓄意的控制直落（fall through）是维护的噩梦，应该罕见并被明确标示出来。

示例

```
switch (eventType) {  
    case Information:  
        update_status_bar();  
        break;  
    case Warning:  
        write_event_log();  
        // 不好 - 隐式的控制直落  
    case Error:  
        display_error_window();  
        break;  
}
```

单个语句带有多个 `case` 标签是可以的：

```
switch (x) {  
    case 'a':  
    case 'b':  
    case 'f':  
        do_something(x);  
        break;  
}
```

在 `case` 标签中使用返回语句也是可以的：

```
switch (x) {  
    case 'a':  
        return 1;  
    case 'b':  
        return 2;  
    case 'c':  
        return 3;  
}
```

例外

在罕见的直落被视为合适行为的情况下。应当明确标示，并使用 `[[fallthrough]]` 标注：

```
switch (eventType) {  
    case Information:  
        update_status_bar();  
        break;  
    case Warning:  
        write_event_log();  
        [[fallthrough]];  
    case Error:  
        display_error_window();  
        break;  
}
```

注解

强制实施

对所有从非空的 `case` 隐式发生的直落进行标记。

ES.79: `default` (仅) 用于处理一般情况

理由

代码清晰性。

提升错误检测的机会。

示例

```
enum E { a, b, c, d };

void f1(E x)
{
    switch (x) {
        case a:
            do_something();
            break;
        case b:
            do_something_else();
            break;
        default:
            take_the_default_action();
            break;
    }
}
```

此处很明显有一种默认的动作，而情况 `a` 和 `b` 则是特殊情况。

示例

不过当不存在默认动作而只想处理特殊情况时怎么办呢？

这种情况下，应当使用空的 `default`，否则没办法知道你确实处理了所有情况：

```
void f2(E x)
{
    switch (x) {
        case a:
            do_something();
            break;
        case b:
            do_something_else();
            break;
        default:
            // 其他情况无需动作
            break;
    }
}
```

如果没有 `default` 的话，维护者以及编译器可能会合理地假定你有意处理所有情况：

```
void f2(E x)
{
    switch (x) {
        case a:
            do_something();
            break;
        case b:
        case c:
            do_something_else();
            break;
    }
}
```

你是忘记了情况 `d` 还是故意遗漏了它？

当有人向枚举中添加一种情况，而又未能对每个针对这些枚举符的 `switch` 中添加时，容易出现这种遗忘 `case` 的情况。

强制实施

针对某个枚举的 `switch` 语句，若其未能处理其所有枚举符且没有 `default`，则对其进行标记。这样做对于某些代码库可能会产生大量误报；此时，可以仅标记那些处理了大多数情况而不是所有情况的 `switch` 语句
(这正是第一个 C++ 编译器曾经的策略)。

ES.84: 不要试图声明没有名字的局部变量

理由

没有这种东西。

我们眼里看起来像是个无名变量的东西，对于编译器来说是一条由一个将会立刻离开作用域的临时对象所组成的语句。

示例，不好

```
void f()
{
    lock_guard<mutex>{mx}; // 不好
    // ...
}
```

这里声明了一个无名的 `lock_guard` 对象，它将在分号处立刻离开作用域。

这并不是一种少见的错误。

特别是，这个特别的例子会导致很难发觉的竞争条件。

注解

无名函数实参是没问题的。

强制实施

标记出仅有临时对象的语句。

ES.85: 让空语句显著可见

理由

可读性。

示例

```
for (i = 0; i < max; ++i); // 不好: 空语句很容易被忽略
v[i] = f(v[i]);

for (auto x : v) {           // 好多了
    // 空
}
v[i] = f(v[i]);
```

强制实施

对并非块语句且不包含注释的空语句进行标记。

ES.86: 避免在原生的 `for` 循环中修改循环控制变量

理由

循环控制的第一行应当允许对循环中所发生的事情进行正确的推理。同时在循环的重复表达式和循环体之中修改循环计数器，是发生意外和 BUG 的一种经常性来源。

示例

```
for (int i = 0; i < 10; ++i) {
    // 未改动 i -- ok
}

for (int i = 0; i < 10; ++i) {
    //
    if /* 某种情况 */) ++i; // 不好
    //
}

bool skip = false;
for (int i = 0; i < 10; ++i) {
    if (skip) { skip = false; continue; }
    //
    if /* 某种情况 */) skip = true; // 有改善: 为两个概念使用了两个变量。
    //
}
```

强制实施

如果变量在循环控制的重复表达式和循环体中都潜在地进行更新（存在非 `const` 使用），则进行标记。

ES.87: 请勿在条件上添加多余的 `==` 或 `!=`

理由

这样可避免啰嗦，并消除了发生某些错误的机会。
有助于使代码风格保持一致性和协调性。

示例

根据定义，`if` 语句，`while` 语句，以及 `for` 语句中的条件，选择 `true` 或 `false` 的取值。
数值与 `0` 相比较，指针值与 `nullptr` 相比较。

```
// 这些都表示“当 p 不是 nullptr 时”
if (p) { ... } // 好
if (p != 0) { ... } // !=0 是多余的；不好：不要对指针用 0
if (p != nullptr) { ... } // !=nullptr 是多余的，不建议如此
```

通常，`if (p)` 可解读为“如果 `p` 有效”，这正是程序员意图的直接表达，
而 `if (p != nullptr)` 则只是一种啰嗦的变通写法。

示例

这条规则对于把声明式用作条件时尤其有用

```
if (auto pc = dynamic_cast<Circle>(ps)) { ... } // 执行是按照 ps 指向某种 Circle 来
进行的，好

if (auto pc = dynamic_cast<Circle>(ps); pc != nullptr) { ... } // 不建议如此
```

示例

要注意，条件中会实施向 `bool` 的隐式转换。

例如：

```
for (string s; cin >> s; ) v.push_back(s);
```

这里会执行 `istream` 的 `operator bool()`。

注解

明确地将整数和 `0` 进行比较通常并非是多余的。
因为（与指针和布尔值相反），整数通常都具有超过两个的有效值。
此外 `0`（零）还经常会用于代表成功。
因此，最好明确地进行比较。

```
void f(int i)
{
    if (i) // 可疑
    // ...
    if (i == success) // 可能更好
    // ...
}
```

一定要记住整数可以有超过两个值。

示例，不好

众所周知，

```
if(strcmp(p1, p2)) { ... } // 这两个 C 风格的字符串相等吗？（错误！）
```

是一种常见的新手错误。

如果使用 C 风格的字符串，那么就必须好好了解 `<cstring>` 中的函数。

即便冗余地写为

```
if(strcmp(p1, p2) != 0) { ... } // 这两个 C 风格的字符串相等吗？（错误！）
```

也不会有效果。

注解

表达相反的条件的最简单的方式就是使用一次取反：

```
// 这些都表示“当 p 为 nullptr 时”
if (!p) { ... } // 好
if (p == 0) { ... } // ==0 是多余的；不好：不要对指针用 0
if (p == nullptr) { ... } // ==nullptr 是多余的，不建议如此
```

强制实施

容易，仅需检查条件中多余的 `!=` 和 `==` 的使用即可。

算术

ES.100: 不要进行有符号和无符号混合运算

理由

避免错误的结果。

示例

```
int x = -3;
unsigned int y = 7;

cout << x - y << '\n'; // 无符号结果，可能是 4294967286
cout << x + y << '\n'; // 无符号结果：4
cout << x * y << '\n'; // 无符号结果，可能是 4294967275
```

在更实际的例子中，这种问题更难于被发现。

注解

不幸的是，C++ 使用有符号整数作为数组下标，而标准库使用无符号整数作为容器下标。

这妨碍了一致性。使用 `gsl::index` 来作为下标类型；[参见 ES.107](#)。

强制实施

- 编译器已知这种情况，有些时候会给出警告。
- (避免噪声) 有符号/无符号的混合比较，若其一个实参是 `sizeof` 或调用容器的 `.size()` 而另一个是 `ptrdiff_t`，则不要进行标记。

ES.101: 使用无符号类型进行位操作

理由

无符号类型支持位操作而没有符号位的意外。

示例

```
unsigned char x = 0b1010'1010;
unsigned char y = ~x; // y == 0b0101'0101;
```

注解

无符号类型对于模算术也很有用。

不过，如果你想要进行模算术时，

应当按需添加代码注释以注明所依赖的回绕行为，因为这样的代码会让许多程序员感觉意外。

强制实施

- 一般来说基本不可能，因为标准库也使用了无符号下标。
???

ES.102: 使用有符号类型进行算术运算

理由

因为大多数算术都假定是有符号的；

当 `y > x` 时，`x - y` 都会产生负数，除了罕见的情况下你确实需要模算术。

示例

当你不期望时，无符号算术会产生奇怪的结果。

这在混合有符号和无符号算术时有其如此。

```
template<typename T, typename T2>
T subtract(T x, T2 y)
{
    return x - y;
}

void test()
{
    int s = 5;
    unsigned int us = 5;
    cout << subtract(s, 7) << '\n'; // -2
    cout << subtract(us, 7u) << '\n'; // 4294967294
    cout << subtract(s, 7u) << '\n'; // -2
    cout << subtract(us, 7) << '\n'; // 4294967294
    cout << subtract(s, us + 2) << '\n'; // -2
```

```
    cout << subtract(us, s + 2) << '\n'; // 4294967294
}
```

我们这次非常明确发生了什么。

但要是你见到 `us - (s + 2)` 或者 `s += 2; ...; us - s` 时，你确实能够预计到打印的结果将是 `4294967294` 吗？

例外

如果你确实需要模算术的话就使用无符号类型——

根据需要为其添加代码注释以说明其依赖溢出行为，因为这样的代码会让许多程序员感觉意外。

示例

标准库使用无符号类型作为下标。

内建数组则用有符号类型作为下标。

这不可避免地带来了意外（以及 BUG）。

```
int a[10];
for (int i = 0; i < 10; ++i) a[i] = i;
vector<int> v(10);
// 比较有符号和无符号数；有些编译器会警告，但我们不能警告
for (gsl::index i = 0; i < v.size(); ++i) v[i] = i;

int a2[-2];           // 错误：负的大小

// OK，但 int 的数值（4294967294）过大，应当会造成一个异常
vector<int> v2(-2);
```

使用 `gsl::index` 作为下标类型；[参见 ES.107](#)。

强制实施

- 对混合有符号和无符号算术进行标记。
- 对将无符号算术的结果作为有符号数赋值或打印进行标记。
- 对负数字面量（比如 `-2`）用作容器下标进行标记。
- （避免噪声）有符号/无符号的混合比较，若其一个实参是 `sizeof` 或调用容器的 `.size()` 而另一个是 `ptrdiff_t`，则不要进行标记。

ES.103: 避免上溢出

理由

上溢出通常会让数值算法变得没有意义。

将值增加超过其最大值将导致内存损坏和未定义的行为。

示例，不好

```
int a[10];
a[10] = 7;    // 不好，数组边界上溢出

for (int n = 0; n <= 10; ++n)
    a[n] = 9;    // 不好，数组边界上溢出
```

示例，不好

```
int n = numeric_limits<int>::max();  
int m = n + 1; // 不好，数值上溢出
```

示例，不好

```
int area(int h, int w) { return h * w; }  
  
auto a = area(10'000'000, 100'000'000); // 不好，数值上溢出
```

例外

如果你确实需要模算术的话就使用无符号类型。

替代方案: 对于可以负担一些开销的关键应用，可以使用带有范围检查的整数和/或浮点类型。

强制实施

???

ES.104: 避免下溢出

理由

将值减小超过其最小值将导致内存损坏和未定义的行为。

示例，不好

```
int a[10];  
a[-2] = 7; // 不好  
  
int n = 101;  
while (n--)  
    a[n - 1] = 9; // 不好（两次）
```

例外

如果你确实需要模算术的话就使用无符号类型。

强制实施

???

ES.105: 避免除整数零

理由

其结果是未定义的，很可能导致程序崩溃。

注解

这同样适用于 %。

示例，不好

```
int divide(int a, int b)
{
    // 不好，应当进行检查（比如一条前条件）
    return a / b;
}
```

示例，好

```
int divide(int a, int b)
{
    // 好，通过前条件进行处置（并当 C++ 支持契约后可以进行替换）
    Expects(b != 0);
    return a / b;
}

double divide(double a, double b)
{
    // 好，通过换为使用 double 来处置
    return a / b;
}
```

替代方案: 对于可以负担一些开销的关键应用，可以使用带有范围检查的整数和/或浮点类型。

强制实施

- 对以可能为零的整型值的除法进行标记。

ES.106: 不要试图用 `unsigned` 来防止负数值

理由

选用 `unsigned` 意味着对于包括模算术在内的整数的常规行为的许多改动，它将抑制掉与溢出有关的警告，并打开了与混合符号相关的错误的大门。使用 `unsigned` 并不会真正消除负数值的可能性。

示例

```
unsigned int u1 = -2;    // 合法: u1 的值为 4294967294
int i1 = -2;
unsigned int u2 = i1;    // 合法: u2 的值为 4294967294
int i2 = u2;            // 合法: i2 的值为 -2
```

真实代码中很难找出这样的（完全合法的）语法构造的问题，而它们是许多真实世界错误的来源。

考虑：

```
unsigned area(unsigned height, unsigned width) { return height*width; } // [参见]
(#Ri-expects)
// ...
int height;
cin >> height;
auto a = area(height, 2);    // 当输入为 -2 时 a 为 4294967292
```

记住把 `-1` 赋值给 `unsigned int` 会变成最大的 `unsigned int`。
而且，由于无符号算术是模算术，其乘法并不会溢出，而是会发生回绕。

示例

```
unsigned max = 100000;      // “不小心写错了”，应该写 10'000
unsigned short x = 100;
while (x < max) x += 100; // 无限循环
```

要是 `x` 是个有符号的 `short` 的话，我们就会得到有关溢出的未定义行为的警告了。

替代方案

- 使用有符号整数并检查 `x >= 0`
- 使用某个正整数类型
- 使用某个整数子值域类型
- `Assert(-1 < x)`

例如

```
struct Positive {
    int val;
    Positive(int x) :val{x} { Assert(0 < x); }
    operator int() { return val; }
};

int f(Positive arg) { return arg; }

int r1 = f(2);
int r2 = f(-2); // 抛出异常
```

注解

???

强制实施

参见 ES.100 的强制实施。

ES.107: 不要对下标使用 `unsigned`，优先使用 `gsl::index`

理由

避免有符号和无符号混乱。

允许更好的优化。

允许更好的错误检测。

避免 `auto` 和 `int` 有关的陷阱。

示例，不好

```
vector<int> vec = /*...*/;

for (int i = 0; i < vec.size(); i += 2)                                // 可能不够大
    cout << vec[i] << '\n';
for (unsigned i = 0; i < vec.size(); i += 2)                            // 有风险的回绕
```

```

cout << vec[i] << '\n';
for (auto i = 0; i < vec.size(); i += 2) // 可能不够大
    cout << vec[i] << '\n';
for (vector<int>::size_type i = 0; i < vec.size(); i += 2) // 嘴巴
    cout << vec[i] << '\n';
for (auto i = vec.size()-1; i >= 0; i -= 2) // BUG
    cout << vec[i] << '\n';
for (int i = vec.size()-1; i >= 0; i -= 2) // 可能不够大
    cout << vec[i] << '\n';

```

示例，好

```

vector<int> vec = /*...*/;

for (gs1::index i = 0; i < vec.size(); i += 2) // ok
    cout << vec[i] << '\n';
for (gs1::index i = vec.size()-1; i >= 0; i -= 2) // ok
    cout << vec[i] << '\n';

```

注解

内建数组允许有符号的下标。

标准库容器使用无符号的下标。

因此没有完美的兼容解决方案（除非将来某一天，标准库容器改为使用有符号下标了）。

鉴于无符号和混合符号方面的已知问题，最好坚持使用（有符号）并且足够大的整数，而 `gs1::index` 保证了这点。

示例

```

template<typename T>
struct My_container {
public:
    // ...
    T& operator[](gs1::index i); // 不是 unsigned
    // ...
};

```

示例

??? 演示改进后的代码生成和潜在可进行的错误检查 ???

替代方案

可以代之以

- 使用算法
- 使用基于范围的 `for`
- 使用迭代器或指针

强制实施

- 非常麻烦，因为标准库容器已经搞错了。
- (避免噪声) 有符号/无符号的混合比较，若其中一个实参是 `sizeof` 或调用容器的 `.size()` 而另一个是 `ptrdiff_t`，则不要进行标记。

Per: 性能

??? 这一节应该放在主指南中吗 ???

本章节所包含的规则对于需要高性能和低延迟的人们很有价值。

就是说，这些规则是有关如何在可预测的短时段中尽可能使用更短时间和更少资源来完成任务的。

本章节中的规则要比（绝大）多数应用程序所需要的规则更多限制并更有侵入性。

请勿在一般的代码中盲目地尝试遵循这些规则：要达成低延迟的目标是需要进行一些额外工作的。

性能规则概览：

- [Per.1: 请勿进行无理由的优化](#)
- [Per.2: 请勿进行不成熟的优化](#)
- [Per.3: 请勿对非性能关键的代码进行优化](#)
- [Per.4: 不能假定复杂代码一定比简单代码更快](#)
- [Per.5: 不能假定低级代码一定比高级代码更快](#)
- [Per.6: 请勿不进行测量就作出性能判断](#)
- [Per.7: 设计应当允许优化](#)
- [Per.10: 依赖静态类型系统](#)
- [Per.11: 把计算从运行时转移到编译期](#)
- [Per.12: 消除多余的别名](#)
- [Per.13: 消除多余的间接](#)
- [Per.14: 最小化分配和回收的次数](#)
- [Per.15: 请勿在关键逻辑分支中进行分配](#)
- [Per.16: 使用紧凑的数据结构](#)
- [Per.17: 在时间关键的结构中应当先声明最常用的成员](#)
- [Per.18: 空间即时间](#)
- [Per.19: 进行可预测的内存访问](#)
- [Per.30: 避免在关键路径中进行上下文切换](#)

Per.1: 请勿进行无理由的优化

理由

如果没有必要优化的话，这样做的结果就是更多的错误和更高的维护成本。

注解

一些人作出优化只是出于习惯或者因为感觉这很有趣。

???

Per.2: 请勿进行不成熟的优化

理由

经过精心优化的代码通常比未优化的代码更大而且更难修改。

???

Per.3: 请勿对非性能关键的代码进行优化

理由

对程序中并非性能关键的部分进行的优化，对于系统性能是没有效果的。

注解

如果你的程序要耗费大量时间来等待 Web 或人的操作的话，对内存中的计算进行优化可能是没什么用处的。

换个角度来说：如果你的程序花费处理时间的 4% 来计算 A 而花费 40% 的时间来计算 B，那对 A 的 50% 的改进其影响只能和 B 的 5% 的改进相比。（如果你甚至不知道 A 或 B 到底花费了多少时间，参见 [Per.1](#) 和 [Per.2](#)。）

Per.4: 不能假定复杂代码一定比简单代码更快

理由

简单的代码可能会非常快。优化器在简单代码上有时候会发生奇迹。

示例，好

```
// 清晰表达意图，快速执行

vector<uint8_t> v(100000);

for (auto& c : v)
    c = ~c;
```

示例，不好

```
// 试图更快，但通常会更慢

vector<uint8_t> v(100000);

for (size_t i = 0; i < v.size(); i += sizeof(uint64_t)) {
    uint64_t& quad_word = *reinterpret_cast<uint64_t*>(&v[i]);
    quad_word = ~quad_word;
}
```

注解

???

???

Per.5: 不能假定低级代码一定比高级代码更快

理由

低级代码有时候会妨碍优化。优化器在高级代码上有时候会发生奇迹。

注解

???

???

Per.6: 请勿不进行测量就作出性能评断

理由

性能领域充斥各种错误认识和伪习俗。

现代的硬件和优化器并不遵循这些幼稚的假设；即便是专家也会经常感觉意外。

注解

要进行高质量的性能测量是很难的，而且需要采用专门的工具。

注解

有些使用了 Unix 的 `time` 或者标准库的 `<chrono>` 的简单的微基准测量，有助于打破大多数明显的错误认识。

如果确实无法精确地测量完整系统的话，至少也要尝试对一些关键操作和算法进行测量。

性能剖析可以帮你发现系统的哪些部分是性能关键的。

你很可能感觉意外。

???

Per.7: 设计应当允许优化

理由

因为我们经常需要对最初的设计进行优化。

因为忽略后续改进的可能性的设计是很难修改的。

示例

来自 C (以及 C++) 的标准：

```
void qsort (void* base, size_t num, size_t size, int (*compar)(const void*, const void*));
```

什么情况会需要对内存进行排序呢？

时即上，我们需要对元素序列进行排序，通常它们存储于容器之中。

对 `qsort` 的调用抛弃了许多有用的信息（比如元素的类型），强制用户对其已知的信息

进行重复（比如元素的大小），并强制用户编写额外的代码（比如用于比较 `double` 的函数）。

这蕴含了程序员工作量的增加，易错，并剥夺了编译器为优化所需的信息。

```
double data[100];
// ... 填充 a ...

// 对从地址 data 开始的 100 块 sizeof(double) 大小
// 的内存，用由 compare_doubles 所定义的顺序进行排序
qsort(data, 100, sizeof(double), compare_doubles);
```

从接口设计的观点来看，`qsort` 抛弃了有用的信息。

这样做可以更好 (C++98)：

```
template<typename Iter>
void sort(Iter b, Iter e); // sort [b:e)

sort(data, data + 100);
```

这里，我们利用了编译器关于数组大小，元素类型，以及如何对 `double` 进行比较的知识。

而以 C++20 的话，我们还可以做得更好：

```
// sortable 指定了 c 必须是一个
// 可以用 < 进行比较的元素的随机访问序列
void sort(sortable auto& c);

sort(c);
```

其中的关键在于传递充分的信息以便能够选择一个好的实现。

这里给出的几个 `sort` 接口仍然有一个缺憾：

它们隐含地依赖于元素类型定义了小于 (`<`) 运算符。

为使接口完整，我们需要另一个接受比较准则的版本：

```
// 用 r 比较 c 的元素
template<random_access_range R, class C> requires sortable<R, C>
void sort(R&& r, C c);
```

`sort` 的标准库规范提供了这两个版本和其他版本。

注解

不成熟的优化被称为[一切罪恶之源](#)，但这并不是轻视性能的理由。

考虑如何使设计可以为改进而进行修正绝不是不成熟的，而性能改进则是一种常见的改进要求。

我们的目标是建立一组习惯，使缺省情况就能得到高效，可维护，且可优化的代码。

特别是，当你编写并非一次性实现细节的函数时，应当考虑

- 信息传递：
 - 优先采用能够为后续的实现改进带来充分信息的简洁[接口](#)。
要注意信息会通过我们所提供的接口来流入和流出一个实现。
- 紧凑的数据：默认情况[使用紧凑的数据](#)，比如 `std::vector`，[并系统化地进行访问](#)。
如果你觉得需要一种有链接的结构的话，应尝试构造接口使这个结构不会被用户所看到。
- 函数的参数传递和返回：
 - 对可改变和不可变的数据加以区分。
 - 不要把资源管理的负担强加给用户。
 - 不要把假性的间接强加给用户。

对通过接口传递信息采用[符合惯例的方式](#)；

不合惯例的，以及“优化过的”数据传递方式可能会严重影响后续的重新实现。

- 抽象：

不要过度泛化；视图提供每一种可能用法（和误用），并把每个设计决策都（通过编译时或运行时间接）

推迟到后面处理的设计，通常是复杂的，膨胀的，难于理解的混乱体。

应当从具体的例子进行泛化，泛化时要保持性能。

不要仅仅基于关于未来需求的推测而进行泛化。

理想情况是零开销泛化。

- 程序库：

使用带有良好接口的程序库。

当没有可用的程序库时，就构建自己的，并模仿一个好程序库的接口风格。

[标准库](#)是寻找模仿的一个好的第一来源。

- 隔离：

通过将你所选择的接口提供给你的代码来将其和杂乱和老旧风格的代码之间进行隔离。

这有时候称为为有用或必须但杂乱的代码“提供包装”。

不要让不良设计“渗入”你的代码中。

示例

考虑：

```
template<class ForwardIterator, class T>
bool binary_search(ForwardIterator first, ForwardIterator last, const T& val);
```

`binary_search(begin(c), end(c), 7)` 能够得出 `7` 是否在 `c` 之中。

不过，它无法得出 `7` 在何处，或者是否多于一个 `7`。

有时候仅把最小数量的信息传递回来（如这里的 `true` 或 `false`）是足够的，但一个好的接口会向调用方传递其所需的信息。因此，标准库还提供了

```
template<class ForwardIterator, class T>
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last, const T& val);
```

`lower_bound` 返回第一个匹配元素（如果有）的迭代器，否则返回第一个大于 `val` 的元素的迭代器，找不到这样的元素时，返回 `last`。

不过 `lower_bound` 还是无法为所有用法返回足够的信息，因此标准库还提供了

```
template<class ForwardIterator, class T>
pair<ForwardIterator, ForwardIterator>
equal_range(ForwardIterator first, ForwardIterator last, const T& val);
```

`equal_range` 返回迭代器的 `pair`，指定匹配的第一个和最后一个之后的元素。

```
auto r = equal_range(begin(c), end(c), 7);
for (auto p = r.first; p != r.second; ++p)
    cout << *p << '\n';
```

显然，这三个接口都是以相同的基本代码实现的。
它们不过是将基本的二叉搜索算法表现给用户的三种方式，
包含从最简单（“让简单的事情简单！”）
到返回完整但不总是必要的信息（“不要隐藏有用的信息”）。
自然，构造这样一组接口是需要经验和领域知识的。

注解

接口的构建不要仅匹配你能想到的第一种实现和第一种用例。
一旦第一个初始实现完成后，应当进行复审；一旦它被部署出去，就很难对错误进行补救了。

注解

对效率的需求并不意味着对[底层代码](#)的需求。
高层代码并不必然缓慢或膨胀。

注解

事物都是有成本的。
不要对成本过于偏执（当代计算机真的非常快），
但需要对你所使用的东西的成本的数量级有大致的概念。
例如，应当对
一次内存访问，
一次函数调用，
一次字符串比较，
一次系统调用，
一次磁盘访问，
以及一个通过网络的消息的成本有大致的概念。

注解

如果你只能想到一种实现的话，可能你并没有某种能够设计一个稳定接口的东西。
有可能它只不过是实现细节——不是每段代码都需要一个稳定接口——停下来想一想。
一个有用的问题是
“如果这个操作需要用多个线程实现的话，需要什么样的接口呢？向量化？”

注解

这条规则并不抵触[不要进行不成熟的优化](#)规则。
它对其进行了补充，鼓励程序员在必要的时候，使后续的——适当并且成熟的一——优化能够进行。

强制实施

很麻烦。
也许查找 `void*` 函数参数能够找到妨碍后续优化的接口的例子。

Per.10: 依赖静态类型系统

理由

类型违规，弱类型（比如 `void*`），以及低级代码（比如把序列当作独立字节进行操作）等会让优化器的工作变得困难很多。简单的代码通常比手工打造的复杂代码能够更好地优化。

???

Per.11: 把计算从运行时转移到编译期

理由

- 减少代码大小和运行时间。
- 通过使用常量来避免数据竞争。
- 编译时捕获错误（并因而消除错误处理代码）。

示例

```
double square(double d) { return d*d; }
static double s2 = square(2);      // 旧式代码: 动态初始化

constexpr double ntimes(double d, int n)    // 假定 0 <= n
{
    double m = 1;
    while (n--) m *= d;
    return m;
}
constexpr double s3 {ntimes(2, 3)}; // 现代代码: 编译期初始化
```

像 `s2` 的初始化这样的代码并不少见，尤其是比 `square()` 更复杂一些的初始化更是如此。不过，与 `s3` 的初始化相比，它有两个问题：

- 我们得忍受运行时的一次函数调用的开销
- `s2` 在初始化开始前可能就被某个别的线程访问了。

注意：常量是不可能发生数据竞争的。

示例

考虑一种流行的提供包装类的技术，在包装类自身之中存储小型对象，而把大型对象保存到堆上。

```
constexpr int on_stack_max = 20;

template<typename T>
struct Scoped {      // 在 Scoped 中存储一个 T
    // ...
    T obj;
};

template<typename T>
struct On_heap {     // 在自由存储中存储一个 T
    // ...
    T* objp;
};

template<typename T>
using Handle = typename std::conditional<(sizeof(T) <= on_stack_max),
                                         Scoped<T>,           // 第一种候选
                                         On_heap<T>            // 第二种候选
                                     >::type;

void f()
{
    Handle<double> v1;          // double 在栈中
```

```
Handle<std::array<double, 200>> v2; // array 保存到自由存储里
// ...
}
```

假定 `scoped` 和 `on_heap` 均提供了兼容的用户接口。

这里我们在编译时计算出了最优的类型。

对于选择所要调用的最优函数，也有类似的技术。

注解

理想情况是，不试图在编译期执行所有的代码。

显然，大多数的运算都依赖于输入，因而它们没办法挪到编译期进行，

而除了这种逻辑限制外，实际情况是，复杂的编译期运算会严重增加编译时间，

并使调试变得复杂。

甚至编译期运算也可能使得代码变慢。

这种情况确实罕见，但当把一种通用运算分解为一组优化的子运算时，可能会导致指令高速缓存的效率变差。

强制实施

- 找出可以（但尚不）是 `constexpr` 的简单函数。
- 找出调用时其全部实参均为常量表达式的函数。
- 找出可以为 `constexpr` 的宏。

Per.12: 消除多余的别名

???

Per.13: 消除多余的间接

???

Per.14: 最小化分配和回收的次数

???

Per.15: 请勿在关键逻辑分支中进行分配

???

Per.16: 使用紧凑的数据结构

理由

性能通常都是由内存访问次数所决定的。

???

Per.17: 在时间关键的结构中应当先声明最常用的成员

???

Per.18: 空间即时间

理由

性能通常都是由内存访问次数所决定的。

???

Per.19: 进行可预测的内存访问

理由

性能对于 Cache 的性能非常敏感，而 Cache 算法则更喜欢对相邻数据进行的（通常是线性的）简单访问行为。

示例

```
int matrix[rows][cols];  
  
// 不好  
for (int c = 0; c < cols; ++c)  
    for (int r = 0; r < rows; ++r)  
        sum += matrix[r][c];  
  
// 好  
for (int r = 0; r < rows; ++r)  
    for (int c = 0; c < cols; ++c)  
        sum += matrix[r][c];
```

Per.30: 避免在关键路径中进行上下文切换

???

CP: 并发与并行

我们经常想要我们的计算机同时运行许多任务（或者至少表现为同时运行它们）。

这有许多不同的原因（例如，需要仅用一个处理器等待许多事件，同时处理许多数据流，或者利用大量的硬件设施）

因此也由许多不同的用以表现并发和并行的基本设施。

我们这里将说明使用 ISO 标准 C++ 中用以表现基本并发和并行的设施的原则和规则。

线程是对并发和并行编程支持的机器级别的基础。

使用线程允许互不相关地运行程序的多个部分，

同时共享相同的内存。并发编程很麻烦，

因为在线程之间保护共享的数据说起来容易做起来难。

使现存的单线程代码可以并发执行，

可以通过策略性地添加 `std::async` 或 `std::thread` 这样简单做到，

也可能需要进行完全的重写，这依赖于原始代码是否是以线程友好
的方式编写的。

本文档中的并发/并行规则的设计有三个

目标：

- 有助于编写可以被修改为能够在线程环境中使用
的代码。

- 展示使用标准库所提供的线程原语的简洁，安全的方式。
- 对当并发和并行无法提供所需要的性能增益时应当如何做提供指导。

同样重要的一点，是要注意 C++ 的并发仍然是未完成的工作。C++11 引入了许多核心并发原语，C++14 和 C++17 对它们进行了改进，而且在使 C++ 编写并发程序更加简单的方面仍有许多关注。我们预计这里的一些与程序库相关的指导方针会随着时间有大量的改动。

这一部分需要大量的工作（显然如此）。请注意我们的规则是从相对非专家们入手的。真正的专家们还请稍等；我们欢迎贡献者，但请为正在努力使他们的并发程序正确且高效的大多数程序员着想。

并发和并行规则概览：

- [CP.1: 假定你的代码将作为多线程程序的一部分而运行](#)
- [CP.2: 避免数据竞争](#)
- [CP.3: 最小化可写数据的明确共享](#)
- [CP.4: 以任务而不是线程的角度思考](#)
- [CP.8: 不要为同步而使用 `volatile`](#)
- [CP.9: 只要可行，就使用工具对并发代码进行验证](#)

参见：

- [CP.con: 并发](#)
- [CP.coro: 协程](#)
- [CP.par: 并行](#)
- [CP.mess: 消息传递](#)
- [CP.vec: 向量化](#)
- [CP.free: 无锁编程](#)
- [CP/etc: 其他并发规则](#)

CP.1: 假定你的代码将作为多线程程序的一部分而运行

理由

很难说现在或者未来什么时候会不会需要使用并发。代码是会被重用的。程序的其他使用了线程的部分可能会使用某个未使用线程的程序库。请注意这条规则对于程序库代码来说最紧迫，而对独立的应用程序来说则最不紧迫。不过，久而久之，代码片段可能出现在意想不到的地方。

示例，不好

```

double cached_computation(int x)
{
    // 不好：这些静态变量导致多线程的使用情况中的数据竞争
    static int cached_x = 0.0;
    static double cached_result = COMPUTATION_OF_ZERO;

    if (cached_x != x) {
        cached_x = x;
        cached_result = computation(x);
    }
    return cached_result;
}

```

虽然 `cached_computation` 在单线程环境中可以正确工作，但在多线程环境中，其两个 `static` 变量将导致数据竞争进而发生未定义的行为。

示例，好

```

struct ComputationCache {
    int cached_x = 0;
    double cached_result = COMPUTATION_OF_ZERO;

    double compute(int x) {
        if (cached_x != x) {
            cached_x = x;
            cached_result = computation(x);
        }
        return cached_result;
    }
};

```

这里的缓存作为 `ComputationCache` 对象的成员数据保存，而非共享的静态状态。

这项重构本质上是将关注点委派给了调用方：

单线程程序可能仍会选择采用一个全局的 `ComputationCache` 对象，而多线程程序可能会为每个线程准备一个 `ComputationCache`，或者为任意定义的“上下文”每个准备一个。

重构后的函数不再试图管理 `cached_x` 的分配。

这方面可以看做是对单一职责原则 (SRP) 的一次应用。

在这个特定的例子中，为线程安全性进行的重构同样改进了其在单线程程序中的可用性。

不难想象，某个单线程程序可能需要在程序的不同部分中使用两个 `ComputationCache` 实例，并且它们并不会覆盖掉互相的缓冲数据。

还有其他的一些方法，可以为针对标准多线程环境（就是唯一的并发形式是 `std::thread` 的环境）编写的

代码添加线程安全性：

- 将状态变量标为 `thread_local` 而非 `static`。
- 实现并发控制逻辑，例如，用一个 `static std::mutex` 来保护对这两个 `static` 变量的访问。
- 拒绝在多线程环境中进行构建和/或运行。
- 提供两个实现，一个用在单线程环境中，另一个用在多线程环境中。

例外

永远不会在多线程环境中执行的代码。

要小心的是：有许多例子，“认为”永远不会在多线程程序中执行的代码却真的在多线程程序中执行了，通常是在多年以后。

一般来说，这种程序将导致进行痛苦的移除数据竞争的工作。

因此，确实有意不在多线程环境中执行的代码，应当清晰地进行标注，而且理想情况下应当利用编译或运行时的强制机制来提早检测到这种使用情况。

CP.2: 避免数据竞争

理由

不这样的话，则任何东西都不保证能工作，而且可能出现微妙的错误。

注解

简而言之，当两个线程并发（未同步）地访问同一个对象，且至少一方为写入方（实施某个非 `const` 操作）时，就会出现数据竞争。

有关如何正确使用同步来消除数据竞争的更多信息，请求教于一本有关并发的优秀书籍（参见 [认真学习文献](#)）。

示例，不好

有大量存在数据竞争的例子，其中的一些现在这个时候就运行在产品软件之中。一个非常简单的例子是：

```
int get_id()
{
    static int id = 1;
    return id++;
}
```

这里的增量操作就是数据竞争的一个例子。这可能以许多方式导致发生错误，包括：

- 线程 A 加载 `id` 的值，OS 上下文切换使 A 离开一段时间，其中有其他线程创建了上百个 ID。线程 A 再次允许执行，而 `id` 被写回到那个位置，其值为 A 所读取的 `id` 值加一。
- 线程 A 和 B 同时加载 `id` 并进行增量。它们都将获得相同的 ID。

局部静态变量是数据竞争的一种常见来源。

示例，不好

```

void f(fstream& fs, regex pattern)
{
    array<double, max> buf;
    int sz = read_vec(fs, buf, max); // 从 fs 读取到 buf 中
    gsl::span<double> s {buf};
    // ...
    auto h1 = async([&] { sort(std::execution::par, s); }); // 产生一个进行排序
    的任务
    // ...
    auto h2 = async([&] { return find_all(buf, sz, pattern); }); // 产生一个查找
    匹配的任务
    // ...
}

```

这里，在 `buf` 的元素上有一个（很讨厌的）数据竞争（`sort` 既会读取也会写入）。

所有的数据竞争都很讨厌。

我们这里设法在栈上的数据上造成了数据竞争。

不是所有数据竞争都像这个这样容易找出来的。

示例，不好

```

// 未用锁进行控制的代码

unsigned val;

if (val < 5) {
    // ... 其他线程可能在这里改动 val ...
    switch (val) {
        case 0: // ...
        case 1: // ...
        case 2: // ...
        case 3: // ...
        case 4: // ...
    }
}

```

这里，若编译器不知道 `val` 可能改变，它最可能将这个 `switch` 实现为一个有五个项目的跳转表。

然后，超出 `[0..4]` 范围的 `val` 值将会导致跳转到程序中的任何可能位置的地址，从那个位置继续执行。

真的，当你遇到数据竞争时什么都可能发生。

实际上可能更为糟糕：通过查看生成的代码，是可以确定这个走偏的跳转针对给定的值将会跳转到哪里的；

这就是一种安全风险。

强制实施

有些事是可能做到的，至少要做一些事。

有一些商用和开源的工具试图处理这种问题，

但要注意的是任何工具解决方案都有其成本和盲点。

静态工具通常会有许多漏报，而动态工具则通常有显著的成本。

我们希望有更好的工具。

使用多个工具可以找到比单个工具更多的问题。

还有其他方式可以缓解发生数据竞争的机会：

- 避免全局数据
- 避免 `static` 变量
- 更多地使用栈上的具体类型（且不要过多地把指针到处传递）
- 更多地使用不可变数据（字面量，`constexpr`，以及 `const`）

CP.3: 最小化可写数据的明确共享

理由

如果不共享可写数据的话，就不会发生数据竞争。

越少进行共享，你就越少有机会忘掉对访问进行同步（而发生数据竞争）。

越少进行共享，你就越少有机会需要等待锁定（并改进性能）。

示例

```
bool validate(const vector<Reading>&);
Graph<Temp_node> temperature_gradients(const vector<Reading>&);
Image altitude_map(const vector<Reading>&);
// ...

void process_readings(const vector<Reading>& surface_readings)
{
    auto h1 = async([&] { if (!validate(surface_readings)) throw Invalid_data{}; });
    auto h2 = async([&] { return temperature_gradients(surface_readings); });
    auto h3 = async([&] { return altitude_map(surface_readings); });
    // ...
    h1.get();
    auto v2 = h2.get();
    auto v3 = h3.get();
    // ...
}
```

没有这些 `const` 的话，我们就必须为潜在的数据竞争而在 `surface_readings` 上的所有异步函数调用进行复审。

使 `surface_readings`（对于这个函数）为 `const` 允许我们仅在函数体代码中进行推理。

注解

不可变数据可以安全并高效地共享。

无须对其进行锁定：不可能在常量上发生数据竞争。

另请参见[CP.mess: 消息传递](#)和[CP.31: 优先采用按值传递](#)。

强制实施

???

CP.4: 以任务而不是线程的角度思考

理由

`thread` 是一种实现概念，一种针对机器的思考方式。

任务则是一种应用概念，有时候你可以使任务和其他任务并发执行。

应用概念更容易进行推理。

理由

```
void some_fun(const std::string& msg)
{
    std::thread publisher([=] { std::cout << msg; });           // 不好：表达性不足
                                                               // 且更易错
    auto pubtask = std::async([=] { std::cout << msg; }); // OK
    // ...
    publisher.join();
}
```

注解

除了 `async()` 之外，标准库中的设施都是底层的，面向机器的，线程和锁层次的设施。这是一种必须的基础，但我们不得不尝试提升抽象的层次：为提供生产率，可靠性，以及性能。这对于采用更高层次的，更加面向应用的程序库（如果可能就建立在标准库设施上）的强有力的理由。

强制实施

???

CP.8: 不要为同步而使用 `volatile`

理由

和其他语言不同，C++ 中的 `volatile` 并不提供原子性，不会在线程之间进行同步，而且不会防止指令重排（无论编译器还是硬件）。它和并发完全没有关系。

示例，不好

```
int free_slots = max_slots; // 当前的对象内存的来源

Pool* use()
{
    if (int n = free_slots--) return &pool[n];
}
```

这里有一个问题：

这在单线程程序中是完全正确的代码，但若有两个线程执行，并且在 `free_slots` 上发生竞争条件时，两个线程就可能拿到相同的值和 `free_slots`。这（显然）是不好的数据竞争，受过其他语言训练的人可能会试图这样修正：

```
volatile int free_slots = max_slots; // 当前的对象内存的来源

Pool* use()
{
    if (int n = free_slots--) return &pool[n];
}
```

这并没有同步效果：数据竞争仍然存在！

C++ 对此的机制是 `atomic` 类型：

```
atomic<int> free_slots = max_slots; // 当前的对象内存的来源

Pool* use()
{
    if (int n = free_slots--) return &pool[n];
}
```

现在的 `--` 操作是原子性的，
而不是可能被另一个线程介入其独立操作之间的读-增量-写序列。

替代方案

在某些其他语言中曾经使用 `volatile` 的地方使用 `atomic` 类型。
为更加复杂的例子使用 `mutex`。

另见

[volatile 的（罕见）恰当用法](#)

CP.9: 只要可行，就使用工具对并发代码进行验证

经验表明，让并发代码正确是特别难的，
而编译期检查、运行时检查和测试在找出并发错误方面，
并没有如同在顺序代码中找出错误时那么有效。
一些微妙的并发错误可能会造成显著的不良后果，包括内存破坏，死锁，和安全漏洞等。

示例

???

注解

线程安全性是一种有挑战性的任务，有经验的程序员通常可以做得更好一些：缓解这些风险的一种重要策略是运用工具。

现存有不少这样的工具，既有商用的也有开源的，既有研究性的也有产品级的。
遗憾的是，人们的需求和约束条件之间有巨大的差别，使我们无法给出特定的建议，
但我们可以提一些：

- 静态强制实施工具：[clang](#)
和一些老版本的 [GCC](#)
都提供了一些针对线程安全性性质的静态代码标注。
统一地采用这项技术可以将许多种类的线程安全性错误变为编译时的错误。
这些代码标注一般都是局部的（将某个特定成员变量标记为又一个特定的互斥体进行防护），
且一般都易于学习使用。但与许多的静态工具一样，它经常会造成漏报；
这些情况应当被发现但却被忽略。
- 动态强制实施工具：[Clang 的 Thread Sanitizer](#)（即 TSAN）
是动态工具的一个强有力的例子：它改变你的程序的构建和执行，向其中添加对内存访问的簿记工作，
精确地识别你的二进制程序的一次特定执行中发生的数据竞争。
使用它的代价在内存上（多数情况为五到十倍），也拖慢 CPU（二到二十倍）。
像这样的动态工具，在集成测试，金丝雀推送，以及在多个线程上操作的单元测试上实施是最好的。

它是与工作负载相关的：一旦 TSAN 识别了一个问题，那它就是实际发生的数据竞争，但它只能识别出一次特定的执行之中发生的竞争。

强制实施

对于特定的应用，应用的构建者来选择哪些支持工具是有价值的。

CP.con: 并发

这个部分所关注的是相对比较专门的通过共享数据进行多线程通信的用法。

- 有关并行算法，参见[并行](#)。
- 有关不使用明确共享的任务间，通信参见[消息传递](#)。
- 有关向量并行代码，参见[向量化](#)。
- 有关无锁编程，参见[无锁](#)。

并发规则概览：

- [CP.20: 使用 RAII，绝不使用普通的 `lock\(\)` / `unlock\(\)`](#)
- [CP.21: 用 `std::lock\(\)` 或 `std::scoped_lock` 来获得多个 `mutex`](#)
- [CP.22: 绝不在持有锁的时候调用未知的代码（比如回调）](#)
- [CP.23: 把联结的 `thread` 看作是有作用域的容器](#)
- [CP.24: 把 `thread` 看作是全局的容器](#)
- [CP.25: 优先采用 `gsl::joining_thread` 而不是 `std::thread`](#)
- [CP.26: 不要 `detach\(\)` 线程](#)
- [CP.31: 少量数据在线程之间按值传递，而不是通过引用或指针传递](#)
- [CP.32: 用 `shared_ptr` 在无关的 `thread` 之间共享所有权](#)
- [CP.40: 最小化上下文切换](#)
- [CP.41: 最小化线程的创建和销毁](#)
- [CP.42: 不要无条件地 `wait`](#)
- [CP.43: 最小化临界区的时间耗费](#)
- [CP.44: 记得为 `lock_guard` 和 `unique_lock` 命名](#)
- [CP.50: `mutex` 要和其所护卫的数据一起定义。一旦可能就使用 `synchronized_value<T>`](#)
- ??? 何时使用 spinlock
- ??? 何时使用 `try_lock()`
- ??? 何时应优先使用 `lock_guard` 而不是 `unique_lock`
- ??? 时序多工
- ??? 何时/如何使用 `new thread`

CP.20: 使用 RAII，绝不使用普通的 `lock()` / `unlock()`

理由

避免源于未释放的锁定的令人讨厌的错误。

示例，不好

```
mutex mtx;

void do_stuff()
{
    mtx.lock();
    // ... 做一些事 ...
    mtx.unlock();
}
```

或早或晚都会有人忘记 `mtx.unlock()`，在 `... 做一些事 ...` 中放入一个 `return`，抛出异常，或者别的什么。

```
mutex mtx;

void do_stuff()
{
    unique_lock<mutex> lck {mtx};
    // ... 做一些事 ...
}
```

强制实施

标记对成员 `lock()` 和 `unlock()` 的调用。???

CP.21: 用 `std::lock()` 或 `std::scoped_lock` 来获得多个 mutex

理由

避免在多个 `mutex` 上造成死锁。

示例

下面将导致死锁：

```
// 线程 1
lock_guard<mutex> lck1(m1);
lock_guard<mutex> lck2(m2);

// 线程 2
lock_guard<mutex> lck2(m2);
lock_guard<mutex> lck1(m1);
```

代之以使用 `lock()`：

```
// 线程 1
lock(m1, m2);
lock_guard<mutex> lck1(m1, defer_lock);
lock_guard<mutex> lck2(m2, defer_lock);

// 线程 2
lock(m2, m1);
lock_guard<mutex> lck2(m2, defer_lock);
lock_guard<mutex> lck1(m1, defer_lock);
```

或者（这样更佳，但仅为 C++17）：

```
// 线程 1
scoped_lock<mutex, mutex> lck1(m1, m2);

// 线程 2
scoped_lock<mutex, mutex> lck2(m2, m1);
```

这样，`thread1` 和 `thread2` 的作者们仍然未在 `mutex` 的顺序上达成一致，但顺序不再是问题了。

注解

在实际代码中，`mutex` 的命名很少便于程序员记得某种有意的关系和有意的获取顺序。

在实际代码中，`mutex` 并不总是便于在连续代码行中依次获取的。

注解

在 C++17 中，可以编写普通的

```
lock_guard lck1(m1, adopt_lock);
```

而让 `mutex` 类型被推断出来。

强制实施

检测多个 `mutex` 的获取。

这一般来说是无法确定的，但找出常见的简单例子（比如上面这个）则比较容易。

CP.22: 绝不在持有锁的时候调用未知的代码（比如回调）

理由

如果不了解代码做了什么，就有死锁的风险。

示例

```
void do_this(Foo* p)
{
    lock_guard<mutex> lck {my_mutex};
    // ... 做一些事 ...
    p->act(my_data);
    // ...
}
```

如果不知道 `Foo::act` 会干什么（可能它是一个虚函数，调用某个还未编写的某个派生类成员），它可能会（递归地）调用 `do_this` 因而在 `my_mutex` 上造成死锁。可能它会在某个别的 `mutex` 上锁定而无法在适当的时间内返回，对任何调用了 `do_this` 的代码造成延迟。

示例

“调用未知代码”问题的一个常见例子是调用了试图在相同对象上进行锁定访问的函数。这种问题通常可以用 `recursive_mutex` 来解决。例如：

```
recursive_mutex my_mutex;

template<typename Action>
void do_something(Action f)
{
    unique_lock<recursive_mutex> lck {my_mutex};
    // ... 做一些事 ...
    f(this);      // f 将会对 *this 做一些事
    // ...
}
```

如果如同其很可能做的那样，`f()` 调用了 `*this` 的某个操作的话，我们就必须保证在调用之前对象的不变式是满足的。

强制实施

- 当持有非递归的 `mutex` 时调用虚函数则进行标记。
- 当持有非递归的 `mutex` 时调用回调则进行标记。

CP.23: 把联结的 `thread` 看作是有作用域的容器

理由

为了维护指针安全性并避免泄漏，需要考虑 `thread` 所使用的指针。

如果 `thread` 联结了，我们可以安全地把指向这个 `thread` 所在作用域及其外围作用域中的对象的指针传递给它。

示例

```
void f(int* p)
{
    // ...
    *p = 99;
    // ...
}
int glob = 33;

void some_fct(int* p)
{
    int x = 77;
    joining_thread t0(f, &x);           // OK
    joining_thread t1(f, p);            // OK
    joining_thread t2(f, &glob);         // OK
    auto q = make_unique<int>(99);
    joining_thread t3(f, q.get());       // OK
```

```
// ...  
}
```

`gs1::joining_thread` 是一种 `std::thread`，其析构函数进行联结且不可被 `detached()`。
这里的“OK”表明对象能够在 `thread` 可以使用指向它的指针时一直处于作用域（“存活”）。
`thread` 运行的并发性并不会影响这里的生存期或所有权问题；
这些 `thread` 可以仅仅被看成是从 `some_fct` 中调用的函数对象。

强制实施

确保 `joining_thread` 不会 `detach()`。
之后，可以实施（针对局部对象的）常规的生存期和所有权强制实施方案。

CP.24: 把 `thread` 看作是全局的容器

理由

为了维护指针安全性并避免泄漏，需要考虑 `thread` 所使用的指针。
如果 `thread` 脱离了，我们只可以安全地把指向静态和自由存储的对象的指针传递给它。

示例

```
void f(int* p)  
{  
    // ...  
    *p = 99;  
    // ...  
}  
  
int glob = 33;  
  
void some_fct(int* p)  
{  
    int x = 77;  
    std::thread t0(f, &x);           // 不好  
    std::thread t1(f, p);           // 不好  
    std::thread t2(f, &glob);        // OK  
    auto q = make_unique<int>(99);  
    std::thread t3(f, q.get());      // 不好  
    // ...  
    t0.detach();  
    t1.detach();  
    t2.detach();  
    t3.detach();  
    // ...  
}
```

这里的“OK”表明对象能够在 `thread` 可以使用指向它的指针时一直处于作用域（“存活”）。
“bad”则表示 `thread` 可能在对象销毁之后使用指向它的指针。
`thread` 运行的并发性并不会影响这里的生存期或所有权问题；
这些 `thread` 可以仅仅被看成是从 `some_fct` 中调用的函数对象。

注解

即便具有静态存储期的对象，在脱离的线程中的使用也会造成问题：
若是这个线程持续到程序终止，则它的运行可能与具有静态存储期的对象的销毁过程之间并发地发生，
而这样对这些对象的访问就可能发生竞争。

注解

如果你不 `detach()` 并使用 `gs1::joining_thread` 的话，本条规则是多余的。
不过，将代码转化为遵循这些指导方针可能很困难，而对于第三方库来说更是不可能的。
这些情况下，这条规则对于生存期安全性和类型安全性来说就是必要的了。

一般来说是无法确定是否对某个 `thread` 执行了 `detach()` 的，但简单的常见情况则易于检测出来。
如果无法证明某个 `thread` 并没有 `detach()` 的话，我们只能假定它确实脱离了，且它的存活将超过其
构造时所处于的作用域；
之后，可以实施（针对全局对象的）常规的生存期和所有权强制实施方案。

强制实施

当试图将局部变量传递给可能 `detach()` 的线程时进行标记。

CP.25: 优先采用 `gs1::joining_thread` 而不是 `std::thread`

理由

`joining_thread` 是一种在其作用域结尾处进行联结的线程。
脱离的线程很难进行监管。
确保脱离的线程（和潜在脱离的线程）中没有错误则更加困难。

示例，不好

```
void f() { std::cout << "Hello "; }

struct F {
    void operator()() const { std::cout << "parallel world "; }
};

int main()
{
    std::thread t1{f};          // f() 在独立线程中执行
    std::thread t2{F()};        // F() 在独立线程中执行
} // 请找出问题
```

示例

```
void f() { std::cout << "Hello "; }

struct F {
    void operator()() const { std::cout << "parallel world "; }
};

int main()
{
    std::thread t1{f};          // f() 在独立线程中执行
    std::thread t2{F()};        // F() 在独立线程中执行
```

```
t1.join();
t2.join();
} // 剩下一个糟糕的 BUG
```

注解

使“不死线程”成为全局的，将其放入外围作用域中，或者放入自由存储中，而不要 `detach()` 它们。
不要 `detach`。

注解

因为老代码和第三方库也会使用 `std::thread`，本条规则可能很难引入。

理由

标记 `std::thread` 的使用：

- 建议使用 `gs1::joining_thread` 或 C++20 的 `std::jthread`。
- 建议当其脱离时使其“[外放所有权](#)”到某个外围作用域中。
- 如果不明确线程是联结还是脱离，则给出警告。

CP.26: 不要 `detach()` 线程

理由

通常，需要存活超出线程创建的作用域的情况是来源于 `thread` 的任务所决定的，但用 `detach` 来实现这点将造成更加难于对脱离的线程进行监控和通信。特别是，要确保线程按预期完成或者按预期的时间存活变得更加困难（虽然不是不可能）。

示例

```
void heartbeat();

void use()
{
    std::thread t(heartbeat);           // 不联结；打算持续运行 heartbeat
    t.detach();
    // ...
}
```

这是一种合理的线程用法，一般会使用 `detach()`。

但这里有些问题。

我们怎么监控脱离的线程以查看它是否存活呢？

心跳里边可能会出错，而在需要心跳的系统中，心跳丢失可能是非常严重的问题。

因此，我们需要与心跳线程进行通信

（例如，通过某个消息流，或者使用 `condition_variable` 的通知事件）。

一种替代的，而且通常更好的方案是，通过将其放入某个其创建点（或激活点）之外的作用域来控制其生存期。

例如：

```
void heartbeat();

gs1::joining_thread t(heartbeat);           // 打算持续运行 heartbeat
```

这个心跳，（除非出错或者硬件故障等情况）将在程序运行时一直运行。

有时候，我们需要将创建点和所有权点分离开：

```
void heartbeat();

unique_ptr<gsl::joining_thread> tick_tock {nullptr};

void use()
{
    // 打算在 tick_tock 存活期间持续运行 heartbeat
    tick_tock = make_unique(gsl::joining_thread, heartbeat);
    // ...
}
```

强制实施

标记 `detach()`。

CP.31: 少量数据在线程之间按值传递，而不是通过引用或指针传递

理由

对少量数据进行复制和访问要比使用某种锁定机制进行共享更廉价。

复制天然会带来唯一所有权（简化代码），并消除数据竞争的可能性。

注解

对“少量”进行精确的定义是不可能的。

示例

```
string modify1(string);
void modify2(string&);

void fct(string& s)
{
    auto res = async(modify1, s);
    async(modify2, s);
}
```

`modify1` 的调用涉及两个 `string` 值的复制；而 `modify2` 的调用则不会。

另一方面，`modify1` 的实现和我们为单线程代码所编写的完全一样，

而 `modify2` 的实现则需要某种形式的锁定以避免数据竞争。

如果字符串很短（比如 10 个字符），对 `modify1` 的调用将会出乎意料地快；

基本上所有的代价都在 `thread` 的切换上。如果字符串很长（比如 1,000,000 个字符），对其两次复制可能并不是一个好主意。

注意这个论点和 `async` 并没有任何关系。它同等地适用于任何对采用消息传递还是共享内存的考虑之上。

强制实施

???

CP.32: 用 `shared_ptr` 在无关的 `thread` 之间共享所有权

理由

如果线程之间是无关的（就是说，互相不知道是否在相同作用域中，或者一个的生存期在另一个之内），

而它们需要共享需要删除的自由存储内存，`shared_ptr`（或者等价物）就是唯一的可以保证正确删除的安全方式。

示例

```
???
```

注解

- 可以共享静态对象（比如全局对象），因为它并不像是需要某个线程来负责其删除那样被谁所拥有。
- 自由存储上不会被删除的对象可以进行共享。
- 由一个线程所拥有的对象可以安全地共享给另一个线程，只要第二个线程存活不会超过这个拥有者线程即可。

强制实施

???

CP.40: 最小化上下文切换

理由

上下文切换是昂贵的。

示例

```
???
```

强制实施

???

CP.41: 最小化线程的创建和销毁

理由

线程创建是昂贵的。

示例

```

void worker(Message m)
{
    // 处理
}

void dispatcher(istream& is)
{
    for (Message m; is >> m; )
        run_list.push_back(new thread(worker, m));
}

```

这会为每个消息产生一个线程，而 `run_list` 则假定在它们完成后对这些任务进行销毁。

我们可以用一组预先创建的工作线程来处理这些消息：

```

Sync_queue<Message> work;

void dispatcher(istream& is)
{
    for (Message m; is >> m; )
        work.put(m);
}

void worker()
{
    for (Message m; m = work.get(); ) {
        // 处理
    }
}

void workers() // 设立工作线程（这里是 4 个工作线程）
{
    joining_thread w1 {worker};
    joining_thread w2 {worker};
    joining_thread w3 {worker};
    joining_thread w4 {worker};
}

```

注解

如果你的系统有一个好的线程池的话，就请使用它。

如果你的系统有一个好的消息队列的话，就请使用它。

强制实施

???

CP.42: 不要无条件地 `wait`

理由

没有条件的 `wait` 可能会丢失唤醒，或者唤醒时只会发现无事可做。

示例，不好

```
std::condition_variable cv;
std::mutex mx;

void thread1()
{
    while (true) {
        // 做一些工作 ...
        std::unique_lock<std::mutex> lock(mx);
        cv.notify_one();      // 唤醒另一个线程
    }
}

void thread2()
{
    while (true) {
        std::unique_lock<std::mutex> lock(mx);
        cv.wait(lock);       // 可能会永远阻塞
        // 做一些工作 ...
    }
}
```

这里，如果某个其他 `thread` 消费了 `thread1` 的通知的话，`thread2` 将会永远等待下去。

示例

```
template<typename T>
class Sync_queue {
public:
    void put(const T& val);
    void put(T&& val);
    void get(T& val);

private:
    mutex mtx;
    condition_variable cond;      // 这用于控制访问
    list<T> q;
};

template<typename T>
void Sync_queue<T>::put(const T& val)
{
    lock_guard<mutex> lck(mtx);
    q.push_back(val);
    cond.notify_one();
}

template<typename T>
void Sync_queue<T>::get(T& val)
{
    unique_lock<mutex> lck(mtx);
    cond.wait(lck, [this] { return !q.empty(); });      // 防止假性唤醒
    val = q.front();
    q.pop_front();
}
```

这样当执行 `get()` 的线程被唤醒时，如果队列为空（比如因为别的线程在之前已经 `get()` 过了），它将立刻回到睡眠中继续等待。

强制实施

对所有没有条件的 `wait` 进行标记。

CP.43: 最小化临界区的时间耗费

理由

获取 `mutex` 时耗费的时间越短，其他 `thread` 不得不等待的机会就会越少，而 `thread` 的挂起和恢复是昂贵的。

示例

```
void do_something() // 不好
{
    unique_lock<mutex> lck(my_lock);
    do0(); // 预备: 不需要锁定
    do1(); // 事务: 需要锁定
    do2(); // 清理: 不需要锁定
}
```

这里我们持有的锁定比所需的要长：

我们不应该在必须锁定之前就获取锁定，而且应当在开始清理之前将其释放掉。

可以将其重写为：

```
void do_something() // 不好
{
    do0(); // 预备: 不需要锁定
    my_lock.lock();
    do1(); // 事务: 需要锁定
    my_lock.unlock();
    do2(); // 清理: 不需要锁定
}
```

但这样损害了安全性并且违反了[使用 RAI](#) 规则。

我们可以为临界区添加语句块：

```
void do_something() // OK
{
    do0(); // 预备: 不需要锁定
    {
        unique_lock<mutex> lck(my_lock);
        do1(); // 事务: 需要锁定
    }
    do2(); // 清理: 不需要锁定
}
```

强制实施

一般来说是不可能的。

对“裸”`lock()` 和 `unlock()` 进行标记。

CP.44: 记得为 `lock_guard` 和 `unique_lock` 命名

理由

无名的局部对象时临时对象，会立刻离开作用域。

示例

```
// 全局互斥体
mutex m1;
mutex m2;

void f()
{
    unique_lock<mutex>(m1); // (A)
    lock_guard<mutex> {m2}; // (B)
    // 关键区中的工作
}
```

这个貌似足够有效，但其实并非如此。在 (A) 点，`m1` 是一个默认构造的局部 `unique_lock`，它隐藏了全局的 `::m1`（且并未锁定它）。在 (B) 点，构造了一个无名 `lock_guard` 临时对象并锁定了 `::m2`，但它立即离开作用域并再次解锁了 `::m2`。函数 `f()` 的余下部分中并没有锁定任何互斥体。

强制实施

标记所有的无名 `lock_guard` 和 `unique_lock`。

CP.50: `mutex` 要和其所保护的数据一起定义，只要可能就使用 `synchronized_value<T>`

理由

对于读者来说，数据应该且如何被保护应当是显而易见的。这可以减少锁定错误的互斥体，或者互斥体未能被锁定的机会。

使用 `synchronized_value<T>` 保证了数据都带有互斥体，并且当访问数据时锁定正确的互斥体。

参见向某个未来的 TS 或 C++ 标准的修订版添加 `synchronized_value` [WG21 提案](#)。

示例

```

struct Record {
    std::mutex m;    // 访问其他成员之前应当获取这个 mutex
    // ...
};

class MyClass {
    struct DataRecord {
        // ...
    };
    synchronized_value<DataRecord> data; // 用互斥体保护数据
};

```

强制实施

??? 可能吗?

CP.coro: 协程

这一部分关注协程的使用。

协程规则概览:

- [CP.51: 不要使用作为协程的有俘获 lambda 表达式](#)
- [CP.52: 不要在持有锁或其它同步原语时跨越挂起点](#)
- [CP.53: 协程的形参不能按引用传递](#)

CP.51: 不要使用作为协程的有俘获 lambda 表达式

理由

对于普通 lambda 来说正确的使用模式，对于协程 lambda 来说是高危的。很明显的变量俘获模式将会造成在首个挂起点之后访问已释放的内存，即便是带引用计数的智能指针和可复制类型也是如此。

lambda 表达式会产生一个带有存储的闭包对象，它通常在运行栈上，并会在某个时刻离开作用域。当闭包对象离开作用域时，它所俘获的也会离开作用域。普通 lambda 的执行在这个时间点都已经完成了，因此这并不是问题。闭包 lambda 则可能会在闭包对象已经销毁之后从挂起中恢复执行，而在这时其所有俘获都将变为“释放后使用”的内存访问。

示例，不好

```

int value = get_value();
std::shared_ptr<Foo> sharedFoo = get_foo();
{
    const auto Lambda = [value, sharedFoo]() -> std::future<void>
    {
        co_await something();
        // "sharedFoo" 和 "value" 已被销毁
        // “共享”指针没有带来任何好处
    };
    Lambda();
} // Lambda 闭包对象此时已离开作用域

```

示例，更好

```
int value = get_value();
std::shared_ptr<Foo> sharedFoo = get_foo();
{
    const auto lambda = [](auto sharedFoo, auto value) -> std::future<void> // 以
    按值传参代替作为俘获
    {
        co_await something();
        // sharedFoo 和 value 此时仍然有效
    };
    lambda(sharedFoo, value);
} // Lambda 闭包对象此时已离开作用域
```

示例，最佳

使用函数作为协程。

```
std::future<void> Class::do_something(int value, std::shared_ptr<Foo> sharedFoo)
{
    co_await something();
    // sharedFoo 和 value 此时仍然有效
}

void SomeOtherFunction()
{
    int value = get_value();
    std::shared_ptr<Foo> sharedFoo = get_foo();
    do_something(value, sharedFoo);
}
```

强制实施

标记作为协程且具有非空俘获列表的 lambda 表达式。

CP.52: 不要在持有锁或其它同步原语时跨越挂起点

理由

这种模式会导致明显的死锁风险。某些种类的等待允许当前线程在异步操作完成前实施一些额外的工作。如果持有锁的线程实施了需要相同的所的工作，那它就会因为试图获取它已经持有的锁而发生死锁。

如果协程在某个与获得所的线程不同的另一个线程中完成，那就是未定义行为。即使协程中明确返回其原来的线程，仍然有可能在协程恢复之前抛出异常，并导致其锁定防护对象（lock guard）未能销毁。

示例，不好

```
std::mutex g_lock;

std::future<void> Class::do_something()
{
    std::lock_guard<std::mutex> guard(g_lock);
    co_await something(); // 危险: 在持有锁时挂起协程
    co_await somethingElse();
}
```

示例, 好

```
std::mutex g_lock;

std::future<void> Class::do_something()
{
{
    std::lock_guard<std::mutex> guard(g_lock);
    // 修改被锁保护的数据
}
co_await something(); // OK: 锁已经在协程挂起前被释放
co_await somethingElse();
}
```

注解

这种模式对于性能也不好。每当遇到比如 `co_await` 这样的挂起点时，都会停止当前函数的执行而去运行别的代码。而在协程恢复之前可能会经过很长时间。这个锁会在整个时间段中持有，并且无法被其他线程获得以进行别的工作。

强制实施

标记所有未能在协程挂起前销毁的锁定防护。

CP.53: 协程的形参不能按引用传递

理由

一旦协程到达其第一个如 `co_await` 这样的挂起点，其同步执行的部分就会返回。这个位置之后，所有按引用传递的形参都是悬挂引用。此后对它们的任何使用都是未定义行为，可能包括向已释放的内存进行写入。

示例, 不好

```
std::future<int> Class::do_something(const std::shared_ptr<int>& input)
{
    co_await something();

    // 危险: 对 input 的引用可能不再有效, 可能已经是已释放内存
    co_return *input + 1;
}
```

示例，好

```
std::future<int> Class::do_something(std::shared_ptr<int> input)
{
    co_await something();
    co_return *input + 1; // input 是一个副本，且到此处仍有效
}
```

注解

这个问题并不适用于仅在第一个挂起点之前访问的引用形参。此后对函数的改动中可能会添加或移除挂起点，而这可能会再次引入这一类的缺陷。一些种类的协程，在协程执行第一行代码之前就会有挂起点，这种情况中的引用形参总是不安全的。一直采用按值传递的方式更安全，因为所复制的形参存活于协程的栈帧中，并在整个协程中都可以安全访问。

注解

输出参数也有这种危险。[F.20: 对于“输出 \(out\) ”值，采用返回值优先于输出参数](#) 不建议使用输出参数。协程应当完全避免输出参数。

强制实施

标记协程的所有引用形参。

CP.par: 并行

这里的“并行”代表的是对许多数据项（或多或少）同时地（“并行进行”）实施某项任务。

并行规则概览：

- ???
- ???
- 适当的时候，优先采用标准库的并行算法
- 使用为并行设计的算法，而不是不必要地依赖于线性求值的算法

CP.mess: 消息传递

标准库的设施是相当底层的，关注于使用 `thread`, `mutex`, `atomic` 等类型的贴近硬件的关键编程。大多数人都不应该在这种层次上工作：它是易错的，而且开发很慢。

如果可能的话，应当使用高层次的设施：消息程序库，并行算法，以及向量化。

这一部分关注如何传递消息，以使程序员不必进行显式的同步。

消息传递规则概览：

- [CP.60: 使用 `future` 从并发任务返回值](#)
- [CP.61: 使用 `async\(\)` 来产生并发任务](#)
- 消息队列
- 消息程序库

???? 是否应该有某个“使用 X 而不是 `std::async`”，其中 X 是某种更好说明的线程池？

??? 在未来的趋势下（甚至是现存的程序库），`std::async` 是否还是值得使用的并行设施？当有人想要对比如 `std::accumulate`（带上额外的累加前条件），或者归并排序进行并行化时，指导方针应该给出什么样的建议呢？

CP.60: 使用 `future` 从并发任务返回值

理由

`future` 为异步任务保持了常规函数调用的返回语义。

它没有显式的锁定，而且正确的（值）返回和错误的（异常）返回都能被简单处理。

示例

```
???
```

注解

???

强制实施

???

CP.61: 使用 `async()` 来产生并发任务

理由

[R.12](#) 告诉我们要避免原始所有权指针，与此相似，

我们也要尽可能避免原始线程和原始承诺（promise）。应使用诸如 `std::async` 之类的工厂函数，它将处理线程的产生和重用，而不会讲原始线程暴露给自己的代码。

示例

```
int read_value(const std::string& filename)
{
    std::ifstream in(filename);
    in.exceptions(std::ifstream::failbit);
    int value;
    in >> value;
    return value;
}

void async_example()
{
    try {
        std::future<int> f1 = std::async(read_value, "v1.txt");
        std::future<int> f2 = std::async(read_value, "v2.txt");
        std::cout << f1.get() + f2.get() << '\n';
    } catch (std::ios_base::failure & fail) {
        // 此处处理异常
    }
}
```

注解

不幸的是，`async()` 并不完美。比如说，它并不使用线程池，

这意味着它可能会因为资源耗尽而失败，而不会将你的任务放入队列以便随后执行。

不过，即便你不能用 `std::async`，你也应当优先编写自己的返回

`future` 的工厂函数，而非使用原始承诺。

示例（不好）

这个例子展示了两种方式，都使用了 `std::future` 却未能避免对原始 `std::thread` 的管理。

```
void async_example()
{
    std::promise<int> p1;
    std::future<int> f1 = p1.get_future();
    std::thread t1([p1 = std::move(p1)]() mutable {
        p1.set_value(read_value("v1.txt"));
    });
    t1.detach(); // 恶行

    std::packaged_task<int()> pt2(read_value, "v2.txt");
    std::future<int> f2 = pt2.get_future();
    std::thread(std::move(pt2)).detach();

    std::cout << f1.get() + f2.get() << '\n';
}
```

示例（好）

这个例子展示一种方法，可以让你在当 `std::async` 自身在产品中不可接受的情形中，模仿 `std::async` 所设立的一般模式的方法。

```
void async_example(workQueue& wq)
{
    std::future<int> f1 = wq.enqueue([]() {
        return read_value("v1.txt");
    });
    std::future<int> f2 = wq.enqueue([]() {
        return read_value("v2.txt");
    });
    std::cout << f1.get() + f2.get() << '\n';
}
```

所有为执行 `read_value` 的代码而产生的线程都被隐藏到对 `workQueue::enqueue` 的调用之内。用户代码仅需处理 `future` 对象，无需处理原始 `thread`, `promise` 或 `packaged_task` 对象。

强制实施

???

CP.vec: 向量化

向量化是一种在不引入显式同步时并发地执行多个任务的技术。

它会并行地把某个操作实施与某个数据结构（向量，数组，等等）的各个元素之上。

向量化引人关注的性质在于它通常不需要对程序进行非局部的改动。

不过，向量化只对简单的数据结构和特别针对它所构造的算法能够得到最好的工作效果。

向量化规则概览：

- ???
- ???

CP.free: 无锁编程

使用 `mutex` 和 `condition_variable` 进行同步相对来说很昂贵。

而且可能导致死锁。

为了性能并消除死锁的可能性，有时候我们不得不使用麻烦的底层“无锁”设施，它们依赖于对内存短暂地获得互斥（“原子性”）访问。

无锁编程也被用于实现如 `thread` 和 `mutex` 这样的高层并发机制。

无锁编程规则概览：

- [CP.100: 除非绝对必要，请勿使用无锁编程](#)
- [CP.101: 不要信任你的硬件-编译器组合](#)
- [CP.102: 仔细研究文献](#)
- 如何/何时使用原子
- 避免饥饿
- 使用无锁数据结构而不是手工构造的专门的无锁访问
- [CP.110: 不要为初始化编写你自己的双检查锁定](#)
- [CP.111: 当确实需要双检查锁定时应当采用惯用的模式](#)
- 如何/何时进行比较并交换 (CAS)

CP.100: 除非绝对必要，请勿使用无锁编程

理由

无锁编程容易出错，要求专家级的语言特性、机器架构和数据结构知识。

示例，不好

```
extern atomic<Link*> head;           // 共享的链表表头

Link* nh = new Link(data, nullptr);   // 为进行插入制作一个连接
Link* h = head.load();                // 读取链表中的共享表头

do {
    if (h->data <= data) break;        // 这样的话，就插入到别处
    nh->next = h;                     // 下一个元素是之前的表头
} while (!head.compare_exchange_weak(h, nh)); // 将 nh 写入 head 或 h
```

请找出这里的 BUG。

通过测试找到它是非常困难的。

请阅读有关 ABA 问题的材料。

例外

原子变量可以简单并安全地使用，只要你所用的是顺序一致性内存模型 (`memory_order_seq_cst`)，而这是默认情况。

注解

高层的并发机制，诸如 `thread` 和 `mutex`，是使用无锁编程来实现的。

替代方案: 使用由他人实现并作为程序库一部分的无锁数据结构。

CP.101: 不要信任你的硬件-编译器组合

理由

无锁编程所使用的底层硬件接口，是属于最难正确实现的，

而且属于最可能会发生最微妙的兼容性问题的领域。

如果你为了性能而进行无锁编程的话，你应当进行回归检查。

注解

指令重排（静态和动态的）会让我们很难有效在这个层次上进行思考（尤其当你使用宽松的内存模型的时候）。

经验，（半）形式化的模型以及模型检查可以提供帮助。

测试——通常需要极端程度——是基础。

“不要飞得太靠近太阳。”

强制实施

准备强有力的规则，使得当硬件，操作系统，编译器，和程序库发生任何改变都能重复测试以进行覆盖。

CP.102: 仔细研究文献

理由

除了原子和少数其他的标准模式之外，无锁编程真的是只有专家才懂的议题。

在发布无锁代码给其他人使用之前，应当先成为一名专家。

参考文献

- Anthony Williams: C++ concurrency in action. Manning Publications.
- Boehm, Adve, You Don't Know Jack About Shared Variables or Memory Models , Communications of the ACM, Feb 2012.
- Boehm, "Threads Basics", HPL TR 2009-259.
- Adve, Boehm, "Memory Models: A Case for Rethinking Parallel Languages and Hardware", Communications of the ACM, August 2010.
- Boehm, Adve, "Foundations of the C++ Concurrency Memory Model", PLDI 08.
- Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber, "Mathematizing C++ Concurrency", POPL 2011.
- Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup: Understanding and Effectively Preventing the ABA Problem in Descriptor-based Lock-free Designs. 13th IEEE Computer Society ISORC 2010 Symposium. May 2010.
- Damian Dechev and Bjarne Stroustrup: Scalable Non-blocking Concurrent Objects for Mission Critical Code. ACM OOPSLA'09. October 2009
- Damian Dechev, Peter Pirkelbauer, Nicolas Rouquette, and Bjarne Stroustrup: Semantically Enhanced Containers for Concurrent Real-Time Systems. Proc. 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (IEEE ECBS). April 2009.

- Maurice Herlihy, Nir Shavit, Victor Luchangco, Michael Spear, "The Art of Multiprocessor Programming", 2nd ed. September 2020

CP.110: 不要为初始化编写你自己的双检查锁定

理由

从 C++11 开始，静态局部变量是以线程安全的方式初始化的。当和 RAII 模式结合时，静态局部变量可以取代为初始化自己编写双检查锁定的需求。`std::call_once` 也可以达成相同的目的。请使用 C++11 的静态局部变量或者 `std::call_once` 来代替自己为初始化编写的双检查锁定。

示例

使用 `std::call_once` 的例子。

```
void f()
{
    static std::once_flag my_once_flag;
    std::call_once(my_once_flag, []()
    {
        // 这个只做一次
    });
    // ...
}
```

使用 C++11 的线程安全静态局部变量的例子。

```
void f()
{
    // 假定编译器遵循 C++11
    static My_class my_object; // 构造函数仅调用一次
    // ...
}

class My_class
{
public:
    My_class()
    {
        // 这个只做一次
    }
};
```

强制实施

??? 是否可能检测出这种惯用法？

CP.111: 当确实需要双检查锁定时应当采用惯用的模式

理由

双检查锁定是很容易被搞乱的。如果确实需要编写自己的双检查锁定，而不顾规则 [CP.110: 不要为初始化编写你自己的双检查锁定](#) 和规则 [CP.100: 除非绝对必要，请勿使用无锁编程](#)，那么应当采用惯用的模式。

使用双检查锁定模式而不违反[CP.110: 不要为初始化编写你自己的双检查锁定](#)的情形，出现于当某个非线程安全的动作既困难也罕见，并且存在某个快速且线程安全的测试可以用于保证该动作并不需要实施的情况，但反过来的情况则无法保证。

示例，不好

使用 `volatile` 并不能使得第一个检查线程安全，另见[CP.200: volatile 仅用于和非 C++ 内存进行通信](#)

```
mutex action_mutex;
volatile bool action_needed;

if (action_needed) {
    std::lock_guard<std::mutex> lock(action_mutex);
    if (action_needed) {
        take_action();
        action_needed = false;
    }
}
```

示例，好

```
mutex action_mutex;
atomic<bool> action_needed;

if (action_needed) {
    std::lock_guard<std::mutex> lock(action_mutex);
    if (action_needed) {
        take_action();
        action_needed = false;
    }
}
```

这对于正确调校的内存顺序可能会带来好处，其中的获取加载要比顺序一致性加载更加高效

```
mutex action_mutex;
atomic<bool> action_needed;

if (action_needed.load(memory_order_acquire)) {
    lock_guard<std::mutex> lock(action_mutex);
    if (action_needed.load(memory_order_relaxed)) {
        take_action();
        action_needed.store(false, memory_order_release);
    }
}
```

强制实施

??? 是否可能检测出这种惯用法？

CP/etc: 其他并发规则

这些规则不适于简单的分类：

- [CP.200: volatile 仅用于和非 C++ 内存进行通信](#)
- [CP.201: ??? 信号](#)

CP.200: volatile 仅用于和非 C++ 内存进行通信

理由

`volatile` 用于涉指那些与“非 C++”代码之间共享的对象，或者不遵循 C++ 内存模型的硬件。

示例

```
const volatile long clock;
```

这说明了一个被某个时钟电路不断更新的寄存器。

`clock` 为 `volatile` 是因为其值将会在没有使用它的 C++ 程序的任何动作下被改变。

例如，两次读取 `clock` 经常会产生两个不同的值，因此优化器最好不要将下面代码中的第二个读取操作优化掉：

```
long t1 = clock;
// ... 这里没有对 clock 的使用 ...
long t2 = clock;
```

`clock` 为 `const` 是因为程序不应当试图写入 `clock`。

注解

除非你是在编写直接操作硬件的最底层代码，否则应当把 `volatile` 当做是某种深奥的功能特性并最好避免使用。

示例

通常 C++ 代码接受的 `volatile` 内存是由别处所拥有的（硬件或其他语言）：

```
int volatile* vi = get_hardware_memory_location();
// 注意：我们获得了指向别人的内存的指针
// volatile 说明“请特别尊重地对待”
```

有时候 C++ 代码会分配 `volatile` 内存，并通过故意地暴露一个指针来将其共享给“别人”（硬件或其他语言）：

```
static volatile long v1;
please_use_this(&v1); // 暴露对这个的一个引用给“别人”（不是 C++）
```

示例，不好

`volatile` 局部变量几乎都是错误的——既然是短暂的，它们如何才能共享给其他语言或硬件呢？因为相同的原因，这几乎同样强有力地适用于成员变量。

```

void f()
{
    volatile int i = 0; // 不好, volatile 局部变量
    // etc.
}

class My_type {
    volatile int i = 0; // 可以的, volatile 成员变量
    // etc.
};

```

注解

于其他一些语言不通, C++ 中的 `volatile` 和同步没有任何关系。

强制实施

- 对 `volatile T` 的局部成员变量进行标记; 几乎肯定你应当用 `atomic<T>` 进行代替。
- ???

CP.201: ??? 信号

???UNIX 信号处理??? 也许值得提及异步信号安全有多么微弱, 以及如何同信号处理器进行通信 (也许最好应当“完全避免”)

E: 错误处理

错误处理涉及:

- 检测某个错误
- 将有关错误的信息传递给某个处理代码
- 维持程序的某个有效状态
- 避免资源泄漏

不可能做到从所有的错误中恢复。如果从某个错误进行恢复是不可能的话, 以明确定义的方式迅速“脱离”则是很重要的。错误处理的策略必须简单, 否则就会成为更糟糕错误的来源。未经测试和很少被执行的错误处理代码自身也是许多 BUG 的来源。

以下规则是设计用以帮助避免几种错误的:

- 类型违规 (比如对 `union` 和强制转换的误用)
- 资源泄漏 (包括内存泄漏)
- 边界错误
- 生存期错误 (比如在对象被 `delete` 后访问它)
- 复杂性错误 (可能由于过于复杂的想法表达而导致的逻辑错误)
- 接口错误 (比如通过接口传递了预期外的值)

错误处理规则概览:

- [E.1: 在设计中尽早开发错误处理策略](#)
- [E.2: 通过抛出异常来明示函数无法完成其所赋予的任务](#)
- [E.3: 仅使用异常来进行错误处理](#)
- [E.4: 围绕不变式来设计错误处理策略](#)
- [E.5: 让构造函数建立不变式, 若其无法做到则抛出异常](#)
- [E.6: 使用 RAI 来避免泄漏](#)

- [E.7: 明示前条件](#)
- [E.8: 明示后条件](#)
- [E.12: 当函数不可能或不能接受以 `throw` 来退出时, 使用 `noexcept`](#)
- [E.13: 不要在作为某个对象的直接所有者时抛出异常](#)
- [E.14: 应当使用为该类所设计的自定义类型 \(而不是内建类型\) 作为异常](#)
- [E.15: 按值抛出并按引用捕获类型层次中的异常](#)
- [E.16: 析构函数, 回收函数, `swap`, 以及异常类型的复制/移动构造决不能失败](#)
- [E.17: 不要试图在每个函数中捕获每个异常](#)
- [E.18: 最小化对 `try / catch` 的显式使用](#)
- [E.19: 当没有合适的资源包装时, 使用 `final_action` 对象来表达清理动作](#)
- [E.25: 当不能抛出异常时, 模拟 RAI 来进行资源管理](#)
- [E.26: 当不能抛出异常时, 考虑采取快速失败](#)
- [E.27: 当不能抛出异常时, 系统化地使用错误代码](#)
- [E.28: 避免基于全局状态 \(比如 `errno`\) 的错误处理](#)
- [E.30: 请勿使用异常说明](#)
- [E.31: 恰当地对 `catch` 子句排序](#)

E.1: 在设计中尽早开发错误处理策略

理由

在一个系统中改造翻新一种一致且完整的处理错误和资源泄漏的策略是很难的。

E.2: 通过抛出异常来明示函数无法完成其所赋予的任务

理由

让错误处理有系统性, 强健, 而且避免重复。

示例

```
struct Foo {
    vector<Thing> v;
    File_handle f;
    string s;
};

void use()
{
    Foo bar {{Thing{1}, Thing{2}, Thing{monkey}}, {"my_file", "r"}, "Here we
go!"};
    // ...
}
```

这里, `vector` 和 `string` 的构造函数可能无法为其元素分配足够的内存, `vector` 的构造函数可能无法复制其初始化式列表中的 `Thing`, 而 `File_handle` 可能无法打开所需的文件。

这些情况中, 它们都会抛出异常来让 `use()` 的调用者来处理。

如果 `use()` 可以处理这种构造 `bar` 的故障的话, 它可以用 `try / catch` 来控制。

这些情况下, `Foo` 的构造函数都会在把控制传递给试图创建 `Foo` 的任何代码之前恰当地销毁以及构造的内存。

注意它并不存在可以包含错误代码的返回值。

`File_handle` 的构造函数可以这样定义:

```
File_handle::File_handle(const string& name, const string& mode)
    : f{fopen(name.c_str(), mode.c_str())}
{
    if (!f)
        throw runtime_error{"File_handle: could not open " + name + " as " +
mode"};
}
```

注解

人们通常说异常应当用于表明意外的事件和故障。

不过这里面存在一点循环，“什么是意外的？”

例如：

- 无法满足的前条件
- 无法构造对象的构造函数（无法建立类的[不变式](#)）
- 越界错误（比如 `v[v.size()] = 7`）
- 无法获得资源（比如网络未连接）

相较而言，终止某个普通的循环则不是意外的。

除非这个循环本应当是无限循环，否则其终止就是正常且符合预期的。

注解

不要用 `throw` 仅仅作为从函数中返回值的另一种方式。

例外

某些系统，比如在执行开始之前就需要保证以（通常很短的）常量最大时间来执行动作的硬实时系统，这样的系统只有当存在工具可以支持精确地预测从一次 `throw` 中恢复的最大时间时才能使用异常。

参见: [RAII](#)

参见: [讨论](#)

注解

在你决定你无法负担或者不喜欢基于异常的错误处理之前，请看一看[替代方案](#)；

它们各自都有自己的复杂性和问题。

同样地，只要可能的话，就应该进行测量之后再发表有关效率的言论。

E.3: 仅使用异常来进行错误处理

理由

以保持错误处理和“常规代码”互相分离。

C++ 实现都倾向于基于假定异常的稀有而进行优化。

示例，请勿如此

```
// 请勿如此：异常并未用于错误处理
int find_index(vector<string>& vec, const string& x)
{
    try {
        for (gsl::index i = 0; i < vec.size(); ++i)
            if (vec[i] == x) throw i; // 找到了 x
    }
    catch (int i) {
        return i;
    }
    return -1; // 未找到
}
```

这种代码要比显然的替代方式更加复杂，而且极可能运行慢得多。

在 `vector` 中寻找一个值是没什么意外情况的。

强制实施

可能应该是启发式措施。

查找从 `catch` 子句“漏掉”的异常值。

E.4: 围绕不变式来设计错误处理策略

理由

要使用一个对象，它必须处于某个（正式或非正式通过不变式所定义的）有效的状态，而要从错误中恢复，每个还未销毁的对象也必须处于有效的状态。

注解

不变式是对象的成员的逻辑条件，构造函数必须进行建立，且为公开的成员函数所假定。

强制实施

???

E.5: 让构造函数建立不变式，若其无法做到则抛出异常

理由

遗留仍未建立不变式的对象将会带来麻烦。

不是任何成员函数都可以对其进行调用。

示例

```

class vector { // 非常简单的 double 向量
    // 当 elem != nullptr 时 elem 指向 sz 个 double
public:
    vector() : elem{nullptr}, sz{0} {}
    vector(int s) : elem{new double[s]}, sz{s} { /* 元素的初始化 */ }
    ~vector() { delete [] elem; }
    double& operator[](int s) { return elem[s]; }
    // ...
private:
    owner<double*> elem;
    int sz;
};

```

类不变式——这里以代码注释说明——是由构造函数建立的。

当 `new` 无法分配所需的内存时将抛出异常。

各运算符，尤其是下标运算符，都是依赖于这个不变式的。

参见: [当构造函数无法构造有效对象时，应当抛出异常](#)

强制实施

对带有 `private` 状态但没有（公开，受保护或私有的）构造函数的类进行标记。

E.6: 使用 RAII 来避免泄漏

理由

资源泄漏通常是不可接受的。

手工的资源释放很容易出错。

RAII (Resource Acquisition Is Initialization, 资源获取即初始化) 是最简单，最系统化的避免泄漏方案。

示例

```

void f1(int i) // 不好：可能会泄漏
{
    int* p = new int[12];
    // ...
    if (i < 17) throw Bad{"in f()", i};
    // ...
}

```

我们可以在抛出异常前小心地释放资源：

```

void f2(int i) // 笨拙且易错：显式的释放
{
    int* p = new int[12];
    // ...
    if (i < 17) {
        delete[] p;
        throw Bad{"in f()", i};
    }
    // ...
}

```

这样很啰嗦。在更大型的可能带有多个 `throw` 的代码中，显式的释放将变得重复且易错。

```
void f3(int i)    // OK: 通过资源包装来进行资源管理（请见下文）
{
    auto p = make_unique<int[]>(12);
    // ...
    if (i < 17) throw Bad{"in f()", i};
    // ...
}
```

注意即使 `throw` 是在所调用的函数中暗中发生，这也能正常工作：

```
void f4(int i)    // OK: 通过资源包装来进行资源管理（请见下文）
{
    auto p = make_unique<int[]>(12);
    // ...
    helper(i);    // 可能抛出异常
    // ...
}
```

除非你确实需要指针语义，否则还是应当使用局部的资源对象：

```
void f5(int i)    // OK: 通过局部对象来进行资源管理
{
    vector<int> v(12);
    // ...
    helper(i);    // 可能抛出异常
    // ...
}
```

这即简单又安全，而且通常更加高效。

注解

当没有合适的资源包装，且定义一个适当的 RAII 对象/包装由于某种原因不可行时，万不得已，可以使用 [final_action 对象](#) 来表达清理动作。

注解

但是当我们所编写的程序不能使用异常时应当怎么办呢？

首先应当质疑这项假设；到处都有许多反异常的错误认识。

据我们所知，只有少量正当理由：

- 我们所在的系统太小，支持异常将会吃掉我们的 2K 内存的大部分。
 - 我们所在的是硬实时系统，而且我们没有工具能保证异常会在所需时间内处理掉。
 - 我们所在的系统中有成吨的遗留代码以难于理解的方式大量地使用指针
(尤其是没有可识别的所有权策略)，因此异常可能会造成泄露。
 - 我们的 C++ 异常机制的实现不合理地糟糕
(很慢，很耗内存，对于动态链接库无法正确工作，等等)。
- 请向你的实现的供应商提出意见；如果没有用户提出意见，就不会出现改进。
- 如果我们质疑经理的古老智慧的话会被炒鱿鱼。

以上原因中只有第一条才是基础问题，因此一旦可能的话，还是要用异常来实现 RAII，或者设计你的 RAII 对象永不失败。

当无法使用异常时，可以模拟 RAII。

就是说，系统化地在对象构造之后检查其有效性，并且仍然在析构函数中释放所有的资源。

一种策略是为每个资源包装添加一个 `valid()` 操作：

```
void f()
{
    vector<string> vs(100);    // 非 std::vector: 添加了 valid()
    if (!vs.valid()) {
        // 处理错误或退出
    }

    ifstream fs("foo");      // 非 std::ifstream: 添加了 valid()
    if (!fs.valid()) {
        // 处理错误或退出
    }

    // ...
} // 析构函数如常进行清理
```

显然这样做增加了代码大小，不允许隐式的“异常”（`valid()` 检查）传播，而且 `valid()` 检查可能被忘掉。

优先采用异常。

参见: [noexcept 的用法](#)

强制实施

???

E.7: 明示前条件

理由

避免接口错误。

参见: [前条件规则](#)

E.8: 明示后条件

理由

避免接口错误。

参见: [后条件规则](#)

E.12: 当函数不可能或不能接受以 `throw` 来退出时，使用 `noexcept`

理由

使错误处理系统化，强健，且高效。

示例

```
double compute(double d) noexcept
{
    return log(sqrt(d <= 0 ? 1 : d));
}
```

这里，我们已知 `compute` 不会抛出异常，因为它仅由不会抛出异常的操作所组成。

通过将 `compute` 声明为 `noexcept`，让编译器和人类阅读者获得信息，使其更容易理解和操作 `compute`。

注解

许多标准库函数都是 `noexcept` 的，这包括所有从 C 标准库中“继承”来的标准库函数。

示例

```
vector<double> munge(const vector<double>& v) noexcept
{
    vector<double> v2(v.size());
    // ... 做一些事 ...
}
```

这里的 `noexcept` 表明我不希望或无法处理无法构造局部的 `vector` 对象的情形。

也就是说，我认为内存耗尽是一种严重的设计错误（类比于硬件故障），因此我希望当其发生时让程序崩溃。

注解

请勿使用传统的[异常说明](#)。

参见

[讨论](#)。

E.13: 不要在作为某个对象的直接所有者时抛出异常

理由

这可能导致一次泄漏。

示例

```
void leak(int x)    // 请勿如此：可能泄漏
{
    auto p = new int{7};
    if (x < 0) throw Get_me_out_of_here{}; // 可能泄漏 *p
    // ...
    delete p;   // 可能不会执行到这里
}
```

避免这种问题的一种方法是坚持使用资源包装：

```
void no_leak(int x)
{
    auto p = make_unique<int>(7);
    if (x < 0) throw Get_me_out_of_here{}; // 将按需删除 *p
    // ...
    // 无须 delete p
}
```

另一种（通常更好）的方案是使用一个局部变量来消除指针的显式使用：

```
void no_leak_simplified(int x)
{
    vector<int> v(7);
    // ...
}
```

注解

如果有需要清理的某个局部“东西”，但其并未表示为带有析构函数的对象，则这样的清理也必须在 `throw` 之前完成。

有时候，[finally\(\)](#) 可以把这种不系统的清理变得更加可管理一些。

E.14: 应当使用为目的所设计的自定义类型（而不是内建类型）作为异常

理由

自定义类型可以把有关某个错误的信息更好地传递给处理器。

这些信息可以编码到类型自身中，而类型则不大可能会和其他人的异常造成冲突。

示例

```
throw 7; // 不好

throw "something bad"; // 不好

throw std::exception(); // 不好 - 未提供信息
```

从 `std::exception` 派生，能够获得选择捕获特定异常或者通过 `std::exception` 进行通盘处理的灵活性：

```
class MyException: public std::runtime_error
{
public:
    MyException(const string& msg) : std::runtime_error(msg) {}
    // ...
};

// ...

throw MyException("something bad"); // 好
```

异常可以不必派生于 `std::exception`：

```
class MyCustomError final {}; // 并未派生于 std::exception  
// ...  
  
throw MyCustomError{}; // 好 - 处理器必须捕获这个类型（或 ...）
```

当检测位置没有可以添加的有用信息时，可以使用派生于 `exception` 的库类型作为通用类型：

```
throw std::runtime_error{"someting bad"}; // 好  
// ...  
  
throw std::invalid_argument("i is not even"); // 好
```

也可以使用 `enum` 类：

```
enum class alert {RED, YELLOW, GREEN};  
  
throw alert::RED; // 好
```

强制实施

识别针对内建类型和 `std::exception` 的 `throw`。

E.15: 按值抛出并按引用捕获类型层次中的异常

理由

按值（而非指针）抛出并按引用捕获，能避免进行复制，尤其是基类子对象的切片。

示例，不好

```
void f()  
{  
    try {  
        // ...  
        throw new widget{}; // 请勿如此：抛出值而不要抛出原始指针  
        // ...  
    }  
    catch (base_class e) { // 请勿如此：可能造成切片  
        // ...  
    }  
}
```

可以代之以引用：

```
catch (base_class& e) { /* ... */ }
```

或者（通常更好的） `const` 引用：

```
catch (const base_class& e) { /* ... */ }
```

大多数处理器并不会改动异常，一般情况下我们都会[建议使用 `const`](#)。

注解

对于如一个 `enum` 值这样的小型值类型来说，按值捕获是合适的。

注解

重新抛出已捕获的异常应当使用 `throw;` 而非 `throw e;`。使用 `throw e;` 将会抛出 `e` 的一个新副本（并于异常被 `catch (const std::exception& e)` 捕获时切片成静态类型 `std::exception`），而非重新抛出原来的 `std::runtime_error` 类型的异常。（但请关注[请勿试图在每个函数中捕获所有的异常](#)，以及[尽可能减少 `try / catch` 的显式使用](#)。）

强制实施

- 对按值捕获具有虚函数的类型进行标记。
- 对抛出原始指针进行标记。

E.16: 析构函数，回收函数，`swap`，以及异常类型的复制/移动构造 决不能失败

理由

如果析构函数，`swap`，内存回收，或者尝试复制/移动异常对象时会失败，就是说如果它会通过异常而退出，或者根本不会实施其所需的动作，我们就将不知道应当如何编写可靠的程序。

示例，请勿如此

```
class Connection {
    // ...
public:
    ~Connection()    // 请勿如此：非常糟糕的析构函数
    {
        if (cannot_disconnect()) throw I_give_up{information};
        // ...
    }
};
```

注解

许多人都曾试图编写违反这条规则的可靠代码，比如当网络连接“拒绝关闭”的情形。

尽我们所知，没人曾找到一个做到这点的通用方案。

虽然偶尔对于非常特殊的例子，你可以通过设置某个状态以便进行将来的清理的方式绕过它。

比如说，我们可能将一个不打算关闭的 `socket` 放入一个“故障 `socket`”列表之中，
让其被某个定期的系统状态清理所检查处理。

我们见过的每个这种例子都是易错的，专门的，而且通常有 BUG。

注解

标准库假定析构函数，回收函数（比如 `operator delete`），和 `swap` 都不会抛出异常。当它们这样做时，基本的标准库不变式将会被打破。

注解

- 回收函数，包括 `operator delete`，必须为 `noexcept`。
- `swap` 函数必须为 `noexcept`。
- 大多数的析构函数都是缺省隐含为 `noexcept` 的。
- 而且，[应该使移动操作为 noexcept](#)。
- 当编写用作异常类型的类型时，确保其复制构造函数不为 `noexcept`。一般来说我们没法机制化地强制这一点，因为我们并不了解一个类型是否有意作为一种异常类型。
- 尝试避免抛出复制构造函数不为 `noexcept` 的类型。一般来说我们没法机制化地强制这一点，因为即便 `throw std::string(...)` 也可能抛异常，虽然实际上并不会。

强制实施

- 识别会 `throw` 的析构函数，回收操作，和 `swap`。
- 识别不为 `noexcept` 的这类操作。

参见: [讨论](#)

E.17: 不要试图在每个函数中捕获每个异常

理由

如果函数无法对异常进行有意义的恢复动作，其捕获这个异常就导致复杂性和浪费。

要让异常传播直到遇到一个可以处理它的函数。

要用 [RAII](#) 来处理栈回溯路径上的清理操作。

示例，请勿如此

```
void f() // 不好
{
    try {
        // ...
    }
    catch (...) {
        // 不做任何事
        throw; // 传播异常
    }
}
```

强制实施

- 标记嵌套的 `try` 块。
- 对带有过高的 `try` 块/函数比率的源代码文件进行标记。（[??? 问题：定义“过高”](#)）

E.18: 最小化对 `try/catch` 的显式使用

理由

`try / catch` 很啰嗦，而且非平凡的使用是易错的。

`try / catch` 可以作为对非系统化和/或低级的资源管理或错误处理的一个信号。

示例，不好

```
void f(zstring s)
{
    Gadget* p;
    try {
        p = new Gadget(s);
        // ...
        delete p;
    }
    catch (Gadget_construction_failure) {
        delete p;
        throw;
    }
}
```

这段代码很混乱。

可能在 `try` 块中的裸指针上发生泄漏。

不是所有的异常都被处理了。

`delete` 一个构造失败的对象几乎肯定是一个错误。

更好的是：

```
void f2(zstring s)
{
    Gadget g {s};
}
```

替代方案

- 合适的资源包装以及 [RAII](#)
- [finally](#)

强制实施

??? 很难，需要启发式方法

E.19: 当没有合适的资源包装时，使用 `final_action` 对象来表达清理动作

理由

[GSL](#) 提供的 `finally` 要比 `try/catch` 更不啰嗦且易于搞错。

示例

```
void f(int n)
{
    void* p = malloc(n);
    auto _ = gsl::finally([p] { free(p); });
    // ...
}
```

注解

`finally` 没有 `try / catch` 那样混乱，但它仍然比较专门化。
优先采用[适当的资源管理对象](#)。
万不得已考虑使用 `finally`。

注解

相对于老式的 [goto exit](#) 技巧来说，使用 `finally` 是处理并非系统化的资源管理中的清理工作的更加系统化并且相当简洁的方案。

强制实施

启发式措施：检测 `goto exit;`。

E.25: 当不能抛出异常时，模拟 RAI 来进行资源管理

理由

即便没有异常，[RAI](#) 通常仍然是最佳且最系统化的处理资源的方式。

注解

使用异常进行错误处理，是 C++ 中唯一完整且系统化的处理非局部错误的方式。
特别是，非侵入地对对象构造的失败进行报告需要使用异常。
以无法被忽略的方式报告错误需要使用异常。
如果无法使用异常，应当尽你所能模拟它们的使用。

大量对异常的惧怕是被误导的。
当用在并非充斥指针和复杂控制结构的代码中的违例情形时，
异常处理几乎总是（在时间和空间上）可以负担，且几乎总会导致更好的代码。
当然这假定存在一个优良的异常处理机制实现，而这并非在所有系统上都存在。
存在另一些情况并不适用于上述问题，但由于其他原因而无法使用异常。
一个例子是一些硬实时系统：必须在固定的时间之内完成操作并得到错误或正确的响应。
在没有适当的时间评估工具的条件下，很难对异常作出保证。
这样的系统（比如飞控系统）通常也会禁止使用动态（堆）内存。

因此，对于错误处理的主要指导方针还是“使用异常和 [RAI](#)。”
本节所处理的情况是，要么你没有高效的异常实现，
或者要面对大量的老式代码
(比如说，大量的指针，不明确定义的所有权，以及大量的不系统化的基于错误代码检查的错误处理)
而且向其引入简单且系统化的错误处理的做法不可行。

在宣称不能使用异常或者抱怨它们成本过高之前，应当考虑一下使用[错误代码](#)的例子。
请考虑使用错误代码的成本和复杂性。
如果你担心性能的话，请进行测量。

示例

假定你想要编写

```
void func(zstring arg)
{
    Gadget g {arg};
    // ...
}
```

当这个 `g` 并未正确构造时，`func` 将以一个异常退出。

当无法抛出异常时，我们可以通过向 `Gadget` 添加 `valid()` 成员函数来模拟 RAI 风格的资源包装：

```
error_indicator func(zstring arg)
{
    Gadget g {arg};
    if (!g.valid()) return gadget_construction_error;
    // ...
    return 0;    // 零代表“正常”
}
```

显然问题现在变成了调用者必须记得测试其返回值。考虑添加 `[[nodiscard]]` 以鼓励这样的做法。

参见: [讨论](#)

强制实施

(仅) 对于这种想法的特定版本是可能的：比如检查资源包装的构造后进行系统化的 `valid()` 测试。

E.26: 当不能抛出异常时，考虑采取快速失败

理由

如果你无法做好错误恢复的话，至少你可以在发生更多后续的损害之前拜托出来。

参见: [模拟 RAI](#)

注解

当你无法系统化地进行错误处理时，考虑以“程序崩溃”作为对任何无法局部处理的错误的回应。

就是说，如果你无法在检测到错误的函数的上下文中处理它，则调用 `abort()`, `quick_exit()`, 或者相似的某个将触发某种系统重启的函数。

在具有大量进程和/或大量计算机的系统中，你总要预计到并处理这些关键程序崩溃，比如说源于硬件故障而引发。

这种情况下，“程序崩溃”只不过把错误处理留给了系统的下一个层次。

示例

```
void f(int n)
{
    // ...
    p = static_cast<x*>(malloc(n * sizeof(x)));
    if (!p) abort();      // 当内存耗尽时 abort
    // ...
}
```

大多数程序都无法得体地处理内存耗尽。这大略上等价于

```
void f(int n)
{
    // ...
    p = new x[n];      // 当内存耗尽时抛出异常（默认情况会调用 terminate）
    // ...
}
```

通常，在退出之前将“崩溃”的原因记录日志是个好主意。

强制实施

很难对付

E.27: 当不能抛出异常时，系统化地使用错误代码

理由

系统化地使用任何错误处理策略都能最小化忘记处理错误的机会。

参见：[模拟 RAII](#)

注解

需要处理几个问题：

- 如何从函数向外传递错误指示？
- 如何在错误退出函数之前释放所有资源？
- 使用什么来作为错误指示？

通常，返回错误指示意味着返回两个值：其结果以及一个错误指示。

错误指示可以是对象的一部分，例如对象可以带有 `valid()` 指示，
也可以返回一对值。

示例

```
Gadget make_gadget(int n)
{
    // ...
}

void user()
{
    Gadget g = make_gadget(17);
    if (!g.valid()) {
        // 错误处理
    }
    // ...
}
```

这种方案符合[模拟 RAII 资源管理](#)。

`valid()` 函数可以返回一个 `error_indicator`（比如说 `error_indicator` 枚举的某个成员）。

示例

要是我们无法或者不想改动 `Gadget` 类型呢？

这种情况下，我们只能返回一对值。

例如：

```
std::pair<Gadget, error_indicator> make_gadget(int n)
{
    // ...
}

void user()
```

```

{
    auto r = make_gadget(17);
    if (!r.second) {
        // 错误处理
    }
    Gadget& g = r.first;
    // ...
}

```

可见，`std::pair` 是一种可能的返回类型。

某些人则更喜欢专门的类型。

例如：

```

Gval make_gadget(int n)
{
    // ...
}

void user()
{
    auto r = make_gadget(17);
    if (!r.err) {
        // 错误处理
    }
    Gadget& g = r.val;
    // ...
}

```

倾向于专门返回类型的一种原因是为其成员提供命名，而不是使用多少有些隐秘的 `first` 和 `second`，而且可以避免与 `std::pair` 的其他使用相混淆。

示例

通常，在错误退出之前必须进行清理。

这样做是很混乱的：

```

std::pair<int, error_indicator> user()
{
    Gadget g1 = make_gadget(17);
    if (!g1.valid()) {
        return {0, g1_error};
    }

    Gadget g2 = make_gadget(17);
    if (!g2.valid()) {
        cleanup(g1);
        return {0, g2_error};
    }

    // ...

    if (all_foobar(g1, g2)) {
        cleanup(g2);
        cleanup(g1);
    }
}

```

```

        return {0, foobar_error};
    }

// ...

cleanup(g2);
cleanup(g1);
return {res, 0};
}

```

模拟 RAII 可能不那么简单，尤其是在带有多个资源和多种可能错误的函数之中。

一种较为常见的技巧是把清理都集中到函数末尾以避免重复（注意这里本不必为 `g2` 增加一层作用域，但是编译 `goto` 版本却需要它）：

```

std::pair<int, error_indicator> user()
{
    error_indicator err = 0;
    int res = 0;

    Gadget g1 = make_gadget(17);
    if (!g1.valid()) {
        err = g1_error;
        goto g1_exit;
    }

    {
        Gadget g2 = make_gadget(31);
        if (!g2.valid()) {
            err = g2_error;
            goto g2_exit;
        }

        if (all_foobar(g1, g2)) {
            err = foobar_error;
            goto exit;
        }
    }

    // ...

g2_exit:
    if (g2.valid()) cleanup(g2);
}

g1_exit:
    if (g1.valid()) cleanup(g1);
    return {res,err};
}

```

函数越大，这种技巧就越有吸引力。

`finally` 可以[略微缓解这个问题](#)。

而且，程序变得越大，系统化地采用一中基于错误指示的错误处理策略就越加困难。

我们[优先采用基于异常的错误处理](#)，并建议[保持函数短小](#)。

参见: [讨论](#)

参见: [返回多个值](#)

强制实施

很难对付。

E.28: 避免基于全局状态（比如 `errno`）的错误处理

理由

全局状态难于管理，且易于忘记检查。

你上次检查 `printf()` 的返回值是什么时候了？

参见: [模拟 RAII](#)

示例，不好

```
int last_err;

void f(int n)
{
    // ...
    p = static_cast<X*>(malloc(n * sizeof(X)));
    if (!p) last_err = -1;      // 当内存耗尽时发生的错误
    // ...
}
```

注解

C 风格的错误处理就是基于全局变量 `errno` 的，因此基本上不可能完全避免这种风格。

强制实施

很难对付。

E.30: 请勿使用异常说明

理由

异常说明使得错误处理变得脆弱，隐含一些运行时开销，并且已经从 C++ 标准中被删除了。

示例

```
int use(int arg)
    throw(X, Y)
{
    // ...
    auto x = f(arg);
    // ...
}
```

当 `f()` 抛出了不同于 `X` 和 `Y` 的异常时将会执行未预期异常处理器，其默认将终止程序。

这没什么问题，但假定我们检查过着并不会发生而 `f` 则被改写为抛出某个新异常 `Z`，

这样将导致程序崩溃，除非我们改写 `use()` (并重新测试所有东西)。

障碍在于 `f()` 可能在某个我们无法控制的程序库中，而对于新的异常 `use()`

没办法对其做任何事，或者对其完全不感兴趣。

我们可以改写 `use()` 使其传递 `z` 出去，但这样的话 `use()` 的调用方可能也需要被改写。

如此事态将很快变得无法掌控。

或者，我们可以在 `use()` 中添加一个 `try-catch` 以将 `z` 映射为某种可以接受的异常。

这种方法也会很快变得无法掌控。

注意，对异常集合的改动通常都发生在系统的最底层

(比如说，当改换了网络库或者某种中间件时)，因此改变将沿着冗长的调用链“冒泡上浮”。

在大型代码库中，这将意味着直到最后一个使用方也被改写之前，没人可以更新某个库到新版本。

如果 `use()` 是某个库的一部分，则也许不可能对其进行更新，因为其改动可能影响到未知的客户代码。

而让异常继续传递直到其到达某个潜在可以处理它的函数的策略，已经在多年的实践中得到了证明。

注解

静态强制检查异常说明并不会带来任何好处。

相关例子请参见 [Stroustrup94](#)。

注解

当不会抛出异常时，请使用 `noexcept`。

强制实施

标记每个异常说明。

E.31: 恰当地对 `catch` 子句排序

理由

`catch` 子句是以其出现顺序依次求值的，而其中一个可能会隐藏掉另一个。

示例，不好

```
void f()
{
    // ...
    try {
        // ...
    }
    catch (Base& b) { /* ... */ }
    catch (Derived& d) { /* ... */ }
    catch (...) { /* ... */ }
    catch (std::exception& e) { /* ... */ }
}
```

若 `Derived` 派生自 `Base` 则 `Derived` 的处理器永远不会被执行。

“捕获任何东西”的处理器保证 `std::exception` 的处理器永远不会被执行。

强制实施

标记出所有的“隐藏处理器”。

Con: 常量与不可变性

常量是不会出现竞争条件的。

当大量的对象不会改变它们的值时，对程序进行推理将变得更容易。

承诺“不改动”作为参数所传递对象的接口，极大地提升了可读性。

常量规则概览：

- [Con.1: 缺省情况下，对象应当是不可变的](#)
- [Con.2: 缺省情况下，成员函数应当为 `const`](#)
- [Con.3: 缺省情况下，应当传递指向 `const` 对象的指针或引用](#)
- [Con.4: 构造之后不再改变其值的对象应当以 `const` 来定义](#)
- [Con.5: 以 `constexpr` 来定义可以在编译期计算的值](#)

Con.1: 缺省情况下，对象应当是不可变的

理由

不可变对象更易于进行推理，应仅当需要改动对象的值时，才使之为非 `const` 对象。

避免出现意外造成的或者很难发觉的值的改变。

示例

```
for (const int i : c) cout << i << '\n';      // 仅进行读取: const  
for (int i : c) cout << i << '\n';           // 不好：仅进行读取
```

例外

按值传递的函数参数很少被改动，但也很少被声明为 `const`。

为了避免造成混淆和大量的误报，不要对函数参数实施这条规则。。

```
void f(const char* const p); // 迂腐  
void g(const int i) { ... } // 迂腐
```

注意，函数参数是局部变量，其改动也是局部的。

强制实施

- 标记未发生改动的非 `const` 变量（排除参数以避免误报）

Con.2: 缺省情况下，成员函数应当为 `const`

理由

除非成员函数会改变对象的可观测状态，否则它应当标记为 `const`。

这样做更精确地描述了设计意图，具有更佳的可读性，编译器可以识别更多的错误，而且有时能够带来更多的优化机会。

示例，不好

```

class Point {
    int x, y;
public:
    int getx() { return x; }      // 不好，应当为 const，它并不改变对象的状态
    // ...
};

void f(const Point& pt)
{
    int x = pt.getx();          // 错误，无法通过编译，因为 getx 并未标记为 const
}

```

注解

传递非 `const` 的指针或引用并非天生就是不好的，但应当只有在所调用的函数预计会修改这个对象时才这样做。代码的读者必须假定接受“普通的”`T*` 或 `T&` 的函数都将会修改其所指代的对象。如果它现在不会，那它可能以后会，且无需强制要求重新编译。

注解

有些代码和程序库提供的函数是声明为 `T*`，但这些函数并不会修改这个 `T`。这对于进行代码现代化转换的人们来说是个问题。你可以：

- 如果倾向于长期解决方案的话，将程序库更新为 `const` 正确的；
- “强制掉 `const`”（[最好避免这样做](#)）；
- 提供包装函数。

例如：

```

void f(int* p);    // 老代码: f() 并不会修改 `*p`
void f(const int* p) { f(const_cast<int*>(p)); } // 包装函数

```

注意，这种包装函数的方案是一种补丁，只能在无法修改 `f()` 的声明时才使用它，比如当它属于某个你无法修改的程序库时。

注解

`const` 成员函数可以改动 `mutable` 对象的值，或者通过某个指针成员改动对象的值。一种常见用法是来维护一个缓存以避免重复进行复杂的运算。例如，这里的 `Date` 缓存（记住）了其字符串表示，以简化其重复使用：

```

class Date {
public:
    // ...
    const string& string_ref() const
    {
        if (string_val == "") compute_string_rep();
        return string_val;
    }
    // ...
private:

```

```
void compute_string_rep() const; // 计算字符串表示并将其存入 string_val
mutable string string_val;
// ...
};
```

另一种说法是 `const` 特性不会传递。

通过 `const` 成员函数改动 `mutable` 成员的值和通过非 `const` 指针来访问的对象的值是有可能的。

由类负责确保这样的改动仅当根据其语义（不变式）对于其用户有意义时才会发生。

参见：[PImpl](#)

强制实施

- 如果未标记为 `const` 的成员函数并未对任何成员变量实施非 `const` 操作的话，对其进行标记。

Con.3: 缺省情况下，应当传递指向 `const` 对象的指针或引用

理由

避免所调用的函数意外地改变了这个值。

如果被调用的函数不会改动状态的话，对程序的推理将变得容易得多。

示例

```
void f(char* p); // f 会不会修改 *p? (假定它会修改)
void g(const char* p); // g 不会修改 *p
```

注解

传递指向非 `const` 对象的指针或引用并不是天生就有问题的，不过只有当所调用的函数本就有意改动对象时才能这样做。

注解

请勿强制掉 `const`。

强制实施

- 如果函数并未修改以指向非 `const` 的指针或引用传递的对象，则对其进行标记。
- 如果函数（利用强制转换）修改了以指向 `const` 的指针或引用传递的对象，则对其进行标记。

Con.4: 构造之后不再改变其值的对象应当以 `const` 来定义

理由

避免意外地改变对象的值。

示例

```
void f()
{
    int x = 7;
    const int y = 9;

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

既然 `x` 并非 `const`，我们就必须假定它可能在循环中的某处会被修改。

强制实施

- 标记并未被修改的非 `const` 变量。

Con.5: 以 `constexpr` 来定义可以在编译期计算的值

理由

更好的性能，更好的编译期检查，受保证的编译期求值，竞争条件可能性为零。

示例

```
double x = f(2);           // 可能在运行时求值
const double y = f(2);     // 可能在运行时求值
constexpr double z = f(2); // 除非 f(2) 可在编译期求值，否则会报错
```

注解

参见 F.4。

强制实施

- 对带有常量表达式初始化式的 `const` 定义进行标记。

T: 模板和泛型编程

泛型编程是使用以类型、值和算法进行参数化的类型和算法所进行的编程。

在 C++ 中，泛型编程是以“模板”语言机制来提供支持的。

泛型函数的参数是以针对参数类型及其所涉及的值的规定的集合来刻画的。

在 C++ 中，这些规定是以被称为概念（concept）的编译期谓词来表达的。

模板还可用于进行元编程；亦即由编译期代码所组成的程序。

泛型编程的中心是“概念”；亦即以编译时谓词表现的对于模板参数的要求。

C++20 已经将“概念”标准化了，不过是在 GCC 6.1 中以一种略有不同的语法首次提供的。

模板使用的规则概览：

- [T.1: 用模板来提升代码的抽象层次](#)
- [T.2: 用模板来表达适用于许多参数类型的算法](#)
- [T.3: 用模板来表达容器和范围](#)
- [T.4: 用模板来表达对语法树的操作](#)

- [T.5: 结合泛型和面向对象技术来增强它们的能力，而不是它们的成本](#)

概念使用的规则概览：

- [T.10: 为所有模板实参指明概念](#)
- [T.11: 尽可能采用标准概念](#)
- [T.12: 对于局部变量，优先采用概念名而不是 `auto`](#)
- [T.13: 对于简单的单类型参数概念，优先采用简写形式](#)
- ???

概念定义的规则概览：

- [T.20: 避免没有有意义的语义的“概念”](#)
- [T.21: 为概念提出一组完整的操作要求](#)
- [T.22: 为概念指明公理](#)
- [T.23: 通过添加新的使用模式，从更一般情形的概念中区分出提炼后的概念](#)
- [T.24: 用标签类或特征类来区分仅在语义上存在差别的概念](#)
- [T.25: 避免互补性的约束](#)
- [T.26: 优先采用使用模式而不是简单的语法来定义概念](#)
- [T.30: 节制地采用概念求反 \(`!C<T>`\) 来表示微小差异](#)
- [T.31: 节制地采用概念析取 \(disjunction\) \(`C1<T> || C2<T>`\) 来表示备选项](#)
- ???

模板接口的规则概览：

- [T.40: 使用函数对象向算法传递操作](#)
- [T.41: 在模板的概念上仅提出基本的性质要求](#)
- [T.42: 使用模板别名来简化写法并隐藏实现细节](#)
- [T.43: 优先使用 `using` 而不是 `typedef` 来定义别名](#)
- [T.44: \(如果可行\) 使用函数模板来对类模板的参数类型进行推断](#)
- [T.46: 要求模板参数至少是半正规的](#)
- [T.47: 避免用常用名字命名高度可见的无约束模板](#)
- [T.48: 如果你的编译器不支持概念的话，可以用 `enable_if` 来模拟](#)
- [T.49: 尽可能避免类型擦除](#)

模板定义的规则概览：

- [T.60: 最小化模板的上下文依赖性](#)
- [T.61: 请勿对成员进行过度参数化 \(SCARY\)](#)
- [T.62: 将无依赖的类模板成员置于一个非模板基类之中](#)
- [T.64: 用特化来提供类模板的其他实现](#)
- [T.65: 用标签分派来提供函数的其他实现](#)
- [T.67: 用特化来提供不规则类型的其他实现](#)
- [T.68: 在模板中用 `{}` 而不是 `{}<T>` 以避免歧义](#)
- [T.69: 在模板中，请勿进行未限定的非成员函数调用，除非有意将之作为定制点](#)

模板和类型层次的规则概览：

- [T.80: 请勿不成熟地对类层次进行模板化](#)
- [T.81: 请勿混合类层次和数组 // ??? 放在“类型层次”部分](#)
- [T.82: 当不想要虚函数时，可以将类层次线性化](#)
- [T.83: 请勿声明虚的成员函数模板](#)
- [T.84: 使用非模板的核心实现来提供 ABI 稳定的接口](#)
- T.???: ???

变参模板的规则概览：

- [T.100: 当需要可以接受可变数量的多种类型参数的函数时，使用变参模板](#)
- [T.101: ??? 如何向变参模板传递参数 ???](#)
- [T.102: ??? 如何处理变参模板的参数 ???](#)
- [T.103: 请勿对同质参数列表使用变参模板](#)
- [T.?: ???](#)

元编程的规则概览：

- [T.120: 仅当确实需要时才使用模板元编程](#)
- [T.121: 模板元编程主要用于模拟概念机制](#)
- [T.122: 用模板（通常为模板别名）来在编译期进行类型运算](#)
- [T.123: 用 `constexpr` 函数来在编译期进行值运算](#)
- [T.124: 优先使用标准库的模板元编程设施](#)
- [T.125: 当需要标准库之外的模板元编程设施时，使用某个现存程序库](#)
- [T.?: ???](#)

其他模板规则概览：

- [T.140: 若操作可被重用，则应为其命名](#)
- [T.141: 当仅在一个地方需要一个简单的函数对象时，使用无名的 lambda](#)
- [T.142: 使用模板变量以简化写法](#)
- [T.143: 请勿编写并非有意非泛型的代码](#)
- [T.144: 请勿特化函数模板](#)
- [T.150: 用 `static_assert` 来检查类是否与概念相符](#)
- [T.?: ???](#)

T.gp: 泛型编程

泛型编程是利用以类型、值和算法进行了参数化的类型和算法所进行的编程。

T.1: 用模板来提升代码的抽象层次

理由

通用性。重用。效率。鼓励用户类型的定义一致性。

示例，不好

概念上说，以下要求是错误的，因为我们所要求的 `T` 不止是如“可以增量”或“可以进行加法”这样的非常低级的概念：

```
template<typename T>
    requires Incrementable<T>
T sum1(vector<T>& v, T s)
{
    for (auto x : v) s += x;
    return s;
}

template<typename T>
    requires Simple_number<T>
T sum2(vector<T>& v, T s)
{
```

```
    for (auto x : v) s = s + x;
    return s;
}
```

由于假设 `Incrementable` 并不支持 `+`，而 `SimpleNumber` 也不支持 `+=`，我们对 `sum1` 和 `sum2` 的实现者过度约束了。而且，这种情况下也错过了一次通用化的机会。

示例

```
template<typename T>
requires Arithmetic<T>
T sum(vector<T>& v, T s)
{
    for (auto x : v) s += x;
    return s;
}
```

通过假定 `Arithmetic` 同时要求 `+` 和 `+=`，我们约束 `sum` 的使用者来提供完整的算术类型。这并非是最小的要求，但它为算法的实现者更多所需的自由度，并确保了任何 `Arithmetic` 类型都可被用于各种各样的算法。

为达成额外的通用性和可重用性，我们还可以用更通用的 `Container` 或 `Range` 概念而不是仅投入到 `vector` 一个容器上。

注解

如果我们所定义的模板提出的要求，完全是由一个算法的一个实现所要求的操作（比如说仅要求 `+=` 而不同时要求 `=` 和 `+`），而没有别的要求，就会对维护者提出过度的约束。我们的目标是最小化模板参数的要求，但一个实现的绝对最小要求很少能作为有意义的概念。

注解

可以用模板来表现几乎任何东西（它是图灵完备的），但（利用模板进行）泛型编程的目标在于，使操作和算法对于一组带有相似语义性质的类型有效地进行通用化。

强制实施

- 对带有“过于简单”的要求（诸如不用概念而直接使用特定的运算符）的算法进行标记。
- 不要对“过于简单”的概念本身进行标记；它们也许只不过是更有用的概念的构造块。

T.2: 用模板来表达适用于许多参数类型的算法

理由

通用性。最小化源代码数量。互操作性。重用。

示例

这正是 STL 的基础所在。一个 `find` 算法可以很轻易地同任何输入范围一起工作：

```
template<typename Iter, typename Val>
// requires Input_iterator<Iter>
//     && Equality_comparable<value_type<Iter>, Val>
Iter find(Iter b, Iter e, Val v)
{
    // ...
}
```

注解

除非确实需要多于一种模板参数类型，否则请不要使用模板。
请勿过度抽象。

强制实施

??? 很难办，可能需要人来进行

T.3: 用模板来表达容器和范围

理由

容器需要一种元素类型，而将之表示为一个模板参数则是通用的，可重用的，且类型安全的做法。
这样也避免了采用脆弱的或者低效的变通手段。约定：这正是 STL 的做法。

示例

```
template<typename T>
// requires Regular<T>
class Vector {
    // ...
    T* elem;    // 指向 sz 个 T
    int sz;
};

Vector<double> v(10);
v[7] = 9.9;
```

示例，不好

```
class Container {
    // ...
    void* elem;    // 指向 sz 个某种类型的元素
    int sz;
};

Container c(10, sizeof(double));
((double*) c.elem)[7] = 9.9;
```

这样做无法直接表现出程序员的意图，而且向类型系统和优化器隐藏了程序的结构。

把 `void*` 隐藏在宏之中只会掩盖问题，并引入新的发生混乱的机会。

例外：当需要 ABI 稳定的接口时，也许你不得不提供一个基本实现，在基于它来呈现（类型安全的）目标。

参见[稳定的基类](#)。

强制实施

- 对出现于低级实现代码之外的 `void*` 和强制转换进行标记。

T.4: 用模板来表达对语法树的操作

理由

???

示例

```
???
```

例外: ???

T.5: 结合泛型和面向对象技术来增强它们的能力，而不是它们的成本

理由

泛型和面向对象技术是互补的。

示例

静态能够帮助动态：使用静态多态来实现动态多态的接口：

```
class Command {  
    // 纯虚函数  
};  
  
// 实现  
template</*...*/>  
class ConcreteCommand : public Command {  
    // 实现虚函数  
};
```

示例

动态有助于静态：提供通用的，便利的，静态绑定的接口，但内部进行动态派发，这样就可以提供统一的对象布局。

这样的例子包括如 `std::shared_ptr` 的删除器的类型擦除。 (不过[请勿过度使用类型擦除](#)。)

```
#include <memory>  
  
class Object {  
public:  
    template<typename T>  
    Object(T&& obj)  
        : concept_(std::make_shared<ConcreteCommand<T>>(std::forward<T>(obj)))  
    {}  
  
    int get_id() const { return concept_->get_id(); }  
  
private:  
    struct Command {  
        virtual ~Command() {}  
    };  
    std::shared_ptr<Command> concept_;
```

```

    virtual int get_id() const = 0;
};

template<typename T>
struct ConcreteCommand final : Command {
    ConcreteCommand(T&& obj) noexcept : object_(std::forward<T>(obj)) {}
    int get_id() const final { return object_.get_id(); }

private:
    T object_;
};

std::shared_ptr<Command> concept_;
```

}

```

class Bar {
public:
    int get_id() const { return 1; }
};

struct Foo {
public:
    int get_id() const { return 2; }
};

Object o(Bar{});
Object o2(Foo{});

```

注解

类模板之中，非虚函数仅会在被使用时才会实例化——而虚函数则每次都会实例化。这会导致代码大小膨胀，而且可能因为实例化从不需要的功能而导致对通用类型的过度约束。请避免这样做，虽然标准的刻面类犯过这种错误。

参见

- [ref ???](#)
- [ref ???](#)
- [ref ???](#)

强制实施

参见参考条目以获得更具体的规则。

T.concepts: 概念规则

概念是一种 C++20 语言设施，用于为模板参数指定要求。在考虑泛型编程，以及未来的 C++ 程序库（无论标准的还是其他的）的基础时，概念都是关键性的。

本部分假定有概念支持。

概念使用的规则概览：

- [T.10: 为所有模板实参指明概念](#)
- [T.11: 尽可能采用标准概念](#)
- [T.12: 优先采用概念名而不是 auto](#)

- [T.13: 对于简单的单类型参数概念，优先采用简写形式](#)
- ???

概念定义的规则概览：

- [T.20: 避免没有有意义的语义的“概念”](#)
- [T.21: 为概念提出一组完整的操作要求](#)
- [T.22: 为概念指明公理](#)
- [T.23: 通过添加新的使用模式，从更一般情形的概念中区分出提炼后的概念](#)
- [T.24: 用标签类或特征类来区分仅在语义上存在差别的概念](#)
- [T.25: 避免互补性的约束](#)
- [T.26: 优先采用使用模式而不是简单的语法来定义概念](#)
- ???

T.con-use: 概念的使用

T.10: 为所有模板实参指明概念

理由

正确性和可读性。

针对模板参数所假定的含义（包括语法和语义），是模板接口的基础部分。

使用概念能够显著改善模板的文档和错误处理。

为模板参数指定概念是一种强力的设计工具。

示例

```
template<typename Iter, typename Val>
    requires input_iterator<Iter>
        && equality_comparable_with<value_type_t<Iter>, Val>
Iter find(Iter b, Iter e, Val v)
{
    // ...
}
```

也可以等价地用更为简洁的方式：

```
template<input_iterator Iter, typename Val>
    requires equality_comparable_with<value_type_t<Iter>, Val>
Iter find(Iter b, Iter e, Val v)
{
    // ...
}
```

注解

普通的 `typename`（或 `auto`）是受最少约束的概念。

应当仅在只能假定“这是一个类型”的罕见情况中使用它们。

通常，这只会在当我们（用模板元编程代码）操作纯粹的表达式树，并推迟进行类型检查时才会需要。

参考: TC++PL4

强制实施

对没有概念的模板类型参数进行标记。

T.11: 尽可能采用标准概念

理由

“标准”概念（即由 [GSL](#) 和 ISO 标准自身所提供的概念），避免了我们思考自己的概念，它们比我们匆忙中能够想出来的要好得多，而且还提升了互操作性。

注解

如果你不是要创建一个新的泛型程序库的话，大多数所需的概念都已经在标准库中定义过了。

示例

```
template<typename T>
    // 请勿定义这个：<iterator> 中已经有 sortable
concept Ordered_container = Sequence<T> && Random_access<Iterator<T>> &&
Ordered<value_type<T>>;

void sort(Ordered_container auto& s);
```

这个 `ordered_container` 貌似相当合理，但它和标准库中的 `sortable` 概念非常相似。

它是更好？更正确？它真的精确地反映了标准对于 `sort` 的要求吗？

直接使用 `sortable` 则更好而且更简单：

```
void sort(sortable auto& s); // 更好
```

注解

在我们推进一个包含概念的 ISO 标准的过程中，“标准”概念的集合是不断演进的。

注解

设计一个有用的概念是很有挑战性的。

强制实施

很难。

- 查找无约束的参数，使用“非常规”或非标准的概念的模板，以及使用“自造的”又没有公理的概念的目标。
- 开发一种概念识别工具（例如，参考[一种早期实验](#)）。

T.12: 对于局部变量，优先采用概念名而不是 `auto`

理由

`auto` 是最弱的概念。概念的名字会比仅用 `auto` 传达出更多的意义。

示例

```
vector<string> v{ "abc", "xyz" };
auto& x = v.front();           // 不好
String auto& s = v.front(); // 好 (String 是 GSL 的一个概念)
```

强制实施

- ???

T.13: 对于简单的单类型参数概念，优先采用简写形式

理由

可读性。直接表达意图。

示例

这样来表达“T 是一种 sortable”：

```
template<typename T>          // 正确但很啰嗦：“参数的类型
    requires sortable<T>        // 为 T，这是某个 sortable
void sort(T&);                // 类型的名字”

template<sortable T>          // 有改善：“参数的类型
void sort(T&);                // 为 Sortable 的类型 T”

void sort(sortable auto&); // 最佳方式：“参数为 sortable”
```

越简练的版本越符合我们的说话方式。注意许多模板不在需要使用 `template` 关键字了。

强制实施

- 当人们从 `<typename T>` 和 `<class T>` 写法进行转换时，使用简短形式是不可行的。
- 之后，如果声明中首先引入了一个 `typename`，之后又用简单的单类型参数概念对其进行约束的话，就对其进行标记。

T.concepts.def: 概念定义规则

定义恰当的概念并不简单。

概念是用于表现应用领域中的基本概念的（正如其名“概念”）。

只是把用在某个具体的类或算法的参数上的一组语法约束聚在一起，并不是概念所设计的用法，也无法获得这个机制的全部好处。

显然，定义概念对于那些可以使用某个实现（比如 C++20 或更新版本）的代码是最有用的，不过定义概念本身就是一种有益的设计技巧，有助于发现概念上的错误并清理实现中的各种概念。

T.20: 避免没有意义的语义的“概念”

理由

概念是用于表现语义的观念的，比如“数”，元素的“范围”，以及“全序的”等等。

简单的约束，比如“带有 `+` 运算符”和“带有 `>` 运算符”，是无法独立进行有意义的运用的，而仅应当被用作有意义的概念的构造块，而不是在用户代码中使用。

示例，不好

```
template<typename T>
// 不好，不充分
concept Addable = requires(T a, T b) { a+b; };

template<Addable N>
auto algo(const N& a, const N& b) // 使用两个数值
{
    // ...
    return a + b;
}

int x = 7;
int y = 9;
auto z = algo(x, y); // z = 16

string xx = "7";
string yy = "9";
auto zz = algo(xx, yy); // zz = "79"
```

也许拼接是有意进行的。不过更可能的是一种意外。而对减法进行同等的定义则会导致可接受类型的集合的非常不同。

这个 `Addable` 违反了加法应当可交换的数学法则：`a+b == b+a`。

注解

给出有意义的语义的能力，在于定义真正的概念的特征，而不是仅给出语法约束。

示例

```
template<typename T>
// 假定数值的运算符 +、-、* 和 / 都遵循常规的数学法则
concept Number = requires(T a, T b) { a+b; a-b; a*b; a/b; };

template<Number N>
auto algo(const N& a, const N& b)
{
    // ...
    return a + b;
}

int x = 7;
int y = 9;
auto z = algo(x, y); // z = 16

string xx = "7";
string yy = "9";
auto zz = algo(xx, yy); // 错误: string 不是 Number
```

注解

带有多个操作的概念要远比单个操作的概念更少和类型发生意外匹配的机会。

强制实施

- 对在其他 `concept` 的定义之外使用的但操作 `concept` 进行标记。
- 对表现为模拟单操作 `concept` 的 `enable_if` 的使用进行标记。

T.21: 为概念提出一组完整的操作要求

理由

易于理解。

提升互操作性。

帮助实现者和维护者。

注解

这是对一般性规则[必须让概念有语义上的意义](#)的一个专门的变体。

示例，不好

```
template<typename T> concept Subtractable = requires(T a, T b) { a-b; };
```

这个是没有语义作用的。

你至少还需要 `+` 来让 `-` 有意义和有用处。

完整集合的例子有

- `Arithmetic`: `+, -, *, /, +=, -=, *=, /=`
- `Comparable`: `<, >, <=, >=, ==, !=`

注解

无论我们是否使用了概念的直接语言支持，本条规则都适用。

这是一种一般性的设计规则，即便对于非模板也同样适用：

```
class Minimal {
    // ...
};

bool operator==(const Minimal&, const Minimal&);
bool operator<(const Minimal&, const Minimal&);

Minimal operator+(const Minimal&, const Minimal&);
// 没有其他运算符

void f(const Minimal& x, const Minimal& y)
{
    if (!x == y) { /* ... */ }      // OK
    if (x != y) { /* ... */ }      // 意外！错误

    while (!x < y) { /* ... */ }   // OK
    while (x >= y) { /* ... */ }   // 意外！错误

    x = x + y;                   // OK
```

```
x += y; // 意外！错误  
}
```

这是最小化的设计，但会使用户遇到意外或受到限制。
可能它还会比较低效。

这条规则支持这样的观点：概念应当反映（数学上）协调的一组操作。

示例

```
class Convenient {  
    // ...  
};  
  
bool operator==(const Convenient&, const Convenient&);  
bool operator<(const Convenient&, const Convenient&);  
// ... 其他比较运算符 ...  
  
Convenient operator+(const Convenient&, const Convenient&);  
// ... 其他算术运算符 ...  
  
void f(const Convenient& x, const Convenient& y)  
{  
    if (!(x == y)) { /* ... */ } // OK  
    if (x != y) { /* ... */ } // OK  
  
    while (!(x < y)) { /* ... */ } // OK  
    while (x >= y) { /* ... */ } // OK  
  
    x = x + y; // OK  
    x += y; // OK  
}
```

定义所有的运算符也许很麻烦，但并不困难。
理想情况下，语言应当默认提供比较运算符以支持这条规则。

强制实施

- 如果类所支持的运算符是运算符集合的“奇异”子集，比如有 `==` 但没有 `!=` 或者有 `+` 但没有 `-`，就对其进行标记。
确实，`std::string` 也是“奇异”的，但要修改它太晚了。

T.22: 为概念指明公理

理由

有意义或有用的概念都有语义上的含义。
以非正式、半正式或正式的方式表达这些语义可以使概念对于读者更加可理解，而且对其进行表达的工作也能发现一些概念上的错误。
对语义的说明是一种强大的设计工具。

示例

```
template<typename T>
// 假定数值的运算符 +、-、* 和 / 都遵循常规的数学法则
// axiom(T a, T b) { a + b == b + a; a - a == 0; a * (b + c) == a * b + a *
c; /*...*/ }
concept Number = requires(T a, T b) {
    {a + b} -> convertible_to<T>;
    {a - b} -> convertible_to<T>;
    {a * b} -> convertible_to<T>;
    {a / b} -> convertible_to<T>;
};
```

注解

这是一种数学意义上的公理：可以假定成立而无需证明。

通常公理是无法证明的，而即便可以证明，通常也超出了编译器的能力。

一条公理也许并不是通用的，不过模板作者可以假定其对于其所实际使用的所有输入均成立（类似于一种前条件）。

注解

这种上下文中的公理都是布尔表达式。

参见 [Palo Alto TR](#) 中的例子。

当前，C++ 并不支持公理（即便 ISO Concepts TS 也不支持），我们不得不长期用代码注释来给出它们。

语言支持一旦出现，公理前面的 // 就可以删掉了。

注解

GSL 中的概念都具有恰当定义的语义；请参见 Palo Alto TR 和 Ranges TS。

例外

一个处于开发之中的新“概念”的早期版本，经常会仅仅定义了一组简单的约束而并没有恰当指定的语义。

为其寻找正确的语义可能是很费功夫和时间的。

不过不完整的约束集合仍然是很有用的：

```
// 对于一般二叉树的平衡器
template<typename Node> concept Balancer = requires(Node* p) {
    add_fixup(p);
    touch(p);
    detach(p);
};
```

这样 `Balancer` 就必须为树的 `Node` 至少提供这些操作，

但我们还是无法指定详细的语义，因为一种新种类的平衡树可能需要更多的操作，而在设计的早期阶段，很难确定把对于所有节点的精确的一般语义所确定下来。

不完整的或者没有恰当指定的语义的“概念”仍然是很有用的。

比如说，它可能允许在初始的试验之中进行某些检查。

不过，请不要把它当作是稳定的。

它的每个新的用例都可能导致不完整概念的改善。

强制实施

- 在概念定义的代码注释中寻找单词“axiom”。

T.23: 通过添加新的使用模式，从更一般情形的概念中区分出提炼后的概念

理由

否则编译器是无法自动对它们进行区分的。

示例

```
template<typename I>
// 注: <iterator> 中定义了 input_iterator
concept Input_iter = requires(I iter) { ++iter; };

template<typename I>
// 注: <iterator> 中定义了 forward_iterator
concept Fwd_iter = Input_iter<I> && requires(I iter) { iter++; };
```

编译器可以基于所要求的操作的集合（这里为前缀 `++`）来确定提炼关系。

这样做减少了这些类型的实现者的负担，

因为他们不再需要任何特殊的声明来“打入概念内部”了。

如果两个概念具有完全相同的要求的话，它们在逻辑上就是等价的（不存在提炼）。

强制实施

- 对与已经出现的另一个概念具有完全相同的要求的概念进行标记（它们中不存在更精炼的概念）。
为对它们进行区分，参见 [T.24](#)。

T.24: 用标签类或特征类来区分仅在语义上存在差别的概念

理由

要求相同的语法但具有不同语义的两个概念之间会造成歧义，除非程序员对它们进行区分。

示例

```
template<typename I> // 提供随机访问的迭代器
// 注: <iterator> 中定义了 random_access_iterator
concept RA_iter = ...;

template<typename I> // 提供对连续数据的随机访问的迭代器
// 注: <iterator> 中定义了 contiguous_iterator
concept Contiguous_iter =
    RA_iter<I> && is_contiguous_v<I>; // 使用 is_contiguous 特征
```

程序员（在程序库中）必须适当地定义（特征） `is_contiguous`。

把标签类包装到概念中可以得到这个方案的更简单的表达方式：

```
template<typename I> concept Contiguous = is_contiguous_v<I>;  
  
template<typename I>  
concept Contiguous_iter = RA_iter<I> && Contiguous<I>;
```

程序员（在程序库中）必须适当地定义（特征）`is_contiguous`。

注解

特征可以是特征类或者类型特征。
它们可以是用户定义的，或者标准库中的。
优先采用标准库中的特征。

强制实施

- 编译器会将对相同的概念的有歧义的使用标记出来。
- 对相同的概念定义进行标记。

T.25: 避免互补性的约束

理由

清晰性。可维护性。
用否定来表达的具有互补要求的函数是很脆弱的。

示例

最初，人们总会试图定义带有互补要求的函数：

```
template<typename T>  
    requires !C<T>    // 不好  
void f();  
  
template<typename T>  
    requires C<T>  
void f();
```

这样会好得多：

```
template<typename T>    // 通用模板  
void f();  
  
template<typename T>    // 用概念进行特化  
    requires C<T>  
void f();
```

仅当`C<T>`无法满足时，编译器将会选择无约束的模板。
如果你并不想（或者无法）定义无约束版本的
`f()`的话，你可以删掉它。

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

编译器将会选取这个重载，或者给出一个适当的错误。

注解

很不幸，互补约束在 `enable_if` 代码中很常见：

```
template<typename T>
enable_if<!C<T>, void> // 不好
f();  
  
template<typename T>
enable_if<C<T>, void>
f();
```

注解

有时候会（不正确地）把对单个要求的互补约束当做是可以接受的。

不过，对于两个或更多的要求来说，所需要的定义的数量是按指数增长的（2, 4, 8, 16,）：

```
C1<T> && C2<T>
!C1<T> && C2<T>
C1<T> && !C2<T>
!C1<T> && !C2<T>
```

这样，犯错的机会也会倍增。

强制实施

- 对带有 `C<T>` 和 `!C<T>` 约束的函数对进行标记。

T.26: 优先采用使用模式而不是简单的语法来定义概念

理由

其定义更可读，而且更直接地对应于用户需要编写的代码。

其中同时兼顾了类型转换。你再不需要记住所有的类型特征的名字。

示例

你可能打算这样来定义概念 `Equality`：

```
template<typename T> concept Equality = has_equal<T> && has_not_equal<T>;
```

显然，直接使用标准的 `equality_comparable` 要更好而且更容易，

但是——只是一个例子——如果你不得不定义这样的概念的话，应当这样：

```
template<typename T> concept Equality = requires(T a, T b) {
    { a == b } -> std::convertible_to<bool>;
    { a != b } -> std::convertible_to<bool>;
    // axiom { !(a == b) == (a != b) }
    // axiom { a = b; -> a == b } // => 的意思是“意味着”
};
```

而不是定义两个无意义的概念 `has_equal` 和 `has_not_equal` 仅用于帮助 `Equality` 的定义。

“无意义”的意思是我们无法独立地指定 `has_equal` 的语义。

强制实施

???

模板接口

这些年以来，使用模板的编程一直忍受着模板的接口及其实现之间的微弱区分性。

在引入概念之前，这种区分是没有直接的语言支持的。

不过，模板的接口是一个关键概念——是用户和实现者之间的一种契约——而且应当进行周密的设计。

T.40: 使用函数对象向算法传递操作

理由

函数对象比“普通”的函数指针能够向接口传递更多的信息。

一般来说，传递函数对象比传递函数指针能带来更好的性能。

示例

```
bool greater(double x, double y) { return x > y; }  
sort(v, greater); // 函数指针: 可能较慢  
sort(v, [](double x, double y) { return x > y; }); // 函数对象  
sort(v, std::greater{}); // 函数对象  
  
bool greater_than_7(double x) { return x > 7; }  
auto x = find_if(v, greater_than_7); // 函数指针: 不灵活  
auto y = find_if(v, [](double x) { return x > 7; }); // 函数对象: 携带所需数据  
auto z = find_if(v, Greater_than<double>(7)); // 函数对象: 携带所需数据
```

当然，也可以使用 `auto` 或概念来使这些函数通用化。例如：

```
auto y1 = find_if(v, [](totally_ordered auto x) { return x > 7; }); // 要求一种有序  
类型  
auto z1 = find_if(v, [](auto x) { return x > 7; }); // 期望类型带有  
>
```

注解

Lambda 会生成函数对象。

注解

性能表现依赖于编译器和优化器技术。

强制实施

- 标记以函数指针作为模板参数。
- 标记将函数指针作为模板的参数进行传递（存在误报风险）。

T.41: 在模板的概念上仅提出基本的性质要求

理由

保持接口的简单和稳定。

示例

考虑一个带有（过度简化的）简单调试支持的 `sort`：

```
void sort(sortable auto& s) // 对序列 s 进行排序
{
    if (debug) cerr << "enter sort( " << s << ")\n";
    // ...
    if (debug) cerr << "exit sort( " << s << ")\n";
}
```

是否该把它重写为：

```
template<sortable s>
    requires Streamable<s>
void sort(s& s) // 对序列 s 进行排序
{
    if (debug) cerr << "enter sort( " << s << ")\n";
    // ...
    if (debug) cerr << "exit sort( " << s << ")\n";
}
```

毕竟，`sortable` 里面并没有要求任何 `iostream` 支持。

而另一方面，在排序的基本概念中也没有任何东西是有关于调试的。

注解

如果我们要求把用到的所有操作都在要求部分中列出的话，接口就会变得不稳定：

每当我们改动了调试设施，使用情况数据收集，测试支持，错误报告等等，

模板的定义就都得进行改动，而模板的所有使用方都不得不进行重新编译。

这样做很是累赘，而且在某些环境下是不可行的。

与之相反，如果我们在实现中使用了某个并未被概念检查提供保证的操作的话，

我们可能会遇到一个延后的编译时错误。

通过不对模板参数的那些不被认为是基本的性质进行概念检查，

我们把其检查推迟到了进行实例化的时候。

我们认为这是值得做出的折衷。

注意，使用非局部的，非待决的名字（比如 `debug` 和 `cerr`），同样会引入可能导致“神秘的”错误的某些上下文依赖。

注解

可能很难确定一个类型的哪些性质是基本的，而哪些不是。

强制实施

???

T.42: 使用模板别名来简化写法并隐藏实现细节

理由

提升可读性。

隐藏实现。

注意，模板别名取代了许多用于计算类型的特征。

它们也可以用于封装一个特征。

示例

```
template<typename T, size_t N>
class Matrix {
    // ...
    using Iterator = typename std::vector<T>::iterator;
    // ...
};
```

这免除了 `Matrix` 的用户必须了解其元素是存储于 `vector` 之中，而且也免除了用户重复书写 `typename std::vector<T>::`。

示例

```
template<typename T>
void user(T& c)
{
    // ...
    typename container_traits<T>::value_type x; // 不好，啰嗦
    // ...
}

template<typename T>
using value_type = typename container_traits<T>::value_type;
```

这免除了 `value_type` 的用户必须了解用于实现 `value_type` 的技术。

```
template<typename T>
void user2(T& c)
{
    // ...
    value_type<T> x;
    // ...
}
```

注解

一种简洁的常用说法是：“包装特征！”

强制实施

- 将 `using` 声明之外用于消除歧义的 `typename` 进行标记。
- ???

T.43: 优先使用 `using` 而不是 `typedef` 来定义别名

理由

提升可读性：使用 `using` 时，新名字在前面，而不是被嵌在声明中的什么地方。

通用性：`using` 可用于模板别名，而 `typedef` 无法轻易作为模板。

一致性：`using` 在语法上和 `auto` 相似。

示例

```
typedef int (*PFI)(int);    // OK, 但很别扭

using PFI2 = int (*)(int);  // OK, 更好

template<typename T>
typedef int (*PFT)(T);      // 错误

template<typename T>
using PFT2 = int (*)(T);   // OK
```

强制实施

- 标记 `typedef` 的使用。不过这样会出现大量的“命中”:-)

T.44: (如果可行) 使用函数模板来对类模板的参数类型进行推断

理由

显式写出模板参数类型既麻烦又无必要。

示例

```
tuple<int, string, double> t1 = {1, "Hamlet", 3.14}; // 明确类型
auto t2 = make_tuple(1, "Ophelia"s, 3.14);           // 更好；推断类型
```

注意这里用 `s` 后缀来确保字符串是 `std::string` 而不是 C 风格字符串。

注解

既然你可以轻易写出 `make_T` 函数，编译器也可以。以后 `make_T` 函数也许将会变得多余。

例外

有时候没办法对模板参数进行推断，而有时候你则想要明确指定参数：

```
vector<double> v = { 1, 2, 3, 7.9, 15.99 };
list<Record*> lst;
```

注解

注意，C++17 强允许模板参数直接从构造函数参数进行推断，而使这条规则变得多余：

[构造函数的模板形参推断\(Rev. 3\)](#)。

例如：

```
tuple t1 = {1, "Hamlet's", 3.14}; // 推断为: tuple<int, string, double>
```

强制实施

当显式指定的类型与所使用的类型精确匹配时进行标记。

T.46: 要求模板参数至少是半正规的

理由

可读性。

避免意外和错误。

大多数用法都支持这样做。

示例

```
class X {
public:
    explicit X(int);
    X(const X&);           // 复制
    X operator=(const X&);
    X(X&&) noexcept;      // 移动
    X& operator=(X&&) noexcept;
    ~X();
    // ... 没有别的构造函数了 ...
};

X x {1};           // 没问题
X y = x;          // 没问题
std::vector<X> v(10); // 错误：没有默认构造函数
```

注解

`SemiRegular` 要求可以默认构造。

强制实施

- 对并非至少为 `SemiRegular` 的类型进行标记。

T.47: 避免用常用名字命名高度可见的无约束模板

示例

无约束的模板参数和任何东西都能完全匹配，因此这样的模板相对于需要少量转换的更特定类型来说可能更优先。

而当使用 ADL 时，这一点将会更加麻烦和危险。

而常用的名字则会让这种问题更易于出现。

示例

```
namespace Bad {
    struct S { int m; };
    template<typename T1, typename T2>
    bool operator==(T1, T2) { cout << "Bad\n"; return true; }
}

namespace T0 {
    bool operator==(int, Bad::S) { cout << "T0\n"; return true; } // 与 int 比较

    void test()
    {
        Bad::S bad{ 1 };
        vector<int> v(10);
        bool b = 1 == bad;
        bool b2 = v.size() == bad;
    }
}
```

这将会打印出 `T0` 和 `Bad`。

这里 `Bad` 中的 `==` 有意设计为造成问题，不过你是否在真实代码中发现过这个问题呢？

问题在于 `v.size()` 返回的是 `unsigned` 整数，因此需要进行转换才能调用局部的 `==`；而 `Bad` 中的 `=` 则不需要任何转换。

实际的类型，比如标准库的迭代器，也可以被弄成类似的反社会倾向。

注解

如果在相同的命名空间中定义了一个无约束模板类型，这个无约束模板是可以通过 ADL 所找到的（如例子中所发生的一样）。就是说，它是高度可见的。

注解

这条规则应当是没有必要的，不过委员会在是否将无约束模板从 ADL 中排除上无法达成一致。

不幸的是，这会导致大量的误报；标准库也大量地违反了这条规则，它将许多无约束模板和类型都放入了单一的命名空间 `std` 之中。

强制实施

如果定义模板的命名空间中同样定义了具体的类型，就对其进行标记（可能在我们有概念支持之前都是不可行的）。

T.48: 如果你的编译器不支持概念的话，可以用 `enable_if` 来模拟

理由

因为这是我们没有直接的概念支持时能够做到的最好的方式。

`enable_if` 可被用于有条件地定义函数，以及用于在一组函数中进行选择。

示例

```
template<typename T>
enable_if_t<is_integral_v<T>>
f(T v)
{
    // ...
}

// 等价于:
template<Integral T>
void f(T v)
{
    // ...
}
```

注解

请当心互补约束。

当用 `enable_if` 来模拟概念重载时，有时候会迫使我们使用这种易错的设计技巧。

强制实施

???

T.49: 尽可能避免类型擦除

理由

类型擦除通过在一个独立的编译边界之后隐藏类型信息而招致一层额外的间接。

示例

```
???
```

例外: 有时候类型擦除是合适的，比如 `std::function`。

强制实施

???

注解

T.def: 模板定义

模板的定义式（无论是类还是函数）都可能包含任意的代码，因而只有相当范围的 C++ 编程技术才能覆盖这个主题。

不过，这个部分所关注的是特定于模板的实现的问题。

特别地说，这里关注的是模板定义式对其上下文之间的依赖。

T.60: 最小化模板的上下文依赖性

理由

便于理解。
最小化源于意外依赖的错误。
便于创建工具。

示例

```
template<typename C>
void sort(C& c)
{
    std::sort(begin(c), end(c)); // 必要并且有意义的依赖
}

template<typename Iter>
Iter algo(Iter first, Iter last)
{
    for (; first != last; ++first) {
        auto x = sqrt(*first); // 潜在的意外依赖: 哪个 sqrt()?
        helper(first, x); // 潜在的意外依赖:
                           // helper 是基于 first 和 x 所选择的
        TT var = 7; // 潜在的意外依赖: 哪个 TT?
    }
}
```

注解

通常模板是放在头文件中的，因此相对于 `.cpp` 文件之中的函数来说，其上下文依赖更加会受到 `#include` 的顺序依赖的威胁。

注解

让模板仅对其参数进行操作是一种把依赖减到最少的方式，但通常这是很难做到的。
比如说，一个算法通常会使用其他的算法，并且调用一些并非仅在参数上做动作的一些操作。
而且别再让我们从宏开始干活！

参见：[T.69.](#)

强制实施

??? 很麻烦

T.61: 请勿对成员进行过度参数化 (SCARY)

理由

不依赖于模板参数的成员，除非给定某个特定的模板参数，否则也是无法使用的。
这样会限制其使用，而且通常会增加代码大小。

示例，不好

```
template<typename T, typename A = std::allocator<T>>
// requires Regular<T> && Allocator<A>
class List {
public:
    struct Link { // 并未依赖于 A
        T elem;
```

```

    Link* pre;
    Link* suc;
};

using iterator = Link*;

iterator first() const { return head; }

// ...
private:
    Link* head;
};

List<int> lst1;
List<int, My_allocator> lst2;

```

这看起来没什么问题，但现在 `Link` 形成依赖于分配器（尽管它不使用分配器）。这迫使冗余的实例化在某些现实场景中可能造成出奇的高的成本。

通常，解决方案是使用自己的最小模板参数集使嵌套类非局部化。

```

template<typename T>
struct Link {
    T elem;
    Link* pre;
    Link* suc;
};

template<typename T, typename A = std::allocator<T>>
// requires Regular<T> && Allocator<A>
class List2 {
public:
    using iterator = Link<T>*;

    iterator first() const { return head; }

    // ...
private:
    Link<T>* head;
};

List2<int> lst1;
List2<int, My_allocator> lst2;

```

人们发现 `Link` 不再隐藏在列表中很可怕，所以我们命名这个技术为 [SCARY](#)。

引自该学术论文：“首字母缩略词 SCARY 描述了看似错误的赋值和初始化（受冲突的通用参数的约束），

但实际上使用了正确的实现（由于最小化的依赖而不受冲突的约束）。”

注解

这同样适用于那些并不依赖于其全部模板形参的 lambda 表达式。

强制实施

- 对并未依赖于全部模板形参的成员类型进行标记。
- 对并未依赖于全部模板形参的成员函数进行标记。
- 对并未依赖于全部模板形参的 lambda 表达式或变量模板进行标记。

T.62: 将无依赖的类模板成员置于一个非模板基类之中

理由

可以在使用基类成员时不需要指定模板参数，也不需要模板实例化。

示例

```
template<typename T>
class Foo {
public:
    enum { v1, v2 };
    // ...
};
```

???

```
struct Foo_base {
    enum { v1, v2 };
    // ...
};

template<typename T>
class Foo : public Foo_base {
public:
    // ...
};
```

注解

这条规则的更一般化的版本是，

“如果类模板的成员依赖于 M 个模板参数中的 N 个，就将它置于只有 N 个参数的基类之中。”

当 N == 1 时，可以如同 [T.61](#) 一样在一个基类和其外围作用域中的一个类之间进行选择。

??? 常量的情况如何？类的静态成员呢？

强制实施

- 标记 ???

T.64: 用特化来提供类模板的其他实现

理由

模板定义了通用接口。

特化是一种为这个接口提供替代实现的强大机制。

示例

??? 字符串的特化 (==)

??? 表示特化?

注解

???

强制实施

???

T.65: 用标签分派来提供函数的其他实现

理由

- 模板定义了通用接口。
- 标签派发允许我们基于参数类型的特定性质选择不同的实现。
- 性能。

示例

这是 `std::copy` 的一个简化版本 (忽略了非连续序列的可能性)

```
struct pod_tag {};
struct non_pod_tag {};

template<class T> struct copy_trait { using tag = non_pod_tag; }; // T 不是“朴素数据”

template<> struct copy_trait<int> { using tag = pod_tag; }; // int 是“朴素数据”

template<class Iter>
out copy_helper(Iter first, Iter last, Iter out, pod_tag)
{
    // 使用 memmove
}

template<class Iter>
out copy_helper(Iter first, Iter last, Iter out, non_pod_tag)
{
    // 使用调用复制构造函数的循环
}

template<class Iter>
out copy(Iter first, Iter last, Iter out)
{
    return copy_helper(first, last, out, typename
copy_trait<value_type<Iter>>::tag{})
}

void use(vector<int>& vi, vector<int>& vi2, vector<string>& vs, vector<string>& vs2)
```

```
{  
    copy(vi.begin(), vi.end(), vi2.begin()); // 使用 memmove  
    copy(vs.begin(), vs.end(), vs2.begin()); // 使用调用复制构造函数的循环  
}
```

这是一种进行编译时算法选择的通用且有力的技巧。

注解

当可以广泛使用 `concept` 之后，这样的替代实现就可以直接进行区分了：

```
template<class Iter>  
    requires Pod<Value_type<Iter>>  
out copy_helper(in, first, in last, out out)  
{  
    // 使用 memmove  
}  
  
template<class Iter>  
out copy_helper(in, first, in last, out out)  
{  
    // 使用调用复制构造函数的循环  
}
```

强制实施

???

T.67: 用特化来提供不规则类型的其他实现

理由

???

示例

```
???
```

强制实施

???

T.68: 在模板中用 `{}` 而不是 `()` 以避免歧义

理由

`()` 会带来文法歧义。

示例

```

template<typename T, typename U>
void f(T t, U u)
{
    T v1(T(u));      // 错误: 啊, v1 是函数而不是变量
    T v2{u};         // 清晰: 显然是变量
    auto x = T(u); // 不清晰: 构造还是强制转换?
}

f(1, "asdf"); // 不好: 从 const char* 强制转换为 int

```

强制实施

- 标记 `()` 初始化式。
- 标记函数风格的强制转换。

T.69: 在模板中, 请勿进行未限定的非成员函数调用, 除非有意将之作为定制点

理由

- 仅提供预计之内的灵活性。
- 避免源于意外的环境改变的威胁。

示例

主要有三种方法使调用代码对模板进行定制化。

```

template<class T>
// 调用成员函数
void test1(T t)
{
    t.f();      // 要求 T 提供 f()
}

template<class T>
void test2(T t)
// 不带限定地调用非成员函数
{
    f(t);      // 要求 f(/*T*/) 在调用方的作用域或者 T 的命名空间中可用
}

template<class T>
void test3(T t)
// 调用一个“特征”
{
    test_traits<T>::f(t); // 要求定制化 test_traits<>
                          // 以获得非默认的函数和类型
}

```

特征通常是以计算一个类型的类型别名,
用以计算一个值的 `constexpr` 函数,
或者针对用户的类型进行特化的传统的特征模板。

注解

当你打算为依赖于某个模板类型参数的值 `t` 调用自己的辅助函数 `helper(t)` 时，
请将函数放入一个 `::detail` 命名空间中，并把调用限定为 `detail::helper(t);`。
无限定的调用将成为一个定制点，它将会调用处于 `t` 的类型所在命名空间中的任何 `helper` 函数；
这可能会导致诸如[意外地调用了无约束函数模板](#)这样的问题。

强制实施

- 在模板中，如果非成员函数的无限定调用传递了具有依赖类型的变量，而在该模板的命名空间中存在相同名字的非成员函数，则对其进行标记。

T.temp-hier: 模板和类型层次规则：

模板是 C++ 为泛型编程提供支持的基石，而类层次则是 C++ 为面向对象编程提供支持的基石。

这两种语言机制可以有效地组合起来，但必须避免一些设计上的陷阱。

T.80: 请勿不成熟地对类层次进行模板化

理由

使一个带有许多函数，尤其是有许多虚函数的类层次进行模板化，会导致代码膨胀。

示例，不好

```
template<typename T>
struct Container {           // 这是一个接口
    virtual T* get(int i);
    virtual T* first();
    virtual T* next();
    virtual void sort();
};

template<typename T>
class Vector : public Container<T> {
public:
    // ...
};

Vector<int> vi;
Vector<string> vs;
```

把 `sort` 定义为容器的成员函数可能是一个比较糟糕的主意，不过这样做并不鲜见，而且它是一个展示不应当做的事情的好例子。

在这之中，编译器不知道 `Vector<int>::sort()` 是不是会被调用，因此它必须为之生成代码。

`Vector<string>::sort()` 也与此相似。

除非这两个函数被调用，否则这就是代码爆炸。

不难想象当一个带有几十个成员函数和几十个派生类的类层次被大量实例化时会怎么样。

注解

许多情况下都可以通过不为基类进行参数化而提供一个稳定的接口；
参见“[稳定的基类](#)”和 [OO 与 GP](#)。

强制实施

- 对依赖于模板参数的虚函数进行标记。 ??? 误报

T.81: 请勿混合类层次和数组

理由

派生类的数组可以隐式地“退化”成指向基类的指针，并带来潜在的灾难性后果。

示例

假定 `Apple` 和 `Pear` 是两种 `Fruit`。

```
void maul(Fruit* p)
{
    *p = Pear{};      // 把一个 Pear 放入 *p
    p[1] = Pear{};   // 把一个 Pear 放入 p[1]
}

Apple aa [] = { an_apple, another_apple }; // aa 包含的是 Apple (显然!)

maul(aa);
Apple& a0 = &aa[0]; // 是 Pear 吗?
Apple& a1 = &aa[1]; // 是 Pear 吗?
```

`aa[0]` 可能会变为 `Pear` (并且没进行过强制转换！)。

当 `sizeof(Apple) != sizeof(Pear)` 时，对 `aa[1]` 的访问就是并未跟数组中的对象的适当起始位置进行对齐的。

这里出现了类型违例，以及很可能出现的内存损坏。

决不要写这样的代码。

注意，`maul()` 违反了 [T* 应指向独立对象的规则](#)。

替代方案: 使用适当的 (模板化) 容器：

```
void maul2(Fruit* p)
{
    *p = Pear{};      // 把一个 Pear 放入 *p
}

vector<Apple> va = { an_apple, another_apple }; // va 包含的是 Apple (显然!)

maul2(va); // 错误：无法把 vector<Apple> 转换为 Fruit*
maul2(&va[0]); // 这是你明确要做的

Apple& a0 = &va[0]; // 是 Pear 吗?
```

注意，`maul2()` 中的赋值违反了[避免发生切片的规则](#)。

强制实施

- 对这种恐怖的东西进行检测！

T.82: 当不想要虚函数时，可以将类层次线性化

理由

???

示例

```
???
```

强制实施

???

T.83: 请勿声明虚的成员函数模板

理由

C++ 是不支持这样做的。

如果支持的话，就只能等到连接时才能生成 VTBL 了。

而且一般来说，各个实现还要搞定动态连接的问题。

示例，请勿如此

```
class Shape {  
    // ...  
    template<class T>  
    virtual bool intersect(T* p);    // 错误：模板不能为虚  
};
```

注解

我们保留这条规则是因为人们总是问这个问题。

替代方案

双派发或访问器模式，或者计算出所要调用的函数。

强制实施

编译器会处理这个问题。

T.84: 使用非模板的核心实现来提供 ABI 稳定的接口

理由

提升代码的稳定性。

避免代码爆炸。

示例

这个应当是基类：

```
struct Link_base { // 稳定
    Link_base* suc;
    Link_base* pre;
};

template<typename T> // 模板化的包装带来了类型安全性
struct Link : Link_base {
    T val;
};

struct List_base {
    Link_base* first; // 第一个元素（如果有）
    int sz;           // 元素数量
    void add_front(Link_base* p);
    // ...
};

template<typename T>
class List : List_base {
public:
    void put_front(const T& e) { add_front(new Link<T>{e}); } // 隐式强制转换为
Link_base
    T& front() { static_cast<Link<T>*>(first).val; } // 显式强制转换回 Link<T>
    // ...
};

List<int> li;
List<string> ls;
```

这样的话就只有一份用于对 `List` 的元素进行入链和解链的操作的代码了。

而类 `Link` 和 `List` 除了进行类型操作之外什么也没做。

除了使用一个独立的“base”类型外，另一种常用的技巧是对 `void` 或 `void*` 进行特化，并让针对 `T` 的通用模板成为在从或向 `void` 的核心实现进行强制转换的一层类型安全封装。

替代方案: 使用一个 [PImpl](#) 实现。

强制实施

???

T.var: 变参模板规则

???

T.100: 当需要可以接受可变数量的多种类型参数的函数时，使用变参模板

理由

变参模板是做到这点的最通用的机制，而且既高效又类型安全。请不要使用 C 的变参。

示例

```
??? printf
```

强制实施

- 对用户代码中 `va_arg` 的使用进行标记。

T.101: ??? 如何向变参模板传递参数 ???

理由

???

示例

```
??? 当心仅能移动参数和引用参数
```

强制实施

???

T.102: 如何处理变参模板的参数

理由

???

示例

```
??? 转发参数，类型检查，引用
```

强制实施

???

T.103: 请勿对同质参数列表使用变参模板

理由

存在更加正规的给出同质序列的方式，比如使用 `initializer_list`。

示例

```
???
```

强制实施

???

T.meta: 模板元编程 (TMP)

模板提供了一种编译期编程的通用机制。

元编程，是其中至少一项输入或者一项输出是类型的编程。

模板提供了编译期的图灵完备（除了内存消耗外）的鸭子类型系统。

其语法和所需技巧都相当可怕。

T.120: 仅当确实需要时才使用模板元编程

理由

模板元编程很难做对，它会拖慢编译速度，而且通常很难维护。

不过，现实世界有些例子中模板元编程提供了比其他专家级的汇编代码替代方案还要更好的性能。

而且，也存在现实世界的例子用模板元编程做到比运行时代码更好地表达基本设计意图的情况。

例如，当确实需要在编译期进行 AST 操作时，（比如说对矩阵操作进行可选的折叠），C++ 中可能没有其他的实现方式。

示例，不好

???

示例，不好

enable_if

请使用概念来替代它。不过请参见[如何在没有语言支持时模拟概念](#)。

示例

??? 好例子

替代方案: 如果结果是一个值而不是类型，请使用 [constexpr 函数](#)。

注解

当你觉得需要把模板元编程代码隐藏到宏之中时，你可能已经跑得太远了。

T.121: 模板元编程主要用于模拟概念机制

理由

不能使用 C++20 时，我们需要用 TMP 来模拟它。

对概念给出要求的用例（比如基于概念进行重载）是 TMP 的最常见（而且最简单）的用法。

示例

```
template<typename Iter>
/*requires*/ enable_if<random_access_iterator<Iter>, void>
advance(Iter p, int n) { p += n; }

template<typename Iter>
/*requires*/ enable_if<forward_iterator<Iter>, void>
advance(Iter p, int n) { assert(n >= 0); while (n--) ++p;}
```

注解

这种代码使用概念时将更加简单：

```
void advance(random_access_iterator auto p, int n) { p += n; }

void advance(forward_iterator auto p, int n) { assert(n >= 0); while (n--) ++p;}
```

强制实施

???

T.122: 用模板（通常为模板别名）来在编译期进行类型运算

理由

模板元编程是在编译期进行类型生成的受到直接支持和部分正规化的唯一方式。

注解

“特征 (Trait) ”技术基本上在计算类型方面被模板别名所代替，而在计算值方面则被 `constexpr` 函数所代替。

示例

```
??? 大型对象 / 小型对象的优化
```

强制实施

???

T.123: 用 `constexpr` 函数来在编译期进行值运算

理由

函数是用于表达计算一个值的最显然和传统的方式。

通常 `constexpr` 函数都比其他的替代方式具有更少的编译期开销。

注解

“特征 (Trait) ”技术基本上在计算类型方面被模板别名所代替，而在计算值方面则被 `constexpr` 函数所代替。

示例

```
template<typename T>
// requires Number<T>
constexpr T pow(T v, int n)    // 幂/指数
{
    T res = 1;
    while (n--) res *= v;
    return res;
}

constexpr auto f7 = pow(pi, 7);
```

强制实施

- 对产生值的模板元程序进行标记。它们应当被替换成 `constexpr` 函数。

T.124: 优先使用标准库的模板元编程设施

理由

标准中所定义的设施，诸如 `conditional`, `enable_if`, 以及 `tuple` 等，是可移植的，可以假定为大家所了解。

示例

```
???
```

强制实施

???

T.125: 当需要标准库之外的模板元编程设施时，使用某个现存程序库

理由

要搞出高级的 TMP 设施是很难的，而使用一个库则可让你进入某个（有希望收到支持的）社区。只有当确实不得不编写自己的“高级 TMP 支持”时，才应当这样做。

示例

```
???
```

强制实施

???

其他模板规则

T.140: 若操作可被重用，则应为其命名

参见 [F.10](#)

T.141: 当仅在一个地方需要一个简单的函数对象时，使用无名的 lambda

参见 [F.11](#)

T.142?: 使用模板变量以简化写法

理由

改善可读性。

示例

```
???
```

强制实施

```
???
```

T.143: 请勿编写并非有意非泛型的代码

理由

一般性。可重用性。请勿无必要地陷入技术细节之中；请使用最广泛可用的设施。

示例

用 `!=` 而不是 `<` 来比较迭代器；`!=` 可以在更多对象上正常工作，因为它并未蕴含有序性。

```
for (auto i = first; i < last; ++i) {    // 通用性较差
    // ...
}

for (auto i = first; i != last; ++i) {    // 好；通用性较强
    // ...
}
```

当然，范围式 `for` 在符合需求的时候当然是更好的选择。

示例

使用能够提供所需功能的最接近基类的类。

```
class Base {
public:
    Bar f();
    Bar g();
};

class Derived1 : public Base {
public:
    Bar h();
};

class Derived2 : public Dase {
public:
```

```

    Bar j();
};

// 不好，除非确实有特别的原因来将之仅限制为 Derived1 对象
void my_func(Derived1& param)
{
    use(param.f());
    use(param.g());
}

// 好，仅使用 Base 的接口，且保证了这个类型
void my_func(Base& param)
{
    use(param.f());
    use(param.g());
}

```

强制实施

- 对使用 `<` 而不是 `!=` 的迭代器比较进行标记。
- 当存在 `x.empty()` 或 `x.is_empty()` 时，对 `x.size() == 0` 进行标记。`empty()` 比 `size()` 能够对于更多的容器工作，因为某些容器是不知道自己的大小的，甚至概念上就是大小无界的。
- 如果函数接受指向更加派生的类型的指针或引用，但仅使用了在某个基类中所声明的函数，则对其进行标记。

T.144: 请勿特化函数模板

理由

根据语言规则，函数模板是无法被部分特化的。函数模板可以被完全特化，不过你基本上需要的都是进行重载而不是特化——因为函数模板特化并不参与重载，它们的行为和你想要的可能是不同的。少数情况下，应当通过你可以进行适当特化的类模板来进行真正的特化。

示例

```
???
```

例外: 当确实有特化函数模板的恰当理由时，请只编写一个函数模板，并使它委派给一个类模板，然后对这个类模板进行特化（这提供了编写部分特化的能力）。

强制实施

- 标记出所有的函数模板特化。代之以函数重载。

T.150: 用 `static_assert` 来检查类是否与概念相符

理由

当你打算使一个类符合某个概念时，应该提早进行验证以减少麻烦。

示例

```
class X {  
public:  
    X() = delete;  
    X(const X&) = default;  
    X(X&&) = default;  
    X& operator=(const X&) = default;  
    // ...  
};
```

在别的地方，也许是某个实现文件中，可以让编译器来检查 `X` 的所需各项性质：

```
static_assert(Default_constructible<X>);      // 错误：X 没有默认构造函数  
static_assert(Copyable<X>);                   // 错误：忘记定义 X 的移动构造函数了
```

强制实施

不可行。

CPL: C 风格的编程

C 和 C++ 是联系很紧密的两门语言。

它们都是源于 1978 年的“经典 C”语言的，且从此之后就在 ISO 标准委员会中进行演化。
为了让它们保持兼容，我们做过许多努力，但它们各自都并非是对方的子集。

C 规则概览：

- [CPL.1: 优先使用 C++ 而不是 C](#)
- [CPL.2: 当一定要用 C 时，应使用 C 和 C++ 的公共子集，并将 C 代码以 C++ 来编译](#)
- [CPL.3: 当一定要用 C 来作为接口时，应在使用这些接口的调用方代码中使用 C++](#)

CPL.1: 优先使用 C++ 而不是 C

理由

C++ 提供更好的类型检查和更多的语法支持。
它能为高层的编程提供更好的支持，而且通常会产生更快速的代码。

示例

```
char ch = 7;  
void* pv = &ch;  
int* pi = pv;    // 非 C++  
*pi = 999;       // 覆盖了 &ch 附近的 sizeof(int) 个字节
```

针对在 C 中从 `void*` 或向它进行的隐式强制转换的相关规则比较麻烦而且并未强制实施。
特别是，这个例子违反了禁止把类型转换为具有更严格对齐的类型的规则。

强制实施

使用 C++ 编译器。

CPL.2: 当一定要用 C 时，应使用 C 和 C++ 的公共子集，并将 C 代码以 C++ 来编译

理由

它们的子集语言，C 和 C++ 编译器都可以编译，而当作为 C++ 编译时，比“纯 C”进行更好的类型检查。

示例

```
int* p1 = malloc(10 * sizeof(int));           // 非 C++
int* p2 = static_cast<int*>(malloc(10 * sizeof(int))); // 非 C, C 风格的 C++
int* p3 = new int[10];                      // 非 C
int* p4 = (int*) malloc(10 * sizeof(int));    // C 和 C++ 均可
```

强制实施

- 当使用某种将代码作为 C 来编译的构建模式时进行标记。
 - C++ 将会确保代码是合法的 C++ 代码，除非使用了 C 扩展的编译器选项。

CPL.3: 当一定要用 C 来作为接口时，应在使用这些接口的代码中使用 C++

理由

C++ 比 C 的表达能力更强，而且为许多种类的编程都提供了更好的支持。

示例

例如，为使用第三方 C 程序库或者 C 系统接口，可以使用 C 和 C++ 的公共子集来定义其底层接口，以获得更好的类型检查。

尽可能将底层接口封装到一个遵循了 C++ 指导方针的接口之中（以获得更好的抽象、内存安全性和资源安全性），并在 C++ 代码中使用这个 C++ 接口。

示例

在 C++ 中可以调用 C：

```
// C 中:
double sqrt(double);

// C++ 中:
extern "C" double sqrt(double);

sqrt(2);
```

示例

在 C 中可以调用 C++：

```
// C 中:  
X call_f(struct Y*, int);  
  
// C++ 中:  
extern "C" X call_f(Y* p, int i)  
{  
    return p->f(i);    // 可能是虚函数调用  
}
```

强制实施

不需要做什么。

SF: 源文件

区分声明（用作接口）和定义（用作实现）。

用头文件来表达接口并强调逻辑结构。

源文件规则概览：

- SF.1: 如果你的项目还未采用别的约定的话，应当为代码文件使用后缀 `.cpp`，而对接口文件使用后缀 `.h`
- SF.2: 头文件不能含有对象定义或非内联的函数定义
- SF.3: 对在多个源文件中使用的任何声明，都应使用头文件
- SF.4: 在文件中的其他所有声明之前包含头文件
- SF.5: `.cpp` 文件必须包含定义了它的接口的一个或多个头文件
- SF.6: `using namespace` 指令，（仅）可以为迁移而使用，可以为基础程序库使用（比如 `std`），或者在局部作用域中使用
- SF.7: 请勿在头文件中的全局作用域使用 `using namespace` 指令
- SF.8: 为所有的头文件使用 `#include` 防卫宏
- SF.9: 避免源文件的循环依赖
- SF.10: 避免依赖于隐含地 `#include` 进来的名字
- SF.11: 头文件应当是自包含的
- SF.12: 对相对于包含文件的文件优先采用引号形式的 `#include`，其他情况下采用角括号形式
- SF.20: 用 `namespace` 表示逻辑结构
- SF.21: 请勿在头文件中使用无名（匿名）命名空间
- SF.22: 为所有的内部/不导出的实体使用无名（匿名）命名空间

SF.1: 如果你的项目还未采用别的约定的话，应当为代码文件使用后缀 `.cpp`，而对接口文件使用后缀 `.h`

参见 [NL.27](#)

SF.2: 头文件不能含有对象定义或非内联的函数定义

理由

对受制于唯一定义规则的实体的包含将导致连接错误。

示例

```
// file.h:  
namespace Foo {  
    int x = 7;  
    int xx() { return x+x; }  
}  
  
// file1.cpp:  
#include <file.h>  
// ... 更多代码 ...  
  
// file2.cpp:  
#include <file.h>  
// ... 更多代码 ...
```

当连接 `file1.cpp` 和 `file2.cpp` 时将出现两个连接器错误。

其他形式: 头文件必须仅包含:

- `#include` 其他的头文件 (可能包括包含防卫宏)
- 模板
- 类定义
- 函数声明
- `extern` 声明
- `inline` 函数定义
- `constexpr` 定义
- `const` 定义
- `using` 别名定义
- ???

强制实施

根据以上白名单来检查。

SF.3: 对在多个源文件中使用的任何声明，都应使用头文件

理由

可维护性。可读性。

示例，不好

```
// bar.cpp:  
void bar() { cout << "bar\n"; }  
  
// foo.cpp:  
extern void bar();  
void foo() { bar(); }
```

`bar` 的维护者在需要改变 `bar` 的类型时，无法找到其全部声明。

`bar` 的使用者不知道他所使用的接口是否完整和正确。顶多会从连接器获得一些（延迟的）错误消息。

强制实施

- 对并未放入 `.h` 而在其他源文件中的实体声明进行标记。

SF.4: 在文件中的其他所有声明之前包含头文件

理由

最小化上下文的依赖并增加可读性。

示例

```
#include <vector>
#include <algorithm>
#include <string>

// ... 我自己的代码 ...
```

示例，不好

```
#include <vector>

// ... 我自己的代码 ...

#include <algorithm>
#include <string>
```

注解

这对于 `.h` 和 `.cpp` 文件都同样适用。

注解

有一种论点是通过在打算保护的代码的后面再 `#include` 头文件，以此将代码同头文件中的声明式和宏等之间进行隔离

(如上面例子中标为“不好”之处)。

不过，

- 这只能对单个文件（在单个层次上）工作：如果采用这个技巧的头文件被别的头文件所包含，这个威胁就会再次出现。
- 命名空间（一个“实现命名空间”）可以针对许多的上下文依赖进行保护。
- 完全的保护和灵活性需要模块。

参见：

- [工作草案，C++ 的模块扩展](#)
- [模块，组件化及其迁移](#)

强制实施

容易。

SF.5: .cpp 文件必须包含定义了它的接口的一个或多个头文件

理由

这使得编译器可以提早进行一致性检查。

示例，不好

```
// foo.h:  
void foo(int);  
int bar(long);  
int foobar(int);  
  
// foo.cpp:  
void foo(int) { /* ... */ }  
int bar(double) { /* ... */ }  
double foobar(int);
```

这个错误直到调用了 `bar` 或 `foobar` 的程序的连接时才会被发现。

示例

```
// foo.h:  
void foo(int);  
int bar(long);  
int foobar(int);  
  
// foo.cpp:  
#include "foo.h"  
  
void foo(int) { /* ... */ }  
int bar(double) { /* ... */ }  
double foobar(int); // 错误：错误的返回类型
```

`foobar` 的返回类型错误在编译 `foo.cpp` 时立即就被发现了。

对 `bar` 的参数类型错误在连接时之前无法被发现，因为可能不会有重载发生，但系统性地使用 `.h` 文件能够增加时其被程序员更早发现的可能性。

强制实施

???

SF.6: using namespace 指令，(仅) 可以为迁移而使用，可以为基础程序库使用（比如 std），或者在局部作用域中使用

理由

`using namespace` 可能造成名字冲突，因而应当节制使用。

然而，将用户代码中的每个命名空间中的名字都进行限定并不总是能够做到（比如在转换过程中）而且有时候命名空间非常基础，并且在代码库中广为使用，坚持进行限定将使其既啰嗦又分散注意力。

示例

```
#include <string>
#include <vector>
#include <iostream>
#include <memory>
#include <algorithm>

using namespace std;

// ...
```

显然地，大量使用了标准库，而且貌似没使用别的程序库，因此要求每一处带有使用 `std::` 会使人分散注意力。

示例

使用 `using namespace std;` 导致程序员可能面临与标准库中的名字造成名字冲突

```
#include <cmath>
using namespace std;

int g(int x)
{
    int sqrt = 7;
    // ...
    return sqrt(x); // 错误
}
```

不过，不大可能导致并非错误的名字解析，
假定使用 `using namespace std` 的人们都了解 `std` 以及这种风险。

注解

`.cpp` 文件也是一种形式的局部作用域。
包含一条 `using namespace x` 的 N 行的 `.cpp` 文件中发生名字冲突的机会，
和包含一条 `using namespace x` 的 N 行的函数，
以及每个都包含一条 `using namespace x` 的总行数为 N 行的 M 个函数，没有多少差别。

注解

[请勿在头文件全局作用域中使用 `using namespace`。](#)

强制实施

对于单个源文件中，对不同命名空间的多个 `using namespace` 指令进行标记。

SF.7: 请勿在头文件中的全局作用域使用 `using namespace`

理由

这样做使 `#include` 一方无法有效地进行区分并使用其他方式。这还可能使所 `#include` 的头文件之间出现顺序依赖，它们以不同次序包含时可能具有不同的意义。

示例

```
// bad.h
#include <iostream>
using namespace std; // bad

// user.cpp
#include "bad.h"

bool copy(/*... some parameters ...*/); // some function that happens to be
named copy

int main()
{
    copy(/*...*/); // now overloads local ::copy and std::copy, could be
ambiguous
}
```

注解

一个例外是 `using namespace std::literals;`。若要在头文件中使用字符串字面量，则必须如此，而且根据[规则](#)——用户必须以 `operator""_x` 来命名他们自己的 UDL——它们并不会与标准库相冲突。

强制实施

标记头文件的全局作用域中的 `using namespace`。

SF.8: 为所有的头文件使用 `#include` 防卫宏

理由

避免文件被多次 `#include`。

为避免包含防卫宏的冲突，不要仅使用文件名来命名防卫宏。

确保还要包含一个关键词和好的区分词，比如头文件所属的程序库或组件的名字。

示例

```
// file foobar.h:
#ifndef LIBRARY_FOOBAR_H
#define LIBRARY_FOOBAR_H
// ... 声明 ...
#endif // LIBRARY_FOOBAR_H
```

强制实施

标记没有 `#include` 防卫的 `.h` 文件。

注解

一些实现提供了如 `#pragma once` 这样的厂商扩展作为包含防卫宏的替代。这并非标准且不可移植。它向程序中注入了宿主机器的文件系统的语义，而且把你锁定到某个特定厂商。

我们的建议是编写 ISO C++：参见[规则 P.2](#)。

SF.9: 避免源文件的循环依赖

理由

循环会使理解变得困难，并拖慢编译速度。
它们还会使（当其可用时）向利用语言支持的模块进行转换工作变得复杂。

注解

要消除循环依赖；请勿仅仅用 `#include` 防卫宏来试图打破它们。

示例，不好

```
// file1.h:  
#include "file2.h"  
  
// file2.h:  
#include "file3.h"  
  
// file3.h:  
#include "file1.h"
```

强制实施

对任何循环依赖进行标记。

SF.10: 避免依赖于隐含地 `#include` 进来的名字

理由

避免意外。
避免当 `#include` 的头文件改变时改变一条 `#include`。
避免意外地变为依赖于所包含的头文件中的实现细节和逻辑上独立的实体。

示例，不好

```
#include <iostream>  
using namespace std;  
  
void use()  
{  
    string s;  
    cin >> s;           // 好  
    getline(cin, s);    // 错误: getline() 未定义  
    if (s == "surprise") { // 错误: == 未定义  
        // ...  
    }  
}
```

`<iostream>` 暴露了 `std::string` 的定义（“为什么？”是一个有趣的问题），
但其并不必然是通过传递包含整个 `<string>` 头文件而做到这一点的，
这带来了常见的新手问题“为什么 `getline(cin, s);` 不成？”，
甚至偶尔出现的“`string` 无法用 `==` 来比较”。

其解决方案是明确地 `#include <string>`：

示例，好

```
#include <iostream>
#include <string>
using namespace std;

void use()
{
    string s;
    cin >> s;           // 好
    getline(cin, s);    // 好
    if (s == "surprise") { // 好
        // ...
    }
}
```

注解

一些头文件正是用于从一些头文件中合并一组声明。

例如：

```
// basic_std_lib.h:

#include <string>
#include <map>
#include <iostream>
#include <random>
#include <vector>
```

用户只用一条 `#include` 就可以获得整组的声明了：

```
#include "basic_std_lib.h"
```

本条反对隐式包含的规则并不防止这种特意的聚集包含。

强制实施

强制实施将需要一些有关头文件中哪些是“导出”给用户所用而哪些是用于实现的知识。
在我们能用到模块之前没有真正的好方案。

SF.11: 头文件应当是自包含的

理由

易用性，头文件应当易于使用，且单独包含即可正常工作。
头文件应当对其所提供的功能进行封装。
避免让头文件的使用方来管理它的依赖项。

示例

```
#include "helpers.h"
// helpers.h 依赖于 std::string 并已包含了 <string>
```

注解

不遵守这条规则将导致头文件的使用方难于诊断所出现的错误。

注解

头文件应当包含其所有依赖项。请小心使用相对路径，各 C++ 实现对于它们的含义是有分歧的。

强制实施

以一项测试来验证头文件自身可通过编译，或者一个仅包含了该头文件的 cpp 文件可通过编译。

SF.12: 对相对于包含文件的文件优先采用引号形式的 `#include`, 其他情况下采用角括号形式

理由

标准 向编译器提供了对于实现

使用角括号 (`<>`) 或引号 (`" "`) 语法的 `#include` 的两种形式的灵活性。

各厂商利用了这点并采用了不同的搜索算法和指定包含路径的方法。

无论如何，指导方针是使用引号形式来（从同一个组件或项目中）包含那些存在于某个相对于含有这条 `#include` 语句的文件的相对路径中的文件，其他情况尽可能使用角括号形式。这样做鼓励明确表现出文件与包含它的文件之间的局部性，或当需要某种不同的搜索算法的情形。这样一眼就可以很容易明白头文件是从某个局部相对文件包含的，还是某个标准库头文件或别的搜索路径（比如另一个程序库或一组常用包含路径）中的某个头文件。

示例

```
// foo.cpp:  
#include <string> // 来自标准程序库，要求使用 <> 形式  
#include <some_library/common.h> // 从另一个程序库中包含的，并非出于局部相对位置的文件；使用 <> 形式  
#include "foo.h" // 处于同一项目中局部相对于 foo.cpp 的文件，使用 "" 形式  
#include "foo_utils/utils.h" // 处于同一项目中局部相对于 foo.cpp 的文件，使用 "" 形式  
#include <component_b/bar.h> // 通过搜索路径定位到的处于同一项目中的文件，使用 <> 形式
```

注解

不遵守这条可能会导致很难诊断的错误：由于包含时指定的错误的范围而选择了错误的文件。

例如，通常 `#include ""` 的搜索算法首先搜索存在于某个局部相对路径中的文件，因此使用这种形式来指代某个并非位于局部相对路径的文件，就一位置一旦在局部相对路径中出现了一个这样的文件（比如进行包含的文件被移动到了别的位置），它就会在原来所包含的文件之前被找到，并使包含文件集合以一种预料之外的方式被改变。

程序库作者们应当把它们的头文件放到一个文件夹中，然后让其客户使用相对路径来包含这些文件：

```
#include <some_library/common.h>.
```

强制实施

检测按 `""` 引用的头文件是否可以按 `<>` 引用。

SF.20: 用 `namespace` 表示逻辑结构

理由

???

示例

```
???
```

强制实施

???

SF.21: 请勿在头文件中使用无名（匿名）命名空间

理由

在头文件中使用无名命名空间差不多都是一个 BUG。

示例

```
// 文件 foo.h:  
namespace  
{  
    const double x = 1.234; // 不好  
  
    double foo(double y) // 不好  
    {  
        return y + x;  
    }  
}  
  
namespace Foo  
{  
    const double x = 1.234; // 好  
  
    inline double foo(double y) // 好  
    {  
        return y + x;  
    }  
}
```

强制实施

- 对头文件中所使用的任何匿名命名空间进行标记。

SF.22: 为所有的内部/不导出的实体使用无名 (匿名) 命名空间

理由

外部实体无法依赖于嵌套的无名命名空间中的实体。

考虑将实现源文件中的所有定义都放入无名命名空间中，除非它定义的是一个“外部/导出”实体。

示例：不好

```
static int f();  
int g();  
static bool h();  
int k();
```

示例：好

```
namespace {  
    int f();  
    bool h();  
}  
int g();  
int k();
```

示例

API 类及其成员不能放在无名命名空间中；而在实现源文件中所定义的任何的“辅助”类或函数则应当放在无名命名空间作用域之中。

???

强制实施

- ???

SL: 标准库

如果只使用纯语言本身的话，任何开发任务都会变得很麻烦（无论以何种语言）。

如果使用了某个合适的程序库的话，则任何开发任务都会变得相当简单。

这些年来标准库一直在持续增长。

现在它在标准中的描述已经比语言功能特性的描述更大了。

因此，可能指导方针的库部分的规模最终将会增长等于甚至超过其他的所有部分。

<< ??? 我们需要另一个层次的规则编号 ??? >>

C++ 标准库组件概览：

- [SL.con: 容器](#)
- [SL.str: 字符串](#)
- [SL.io: I/O 流 \(iostream\)](#)
- [SL.regex: 正则表达式](#)
- [SL.chrono: 时间](#)
- [SL.C: C 标准库](#)

标准库规则概览：

- [SL.1: 尽可能使用程序库](#)
- [SL.2: 优先使用标准库而不是其他程序库](#)
- [SL.3: 请勿向命名空间 std 中添加非标准实体](#)
- [SL.4: 以类型安全的方式使用标准库](#)
- ???

SL.1: 尽可能使用程序库

理由

节约时间。避免重复发明轮子。

避免重复他人的工作。

如果其他人的工作有了改进，则可以从中获得好处。

当你进行了改进之后可以帮助其他人。

SL.2: 优先使用标准库而不是其他程序库

理由

了解标准库的人更多。

相对于你自己的代码或者大多数其他程序库来说，标准库更加倾向于稳定，进行了良好维护，而且广泛可用。

SL.3: 请勿向命名空间 std 中添加非标准实体

理由

向 std 中添加东西可能会改变本来是遵循标准的代码的含义。

添加到 std 的东西可能会与未来版本的标准产生冲突。

示例

???

强制实施

有可能，但很麻烦而且在一些平台上很可能导致一些问题。

SL.4: 以类型安全的方式使用标准库

理由

因为，很显然，违反这条规则将导致未定义的行为，内存损坏，以及其他所有种类的糟糕的错误。

注解

本条规则是半哲学性的元规则，需要许多具体规则予以支持。

我们需要将之作为对于更加专门的规则的总括。

更加专门的规则概览：

- [SL.4: 以类型安全的方式使用标准库](#)

SL.con: 容器

???

容器规则概览：

- [SL.con.1: 优先采用 STL 的 `array` 或 `vector` 而不是 C 数组](#)
- [SL.con.2: 除非有理由使用别的容器，否则默认情况应优先采用 STL 的 `vector`](#)
- [SL.con.3: 避免边界错误](#)
- [SL.con.4: 请勿对非可平凡复制的实参使用 `memset` 或 `memcpy`](#)

SL.con.1: 优先采用 STL 的 `array` 或 `vector` 而不是 C 数组

理由

C 数组不那么安全，而且相对于 `array` 和 `vector` 也没有什么优势。

对于定长数组，应使用 `std::array`，它传递给函数时并不会退变为指针并丢失其大小信息。

而且，和内建数组一样，栈上分配的 `std::array` 会在栈上保存它的各个元素。

对于变长数组，应使用 `std::vector`，它还可以改变大小并处理内存分配。

示例

```
int v[SIZE]; // 不好  
std::array<int, SIZE> w; // ok
```

示例

```
int* v = new int[initial_size]; // 不好，有所有权的原生指针  
delete[] v; // 不好，手工 delete  
  
std::vector<int> w(initial_size); // ok
```

注解

在不拥有而引用容器中的元素时使用 `gsl::span`。

注解

在栈上分配的固定大小的数组和把元素都放在自由存储上的 `vector` 之间比较性能是没什么意义的。

你同样也可以在栈上的 `std::array` 和通过指针访问 `malloc()` 的结果之间进行这样的比较。

对于大多数代码来说，即便是栈上分配和自由存储分配之间的差异也没那么重要，但 `vector` 带来的便利和安全性却是重要的。

如果有人编写的代码中这种差异确实重要，那么他显然可以在 `array` 和 `vector` 之间做出选择。

强制实施

- 如果 C 数组的声明所在的函数或类也声明了 STL 的某个容器（这是为了避免在老式的非 STL 代码中的大量警告噪音），则对其进行标记。修正：最少要把 C 数组改成 `std::array`。

SL.con.2: 除非有理由使用别的容器，否则默认情况应优先采用 STL 的 `vector`

理由

`vector` 和 `array` 是仅有的能够提供以下各项优势的标准容器：

- 最快的通用访问（随机访问，还包括对于向量化友好性）；
- 最快的默认访问模式（从头到尾或从尾到头方式是对预读器友好的）；
- 最少的空间耗费（连续布局中没有每个元素的开销，而且是 cache 友好的）。

通常你都需要对容器进行元素的添加和删除，因此默认应当采用 `vector`；如果并不需要改动容器的大小的话，则应采用 `array`。

即便其他容器貌似更加合适，比如 `map` 的 $O(\log N)$ 查找性能，或者 `list` 的中部高效插入，对于几个 KB 以内大小的容器来说，`vector` 仍然经常性能更好。

注解

`string` 不应当用作独立字符的容器。`string` 是文本字符串；如果需要字符的容器的话，应当采用 `vector<char_type*>` 或者 `array<char_type*>`。

例外

如果你有正当的理由来使用别的容器的话，就请使用它。例如：

- 若 `vector` 满足你的需求，但你并不需要容器大小可变，则应当代之以 `array`。
- 若你需要支持字典式查找的容器并保证 $O(K)$ 或 $O(\log N)$ 的查找效率，而且容器将会比较大（超过几个 KB），你需要经常进行插入使得维护有序的 `vector` 的开销不大可行，则请代之以使用 `unordered_map` 或者 `map`。

注解

使用 `()` 初始化来将 `vector` 初始化为具有特定数量的元素。

使用 `{}` 初始化来以一个元素列表来对 `vector` 进行初始化。

```
vector<int> v1(20); // v1 具有 20 个值为 0 的元素 (vector<int>{})  
vector<int> v2 {20}; // v2 具有 1 个值为 20 的元素
```

[优先采用 {} 初始化式语法。](#)

强制实施

- 如果 `vector` 构造之后大小不会改变（比如因为它是 `const` 或者因为没有对它调用过非 `const` 函数），则对其进行标记。修正：代之以使用 `array`。

SL.con.3: 避免边界错误

理由

越过已分配的元素的范围进行读写，通常都会导致糟糕的错误，不正确的结果，程序崩溃，以及安全漏洞。

注解

应用于一组元素的范围的标准库函数，都有（或应当有）接受 `span` 的边界安全重载。

如 `vector` 这样的标准类型，在边界剖面配置下（以某种不兼容的方式，如添加契约）可以被修改为实施边界检查，或者使用 `at()`。

理想情况下，边界内保证应当可以被静态强制实行。

例如：

- 基于范围的 `for` 的循环不会越过其所针对的容器的范围
- `v.begin()`, `v.end()` 可以很容易确定边界安全性

这种循环和任何的等价的无检查或不安全的循环一样高效。

通常，可以用一个简单的预先检查来消除检查每个索引的需要。

例如

- 对于 `v.begin()`, `v.begin() + i`, `i` 可以很容易针对 `v.size()` 检查

这种循环比每次都待检查元素访问要快得多。

示例，不好

```
void f()
{
    array<int, 10> a, b;
    memset(a.data(), 0, 10);           // 不好，且包含长度错误 (length = 10 *
    sizeof(int))
    memcmp(a.data(), b.data(), 10);   // 不好，且包含长度错误 (length = 10 *
    sizeof(int))
}
```

而且，`std::array<>::fill()` 或 `std::fill()`，甚或是空的初始化式，都是比 `memset()` 更好的候选。

示例，好

```
void f()
{
    array<int, 10> a, b, c{};        // c 被初始化为零
    a.fill(0);
    fill(b.begin(), b.end(), 0);     // std::fill()
    fill(b, 0);                     // std::ranges::fill()

    if (a == b) {
        // ...
    }
}
```

Example

如果代码使用的是未修改的标准库，仍然有一些变通方案来以边界安全的方式使用 `std::array` 和 `std::vector`。代码中可以调用各个类的 `.at()` 成员函数，这将抛出 `std::out_of_range` 异常。或者，代码中可以调用 `at()` 自由函数，这将在边界违例时导致快速失败（或者某个自定义动作）。

```

void f(std::vector<int>& v, std::array<int, 12> a, int i)
{
    v[0] = a[0];           // 不好
    v.at(0) = a[0];        // OK (替代方案 1)
    at(v, 0) = a[0];      // OK (替代方案 2)

    v.at(0) = a[i];        // 不好
    v.at(0) = a.at(i);    // OK (替代方案 1)
    v.at(0) = at(a, i);   // OK (替代方案 2)
}

```

强制实施

- 对于没有边界检查的标准库函数的任何调用都给出诊断消息。
??? 在这里添加一组禁用函数的连接列表

本条规则属于[边界剖面配置](#)。

SL.con.4: 请勿对非可平凡复制的实参使用 `memset` 或 `memcpy`

理由

这样做会破坏对象语义（例如，其会覆写掉 `vptr`）。

注解

`(w)memset`, `(w)memcpy`, `(w)memmove`, 以及 `(w)memcmp` 与此相似。

示例

```

struct base {
    virtual void update() = 0;
};

struct derived : public base {
    void update() override {}
};

```

```

void f (derived& a, derived& b) // 虚表再见!
{
    memset(&a, 0, sizeof(derived));
    memcpy(&a, &b, sizeof(derived));
    memcmp(&a, &b, sizeof(derived));
}

```

应当代之以定义适当的默认初始化、复制，以及比较函数

```

void g(derived& a, derived& b)
{
    a = {};// 默认初始化
    b = a; // 复制
    if (a == b) do_something(a,b);
}

```

强制实施

- 对在不可平凡复制的类型使用这些函数进行标记

TODO 注释:

- 对于标准库的影响需要和 WG21 之间进行紧密的协调，即便不需要标准化也应当至少保证兼容性。
- 我们正在考虑为标准库（尤其是 C 标准库）中如 `memcmp` 这样的函数指定边界安全的重载，并在 GSL 中提供它们。
- 对于标准中没有进行完全的边界检查的现存函数和如 `vector` 这样的类型来说，我们的目标是在启用了边界剖面配置的代码中调用时，这些功能应当进行边界检查，而从遗留代码中调用时则没有检查，可能需要利用契约来实现（正由几个 WG21 成员进行提案工作）。

SL.str: 字符串

文本处理是一个大的主题。

`std::string` 无法全部覆盖这些。

这一部分主要尝试澄清 `std::string` 和 `char*`、`zstring`、`string_view` 和 `gsl::span<char>` 之间的关系。

有关非 ASCII 字符集和编码的重要问题（比如 `wchar_t`，Unicode，以及 UTF-8 等）将在别处讨论。

参见：[正则表达式](#)

在这里，我们用“字符序列”或“字符串”来代表（终将）作为文本来读取的字符序列。

We don't consider ???

字符串概览：

- [SL.str.1: 使用 `std::string` 以拥有字符序列](#)
- [SL.str.2: 使用 `std::string_view` 或 `gsl::span<char>` 以指代字符序列](#)
- [SL.str.3: 使用 `zstring` 或 `czstring` 以指代 C 风格、以零结尾的字符序列](#)
- [SL.str.4: 使用 `char*` 以指代单个字符](#)
- [SL.str.5: 使用 `std::byte` 以指代并不必须表示字符的字节值](#)
- [SL.str.10: 当需要实施相关于文化地域的操作时，使用 `std::string`](#)
- [SL.str.11: 当需要改动字符串时，使用 `gsl::span<char>` 而不是 `std::string_view`](#)
- [SL.str.12: 为作为标准库的 `string` 类型的字符串字面量使用后缀 `s`](#)

参见：

- [F.24 span](#)
- [F.25 zstring](#)

SL.str.1: 使用 `std::string` 以拥有字符序列

理由

`string` 能够正确处理资源分配，所有权，复制，渐进扩容，并提供许多有用的操作。

示例

```
vector<string> read_until(const string& terminator)
{
    vector<string> res;
    for (string s; cin >> s && s != terminator; ) // 读取一个单词
        res.push_back(s);
    return res;
}
```

注意已经为 `string` 提供了 `>>` 和 `!=` (作为有用操作的例子) , 并且没有显示的内存分配, 回收, 或者范围检查 (`string` 会处理这些) 。

C++17 中, 我们可以使用 `string_view` 而不是 `const string&` 作为参数, 以允许调用方更大的灵活性:

```
vector<string> read_until(string_view terminator) // C++17
{
    vector<string> res;
    for (string s; cin >> s && s != terminator; ) // 读取一个单词
        res.push_back(s);
    return res;
}
```

示例, 不好

不要使用 C 风格的字符串来进行需要不单纯的内存管理的操作:

```
char* cat(const char* s1, const char* s2) // 当心!
    // return s1 + '.' + s2
{
    int l1 = strlen(s1);
    int l2 = strlen(s2);
    char* p = (char*)malloc(l1 + l2 + 2);
    strcpy(p, s1, l1);
    p[l1] = '.';
    strcpy(p + l1 + 1, s2, l2);
    p[l1 + l2 + 1] = 0;
    return p;
}
```

我们搞对了吗?

调用者能记得要对返回的指针调用 `free()` 吗?

这段代码能通过安全性评审吗?

注解

没有测量就不要假设 `string` 比底层技术慢, 要记得并非所有代码都是性能攸关的。

[请勿进行不成熟的优化](#)

强制实施

???

SL.str.2: 使用 `std::string_view` 或 `gsl::span<char>` 以指代字符序列

理由

`std::string_view` 或 `gsl::span<char>` 提供了简易且（潜在）安全的对字符序列的访问，并与序列的分配和存储方式无关。

示例

```
vector<string> read_until(string_view terminator);

void user(zstring p, const string& s, string_view ss)
{
    auto v1 = read_until(p);
    auto v2 = read_until(s);
    auto v3 = read_until(ss);
    // ...
}
```

注解

`std::string_view` (C++17) 是只读的。

强制实施

???

SL.str.3: 使用 `zstring` 或 `czstring` 以指代 C 风格、以零结尾的字符序列

理由

可读性。

明确意图。

普通的 `char*` 可以是指向单个字符的指针，指向字符数组的指针，指向 C 风格（零结尾）字符串的指针，甚或是指向小整数的指针。

对这些情况加以区分能够避免误解和 BUG。

示例

```
void f1(const char* s); // s 可能是个字符串
```

我们所知的只不过是它可能是 `nullptr` 或者指向至少一个字符

```
void f1(zstring s);      // s 是 C 风格字符串或者 nullptr
void f1(czstring s);    // s 是 C 风格字符串常量或者 nullptr
void f1(std::byte* s); // s 是某个字节的指针 (C++17)
```

注解

除非确实有理由，否则不要把 C 风格的字符串转换为 `string`。

注解

与其他的“普通指针”一样，`zstring` 不能表达所有权。

注解

已经存在了上亿行的 C++ 代码，它们大多使用 `char*` 和 `const char*` 却并不注明其意图。各种不同的方式都在使用它们，包括以之表示所有权，以及（代替 `void*`）作为通用的内存指针。很难区分这些用法，因此这条指导方针很难被遵守。而这是 C 和 C++ 程序中的最主要的 BUG 来源之一，因此一旦可行就遵守这条指导方针是值得的。

强制实施

- 标记在 `char*` 上使用的 `[]`
- 标记在 `char*` 上使用的 `delete[]`
- 标记在 `char*` 上使用的 `free()`

SL.str.4: 使用 `char*` 以指代单个字符

示例

现存代码中对 `char*` 的各种不同用法，是一种主要的错误来源。

示例，不好

```
char arr[] = {'a', 'b', 'c'};

void print(const char* p)
{
    cout << p << '\n';
}

void use()
{
    print(arr);    // 运行时错误；可能非常糟糕
}
```

数组 `arr` 并非 C 风格字符串，因为它不是零结尾的。

替代方案

参见 `zstring`, `string`, 以及 `string_view`。

强制实施

- 标记在 `char*` 上使用的 `[]`

SL.str.5: 使用 `std::byte` 以指代并不必须表示字符的字节值

理由

用 `char*` 来表示指向不一定是字符的东西的指针会造成混乱，并会妨碍有价值的优化。

示例

```
???
```

注解

C++17

强制实施

???

SL.str.10: 当需要实施相关于文化地域的操作时，使用 `std::string`

理由

`std::string` 支持标准库的 [locale 功能](#)

示例

```
???
```

注解

???

强制实施

???

SL.str.11: 当需要改动字符串时，使用 `gsl::span<char>` 而不是 `std::string_view`

理由

`std::string_view` 是只读的。

示例

???

注解

???

强制实施

编译器会标记出试图写入 `string_view` 的地方。

SL.str.12: 为作为标准库的 `string` 类型的字符串字面量使用后缀 `s`

理由

直接表达想法能够最小化犯错机会。

示例

```
auto pp1 = make_pair("Tokyo", 9.00);           // {C 风格字符串,double} 有意如此?  
pair<string, double> pp2 = {"Tokyo", 9.00};    // 稍微啰嗦  
auto pp3 = make_pair("Tokyo"s, 9.00);          // {std::string,double}      // C++14  
pair pp4 = {"Tokyo"s, 9.00};                    // {std::string,double}      // C++17
```

强制实施

???

SL.io: I/O 流 (iostream)

`iostream` 是一种类型安全的，可扩展的，带格式的和无格式的流式 I/O 的 I/O 程序库。

它支持多种（且用户可扩展的）缓冲策略以及多种文化地域。

它可以用于进行便利的 I/O，内存读写（字符串流），
以及用户定义的扩展，诸如跨网络的流（asio：尚未标准化）。

I/O 流规则概览：

- [SL.io.1: 仅在必要时才使用字符层面的输入](#)
- [SL.io.2: 当进行读取时，总要考虑非法输入](#)
- [SL.io.3: 优先使用 iostream 进行 I/O](#)
- [SL.io.10: 除非你使用了 printf 族函数，否则要调用 `ios_base::sync_with_stdio\(false\)`](#)
- [SL.io.50: 避免使用 `endl`](#)
- [???](#)

SL.io.1: 仅在必要时才使用字符层面的输入

理由

除非你确实仅处理单个的字符，否则使用字符级的输入将导致用户代码实施潜在易错的
且潜在低效的从字符进行标记组合的工作。

示例

```
char c;  
char buf[128];  
int i = 0;  
while (cin.get(c) && !isspace(c) && i < 128)  
    buf[i++] = c;  
if (i == 128) {  
    // ... 处理过长的字符串 ....  
}
```

更好的做法（简单得多而且可能更快）：

```
string s;
s.reserve(128);
cin>>s;
```

而且额能并不需要 `reserve(128)`。

强制实施

???

SL.io.2: 当进行读取时，总要考虑非法输入

理由

错误通常最好尽快处理。

如果输入无效，所有的函数都必须编写为对付不良的数据（而这并不现实）。

示例

```
???
```

强制实施

???

SL.io.3: 优先使用 `iostream` 进行 I/O

理由

`iosteam` 安全，灵活，并且可扩展。

示例

```
// 写出一个复数:
complex<double> z{ 3,4 };
cout << z << '\n';
```

`complex` 是一个用户定义的类型，而其 I/O 的定义无需改动 `iostream` 库。

示例

```
// 读取一系列复数:
for (complex<double> z; cin>>z)
    v.push_back(z);
```

例外

??? 性能 ???

讨论: `iostream` vs. `printf()` 家族

人们通常说（并且通常是正确的）`printf` 家族比 `iostream` 有两个优势：

格式化的灵活性和性能。

这需要与 `iostream` 在处理用户定义类型方面的扩展性，针对安全性的违反方面的韧性，
隐含的内存管理，以及 `locale` 处理等优势之间进行权衡。

如果需要 I/O 性能的话，你几乎总能做到比 `printf()` 更好。

`gets()`，使用 `%s` 的 `scanf()`，和使用 `%s` 的 `printf()` 在安全性方面冒风险（容易遭受缓冲区溢出问题而且通常很易错）。

C11 定义了一些“可选扩展”，它们对其实参进行一些额外检查。

如果您的 C 程序库中包含 `gets_s()`、`scanf_s()` 和 `printf_s()`，它们也许是更安全的替代方案，但仍然并非是类型安全的。

强制实施

可选地标记 `<cstdio>` 和 `<stdio.h>`。

SL.io.10: 除非你使用了 `printf` 族函数，否则要调用 `ios_base::sync_with_stdio(false)`

理由

`iostreams` 和 `printf` 风格的 I/O 之间的同步是由代价的。

`cin` 和 `cout` 默认是与 `printf` 相同步的。

示例

```
int main()
{
    ios_base::sync_with_stdio(false);
    // ... 使用 iostreams ...
}
```

强制实施

???

SL.io.50: 避免使用 `endl`

理由

`endl` 操纵符大致相当于 `'\n'` 和 `"\n"`；

其最常用的情况只不过会以添加多余的 `flush()` 的方式拖慢程序。

与 `printf` 式输出相比，这种拖慢程度是比较显著的。

示例

```
cout << "Hello, world!" << endl;      // 两次输出操作和一次 flush
cout << "hello, world!\n";                // 一次输出操作且没有 flush
```

注解

对于 `cin/cout`（或同等设备）的交互来说，没什么原因必须进行冲洗；它们是自动进行的。

对于向文件写入来说，也很少需要 `flush`。

注解

对于字符串流（指 `ostringstream`），插入一个 `endl` 完全等价于插入一个 '`\n`' 字符，但正是这种情况下，`endl` 可能会明显比较慢。

`endl` 并不关注产生平台专有的行结尾序列（比如 Windows 上的 "`\r\n`"）。因此，字符串流的 `s << endl` 只会插入单个 '`\n`' 字符。

注解

除了（偶尔会比较重要的）性能问题外，从 "`\n`" 和 `endl` 之间进行选择基本上完全是审美问题。

SL.regex: 正则表达式

`<regex>` 是标准 C++ 的正则表达式库。它支持许多正则表达式的模式约定。

SL.chrono: 时间

`<chrono>`（在命名空间 `std::chrono` 中定义）提供了 `time_point` 和 `duration`，并同时提供了用于以各种不同单位输出时间的函数。它还提供了用于注册 `time_point` 的时钟。

SL.C: C 标准库

???

C 标准库规则概览：

- [SL.C.1: 请勿使用 setjmp/longjmp](#)
- [???](#)
- [???](#)

SL.C.1: 请勿使用 setjmp/longjmp

理由

`longjmp` 会忽略析构函数，由此使得依赖于 RAII 的所有资源管理策略全部失效。

强制实施

标记出现的所有 `longjmp` 和 `setjmp`

A: 架构设计的观念

本部分包括有关高层次的架构性观念和程序库的观念。

架构性规则概览：

- [A.1: 分离稳定的代码和不稳定的代码](#)
- [A.2: 将潜在可复用的部分作为程序库](#)
- [A.4: 程序库之间不能有循环依赖](#)
- [???](#)
- [???](#)

- [???](#)
- [???](#)
- [???](#)
- [???](#)

A.1: 分离稳定的代码和不稳定的代码

对较不稳定的代码进行隔离，有助于其单元测试，接口改进，重构，以及最终弃用。

A.2: 将潜在可复用的部分作为程序库

理由

注解

程序库是一些共同进行维护，文档化，并发布的声明式和定义式的集合体。

程序库可以是一组头文件（“仅有头文件的程序库”），或者一组头文件加上一组目标文件构成。

你可以静态或动态地将程序库连接到程序中，或者你还可以 `#included` 仅头文件的库。

A.4: 程序库之间不能有循环依赖

理由

- 循环依赖导致构建过程变得复杂。
- 循环依赖难于理解，可能会引入不确定性（未定义行为）。

注解

一个程序库可以在它的组件的定义之间包含循环引用。

例如：

???

不过，程序库不能对依赖于它的其他程序库产生依赖。

NR: 伪规则和错误的看法

本部分包含一些在不少地方流行的规则和指导方针，但是我们慎重地建议不要采纳它们。

我们完全了解这些规则曾经在某些时间和场合是有意义的，而且我们自己也曾经采用过它们。

不过，在我们所推荐并以各项指导方针所支持的编程风格的情况下，这些“伪规则”是有害的。

即便是今天，仍有一些情况下这些规则是有意义的。

比如说，缺少合适的工具支持会导致异常在硬实时系统中的不适用，

但请不要盲目地信任“通俗智慧”（比如有关“效率”的未经数据支持的观点）；

这种“智慧”也许是基于几十年前的信息，或者是来自于与 C++ 有非常不同性质的语言的经验（比如 C 或者 Java）。

对于这些伪规则的替代方案的正面观点都在各个规则的“替代方案”部分中给出。

伪规则概览：

- [NR.1: 请勿坚持认为声明都应当放在函数的最上面](#)
- [NR.2: 请勿坚持使函数中只保留一个 `return` 语句](#)
- [NR.3: 请勿避免使用异常](#)
- [NR.4: 请勿坚持把每个类定义放在其自己的源文件中](#)

- [NR.5: 请勿采用两阶段初始化](#)
- [NR.6: 请勿把所有清理操作放在函数末尾并使用 `goto exit`](#)
- [NR.7: 请勿使所有数据成员 `protected`](#)
- ???

NR.1: 请勿坚持认为声明都应当放在函数的最上面

理由

“所有声明都在开头”的规则，是来自不允许在语句之后对变量和常量进行初始化的老编程语言的遗产。这样做会导致更长的程序，以及更多由于未初始化的或者错误初始化的变量所导致的错误。

示例，不好

```
int use(int x)
{
    int i;
    char c;
    double d;

    // ... 做一些事 ...

    if (x < i) {
        // ...
        i = f(x, d);
    }
    if (i < x) {
        // ...
        i = g(x, c);
    }
    return i;
}
```

未初始化变量和其使用点的距离越长，出现 BUG 的机会就越大。

幸运的是，编译器可以发现许多“设值前使用”的错误。

不幸的是，编译器无法捕捉到所有这样的错误，而且一些 BUG 并不都像这个小例子中的这样容易发现。

替代方案

- [坚持为对象进行初始化。](#)
- [ES.21: 不要在确实需要使用变量（或常量）之前就引入它。](#)

NR.2: 请勿坚持使函数中只保留一个 `return` 语句

理由

单返回规则会导致不必要的复杂的代码，并引入多余的状态变量。

特别是，单返回规则导致更难在函数开头集中进行错误检查。

示例

```
template<class T>
// requires Number<T>
string sign(T x)
{
    if (x < 0)
        return "negative";
    if (x > 0)
        return "positive";
    return "zero";
}
```

为仅使用一个返回语句，我们得做类似这样的事：

```
template<class T>
// requires Number<T>
string sign(T x)      // 不好
{
    string res;
    if (x < 0)
        res = "negative";
    else if (x > 0)
        res = "positive";
    else
        res = "zero";
    return res;
}
```

这不仅更长，而且很可能效率更差。

越长越复杂的函数，对其进行变通就越是痛苦。

当然许多简单的函数因为它们本来就简单的逻辑都天然就只有一个 `return`。

示例

```
int index(const char* p)
{
    if (!p) return -1; // 错误指标：替代方案是 "throw nullptr_error{}"
    // ... 进行查找以找出 p 的索引
    return i;
}
```

如果我们采纳这条规则的话，得做类似这样的事：

```

int index2(const char* p)
{
    int i;
    if (!p)
        i = -1; // 错误指标
    else {
        // ... 进行查找以找出 p 的索引
    }
    return i;
}

```

注意我们（故意地）违反了禁止未初始化变量的规则，因为这种风格通常都会导致这样。而且，这种风格也会倾向于采用 [goto exit](#) 伪规则。

替代方案

- 保持函数短小简单。
- 随意使用多个 `return` 语句（以及抛出异常）。

NR.3: 请勿避免使用异常

理由

一般有四种主要的不用异常的理由：

- 异常是低效的
- 异常会导致泄漏和错误
- 异常的性能无法预测
- 异常处理的运行时支持耗费过多空间

我们没有能够满足所有人的解决这个问题的办法。

无论如何，针对异常的讨论已经持续了四十多年了。

一些语言没有异常就无法使用，而另一些并不支持异常。

这在使用和不使用异常的方面都造成了强大的传统，并导致激烈的争论。

不过，我们可以简要说明，为什么我们认为对于通用编程以及这里的指导方针的情况来说，
异常是最佳的候选方案。

简单的论据，无论支持还是反对，都是缺乏说服力的。

确实存在一些特殊的应用，其中异常就是不合适的。

（例如，硬实时系统，且缺乏可靠的对于异常处理的耗费进行估计的支持）。

我们依次来考虑针对异常的主要反对观点：

- 异常是低效的：

和什么相比？

当进行比较时，请确保处理了同样的错误集合，并且它们都进行了等价的处理。

尤其是，不要对一个见到异常就立刻终止的程序和一个在记录错误日志之前
小心地进行资源清理的程序之间进行比较。

确实，某些系统的异常处理实现很糟糕；有时候，这样的实现迫使我们使用
其他错误处理方案，但这并不是异常的基本问题。

当使用某个有效的论据时——无论什么样的上下文——请小心你能拿出确实提供了所讨论的问题的
内部情况的健全的数据。

- 异常会导致泄漏和错误。

不会。

如果你的程序时一大堆乱糟糟的指针而没有总体的资源管理策略，

那么无论你干什么都会有问题。

如果你的系统是由上百万行这样的代码构成的，

那你可能是无法使用异常的，

不过这是一个有关过度和放纵的使用指针的问题，而不是异常的问题。

我们的观点是，你需要用 RAI 来让基于异常的错误处理变得简单且安全——比其他方案都要更简单和安全。

- 异常的性能无法预测。

如果你是在硬实时系统上，而你必须确保一个任务要在给定的时间内完成，

你需要一些工具来支撑这样的保证。

就我们所知，还没有出现这样的工具（至少对大多数程序员没有）。

- 异常处理的运行时支持耗费过多空间

小型（通常为嵌入式）系统中可能如此。

不过在放弃异常之前，请考虑采用统一的利用错误码的错误处理将耗费的空间有多少，

以及错误未被捕获将造成的损失由多少。

许多（可能是大多数）的和异常有关的问题都源自于需要和杂乱的老代码进行交互的历史性原因。

而支持使用异常的基本论点是：

- 它们把错误返回和普通返回进行了清晰的区分
- 它们无法被忘记或忽略
- 它们可以系统化地使用

请记住

- 异常是用于报告错误的（C++ 中；其他语言可能有异常的不同用法）。
- 异常不是用于可以局部处理的错误的。
- 不要试图在每个函数中捕获每一种异常（这样做是冗长的，臃肿的，而且会导致代码缓慢）。
- 异常不是用于那些当发生无法恢复的错误之后需要立即终止模块或系统的错误的。

示例

???

替代方案

- [RAII](#)
- 契约/断言：使用 GSL 的 `Requires` 和 `Ensures`（直到对契约的语言支持可以使用）

NR.4: 请勿坚持把每个类定义放在其自己的源文件中

理由

将每个类都放进其自己的文件所导致的文件数量难于管理，并会拖慢编译过程。

单个的类很少是一种良好的维护和发布的逻辑单位。

示例

???

替代方案

- 使用命名空间来包含逻辑上聚合的类和函数。

NR.5: 请勿采用两阶段初始化

理由

将初始化拆分为两步会导致不变式的弱化，
更复杂的代码（必须处理半构造对象），
以及错误（当未能一致地正确处理半构造对象时）。

示例，不好

```
// 老式传统风格：有许多问题

class Picture
{
    int mx;
    int my;
    int * data;
public:
    // 主要问题：构造函数未进行完全构造
    Picture(int x, int y)
    {
        mx = x;           // 也不好：在构造函数体中而非
                           // 成员初始化式中进行赋值
        my = y;
        data = nullptr; // 也不好：在构造函数中而非
                           // 成员初始化式中进行常量初始化
    }

    ~Picture()
    {
        cleanup();
    }

    // ...

// 不好：两阶段初始化
bool init()
{
    // 不变式检查
    if (mx <= 0 || my <= 0) {
        return false;
    }
    if (data) {
        return false;
    }
    data = (int*) malloc(mx*my*sizeof(int)); // 也不好：拥有原始指针，还用了
    malloc
    return data != nullptr;
}

// 也不好：没有理由让清理操作作为单独的函数
void cleanup()
```

```

    {
        if (data) free(data);
        data = nullptr;
    }
};

Picture picture(100, 0); // 此时 picture 尚未就绪可用
// 这里将失败
if (!picture.Init()) {
    puts("Error, invalid picture");
}
// 现在有一个无效的 picture 对象实例。

```

示例，好

```

class Picture
{
    int mx;
    int my;
    vector<int> data;

    static int check_size(int size)
    {
        // 不变式检查
        Requires(size > 0);
        return size;
    }

public:
    // 更好的方式是以一个 2D 的 Size 类作为单个形参
    Picture(int x, int y)
        : mx(check_size(x))
        , my(check_size(y))
    {
        // 现在已知 x 和 y 为有效的大小
        , data(mx * my) // 出错时将抛出 std::bad_alloc
    }
    // 图片就绪可用
}

// 编译器生成的析构函数会完成工作。 (另见 C.21)

// ...
};

Picture picture1(100, 100);
// picture1 已就绪可用......

// y 并非有效大小值,
// 缺省的契约违规行为将会调用 std::terminate
Picture picture2(100, 0);
// 不会抵达这里.....

```

替代方案

- 始终在构造函数中建立类不变式。
- 不要在需要对象之前就定义它。

NR.6: 请勿把所有清理操作放在函数末尾并使用 `goto exit`

理由

`goto` 是易错的。

这种技巧是进行 RAI^I 式的资源和错误处理的前异常时代的技巧。

示例，不好

```
void do_something(int n)
{
    if (n < 100) goto exit;
    // ...
    int* p = (int*) malloc(n);
    // ...
    if (some_error) goto_exit;
    // ...
exit:
    free(p);
}
```

请找出其中的 BUG。

替代方案

- 使用异常和 [RAII](#)
- 对于非 RAII 资源，使用 [finally](#)。

NR.7: 请勿使所有数据成员 `protected`

理由

`protected` 数据是一种错误来源。

`protected` 数据可以被各种地方的无界限数量的代码所操纵。

`protected` 数据是在类层次中等价于全局对象的东西。

示例

```
???
```

替代方案

- [使成员数据 `public` 或者（更好地）`private`。](#)

RF: 参考材料

已经为 C++，尤其是对 C++ 的使用编写过了许多的编码标准、规则和指导方针。
它们中许多都

- 关注的是低级问题，比如标识符的拼写

- 是由 C++ 的新手编写的
- 将“禁止程序员作出不常见行为”作为其首要目标
- 将维持许多编译器的可移植性作为目标（有些已经是 10 年前的了）
- 是为了维持好几十年的代码库而编写的
- 是仅关注单一的应用领域的
- 只会产生反效果
- 被忽略了（程序员为了完成工作不得不忽略它们）

不良的编码标准要比没有编码标准还要差。

不过一组恰当的指导方针比没有标准要好得多：“形式即解放。”

我们为什么不能有一种允许所有我们想要的同时又禁止所有我们不期望的东西的语言（“完美的语言”）呢？

本质上说，这是由于可负担的语言（及其工具链）同时也要为那些需求与你不同的人提供服务，并且要为你今后比今天更多的需求提供服务。

而且，你的需求会随时间而改变，而为此你则需要采用一种通用语言。

今日貌似理想的语言在未来可能会变得过于受限了。

编码指导方针可以使语言能够适应于特定的需求。

因此，并不存在适用于每个人的单一编码风格。

我们预计不同的组织会提供更具限制性和更严格的编码风格附加规定。

参考材料部分：

- [RF.rules: 编码规则](#)
- [RF.books: 带有编码指导方针的书籍](#)
- [RF.C++: C++ 编程 \(C++11/C++14/C++17\)](#)
- [RF.web: 网站](#)
- [RS.video: 有关“当代 C++”的视频](#)
- [RF.man: 手册](#)
- [RF.core: 核心指导方针相关材料](#)

RF.rules: 编码规则

- [AUTOSAR Guidelines for the use of the C++14 language in critical and safety-related systems v17.10](#)
- [Boost Library Requirements and Guidelines.](#)
???.
- [Bloomberg: BDE C++ Coding.](#)
着重强调了代码的组织和布局。
- Facebook: ???
- [GCC Coding Conventions.](#)
C++03 以及（相当）一部分向后兼容。
- [Google C++ Style Guide.](#)
面向 C++17 和（同样）较老的代码库。Google 的专家们现在正展开活跃的合作，以改进这里的各项指导方针，有希望能够合并这些成果，以使它们能够成为他们也同样推荐采纳的一组现代的通用指导方针。
- [JSF++: JOINT STRIKE FIGHTER AIR VEHICLE C++ CODING STANDARDS.](#)
文档编号 2RDU00001 Rev C. December 2005.
针对飞行控制软件。
针对硬实时。
这意味着它需要非常多的限制（“程序如果发生故障就会有人挂掉”）。
例如，飞机起飞后禁止进行任何自由存储的分配和回收（禁止内存溢出并禁止发生碎片化）。

禁止使用异常（因为没有可用工具可以保证异常能够在固定的短时间段内被处理）。

所使用的程序库必须是已被证明可以用于关键任务应用的。

它和这个指导方针集合的相似性并不让人惊讶，因为 Bjarne Stroustrup 正是 JSF++ 的作者之一。

建议采纳，但请注意其非常特定的关注领域。

- [MISRA C++ 2008: Guidelines for the use of the C++ language in critical systems] (<https://www.misra.org.uk/Buyonline/tbid/58/Default.aspx>)。

- [Using C++ in Mozilla Code](#).

如其名称所示，它关注于跨许多（老）编译器的兼容性。

因此，它是很具有限制性的。

- [Geosoft.no: C++ Programming Style Guidelines](#).

???.

- [Possibility.com: C++ Coding Standard](#).

???.

- [SEI CERT: Secure C++ Coding Standard](#).

针对安全关键代码所编写的一组非常好的规则（还带有示例和原理说明）。

它们的许多规则都广泛适用。

- [High Integrity C++ Coding Standard](#).

- [Ilfm](#).

有些简略，基于 C++14，而且是（有理由地）针对其应用领域的。

- ???

RF.books: 带有编码指导方针的书籍

- [Meyers96](#) Scott Meyers: *More Effective C++*. Addison-Wesley 1996.

- [Meyers97](#) Scott Meyers: *Effective C++, Second Edition*. Addison-Wesley 1997.

- [Meyers01](#) Scott Meyers: *Effective STL*. Addison-Wesley 2001.

- [Meyers05](#) Scott Meyers: *Effective C++, Third Edition*. Addison-Wesley 2005.

- [Meyers15](#) Scott Meyers: *Effective Modern C++*. O'Reilly 2015.

- [SuttAlex05](#) Sutter and Alexandrescu: *C++ Coding Standards*. Addison-Wesley 2005. 与其说是一组规则，不如说是一组元规则。前 C++11 时代。

- [Stroustrup05](#) Bjarne Stroustrup: [A rationale for semantically enhanced library languages](#). LCSD05. October 2005.

- [Stroustrup14](#) Stroustrup: [A Tour of C++](#).

Addison Wesley 2014.

每章的结尾都有一个包含一组建议的忠告部分。

- [Stroustrup13](#) Stroustrup: [The C++ Programming Language \(4th Edition\)](#).

Addison Wesley 2013.

每章的结尾都有一个包含一组建议的忠告部分。

- Stroustrup: [Style Guide](#)

for [Programming: Principles and Practice using C++](#).

大多是一些低级的命名和代码布局规则。

主要作为教学工具。

RF.C++: C++ 编程 (C++11/C++14)

- [TC++PL4](#):

面向有经验的程序员的，对 C++ 语言和标准库的全面彻底的描述。

- [Tour++](#):

面向有经验的程序员的，对 C++ 语言和标准库的简介。

- [Programming: Principles and Practice using C++](#):

面向初学者和新手们的教材。

RF.web: 网站

- [isocpp.org](#)
- [Bjarne Stroustrup 的个人主页](#)
- [WG21](#)
- [Boost](#)
- [Adobe open source](#)
- [Poco libraries](#)
- Sutter's Mill?
- ???

RS.video: 有关“当代 C++”的视频

- Bjarne Stroustrup: [C++11?Style](#). 2012.
- Bjarne Stroustrup: [The Essence of C++: With Examples in C++84, C++98, C++11, and?C++14](#). 2013
- [CppCon '14](#) 的全部演讲
- Bjarne Stroustrup: [The essence of C++ 在爱丁堡大学](#). 2014
- Bjarne Stroustrup: [The Evolution of C++ Past, Present and Future](#). CppCon 2016 keynote.
- Bjarne Stroustrup: [Make Simple Tasks Simple!](#). CppCon 2014 keynote.
- Bjarne Stroustrup: [Writing Good C++14](#). CppCon 2015 keynote about the Core Guidelines.
- Herb Sutter: [Writing Good C++14... By Default](#). CppCon 2015 keynote about the Core Guidelines.
- CppCon 15
- ??? C++ Next
- ??? Meting C++
- ??? more ???

RF.man: 手册

- ISO C++ Standard C++11.
- ISO C++ Standard C++14.
- [ISO C++ Standard C++17](#). 委员会草案。
- [Palo Alto "Concepts" TR](#).
- [ISO C++ Concepts TS](#).
- [WG21 Ranges report](#). 草案。

RF.core: 核心指导方针相关材料

这个部分包含一些用于展示核心指导方针及其背后的思想的有用材料：

- [Our documents directory](#)
- Stroustrup, Sutter, and Dos Reis: [A brief introduction to C++'s model for type- and resource-safety](#). A paper with lots of examples.
- Sergey Zubkov: [a Core Guidelines talk](#)
and here are the [slides](#). In Russian. 2017.
- Neil MacIntosh: [The Guideline Support Library: One Year Later](#). CppCon 2016.
- Bjarne Stroustrup: [Writing Good C++14](#). CppCon 2015 keynote.
- Herb Sutter: [Writing Good C++14... By Default](#). CppCon 2015 keynote.
- Peter Sommerlad: [C++ Core Guidelines - Modernize your C++ Code Base](#). ACCU 2017.

- Bjarne Stroustrup: [No Littering!](#). Bay Area ACCU 2016.
It gives some idea of the ambition level for the Core uidelines.

CppCon 的展示的幻灯片是可以获得的（其链接，还有上传的视频）。

极大欢迎对于这个列表的贡献。

鸣谢

感谢对规则、建议、支持信息和参考材料等等作出了各种贡献的许多人：

- Peter Juhl
- Neil MacIntosh
- Axel Naumann
- Andrew Pardoe
- Gabriel Dos Reis
- Zhuang, Jiangang (Jeff)
- Sergey Zubkov

请查看 [github](#) 的贡献者列表。

Pro: 剖面配置

理想情况，我们应当遵循所有这些指导方针。

这样能够得到最简洁，最规范，最少易错性，而且通常是最快的代码。

不幸的是这通常是不可能的，因为我们不得不让代码适合于大型的代码库并使用一些现存的程序库。

常常是，这样的代码已经编写好几十年了，并且并不遵循这些指导方针。

我们必须以[渐次采纳](#)为目标。

无论采用何种渐次采纳的策略，我们都应当能够首先采用一些相关指导方针的集合来处理某些问题的集合，遗留其他的以后处理。

当发现某些而不是全部的指导方针对于代码库有关时，以及当在某个专门化的应用领域

采用一组专门化的指导方针的集合时，会出现类似的“相关指导方针”的主意。

我们称这样的相关指导方针的集合为一个“剖面配置”。

我们对这种指导方针集合的目标是其内聚性，它们一起可以有助于我们达成某个特定的目标，如“消除范围错误”

或“静态类型安全性”。

每个剖面配置都被设计用于消除一个类别的错误。

而“随意”实施一些独立的规则，相对于提供确定的改善来说，更像是对代码库的破坏。

“剖面配置”是确定的并且可移植实施的规则（限制）子集，它们是专门设计以达成某项特定保证的。

“确定性”意味着它们仅需要进行局部分析，并且可以在一台计算机中进行实现（虽然并不比如此）。

“可移植实施性”表明它们和语言规则相似，因而程序员们可以期望不同实施工具对于相同的代码给出相同答案。

编写成在这样的语言剖面配置下仍免于警告的代码，可以认为是遵循这个剖面配置的。

而遵从的代码则可以认为对于该剖面配置的目标安全属性来说是安全的。

遵从的代码不会成为这种性质的错误的根本原因，

虽然程序中可能从其他的代码引入这样的错误。

剖面配置还可能会引入一些额外的库类型，以简化遵从性并鼓励编写正确的代码。

剖面配置概览：

- [Pro.type: 类型安全性](#)
- [Pro.bounds: 边界安全性](#)

- [Pro.lifetime: 生存期安全性](#)

未来，我们打算定义更多的剖面配置，并向现有剖面配置中添加更多的检查。

候选者有：

- 窄化算术提升和转换（可能会成为一个单独的安全算术剖面配置的一部分）
- 从负浮点数向无符号整型类型进行算术强制转换（同上）
- 经选择的未定义行为：从 Gabriel Dos Reis 为 WG21 研究小组开发的 UB 列表入手
- 经选择的未指明行为：处理可移植性问题。
- `const` 违反：大多数情况已经由编译器完成，但我们可以捕捉不适当的强制转换和 `const` 的不当使用。

剖面配置的开启是由实现所定义的；典型情况下，是在分析工具之中进行的设置。

要抑制对某个剖面配置检查，可以在语言构造上放一个 `suppress` 标注。例如：

```
[[suppress(bounds)]] char* raw_find(char* p, int n, char x)    // 在 p[0]..p[n - 1] 中寻找 x
{
    // ...
}
```

这样 `raw_find()` 就可以在内存中到处爬了。

显然，进行抑制应当是非常罕见的。

Pro.safety: 类型安全性剖面配置

这个剖面配置将能够简化正确使用类型的代码编写，并避免因疏忽产生类型双关。

它是关注于移除各种主要的类型违例的因素（包括对强制转换和联合的不安全使用）而达成这点的。

针对本部分的目的而言，

类型安全性被定义为这样的性质：对变量的使用不会不遵守其所定义的类型的规则。

通过类型 `T` 所访问的内存，不应该是某个实际上包含了无关类型 `U` 的对象的有效内存。

注意，当和**边界安全性**、**生存期安全性**组合起来时，安全性才是完整的。

这个剖面配置的实现应当在源代码中识别出下列模式，将之作为不符合并给出诊断信息。

类型安全性剖面配置概览：

- Type.1: [避免强制转换](#)：
 1. 请勿使用 `reinterpret_cast`；此为[避免强制转换](#)和[优先使用具名的强制转换](#)的严格的版本。
 2. 请勿在算术类型上使用 `static_cast`；此为[避免强制转换](#)和[优先使用具名的强制转换](#)的严格的版本。
 3. 当源指针类型和目标类型相同时，请勿进行指针强制转换；此为[避免强制转换](#)的严格的版本。
 4. 当指针转换可以隐式转换时，请勿使用指针强制转换；此为[避免强制转换](#)的严格的版本。
- Type.2: 请勿使用 `static_cast` 进行向下强制转换：
[代之以使用 `dynamic_cast`](#)。
- Type.3: 请勿使用 `const_cast` 强制掉 `const`（亦即不要这样做）：
[不要强制掉 `const`](#)。
- Type.4: 请勿使用 C 风格的强制转换 `(T)expression` 和函数式风格强制转换 `T(expression)`：
[优先使用构造语法，具名的强制转换](#)，或 `T{expression}`。

- Type.5: 请勿在初始化之前使用变量:

[坚持进行初始化。](#)

- Type.6: 坚持初始化成员变量:

[坚持进行初始化,](#)

可以采用[默认构造函数](#)或者

[默认成员初始化式。](#)

- Type.7: 避免裸 union:

[代之以使用 variant。](#)

- Type.8: 避免 varargs:

[不要使用 va_arg 参数。](#)

影响

在类型安全性剖面配置下，你可以相信每个操作都将在有效的对象上进行。

可能抛出异常以报告无法（在编译时）被静态地检测到的错误。

要注意的是，这种类型安全性仅当我们同样具有[边界安全性和生存期安全性](#)时才是完整的。

而没有这些保证的话，一个内存区域可能以与其所存储的单个或多个对象，或对象的一部分无关的方式被访问。

Pro.bounds: 边界安全性剖面配置

这个剖面配置将能简化对于在分配的内存块的边界之中进行操作的编码工作。

它是通过关注于移除边界违例的主要根源——即指针算术和数组索引——而做到这点的。

这个剖面配置的核心功能之一就是限制指针只能指向单个对象而不是数组。

我们将边界安全性定义为这样一种性质：程序不通过一个对象来对分配给这个对象的内存范围之外的内存进行访问。

仅当边界安全性与[类型安全性和生存期安全性](#)组合起来时才是完整的，

它们还会包含其他允许发生边界违例的不安全操作。

边界安全性剖面配置概览：

- Bounds.1: 请勿使用指针算术。请使用 span 代替:

[\(仅\) 传递单个对象的指针，并保持指针算术的简单性。](#)

- Bounds.2: 仅使用常量表达式对数组进行索引操作:

[\(仅\) 传递单个对象的指针，并保持指针算术的简单性。](#)

- Bounds.3: 避免数组向指针的退化:

[\(仅\) 传递单个对象的指针，并保持指针算术的简单性。](#)

- Bounds.4: 请勿使用不进行边界检查的标准库函数和类型:

[以类型安全的方式使用标准库](#)

影响

边界安全性意味着，当访问对象（尤其是数组）时不会越过对象的内存分配范围。

这消除了一大类的隐伏且难于发现的错误，包括（不）著名的“缓冲区溢出”错误。

这避免了安全漏洞，以及（当越界写入时发生）内存损坏错误的大量来源。

即使越界访问“只是读取操作”，它也可能导致不变式的违反（当所访问的不是预期的类型时）和“神秘的值”。

Pro.lifetime: 生存期安全性剖面配置

通过已经不指向任何东西的指针进行访问，是错误的一种主要来源，而且在许多传统的 C 或 C++ 风格的编程中这很难避免。

例如，指针可能未初始化，值为 `nullptr`，指向越过指针范围，或者指向已删除的对象。

[参见此处的设计说明书的当前版本](#)

生存期安全性剖面配置概览：

- Lifetime.1: 不要解引用无效指针：

[检测或避免。](#)

影响

一旦强制实施了编码风格规则，静态分析，以及程序库支持的组合方案之后，本剖面配置将能

- 消除 C++ 中的恶劣错误的一种主要来源
- 消除潜在安全漏洞的一种主要来源
- 通过消除多余的“偏执”检查而改善性能
- 提升代码正确性的信心
- 通过强制遵循一种关键的 C++ 语言规则而避免未定义的行为

GSL: 指导方针支持库

GSL 是一个小型的程序库，其中的设施被设计用于支持本指导方针。

不使用这些设施的话，这些指导方针不得不变得对语言细节过于限制。

核心指导方针支持库是定义在 `gsl` 命名空间中的，其中的名字可能是对标准库和其他著名程序库的名字的别名。通过 `gsl` 命名空间进行的（编译期）间接，使我们可以进行试验，以及对这些支持设施提供局部变体。

GSL 只有头文件，可以在 [GSL: 指导方针支持库](#) 找到。

支持库中的设施被设计为极为轻量化（零开销），它们相比于传统方案并不会带来任何开销。

当需要时，它们还可以用其他功能“工具化”（比如一些检查）来帮助进行诸如调试等任务。

各指导方针中，除了使用 GSL 中的类型之外，还使用了标准程序库（如 C++17）中的类型。

例如，我们假定有一个 `variant` 类型，但它当前尚未在 GSL 中。

总之，请使用[通过表决进入 C++17 的版本](#)。

由于诸如当前 C++ 版本的限制等技术原因，您所使用的程序库中可能不支持下面列出的某些 GSL 类型。请查阅您的 GSL 文档以获得更多信息。

对于以下的每个 GSL 类型，我们都为该类型给出了不变式。只要用户代码仅使用类型所提供的成员或自由函数（就是说，用户代码不会以违反任何其他指南规则的方式，绕过类型的接口来改动对象的值或位），该不变式均有效。

GSL 组件概览：

- [GSL.view: 视图](#)
- [GSL.owner](#)
- [GSL.assert: 断言](#)
- [GSL.util: 工具](#)
- [GSL.concept: 概念](#)

我们计划提供一个“ISO C++ 标准风格的”半正式的 GSL 规范。

我们依赖于 ISO C++ 标准库，并希望 GSL 的一些部分能够被吸收到标准库之中。

GSL.view: 视图

这些类型使用户可以区分带有和没有所有权的指针，并区分指向单个对象的指针和指向序列的第一个元素的指针。

“视图”都不是所有者。

引用都不是所有者（参见 [R.4](#)）。注意：有许多机会能让引用存活超过其所指代的对象，如按引用返回局部变量，持有 `vector` 的某个元素的引用然后进行 `push_back`，绑定到 `std::max(x, y + 1)`，等等。生存期安全性剖面配置的目标就是处理这些事情，但即便如此 `owner<T&>` 也没有意义且不建议使用。

它们的名字基本上遵循 ISO 标准库风格（小写字母和下划线）：

- `T*` // `T*` 不是所有者，可能为 `null`；假定为指向单个元素。
- `T&` // `T&` 不是所有者，不可能为“`null` 引用”；引用总是绑定到对象上。

“原生指针”写法（如 `int*`）假定为具有其最常见的含义；亦即指向一个对象的指针，但并不拥有它。所有者应当被转换为资源包装（如 `unique_ptr` 或 `vector<T>`），或标为 `owner<T*>`。

- `owner<T*>` // `T*`，拥有所指向/指代的对象；可能为 `nullptr`。

`owner` 用于对代码中有所有权的指针进行标记，它们无法更改为使用适当的资源包装。

其原因可能包括：

- 转换的成本。
- 需要为某个 ABI 使用指针。
- 这个指针时某种资源包装的实现的一部分。

`owner<T>` 和 `T` 的某种资源包装的区别在于它仍然需要明确进行 `delete`。

`owner<T>` 假定为指代自由存储（堆）上的某个对象。

当某个东西不应当为 `nullptr` 时，可以这样做：

- `not_null<T>` // `T` 通常是某个指针类型（例如 `not_null<int*>` 和 `not_null<owner<Foo*>>`），且不能为 `nullptr`。
`T` 可以是 `==nullptr` 有意义的任何类型。
- `span<T>` // `[p:p+n]`，构造函数接受 `{p, q}` 和 `{p, n}`；`T` 为指针类型
- `span_p<T>` // `{p, predicate} [p:q]`，其中 `q` 为首个使 `predicate(*p)` 为真的元素

`span<T>` 指代零或更多可改动的 `T`，除非 `T` 为 `const` 类型。对 `span` 元素的所有访问，尤其是通过 `operator[]` 进行的访问，默认保证进行边界检查。

注：有提案将 GSL 的 `span`（起初叫做 `array_view`）加入 C++ 标准库且已被采纳（名字和接口有改动），唯一不同是 `std::span` 不提供边界检查保证。因此，GSL 修改了 `span` 的名字和接口以跟踪 `std::span`，并应当与 `std::span` 完全相同，而其仅有差别应当为，GSL 的 `span` 默认是完全边界安全的。如果边界检查影响其接口，那么应当通过 ISO C++ 委员会带回改动的提案，以保持 `gsl::span` 和 `std::span` 接口兼容。如果 `std::span` 未来的演化添加了边界检查，则 `gsl::span` 即可移除。

“指针算术”最好在 `span` 之内进行。

指向多个 `char` 但并非 C 风格字符串的 `char*`（比如指向某个输入缓冲区的指针）应当表示为一个 `span`。

- `zstring` // `char*`，假定为 C 风格字符串；亦即以零结尾的 `char` 的序列或者是 `nullptr`

- `czstring` // `const char*`, 假定为 C 风格字符串; 亦即以零结尾的 `const char` 的序列或者是 `nullptr`

逻辑上来说, 最后两种别名是没有必要的, 但我们并不总是依照逻辑的, 它们可以在指向单个 `char` 的指针和指向 C 风格字符串的指针之间明确地进行区分。

并未假定为零结尾的字符序列应当是 `span<char>`, 或当因 ABI 问题而不可能时是 `char*`, 而不是 `zstring`。

对于不能为 `nullptr` 的 C 风格字符串, 应使用 `not_null<zstring>`。 ??? 我们需要为 `not_null<zstring>` 命名吗? 还是说它的难看是有用的?

GSL.owner: 所有权指针

- `unique_ptr<T>` // 唯一所有权: `std::unique_ptr<T>`
- `shared_ptr<T>` // 共享所有权: `std::shared_ptr<T>` (引用计数指针)
- `stack_array<T>` // 栈分配数组。元素的数量在构造时确定并固定下来。其元素可改变, 除非 `T` 为 `const` 类型。
- `dyn_array<T>` // ??? 有必要吗 ??? 堆分配数组。元素的数量在构造时确定并固定下来。其元素可改变, 除非 `T` 为 `const` 类型。基本上这是一个进行分配并拥有其元素的 `span`。

GSL.assert: 断言

- `Requires` // 前条件断言。当前放置于函数体内。今后应当移动到声明中。
// `Requires(p)` 当不满足 `p == true` 时会终止程序
// `Requires` 处于一组选项的控制之下 (强制, 错误消息, 对终止程序的替代)
- `Ensures` // 后条件断言。当前放置于函数体内。今后应当移动到声明中。

现在这些断言还是宏 (天呐!) 而且必须 (只) 被用在函数定义式之内。

等待标准委员会对于契约和断言语法的确定。

参见使用属性语法的[契约提案](#),

比如说, `Requires(p)` 将变为 `[[requires: p]]`。

GSL.util: 工具

- `finally` // `finally(f)` 创建一个 `final_action{f}`, 其析构函数将执行 `f`
- `narrow_cast` // `narrow_cast<T>(x)` 就是 `static_cast<T>(x)`
- `narrow` // `narrow<T>(x)` 在满足无符号提升下的 `static_cast<T>(x) == x` 时为 `static_cast<T>(x)`, 否则抛出 `narrowing_error` (例如, `narrow<unsigned>(-42)` 会抛出异常)
- `[[implicit]]` // 放在单参数构造函数上的“记号”, 以明确说明它们并非显式构造函数。
- `move_owner` // `p = move_owner(q)` 含义为 `p = q` 但 ???
- `joining_thread` // RAI 风格版本的进行联结的 `std::thread`
- `index` // 用于进行所有的容器和数组索引的类型 (当前是 `ptrdiff_t` 的别名)

GSL.concept: 概念

这些概念 (类型谓词) 借用于

Andrew Sutton 的 Origin 程序库,

Range 提案,

以及 ISO WG21 的 Palo Alto TR。

其中许多都与已在 C++20 中成为 ISO C++ 标准的概念十分相似。

- `String`
- `Number`
- `Boolean`
- `Range` // C++20 中为 `std::ranges::range`
- `Sortable` // C++20 中为 `std::sortable`
- `EqualityComparable` // C++20 中为 `std::equality_comparable`
- `Convertible` // C++20 中为 `std::convertible_to`
- `Common` // C++20 中为 `std::common_with`
- `Integral` // C++20 中为 `std::integral`
- `SignedIntegral` // C++20 中为 `std::signed_integral`
- `SemiRegular` // C++20 中为 `std::semiregular`
- `Regular` // C++20 中为 `std::regular`
- `TotallyOrdered` // C++20 中为 `std::totally_ordered`
- `Function` // C++20 中为 `std::invocable`
- `RegularFunction` // C++20 中为 `std::regular_invocable`
- `Predicate` // C++20 中为 `std::predicate`
- `Relation` // C++20 中为 `std::relation`
- ...

GSL.ptr: 智能指针概念

- `Pointer` // 带有 `*`, `->`, `==`, 以及默认构造的类型 (默认构造被假定为设值为唯一的“null”值)
- `Unique_pointer` // 符合 `Pointer` 的类型, 可移动但不可复制
- `Shared_pointer` // 符合 `Pointer` 的类型, 可复制

NL: 命名和代码布局建议

维持一致的命名和代码布局是很有用的。

即便不为其他原因, 也可以减少“我的代码风格比你的好”这类的纷争。

然而, 人们使用许多许多的不同代码风格, 并狂热地坚持它们 (的优缺点)。

而且, 大多数的现实项目都包含来自于许多来源的代码, 因而通常不可能把所有的代码都标准化为某个单一的代码风格。

经过许多的用户请求给予指导后, 我们给出一组规则, 当你没有更好的选择时可以使用它们, 但真正的目标在于一致性, 而不是任何一组特定的规则。

IDE 和工具可以提供辅助 (当然也可能造成妨碍)。

命名和代码布局规则:

- [NL.1: 不要在代码注释中说明可以由代码来清晰表达的东西](#)
- [NL.2: 在代码注释中说明意图](#)
- [NL.3: 保持代码注释简明干脆](#)
- [NL.4: 保持一种统一的缩进风格](#)
- [NL.5: 避免在名字中编码类型信息](#)
- [NL.7: 使名字的长度大约正比于其作用域的长度](#)
- [NL.8: 使用一种统一的命名风格](#)
- [NL.9: 将 `ALL_CAPS` \(全大写\) 仅用于宏的名字](#)
- [NL.10: 优先采用 `underscore_style` \(下划线风格\) 的名字](#)
- [NL.11: 使字面量可阅读](#)
- [NL.15: 节制地使用空格](#)

- [NL.16: 使用一种常规的类成员声明次序](#)
- [NL.17: 使用从 K&R 衍生出的代码布局](#)
- [NL.18: 使用 C++ 风格的声明符布局](#)
- [NL.19: 避免使用容易误读的名字](#)
- [NL.20: 不要把两个语句放在同一行中](#)
- [NL.21: 每个声明式（仅）声明一个名字](#)
- [NL.25: 请勿将 `void` 用作参数类型](#)
- [NL.26: 采用符合惯例的 `const` 写法](#)
- [NL.27: 为代码文件使用后缀 `.cpp`，而对接口文件使用后缀 `.h`](#)

这些问题的大部分都是审美问题，程序员都有很强的个人倾向。

IDE 也都会提供某些默认方案和一组替代方案。

这些规则是作为缺省建议的，如果没有别的理由，请采用它们。

我们收到一些意见称命名和代码布局非常个人化和任意性，我们不应该试图为之“立法”。

我们并不是在“立法”（参见前一个段落）。

不过，我们也收到了大量的针对某些命名和代码布局约定的请求，要求当没有外来限制的时候应当采用它们。

更专门和详细的规则更加易于强制实施。

这些规则恐怕会和 [PPP Style Guide](#) 中的建议有很强的相似性，

它是为支持 Stroustrup 的 [Programming: Principles and Practice using C++](#) 而编制的。

NL.1: 不要在代码注释中说明可以由代码来清晰表达的东西

理由

编译器不会读注释。

注释没有代码那么精确。

注释不会像代码那样进行一致地更新。

示例，不好

```
auto x = m * v1 + vv; // 将 m 乘以 v1 并将其结果加上 vv
```

强制实施

构建一个 AI 程序来解释口语英文文字，看看它所说的是否可以用 C++ 来更好地表达。

NL.2: 在代码注释中说明意图

理由

代码表示的是做了什么，而不是想要做成什么。通常来说意图比实现能够更清晰简明地进行说明。

示例

```
void stable_sortSortable& c)
    // 对 c 根据由 < 决定的顺序进行排序，保持相等元素（由 == 定义）的
    // 原始相对顺序
{
    // ... 相当多的不平常的代码行 ...
}
```

注解

如果代码注释和代码有冲突，则它们都可能是错的。

NL.3: 保持代码注释简明干脆

理由

冗长啰嗦会拖慢理解速度，而且到处散布在代码文件里也会让代码难于阅读。

注解

使用明白易懂的英文。

也许我可以流利使用丹麦语，但大多数程序员不行；我的代码的维护者也不行。

避免使用网络用语，注意你的文法，标点，以及大小写。

目标是专业性，而不是“够酷”。

强制实施

不可能。

NL.4: 保持一种统一的缩进风格

理由

可读性。避免“微妙的错误”。

示例，不好

```
int i;
for (i = 0; i < max; ++i); // 可能出现的 BUG
if (i == j)
    return i;
```

注解

总是把 `if (...)`, `for (...)`, 以及 `while (...)` 之后的语句进行缩进是一个好主意：

```
if (i < 0) error("negative argument");

if (i < 0)
    error("negative argument");
```

强制实施

使用一种工具。

NL.5: 避免在名字中编码类型信息

原理

当名字反映类型而不是功能时，它将变得难于为提供其功能而改变其所使用的类型。

而且，当改变变量的类型时，使用它的代码也得修改。

最小化无意进行的转换。

示例，不好

```
void print_int(int i);
void print_string(const char*);

print_int(1);           // 重复，人工进行类型匹配
print_string("xyzzy"); // 重复，人工进行类型匹配
```

示例，好

```
void print(int i);
void print(string_view); // 对任意字符串式的序列都能工作

print(1);           // 简洁，自动类型匹配
print("xyzzy");    // 简洁，自动类型匹配
```

注解

带有类型编码的名字要么啰嗦要么难懂。

```
prints // 打印一个 std::string
prints // 打印一个 C 风格字符串
printi // 打印一个 int
```

在无类型语言中曾经采用过像匈牙利记法这样的技巧来在名字中编码类型，但在像 C++ 这样的强静态类型语言中，这通常是不必要而且实际上是有害的，因为这些标注会过时（这些累赘和注释类似，而且和它们一样会烂掉），而且它们干扰了语言的恰当用法（应当代之以使用相同的名字和重载决议）。

注解

一些代码风格会使用非常一般的（而不是特定于类型的）前缀来代表变量的一般用法。

```
auto p = new User();
auto p = make_unique<User>();
// 注: "p" 并非是说“User 类型的原始指针”，
//      而只是一般性的“这是一次间接访问”

auto cntHits = calc_total_of_hits(/*...*/);
// 注: "cnt" 并非用于编码某个类型，
//      而只是一般性的“这是某种东西的一个计数”
```

这样做是没有害处的，且并不属于本条指导方针，因为其并未编码类型信息。

注解

一些代码风格会对成员和局部变量，以及全局变量之间进行区分。

```
struct S {
    int m_;
    S(int m) : m_{abs(m)} { }
};
```

这样做是没有害处的，且并不属于本条指导方针，因为其并未编码类型信息。

注解

像 C++ 这样，一些代码风格对类型和非类型之间进行区分。
例如，对类型名字首字母大写，而函数和变量名字则不这样做。

```
typename<typename T>
class HashTable {    // 将 string 映射为 T
    // ...
};

HashTable<int> index;
```

这样做是没有害处的，且并不属于本条指导方针，因为其并未编码类型信息。

NL.7: 使名字的长度大约正比于其作用域的长度

原理: 作用域越大，搞混的机会和意外的名字冲突的机会就越大。

示例

```
double sqrt(double x);    // 返回 x 的平方根；x 必须是非负数

int length(const char* p); // 返回零结尾的 C 风格字符串的字符数量

int length_of_string(const char zero_terminated_array_of_char[])    // 不好：啰嗦

int g;        // 不好：全局变量具有密秘的名字

int open;    // 不好：全局变量使用短小且常用的名字
```

为指针使用 `p`，以及为浮点变量使用 `x` 是符合惯例的，在受限的作用域中不会造成混乱。

强制实施

???

NL.8: 使用一种统一的命名风格

原理: 命名和命名风格的一致性会提高可读性。

注解

命名风格有好多，当你使用多个程序库时，你无法遵循所有它们不同的命名约定。
应当选用一种“自有风格”，但保持“导入”的程序为其原有风格不变。

示例

ISO 标准仅使用小写字母和数字，并用下划线进行词的连接。

- `int`
- `vector`
- `my_map`

避免使用双下划线 `_`。

示例

Stroustrup:

采用 ISO 标准，但在自己的类型和概念上采用大写字母：

- `int`
- `vector`
- `My_map`

示例

CamelCase：多词标识符的每个词首字母大写：

- `int`
- `vector`
- `MyMap`
- `myMap`

一些命名约定会将首字母大写，而另一些不会。

注解

应当试图在对缩略词的使用和标识符长度上保持一致风格。

```
int mtbf {12};  
int mean_time_between_failures {12}; // 你自己决定
```

强制实施

除使用具有不同命名约定的程序库之外应当是可能做到的。

NL.9: 将 ALL_CAPS (全大写) 仅用于宏的名字

理由

避免在宏和遵循作用域和类型规则的名字之间造成混乱。

示例

```
void f()  
{  
    const int SIZE{1000}; // 不好，应代之以 'size'  
    int v[SIZE];  
}
```

注解

这条规则适用于非宏的符号常量：

```
enum bad { BAD, WORSE, HORRIBLE }; // 不好
```

强制实施

- 对带有小写字母的宏进行标记
- 对 `ALL_CAPS` 非宏名字进行标记

NL.10: 优先采用 `underscore_style` (下划线风格) 的名字

理由

用下划线来分隔名字的各部分就是 C 和 C++ 的原始风格，并被用于 C++ 标准库中。

注解

这条规则仅作为当你有选择权时的缺省方案。

通常你是没有什么选择权的，而只能遵循某个已经设立的风格以维持[一致性](#)。

对一致性的需要优先于个人喜好。

这个推荐适用于[当你没有约束条件或者没有更好的想法时](#)的情况。

经过很多要求给予指导后，添加这个规则。

示例

[Stroustrup](#):

采用 ISO 标准，但在自己的类型和概念上采用大写字母：

- `int`
- `vector`
- `My_map`

强制实施

不可能。

NL.11: 使字面量可阅读

理由

可读性。

示例

用数字分隔符来避免长串的数字

```
auto c = 299'792'458; // m/s2
auto q2 = 0b0000'1111'0000'0000;
auto ss_number = 123'456'7890;
```

示例

需要清晰性时使用字面量后缀

```
auto hello = "Hello!"s; // std::string
auto world = "world"; // C 风格字符串
auto interval = 100ms; // 使用 <chrono>
```

注解

不能在代码中到处当做“魔法常量”一样乱用字面量，
但当定义它们时使它们更可读仍是个好主意。
在较长的整数串中很容易出现拼写错误。

强制实施

标记长数字串。麻烦的是“长”的定义；也许应当是 7。

NL.15: 节制地使用空格

理由

太多的空格会让文本更大更分散。

示例, 不好

```
#include < map >

int main(int argc, char * argv [ ])
{
    // ...
}
```

示例

```
#include <map>

int main(int argc, char* argv[])
{
    // ...
}
```

注解

一些 IDE 有其自己的看法，并会添加分散的空格。

这个推荐适用于[当你没有约束条件或者没有更好的想法时](#)的情况。
经过很多要求给予指导后，添加这个规则。

注解

我们将恰当放置的空白评价为能够明显有助于可读性。但请勿过度。

NL.16: 使用一种常规的类成员声明次序

理由

一种常规的成员次序会提高可读性。

以如下次序声明类

- 类型：类，枚举，别名（`using`）
- 构造函数，赋值，析构函数
- 函数
- 数据

采用先是 `public`，然后是 `protected`，之后是 `private` 的次序。

这个推荐适用于[当你没有约束条件或者没有更好的想法时](#)的情况。

经过很多要求给予指导后，添加这个规则。

示例

```
class X {  
public:  
    // 接口  
protected:  
    // 供派生类实现使用的不带检查的函数  
private:  
    // 实现细节  
};
```

示例

有时候，成员的默认顺序，与将公开接口从实现细节中分离出来的需求之间有冲突。
这种情况下，私有类型和函数可以和私有数据放在一起。

```
class X {  
public:  
    // 接口  
protected:  
    // 供派生类实现使用的不带检查的函数  
private:  
    // 实现细节（类型，函数和数据）  
};
```

示例，不好

避免让具有某一种访问（如 `public`）的多个声明块被具有不同访问（如 `private`）的其他声明块分隔开。

```
class X {  
public:  
    void f();  
public:  
    int g();  
    // ...  
};
```

用宏来声明成员组的做法通常会导致违反所有的次序规则。

不过，宏的使用掩盖了其所表达的东西。

强制实施

对背离上述建议次序的代码进行标记。将会有大量的老代码不符合这条规则。

NL.17: 使用从 K&R 衍生出的代码布局

理由

这正是 C 和 C++ 的原始代码布局。它很好地保持了纵向空间。它对不同语言构造（如函数和类）进行了很好的区分。

注解

在 C++ 的语境中，这种风格通常被称为“Stroustrup”。

这个推荐适用于[当你没有约束条件或者没有更好的想法时](#)的情况。

经过很多要求给予指导后，添加这个规则。

示例

```
struct Cable {
    int x;
    // ...
};

double foo(int x)
{
    if (0 < x) {
        // ...
    }

    switch (x) {
    case 0:
        // ...
        break;
    case amazing:
        // ...
        break;
    default:
        // ...
        break;
    }

    if (0 < x)
        ++x;

    if (x < 0)
        something();
    else
        something_else();

    return some_value;
}
```

注意 `if` 和 `()` 之间有一个空格

注解

每个语句, `if` 的分支, 以及 `for` 的代码体都使用单独的代码行。

注解

`class` 和 `struct` 的 `{` 并不在单独的代码行上, 但函数的 `{` 在单独的代码行上。

注解

对你自定义的类型的名字进行首字母大写, 以将其与标准库类型相区分。

注解

不要对函数名大写。

强制实施

如果想要强制实施的话, 请使用某个 IDE 进行格式化。

NL.18: 使用 C++ 风格的声明符布局

理由

C 风格的布局强调其在表达式中的用法和文法, 而 C++ 风格强调的是类型。对表达式用法的说辞并不适用于引用。

示例

```
T& operator[](size_t); // OK
T &operator[](size_t); // 奇怪
T & operator[](size_t); // 不确定
```

注解

这个推荐适用于[当你没有约束条件或者没有更好的想法时](#)的情况。

经过很多要求给予指导后, 添加这个规则。

强制实施

由于历史原因而不可能。

NL.19: 避免使用容易误读的名字

理由

可读性。

并非每个人都有能将字符轻易区分开的屏幕和打印机。

我们很容易搞混拼写相似和略微拼错的单词。

示例

```
int o001lL = 6; // 不好

int splunk = 7;
int splonk = 8; // 不好: splunk 和 splonk 很容易搞混
```

强制实施

???

NL.20: 不要把两个语句放在同一行中

理由

可读性。

当一行里有多个语句时，相当容易忽视某个语句。

示例

```
int x = 7; char* p = 29;      // 请勿如此
int x = 7; f(x);    ++x;      // 请勿如此
```

强制实施

容易。

NL.21: 每个声明式（仅）声明一个名字

理由

可读性。

最小化声明符语法造成的混乱。

注解

相关细节，参见 [ES.10](#)。

NL.25: 请勿将 `void` 用作参数类型

理由

这很啰嗦，而且仅在考虑 C 兼容性时才有必要。

示例

```
void f(void);  // 不好
void g();      // 好多了
```

注解

即便是 Dennis Ritchie 自己都认为 `void f(void)` 很讨厌。

你可以反驳称，在 C 中当函数原型很少见时禁止这样的代码：

```
int f();
f(1, 2, "weird but valid C89"); // 希望 f() 被定义为 int f(a, b, c) char* c; { /*
... */ }
```

可能造成很大的问题，但这并不适于 21 世纪和 C++。

NL.26: 采用符合惯例的 `const` 写法

理由

更多程序员更加熟悉惯例写法。

大型代码库中的一致性。

示例

```
const int x = 7;      // OK
int const y = 9;      // 不好

const int *const p = nullptr;    // OK, 指向常量 int 的常量指针
int const *const p = nullptr;    // 不好, 指向常量 int 的常量指针
```

注解

我们知道你可能会说“不好”的例子比标有“OK”的更符合逻辑，

但它们会让更多人搞混，尤其是那些依赖于采用了远为常用，符合惯例的 OK 风格的教学材料的新手们。

一如往常，请记住这些命名和代码布局规则的目标在于一致性，而审美则会有广泛的变化。

这个推荐适用于[当你没有约束条件或者没有更好的想法时](#)的情况。

经过很多要求给予指导后，添加这个规则。

强制实施

标记用作类型的后缀的 `const`。

NL.27: 为代码文件使用后缀 `.cpp`，而对接口文件使用后缀 `.h`

理由

这是一条历史悠久的约定。

不过一致性更加重要，因此如果你的项目用了别的约定的话，应当遵守它。

注解

这项约定反应了一种常见使用模式：

头文件更容易和 C 语言共享使用，并可以作为 C++ 和 C 编译，它们通常用 `.h` 后缀，

并且对于有意要和 C 共用的头文件来说，让所有头文件都使用 `.h` 而不是别的扩展名要更加容易。

另一方面，实现文件则很少会和 C 共用，通常应当和 `.c` 文件相区别，

因此一般最好为所有的 C++ 实现文件用别的扩展名（如 `.cpp`）来命名。

特定的名字 `.h` 和 `.cpp` 并不是必要的（只是作为缺省建议），其他的名字也被广泛采用。

例子包括 `.hh`, `.c`, 和 `.cxx` 等。请以类似方式使用这些名字。

本文档中我们把 `.h` 和 `.cpp` 作为头文件和实现文件的简便提法，

虽然实际上的扩展名可能是不同的。

也许你的 IDE（如果你使用的话）对后缀有较强的倾向。

示例

```
// foo.h:  
extern int a; // 声明  
extern void foo();  
  
// foo.cpp:  
int a; // 定义  
void foo() { ++a; }
```

`foo.h` 提供了 `foo.cpp` 的接口。最好避免全局变量。

示例，不好

```
// foo.h:  
int a; // 定义  
void foo() { ++a; }
```

一个程序中两次 `#include <foo.h>` 将导致因为对唯一定义规则的两次违反而出现一个连接错误。

强制实施

- 对不符合约定的文件名进行标记。
- 检查 `.h` 和 `.cpp` (或等价文件) 遵循下列各规则。

FAQ: 常见问题及其回答

本节中包括了对关于这些指导方针的常见问题的回答。

FAQ.1: 这些指导方针的想要达成什么目标？

请参见[本页面开头](#)。这是一个开源项目，旨在为采用当今的 C++ 标准来编写 C++ 代码而维护的一组现代的权威指导方针。这些指导方针的设计是现代的，尽可能使机器可实施的，并且是为贡献和分支保持开放，以使各种组织机构可以便于将它们整合到其自己组织的编码指导方针之中。

FAQ.2: 这项工作是何时何地首次公开的？

是在 [Bjarne Stroustrup 在他为 CppCon 2015 的开场主旨演讲，“Writing Good C++14”](#)。另请参见[相应的 isocpp.org 博客条目](#)，关于类型和内存安全性指导方针的原理请参见 [Herb Sutter 的后续 CppCon 2015 演讲，“Writing Good C++14 ... By Default”](#)。

FAQ.3: 谁是这些指导方针的作者和维护者？

最初的主要作者和维护者是 Bjarne Stroustrup 和 Herb Sutter，而迄今为止的指导方针则是由来自 CERN, Microsoft, Morgan Stanley，以及许多其他组织机构的专家所贡献的。指导方针发布时，其正处于 "0.6" 状态，我们欢迎人们进行贡献。正如 Stroustrup 在其声明中所说：“我们需要帮助！”

FAQ.4: 我如何进行贡献呢？

参见 [CONTRIBUTING.md](#)。我们感激志愿者的帮助！

FAQ.5: 怎样成为一名编辑或维护者?

通过先进行大量贡献并使你的贡献被认可具有一致的质量。参见 [CONTRIBUTING.md](#)。我们感激志愿者的帮助!

FAQ.6: 这些指导方针被 ISO C++ 标准委员会采纳了吗? 它们是否代表委员会的一致意见?

不是这样。这些指导方针不在标准之内。它们是为标准服务的，而当前维护的指导方针是为了更有效地使用当前的标准 C++ 的。我们的目标是使其与委员会所设计的标准保持同步。

FAQ.7: 既然这些指导方针并不是委员会所采纳的, 它们为何在 github.com/isocpp 之下呢?

因为 `isocpp` 是标准 C++ 基金会; 而标准委员会的仓库则处于 github.com/cplusplus 之下。我们需要一个中立组织来持有版权和许可以明确其并不是由某个人或供应商所控制的。这个自然实体就是基金会, 其设立是为了推进使用并持续更新对现代标准 C++ 的理解, 以及推进标准委员会的工作。其所遵循的正是与 isocpp.org 为 [C++ FAQ](#) 所做的相同模式, 它是有 Bjarne Stroustrup, Marshall Cline, 和 Herb Sutter 所发起的工作, 并以相同的方式贡献为了开放项目。

FAQ.8: 会有 C++98 版本的指导方针吗? C++11 版本呢?

不会。这些指导方针的目标是更好地使用现代标准 C++, 以及假定你有一个现代的遵循标准的编译器时如何进行代码编写的。

FAQ.9: 这些指导方针中会提出新的语言功能吗?

不会。这些指导方针的目标是更好地使用现代标准 C++, 它们自我限定为仅建议使用这些功能。

FAQ.10: 这些指导方针的书写使用的是哪个版本的 Markdown?

这些编码指导方针使用的是 [CommonMark](#), 以及 `<a>` HTML 锚定元素。

我们正在考虑以下这些来自 [GitHub Flavored Markdown \(GFM\)](#) 的扩展:

- 有围栏代码块 (正在讨论是否统一使用缩进还是围栏代码块)
- 表格 (我们虽然还没用到, 但很需要它们, 这是一种 GFM 扩展)

避免使用其他 HTML 标签和其他扩展。

注意: 我们还没对这种风格达成一致。

FAQ.50: 什么是 GSL (指导方针支持程序库) ?

GSL 是在指导方针中所指定的类型和别名的一个小集合。当写下本文时, 对它们的说明还过于松散; 我们计划添加一个 WG21 风格的接口规范来确保不同实现之间保持一致, 并作为一项可能的标准化提案, 按常规遵循标准委员会进行采纳、改进、修订或否决。

FAQ.51: github.com/Microsoft/GSL 是 GSL 吗?

不是。它只是由 Microsoft 所贡献的第一个实现。我们鼓励其他供应商提供其他的实现, 对该实现的分支和贡献也是被鼓励的。书写本文作为一项公开项目的一周中, 已经出现了至少一个 GPLv3 的开源实现。我们计划制定一个 WG21 风格的接口规范来确保不同实现之间保持一致。

FAQ.52: 为何不在指导方针之中提供一个真正的 GSL 实现呢?

我们不愿去保佑某个特定的实现，因为我们不希望让人们以为只有一个实现，而疏忽大意地扼杀了其他并行的实现。而如果在指导方针中包含一个真正实现的话，无论是谁提供了它都会变得过于有影响力。我们更倾向于采用委员会的更具长期性的方案，即指定其接口而不是实现。但同时我们也需要至少存在一个实现；希望可以有很多。

FAQ.53: 为什么不把 GSL 类型提交给 Boost 呢?

因为我们想要立刻使用它们，也因为我们想要在一旦标准库中出现了满足其需要的类型时立刻将它们撤销掉。

FAQ.54: ISO C++ 标准委员会采纳了 GSL (指导方针支持程序库) 吗?

没有。GSL 的存在只为提供少量标准库中还没有的类型和别名。如果委员会决定了（这些类型或者满足其需要的其他类型的）标准化的版本，就可以将它们从 GSL 中删除了。

FAQ.55: 既然你是尽可能使用标准类型，为什么 GSL 的 `span<char>` 同 Library Fundamentals 1 Technical Specification 和 C++17 工作文本中的 `string_view` 不同呢？为什么不使用委员会采纳的 `string_view`？

有关 C++ 标准库的视图的分类的统一观点是，“视图（view）”意味着“只读”，而“跨距（span）”意味着“可读写”。如果你只需要一组字符的不需要保证边界检查的只读视图，并且你可以用 C++17，那就使用 C++17 的 `std::string_view`。否则，如果你需要的是不需要保证边界检查的可读写视图，并且可以用 C++20，那就用 C++20 的 `std::span<char>`。否则，就用 `gsl::span<char>`。

FAQ.56: `owner` 和提案的 `observer_ptr` 一样吗？

不一样。`owner` 有所有权，它是一个别名，而且适用于任何间接类型。而 `observer_ptr` 的主要意图则是明确某个没有所有权的指针。

FAQ.57: `stack_array` 和标准的 `array` 一样吗？

不一样。`stack_array` 保证在栈上分配。虽然 `std::array` 直接在其自身内部包含存储，但 `array` 对象可以放在包括堆在内的任何地方。

FAQ.58: `dyn_array` 和 `vector` 或者提案的 `dynarray` 一样吗？

不一样。`dyn_array` 是不可改变大小的，是一种指代堆分配的固定大小数组的一种安全方式。与 `vector` 不同，它是为了取代数组 `new[]` 的。与委员会中提案的 `dynarray` 不同，它并不会参与编译器和语言的魔法，来在当它作为分配于栈上的对象的成员时也在栈上分配；它只不过指代一个“动态的”或基于堆的数组而已。

FAQ.59: `Expect`s 和 `assert` 一样吗?

不一样。它是一种对于契约前条件语言支持的占位符。

FAQ.60: `Ensures` 和 `assert` 一样吗?

不一样。它是一种对于契约后条件语言支持的占位符。

附录 A: 程序库

这个部分列出了一些推荐的程序库，并且特别推荐了其中的几个。

??? 这个对一般性指南来说合适吗？我觉得不是 ???

附录 B: 代码的现代化转换

理想情况下，我们的所有代码都应当遵循全部的规则。

而实际情况则是，我们不得不对付大量的老代码：

- 那些在我们的指导方针被建立或者被了解到之前所编写的代码
- 那些依据老的或者不同的标准所编写的程序库
- 那些在“不寻常”的约束下编写的代码
- 那些我们还未来得及使其现代化的代码

如果有上百万行的新代码的话，“立刻改掉它们”的想法一般都是不现实的。

因此，我们需要一种方式能够渐进地对代码库进行现代化转换。

将老代码升级为现代风格可能是很让人却步的工作。

老代码通常即混乱（难于理解）又可以（在当前使用范围内）正确工作。

很有可能，原来的程序员并不在场，而且测试用例也不完整。

代码的混乱性显著地增大了为进行任何改动所需要的工作量和引入错误的风险。

通常，混乱的老代码的运行会有不必要的缓慢，因为它需要过期的编译器，并且无法得益于当代硬件的改进。

许多情况下，都需要某种自动进行“现代化转换”的工具支持来进行主要的升级工作。

对代码现代化转换的目的在于简化新功能的添加，简化维护工作，以及增加性能（吞吐量或延迟），和更好地利用当代的硬件能力。

而让代码“看起来更好”或“遵循现代风格”自身并不能成为改动的理由。

每一种改动都蕴含着风险，而是由过时的代码库则会蕴含一些成本（包含丢失机会带来的成本）。

成本的缩减必须超过风险。

如何做呢？

并不存在唯一的代码现代化转换的方案。

如何最好地执行，依赖于代码，更新的进度压力，开发者的背景，以及可用的工具。

下面是一些（非常一般的）想法：

- 理想情况是“对全部代码一起进行升级”。这将在最短的总时间内获得最大的好处。
在大多数情况下，这也是不可能的。
- 我们可以对代码库以模块为单位进行转换，不过任何影响接口（尤其是 ABI）的规则，如 [使用 span](#)，都无法按模块来达成。
- 我们可以“自底向上”转换代码，并最先应用我们估计在给定的代码库上将会带来最大好处和最少麻烦的那些规则。

- 我们可以从关注接口开始，比如说，保证没有资源的泄漏，没有指针误用等。这可能会导致涉及整个代码库的一些改动，不过它们是最可能会带来巨大好处的改动。以后，隐藏在这些接口后面的代码可以渐进地进行现代化转换而不会影响其他的代码。

无论你选择哪种方式，都要注意，对指导方针的最高度的遵循性才会带来大多数的好处。这些指导方针并不是一组无关规则的随机集合，并不能让你随意选取并期望取得成功。

我们衷心希望听到有关它们的使用经验，以及有关工具是如何使用的。如果有分析工具（即便是代码变换工具）的支持的话，代码现代化转换后可以变得更快，更简单，而且更安全。

附录 C: 讨论

这个部分包含了对规则和规则集合的跟进材料。

尤其是，我们列出了更多的原理说明，更长的例子，以及对替代方案的探讨等。

讨论: 以成员的声明顺序进行成员变量的定义和初始化

成员变量总是以它们在类定义中的声明顺序进行初始化，因此在构造函数初始化列表中应当以该顺序来书写它们。以别的顺序书写它们只会让代码混淆，因为它并不会以你所见到的顺序来运行，而这会导致难于发现与顺序有关的 BUG。

```
class Employee {
    string email, first, last;
public:
    Employee(const char* firstName, const char* lastName);
    // ...
};

Employee::Employee(const char* firstName, const char* lastName)
: first(firstName),
last(lastName),
// 不好: first 和 last 还未构造
email(first + "." + last + "@acme.com")
{}
```

在这个例子中，`email` 比 `first` 和 `last` 构造得早，因为它是先声明的。这意味着其构造函数对 `first` 和 `last` 的使用过早了——不只在它们被设为所需的值之前，而完全是在它们被构造之前就使用了。

如果类定义和构造函数体是在不同文件中的话，这种由成员变量声明顺序对构造函数的正确性造成的影响将更难于发现。

参考:

[[Cline99]](#Cline99) §22.03-11, [[Dewhurst03]](#Dewhurst03) §52-53, [[Koenig97]](#Koenig97) §4, [[Lakos96]](#Lakos96) §10.3.5, [[Meyers97]](#Meyers97) §13, [[Murray93]](#Murray93) §2.1.3, [[Sutter00]](#Sutter00) §47

讨论：使用 =, {}, 和 () 作为初始化式

???

讨论：当需要在初始化过程中使用“虚函数行为”时，使用工厂函数

如果你的设计需要从基类的构造函数或析构函数中对 `f` 或者 `g` 这样的函数向派生类进行虚函数派发的话，你其实需要的是其他技巧，比如后构造函数——一种必须由调用者调用以完成初始化过程的成员函数，它可以安全地调用 `f` 和 `g`，这是由于成员函数中的虚函数调用能够正常工作。“参考”部分中列出了一些这样的技巧。以下是一个不完整的可选项列表：

- **推卸责任**：仅仅给出文档说明，要求用户代码在对象构造之后必须立刻调用后初始化函数。
- **惰性后初始化**：在第一个调用的成员函数中进行。用基类中的一个布尔标记说明后初始化是否已经执行过。
- **使用虚基类语义**：语言规则要求由最终派生类的构造函数来决定调用哪个基类构造函数；你可以利用这点。（参见[[Taligent94]](#Taligent94)。）
- **使用工厂函数**：以这种方式，你可以轻易确保进行对后构造函数的调用。

以下是对最后一种选项的一个例子：

```
class B {  
public:  
    B()  
    {  
        /* ... */  
        f(); // 不好：C.82：不要在构造函数和析构函数中调用虚函数  
        /* ... */  
    }  
  
    virtual void f() = 0;  
};  
  
class B {  
protected:  
    class Token {};  
  
public:  
    // 需要公开构造函数以便 make_shared 可以访问它。  
    // 通过要求一个 Token 达成受保护访问等级。  
    explicit B(Token) { /* ... */ } // 创建不完全初始化的对象  
    virtual void f() = 0;  
  
    template<class T>  
    static shared_ptr<T> create() // 创建共享对象的接口  
    {  
        auto p = make_shared<T>(typename T::Token{});  
        p->post_initialize();  
        return p;  
    }  
  
protected:  
    virtual void post_initialize() // 构造之后立即调用  
    { /* ... */ f(); /* ... */ } // 好：虚函数分派是安全的  
}
```

```

};

class D : public B { // 某个派生类
protected:
    class Token {};

public:
    // 需要公开构造函数以便 make_shared 可以访问它。
    // 通过要求一个 Token 达成受保护访问等级。
    explicit D(Token) : B(B::Token{}) {}
    void f() override { /* ... */ };

protected:
    template<class T>
    friend shared_ptr<T> B::create();
};

shared_ptr<D> p = D::Create<D>(); // 创建一个 D 对象

```

这种设计需要遵守以下纪律：

- 像 `D` 这样的派生类不能暴露可公开调用的构造函数。否则的话，`D` 的使用者就能够创建 `D` 对象而不调用 `post_initialize` 了。
- 分配被限定为使用 `operator new`。不过，`B` 可以覆盖 `new`（参见 [SuttAlex05](#) 条款 45 和 46）。
- `D` 必须定义一个带有与 `B` 所选择的相同的参数的构造函数。不过，定义多个重载的 `create` 可以缓和这个问题；而且还可以是这些重载对参数类型进行模板化。

一旦满足了上述要求，这个设计就可以保证对于任意完全构造的 `B` 的派生类对象，都将调用 `post_initialize`。`post_initialize` 不必是虚函数；它可以随意进行虚函数调用。

总之，不存在完美的后构造技巧。最差的方式是完全回避问题而只是让调用方来人工调用后构造函数。即便是最佳方案也需要采用一种不同的对象构造语法（易于进行编译期检查）以及需要派生类的作者的协作（这无法进行编译期进行检查）。

参考: [[Alexandrescu01]](#Alexandrescu01) §3, [[Boost]](#Boost), [[Dewhurst03]](#Dewhurst03) §75, [[Meyers97]](#Meyers97) §46, [[Stroustrup00]](#Stroustrup00) §15.4.3, [[Taligent94]](#Taligent94)

讨论: 基类的析构函数应当要么是 `public` 和 `virtual`, 要么是 `protected` 且非 `virtual`

析构应不应该表现为虚函数？就是说，是否允许通过指向 `base` 类的指针来进行析构呢？如果是的话，`base` 的析构函数为被调用则必须是 `public` 的，而且必须 `virtual`，否则调用就会导致未定义行为。否则的话，它应当是 `protected` 的，这样就只有派生类可以在它们自己的析构函数中调用它，且应当是非 `virtual` 的，因为它并不需要表现为虚函数的行为。

示例

基类的一般情况是为了具有 `public` 的派生类，因而调用方代码基本上可以确定要用到某种比如 `shared_ptr<base>` 这样的东西：

```

class Base {
public:
    ~Base(); // 不好，非 virtual
    virtual ~Base(); // 好
    // ...
};

class Derived : public Base { /* ... */ };

{
    unique_ptr<Base> pb = make_unique<Derived>();
    // ...
} // 只有当 ~Base 是虚函数时 ~pb 才会调用正确的析构函数

```

少数比如策略类这类的情况下，类被用作基类是为方便起见，而并非是其多态行为。建议将它们的析构函数作为 `protected` 和非 `virtual` 函数：

```

class My_policy {
public:
    virtual ~My_policy(); // 不好，public 并且 virtual
protected:
    ~My_policy(); // 好
    // ...
};

template<class Policy>
class customizable : Policy { /* ... */ }; // 注：private 继承

```

注解

这个简单的指导方针演示了一种微妙的问题，而且反映了继承的现代用法以及面向对象的设计原则。

对于某个基类 `Base`，调用方代码可能通过指向 `Base` 的指针来销毁派生类对象，比如使用一个 `unique_ptr<Base>`。如果 `Base` 的析构函数是 `public` 的且非 `virtual`（默认情况），它就可能意外地在实际指向一个派生类对象的指针上进行调用，这种情况下想要进行的删除的行为是未定义的。这种状况曾导致一些老编码标准提出通用的要求，让所有基类析构函数都必须 `virtual`。这种做法杀伤力过大了（虽然这是常见的）；其实，规则应当是当且仅当基类析构函数是 `public` 时才要求它是 `virtual` 的。

编写一个基类就是在定义一种抽象（参见条款 35 到 37）。注意对于参与这个抽象的每个成员函数来说，你都需要作出以下决定：

- 它是否应当表现为虚函数。
- 它是应当对所有使用 `Base` 指针的调用方公开，还是作为隐藏的内部实现细节。

如条款 39 中所述，对于普通成员函数来说，其选择可以是：允许通过 `Base` 指针对其进行非虚调用（但当它调用了虚函数时可具有虚行为，比如在 NVI 或者模板方法模式中那样），进行虚调用，或者完全不能调用。NVI 模式是一种避免公开虚函数的技巧。

析构可以仅仅被看做是另一种操作，虽然它带有特殊的语义，并且非虚调用要么很危险要么是错误的。因而，对于基类析构函数来说，其选择有：允许通过 `Base` 指针进行虚函数调用，或者完全不能调用；“非虚调用”是不行的。这样的话，基类析构函数当其可以被调用（即为 `public`）时应当是 `virtual` 的，否则就为非 `virtual`。

注意 NVI 模式并不适用于析构函数，因为构造函数和析构函数无法进行深析构调用。（参见条款 39 和 55。）

推论：当编写基类时，一定要明确编写析构函数，因为隐式生成的析构函数是 public 和非 virtual 的。当预置函数体没问题时你当然可以用 `=default`，而仅仅为其指定正确的可见性和虚函数性质即可。

例外

某些组件体系架构（如 COM 和 CORBA）并不适用标准的删除机制，而是为对象的处置设立了不同的方案。请遵循相应的模式和惯用法，并在适当时采纳本条指导方针。

再考虑一下这种罕见情况：

- `B` 既是一个基类，也是可以被实例化的具体类，因而其析构函数必须为 public 以便 `B` 的对象可以创建和销毁。
- 而 `B` 也没有虚函数，且并不打算按多态方式使用，因此虽然其析构函数是 public 它也不必是 virtual 的。

这样的话，虽然析构函数必须为 public，也会有很强的压力来阻止它变为 virtual，因为作为第一个 virtual 函数，若其所添加的功能永远不会被利用的话，它就会损害所有的运行时类型开销。

在这种罕见情况下，可以是析构函数 public 且非 virtual，但要明确说明其派生类对象绝不能当作 `B` 来多态地使用。我们对 `std::unary_function` 正是这样做的。

不过，一般来说应当避免具体的基类（参见条款 35）。例如，`unary_function` 不过是聚合了一组 `typedef`，它不可能会被有意单独实例化。给它提供 public 的析构函数完全没有任何意义；更好的设计应当是遵循本条款的建议来给它一个 protected 非虚析构函数猜到。

参考: [[SuttAlex05]](#SuttAlex05) Item 50, [[Cargill92]](#Cargill92) pp. 77-79, 207? [[Cline99]](#Cline99) §21.06, 21.12-13? [[Henricson97]](#Henricson97) pp. 110-114? [[Koenig97]](#Koenig97) Chapters 4, 11? [[Meyers97]](#Meyers97) §14? [[Stroustrup00]](#Stroustrup00) §12.4.2? [[Sutter02]](#Sutter02) §27? [[Sutter04]](#Sutter04) §18

讨论: noexcept 的用法

???

讨论: 虚构函数，回收函数和 swap 不允许失败

绝不能允许从虚构函数，资源回收函数（如 `operator delete`），或者 `swap` 函数中用 `throw` 来报告错误。如果这些操作可以失败的话，就几乎不可能编写有用的代码了，而且即便真的发生了某种错误，也几乎不可能有进行重试的任何意义。特别是，C++ 标准库是直截了当地禁止使用可能在析构函数中抛出异常的类型的。现在，大多数析构函数缺省就隐含带有 `noexcept` 了。

示例

```
class Nefarious {
public:
    Nefarious() { /* 可能抛出异常的代码 */ }      // 好
    ~Nefarious() { /* 可能抛出异常的代码 */ }     // 不好，可能抛出异常
    // ...
};
```

1. `Nefarious` 对象很难安全地使用，即便是作为局部变量也是如此：

```
void test(string& s)
{
    Nefarious n;           // 要有麻烦了
    string copy = s;       // 复制 string
} // 先后销毁 copy 和 n
```

这里，对 `s` 的复制可能抛出异常，且当其抛出了异常而 `n` 的析构函数也抛出了异常时，程序就会因调用 `std::terminate` 而退出，因为无法同时传播两个异常。

2. 以 `Nefarious` 为成员或者基类的类同样很难安全地使用，因为它们的析构函数必须调用 `Nefarious` 的析构函数，且同样遭受其糟糕行为的毒害：

```
class Innocent_bystander {
    Nefarious member;      // 噢，毒害了外围类的析构函数
    // ...
};

void test(string& s)
{
    Innocent_bystander i; // 要有更多麻烦了
    string copy2 = s;     // 复制 string
} // 依次销毁 copy 和 i
```

这里，当 `copy2` 的构造中抛出了异常时，我们会遇到同样的问题，因为 `i` 的析构函数现在也会抛出异常，且因此会使我们调用 `std::terminate`。

3. 你也无法可靠地创建全局或静态的 `Nefarious` 对象：

```
static Nefarious n;      // 噢，无法捕获任何析构函数异常
```

4. 你无法可靠地创建 `Nefarious` 的数组：

```
void test()
{
    std::array<Nefarious, 10> arr; // 这行代码会导致 std::terminate()
}
```

当出现可能抛出异常的析构函数时，数组的行为是未定义的，因为根本不可能发明出合理的回退行为。请想象一下：编译器如何才能生成用来构造 `arr` 的代码，如果第四个对象的构造函数抛出了异常，这段代码必须放弃，且在其清理模式中将试图调用已经构造完成的每个对象的析构函数……而这些析构函数中的一个或更多会抛出异常呢？不存在令人满意的答案。

5. 你无法在标准容器中使用 `Nefarious`：

```
std::vector<Nefarious> vec(10); // 这行代码会导致 std::terminate()
```

标准库禁止其所使用的任何析构函数抛出异常。你无法把 `Nefarious` 对象存储到标准容器中，或者在标准库的任何其他组件上使用它们。

注解

它们是绝不能失败的关键函数，因为在事务性编程中需要它们提供两种关键操作：当处理过程中遇到问题时撤回工作，以及当未发生问题时提交工作。如果没有办法可以用无失败操作来安全地撤回的话，就不可能实现无失败的回滚操作。如果没有办法可以用无失败操作（显然 `swap` 可以，但并不仅限于它）来安全地提交状态的改变的话，就不可能实现无失败的提交操作。

请考虑以下在 C++ 标准中所找到的建议和要求：

当在栈回溯过程中所调用的析构函数因为异常而退出时，将调用 `terminate` (15.5.1)。因此析构函数通常应当捕获异常，并防止它们被传播出析构函数。 --[[C++03]](#Cplusplus03) §15.2(3)

C++ 标准库中所定义的任何析构函数（也包括用于实例化标准库模板的任何类型的析构函数）的操作都不会抛出异常。 --[[C++03]](#Cplusplus03) §17.4.4.8(3)

包括专门重载的 `operator delete` 和 `operator delete[]` 在内的回收函数也属于这一类别，因为一般它们也被用在清理过程，尤其是在异常处理过程中，用以对部分完成的工作进行撤回。
除了析构函数和回收函数之外，一般的错误安全性技术也依赖于永不失败的 `swap` 操作——这种情况下，它们不仅用于实现确保成功的回滚操作，也用于实现确保成功的提交操作。例如，以下是对类型 `T` 的一种惯用的 `operator=` 实现，它在复制构造之后，调用了无失败的 `swap`：

```
T& T::operator=(const T& other)
{
    auto temp = other;
    swap(temp);
    return *this;
}
```

(另见条款 56。 ???)

幸运的是，当进行资源释放时，发生故障的范围肯定会比较小。如果使用异常作为错误报告机制的话，请确保这样的函数会处理其内部的处理中可能会产生的所有异常和其他错误。（对于异常，可以直接把你的析构函数中的所有相关部分都包围到一个 `try/catch(...)` 块中。）这点非常重要，因为析构函数可能会在某种紧要关头被调用，比如当无法分配某种系统资源（如内存、文件、锁、端口、窗口，或者其他系统对象）的时候。

当使用异常作为错误处理机制的时候，请始终明示这种行为，将这些函数声明为 `noexcept`。（参见条款 75。）

参考: [[SuttAlex05]](#SuttAlex05) Item 51; [[C++03]](#Cplusplus03) §15.2(3), §17.4.4.8(3)?
[[Meyers96]](#Meyers96) §11? [[Stroustrup00]](#Stroustrup00) §14.4.7, §E.2-4? [[Sutter00]]
([#Sutter00] §8, §16? [[Sutter02]](#Sutter02) §18-19

统一对复制、移动和销毁操作进行定义

理由

???

注解

一旦定义了复制构造函数，就得定义复制赋值运算符。

注解

一旦定义了移动构造函数，就也得定义移动赋值运算符。

示例

```
class X {  
public:  
    X(const X&) { /* stuff */ }  
  
    // 不好：未同时定义复制赋值运算符  
  
    X(X&&) noexcept { /* stuff */ }  
  
    // 不好：未同时定义移动赋值运算符  
  
    // ...  
};  
  
X x1;  
X x2 = x1; // ok  
x2 = x1; // 陷阱：要么不能通过编译，要么会做出不好的事
```

一旦定义了析构函数，就不能再使用编译器所生成的复制或移动操作了；你可能需要定义或者抑制掉移动或复制操作。

```
class X {  
HANDLE hnd;  
// ...  
public:  
    ~X() { /* 自定义的行为，比如关闭 hnd */ }  
    // 可疑：未提到过复制或移动操作--hnd 会怎么样?  
};  
  
X x1;  
X x2 = x1; // 陷阱：要么不能通过编译，要么会做出不好的事  
x2 = x1; // 陷阱：要么不能通过编译，要么会做出不好的事
```

如果定义了复制操作，且有任何基类或成员的该性定义了移动操作的话，应当同样定义移动操作。

```
class X {  
    string s; // 定义了更高效的移动操作  
    // ... 其他数据成员 ...  
public:  
    X(const X&) { /* stuff */ }  
    X& operator=(const X&) { /* stuff */ }  
  
    // 不好：并未一同定义移动构造函数和移动赋值  
    // (为何不把那些自定义的“stuff”重复一下呢？)  
};  
  
X test()  
{  
    X local;
```

```
// ...
return local; // 陷阱：可能会低效甚或产生错误的行为
}
```

一旦定义了复制构造函数，复制赋值运算符，或者析构函数中任何一个，你就可能需要也定义其他的。

注解

如果需要定义这五个函数中的任何一个，这就意味着你需要得到与其预置行为不同的行为——而这五者则是非对称相关的。如下所述：

- 当编写或禁用复制构造函数或复制赋值运算符之一时，很可能需要对另一个同样对待：若其中之一有“特别的”任务，则很可能另一个也应当如此，因为这两个函数应当具有相似的效果。（参见条款 53，其中对这点进行专门的展开说明。）
- 当明确编写复制函数时，很可能需要编写析构函数：若复制构造函数所做的“特别的”任务为分配或复制某中资源（诸如内存、文件、socket等），则需要在析构函数中对其进行回收。
- 当明确编写析构函数时，很可能需要明确编写或禁用复制操作：若不得不编写一个不平凡的析构函数的话，这通常是由于你需要人工释放对象所持有的某个资源。若是如此的话，很可能需要特别小心这些资源的复制，而你就需要关注对象进行复制和赋值的方式，或者完全禁止复制操作。

许多情况下，持有以 RAI 的“拥有者”对象恰当封装了的资源，是能够吧自己编写这些操作的需要消除掉的。（参见条款 13。）

应当优先采用编译器生成（包括 `=default`）的特殊成员；只有它们才被归类为“平凡的”，而且至少有一家主要的标准库供应商针对带有平凡特殊成员的类进行了大量地优化。这可能会成为一种常规实践。

例外：当特殊函数的声明仅为了使其非公开或者为虚函数而没有特殊的语义时，它并不导致需要其他的特殊成员。

少数情况下，带有奇怪类型的成员（诸如引用成员）的类也是例外，因为它们的复制语义很古怪。

在持有引用的类中，你可能需要编写复制构造函数和赋值运算符，但预置的析构函数仍能够做出正确的处理。（需要注意，基本上使用引用成员几乎总是错误的。）

参考： [[SuttAlex05]](#SuttAlex05) Item 52; [[Cline99]](#Cline99) §30.01-14? [[Koenig97]](#Koenig97) §4? [[Stroustrup00]](#Stroustrup00) §5.5, §10.4? [[SuttHysl04b]](#SuttHysl04b)

资源管理规则概览：

- 提供强资源安全性；亦即，绝不让你认为是资源的任何东西发生泄漏
- 绝不在持有未被句柄所拥有的资源时返回或抛出异常
- “原生”的指针或引用不可能是资源句柄
- 绝不让指针的生存期超过其所指向的对象
- 用模板来表现容器（和其他资源句柄）
- 按值返回容器（依靠移动或复制消除来获得性能）
- 若类为资源句柄，则它需要构造函数，析构函数，复制以及移动操作
- 若类为容器，则应为其提供一个初始化式列表构造函数

讨论：提供强资源安全性；亦即，绝不让你认为是资源的任何东西发生泄漏

理由

避免泄漏。泄漏会导致性能损耗，发生神秘的错误，系统崩溃，以及安全性的违犯。

其他形式: 使所有资源都表示为某种可以自我管理生存期的类的对象。

示例

```
template<class T>
class Vector {
private:
    T* elem; // 自由存储中的 sz 个元素，由类对象所拥有
    int sz;
    // ...
};
```

这个类是一个资源句柄。它管理各个 `T` 对象的生存期。为此，`Vector` 必然要对[一组特殊操作](#)（几个构造函数，析构函数，等等）进行定义或弃置。

示例

```
??? “奇异的”非内存资源 ???
```

强制实施

防止泄漏的基本技巧是让所有的资源都被某种带有回档析构函数的资源句柄所拥有。检查工具能够查找出“裸 `new`”。给定一组 C 风格的分配函数（如 `fopen()`），检查工具也能够查找出未被资源句柄管理的使用点。一般来说，可以带着怀疑看待“裸指针”，对其进行标记和分析。如果没有人为输入的话，时无法产生资源的完整列表的（“资源”的定义有些过于宽泛），不过可以用一个资源列表来对工具进行“参数化”。

讨论：绝不在持有未被句柄所拥有的资源时返回或抛出异常

理由

这会导致泄漏。

示例

```
void f(int i)
{
    FILE* f = fopen("a file", "r");
    ifstream is { "another file" };
    // ...
    if (i == 0) return;
    // ...
    fclose(f);
}
```

当 `i == 0` 时 `a file` 的文件句柄就会泄漏。另一方面，`another file` 的 `ifstream` 则将会（在销毁时）正确关闭它的文件。如果你必须显式使用指针而不是带有特定语义的资源句柄的话，可以使用带有自定义删除器的 `unique_ptr` 或 `shared_ptr`：

```
void f(int i)
{
    unique_ptr<FILE, int(*)(FILE*)> f(fopen("a file", "r"), fclose);
    // ...
    if (i == 0) return;
    // ...
}
```

这样更好：

```
void f(int i)
{
    ifstream input {"a file"};
    // ...
    if (i == 0) return;
    // ...
}
```

强制实施

检查器必须将任何“裸指针”当作可疑处理。

检查器可能必须依赖于人工提供的资源列表进行工作。

上手时，我们知道标准库容器，`string`，以及智能指针。

`span` 和 `string_view` 的使用能够提供巨大的帮助（它们并非资源句柄）。

讨论：“原生”的指针或引用不可能是资源句柄

理由

使得能够区分所有者和视图。

注解

这和你如何“拼写”指针是两回事：`T*`，`T&`，`Ptr<T>` 和 `Range<T>` 都不是所有者。

讨论：绝不让指针的生存期超过其所指向的对象

理由

避免极难找到的错误。这种指针的解引用时未定义行为，能够导致发生对类型系统的违犯。

示例

```
string* bad()    // 确实很坏
{
    vector<string> v = { "This", "will", "cause", "trouble", "!" };
    // 导致指向已经销毁的对象 (v) 的已经销毁的成员的一个指针被泄漏出去
    return &v[0];
}

void use()
{
    string* p = bad();
    vector<int> xx = {7, 8, 9};
    // 未定义行为: x 可能不是字符串 "This"
```

```
    string x = *p;
    // 未定义行为：我们不知道在位置 p 上分配的到底是什么（如果有的话）
    *p = "Evil!";
}
```

v 中的各个 string 都在 bad() 退出之时被销毁了，v 自身也是如此。其所返回的指针指向自由存储上的未分配内存。（由 p 所指向的）这块内存，在执行 *p 之时可能已经被重新分配了。此时很可能并不存在可以读取的 string 对象，而通过 p 进行写入则会轻易损坏某些无关类型的对象。

强制实施

大多数编译器已经能对简单情况进行警告，而且它们带有可以更进一步的信息。将函数所返回的任何指针都当作是可疑的。用容器、资源句柄和视图（例如 span，它不是资源句柄）来减少需要检查的情形。上手时，可将带有析构函数的类都当作是资源句柄处理。

讨论：用模板来表现容器（和其他资源句柄）

理由

提供静态类型安全的元素操作。

示例

```
template<typename T> class Vector {
    // ...
    T* elem;    // 指向 sz 个 T 类型的元素
    int sz;
};
```

讨论：按值返回容器（依靠移动或复制消除来获得性能）

理由

简化代码并消除一种进行显式内存管理的需要。将对象递交给外围作用域，由此扩展其生存期。

参见：[F.20, 有关“输出（Out）”值的一般条款](#)

示例

```
vector<int> get_large_vector()
{
    return ...;
}

auto v = get_large_vector(); // 按值返回没有问题，大多数现代编译器都会进行复制消除
```

例外

见 [F.20](#) 中的例外。

强制实施

检查函数所返回额指针和引用，看看它们是否被赋值给资源句柄（如 unique_ptr）。

讨论：若类为资源句柄，则它需要构造函数，析构函数，复制以及移动操作

理由

以提供对资源的生存期的完全控制。以提供一组协调的对资源的操作。

示例

```
??? 折腾指针
```

注解

若所有的成员都为资源句柄，则尽可能要依赖预置的特殊操作。

```
template<typename T> struct Named {
    string name;
    T value;
};
```

现在 `Named` 带有一个默认构造函数，一个析构函数，以及高效的复制和移动操作，只要 `T` 也提供了它们。

强制实施

一般来说，工具是无法知道类是否是资源句柄的。不过，如果类带有某种[默认操作](#)的话，它就得拥有全部，而如果类中有成员为资源句柄的话，它也应被当做是资源句柄。

讨论：若类为容器，则应为其提供一个初始化式列表构造函数

理由

提供一组初始元素是一种常见情形。

示例

```
template<typename T> class Vector {
public:
    Vector(std::initializer_list<T>);
    // ...
};

Vector<string> vs { "Nygaard", "Ritchie" };
```

强制实施

类怎么算作是容器呢？ ???

附录 D: 支持工具

这个部分列出了直接支持采用 C++ 核心指导方针的一些工具。这个列表并非要穷尽那些有助于编写良好的 C++ 代码的工具。

如果一个工具被专门设计以支持并关联到 C++ 核心指导方针，那它就是包括进来的候选者。

工具: [Clang-tidy](#)

Clang-tidy 有一组专门用于强制实施 C++ 核心指导方针的规则。这些规则的命名模式为 `cppcoreguidelines-*`。

工具: [CppCoreCheck](#)

微软编译器的 C++ 代码分析中包含一组专门用于强制实施 C++ 核心指导方针的规则。

词汇表

这是在指导方针中用到的一些术语的相对非正式的定义

(基于 [Programming: Principles and Practice using C++](#) 中的词汇表)。

有关 C++ 的许多主题的更多信息，可以在[标准 C++ 基金会的网站](#) 找到。

- **ABI:** 应用二进制接口，对于特定硬件平台与操作系统的组合的一种规范。与 API 相对。
- **抽象类 (abstract class)** : 不能直接用于创建对象的类；通常用于为派生类定义接口。
当类带有纯虚函数或只有受保护的构造函数时，它就是抽象的。
- **抽象 (abstraction)** : 对事物的描述，有选择并有意忽略（隐藏）了细节（如实现细节）；选择性忽略。
- **地址 (address)** : 用以在计算机的内存中找到某个对象的值。
- **算法 (algorithm)** : 用以解决某个问题的过程或公式；有限的一系列计算步骤以产生一个结果。
- **别名 (alias)** : 指代某个对象的替代方式；通常为名字，指针，或者引用。
- **API:** 应用编程接口，一组构成不同软件之间之间的交互的函数。与 ABI 相对。
- **应用程序 (application)** : 程序或程序的集合，用户将其看作一个实体。
- **近似 (approximation)** : 事物（比如值或者设计），接近于完美的或者理想的（值或设计）。
通常近似都是在理想情形中进行各种权衡的结果。
- **参数/实参 (argument)** : 传递给函数或模板的值，其中以形参来进行访问。
- **数组 (array)** : 同质元素序列，通常是数值，例如 `[0:max]`。
- **断言 (assertion)** : 插入到程序中的语句，以声称（断言）在程序的这个位置某事物必定为真。
- **基类 (base class)** : 目的是为从其进行派生的类型（比如它带有非 `final` 的虚函数），且有意仅间接使用该类型的对象（如通过指针）。[严格地说，“基类”可被定义为“从之进行派生的类型”，但我们这里以类设计者意图的角度来给出定义] 通常基类带有一个或更多的虚函数。
- **位 (bit)** : 计算机中信息的基本单位。一个位的值可以为 0 或 1。
- **bug:** 程序中的错误。
- **字节 (byte)** : 大多数计算机中进行寻址的基本单位。通常一个字节有 8 位。
- **类 (class)** : 一种用户定义的类型，可以包含数据成员，函数成员，以及成员类型。
- **代码 (code)** : 程序或程序的部分；有歧义地同时用于源代码和目标代码。
- **编译器 (compiler)** : 一种将源代码变为目标代码的程序。
- **复杂度 (complexity)** : 对某个问题构造解决方案的难度，或解决方案自身的一种难于精确定义的记法或度量。
有时候复杂度只是（简单地）表示对执行某个算法所需操作的数量的估计。
- **计算 (computation)** : 执行一些代码，通常接受一些输入并产生一些输出。
- **概念 (concept)** : (1) 提法，想法；(2) 一组要求，通常针对模板参数。
- **具体类 (concrete type)** : 并非基类的类型，且有意直接使用该类型的对象（而非仅通过指针/间接），其大小是已知的，通常可以按程序员的意图在任何地方分配（比如静态地在运行栈上分配）。
- **常量 (constant)** : (在给定作用域中) 不能改变的值；不可变。
- **构造函数 (constructor)** : 初始化（“构造”）一个对象的操作。
通常构造函数会建立起不变式，并且通常会获取对象被使用时所需的资源（并通常将由析构函数所

释放)。

- 容器 (*container*) : 持有一些元素 (其他对象) 的对象。
- 复制 (*copy*) : 制造两个对象使其值比较为相等的操作。另见移动。
- 正确性 (*correctness*) : 如果程序或程序片段符合其说明, 则其为正确的。
不幸的是, 说明可能不完整或不一致, 或者也可能无法满足用户的合理预期。
因此为了产生可接受的代码, 我们有时候比仅仅遵守形式说明要做更多的事。
- 成本 (*cost*) : 产生一个程序, 或者执行它的耗费 (如开发时间, 执行时间或空间等)。
理想情况下, 成本应当是复杂度的函数。
- 定制点 (*customization point*) :
- 数据 (*data*) : 计算中所用到的值。
- 调试 (*debugging*) : 寻找并移除程序中的错误的行为; 通常远没有测试那样系统化。
- 声明式 (*declaration*) : 程序中对一个名字及其类型的说明。
- 定义式 (*definition*) : 实体的声明式, 提供了程序使用该实体所需的所有信息。
简化版定义: 分配了内存额声明。
- 派生类 (*derived class*) : 派生自一个或多个基类的类。
- 设计 (*design*) : 对软件的某个片段应当如何运作以满足其说明的一个总体描述。
- 析构函数 (*destructor*) : 当对象销毁 (如在作用域结束时) 被隐式执行 (调用) 的操作。它通常进行资源的释放。
- 封装 (*encapsulation*) : 将某些事物 (如实现细节) 保护为私有的, 不接受未授权的访问。
- 错误 (*error*) : 对程序行为的合理期望 (通常表现为某种需求或者一份用户指南) 和程序的实际行为之间的不一致。
- 可执行程序 (*executable*) : 预备在计算机上运行 (执行) 的程序。
- 功能蔓延 (*feature creep*) : 为“预防万一”而向程序添加过量的功能的倾向。
- 文件 (*file*) : 计算机中的持久信息的容器。
- 浮点数 (*floating-point number*) : 计算机对实数 (如 7.93 和 10.78e-3) 的近似。
- 函数 (*function*) : 命名的代码单元, 可以从程序的不同部分执行 (调用); 计算的逻辑单元。
- 泛型编程 (*generic programming*) : 关注于算法的设计和高效实现的一种编程风格。
泛型算法能够对所有符合其要求的参数类型正确工作。在 C++ 中, 泛型编程通常使用模板进行。
- 全局变量 (*global variable*) : 技术上说, 命名空间作用域中的具名对象。
- 句柄 (*handle*) : 一个类, 允许通过一个成员指针或引用来访问另一个对象。另见资源, 复制, 移动。
- 头文件 (*header*) : 包含用于在程序的各个部分中共享接口的声明的文件。
- 隐藏 (*hiding*) : 防止一个信息片段被直接看到或访问的行为。
例如, 嵌套 (内部) 作用域中的名字会防止外部 (外围) 作用域中相同的名字被直接使用。
- 理想的 (*ideal*) : 我们力争达成的事物的完美版本。我们经常不得不进行各种权衡最后获得一个近似。
- 实现 (*implementation*) : (1) 编写代码并测试的活动; (2) 用以实现一个程序的代码。
- 无限循环 (*infinite loop*) : 终止条件永不为真的循环。参见重复。
- 无限递归 (*infinite recursion*) : 无法终止的递归, 直到机器耗尽内存无法维持其调用。
在现实中这种递归不可能是无限的, 它会因某种硬件错误而终止。
- 信息隐藏 (*information hiding*) : 分离接口和实现, 以此将用户不感兴趣的实现细节隐藏起来, 并提供一种抽象的活动。
- 初始化 (*initialize*) : 为一个对象给定其第一个 (初始) 值。
- 输入 (*input*) : 计算中所使用的值 (比如函数参数以及通过键盘所输入的字符)。
- 整数 (*integer*) : 整数, 比如 42 和 -99。
- 接口 (*interface*) : 一个或一组声明, 说明了一个代码片段 (比如函数或者类) 应当如何进行调用。
- 不变式 (*invariant*) : 程序中的某些点必然总为真的事物; 通常用于描述对象的状态 (值的集合), 或者循环进入其重复的语句之前的状态。
- 重复 (*iteration*) : 重复执行代码片段的行为; 参见递归。

- **迭代器 (iterator)** : 用以标识序列中的一个元素的对象。
- **ISO**: 国际标准化组织。C++ 语言是一项 ISO 标准: ISO/IEC 14882。更多信息请参考 [iso.org](#)。
- **程序库 (library)** : 类型、函数、类等等的集合，它们实现了一组设施（抽象），预备可能被用作不止一个程序的组成部分。
- **生存期 (lifetime)** : 从对象的初始化直到它变为不可用（离开作用域，被删除，或程序终止）的时间。
- **连接器 (linker)** : 用以将目标代码文件和程序库合并构成一个可执行程序的程序。
- **字面量 (literal)** : 直接指定一个值的写法，比如 12 指定的是整数值“十二”。
- **循环 (loop)** : 重复执行的代码片段；在 C++ 中，通常是 `for` 语句或者 `while` 语句。
- **移动 (move)** : 将值从一个对象转移到另一个对象，并遗留一个表示“空”的值的操作。另见复制。
- **仅可移动类型 (move-only type)** : 可以移动但不能复制的具体类型。
- **可变的 (mutable)** : 可以改动；不可变、常量和不变量的反义词。
- **对象 (object)** : (1) 已经初始化的一块具有已知类型的内存区域，持有该类型的一个值；(2) 一块内存区域。
- **目标代码 (object code)** : 编译器的输出，预备作为连接器的输入（连接器以其产生可执行代码）。
- **目标文件 (object file)** : 包含目标代码的文件。
- **面向对象编程 (object-oriented programming)** : (OOP) 一种关注类和类层次的设计和使用的编程风格。
- **操作 (operation)** : 能够实施某种活动的事物，比如函数或运算符。
- **输出 (output)** : 由计算所产生的值（例如函数的结果，或者在屏幕上写下的一行行字符等）。
- **溢出 (overflow)** : 产生无法被其预期目标所存储的值。
- **重载 (overload)** : 定义两个函数或运算符，使其具有相同名字但不同的参数（操作数）类型。
- **覆盖 (override)** : 在派生类中用声明和基类中的某个虚函数具有相同名字和参数类型的函数，以此使该函数可以通过由基类所定义的接口来进行调用。
- **所有者 (owner)** : 负责释放某个资源的对象。
- **范式 (paradigm)** : 设计和编程风格的一种多少有些做作的术语；通常会被用于（错误地）暗示有一种范式被其他的都更优秀。
- **形参 (parameter)** : 对函数或模板的一个明确输入的声明。当进行调用时，函数可以通过其形参的名字来访问向其所传递的各个实参。
- **指针 (pointer)** : (1) 值，用于标识内存中的一个有类型的对象；(2) 持有这种值的变量。
- **后条件 (post-condition)** : 当从一个代码片段（如函数或者循环）退出时必须满足的条件。
- **前条件 (pre-condition)** : 当进入一个代码片段（如函数或者循环）时必须满足的条件。
- **程序 (program)** : 足够完整以便能够在计算机上执行的代码（可能带有关联的数据）。
- **编程 (programming)** : 将问题的解决方案表现为代码的工艺。
- **编程语言 (programming language)** : 用于表达程序的语言。
- **伪代码 (pseudo code)** : 以非正式的写法而非编程语言所编写的对计算的一种描述。
- **纯虚函数 (pure virtual function)** : 必须在派生类中予以覆盖的虚函数。
- **RAII**: (“资源获取即初始化，Resource Acquisition Is Initialization”) 一种基于作用域进行资源管理的基本技术。
- **范围 (range)** : 值的序列，可以以一个开始点和一个结尾点进行描述。例如，`[0:5]` 的意思是值 0, 1, 2, 3, 和 4。
- **递归 (recursion)** : 函数调用其自身的行为；另见重复。
- **引用 (reference)** : (1) 一种值，描述内存中具有类型的值的位置；(2) 持有这种值的变量。
- **正则表达式 (regular expression)** : 对字符串的模式的一种表示法。
- **正规**: 可以进行相等性比较的半正规类型（参见 `std::regular` 概念）。进行复制之后，副本对象与原对象比较为相等。正规类型的行为与如 `int` 这样的内建类型相似，且可以用 `==` 进行比较。特别是，正规类型的对象可以进行复制，且复制的结果是与原对象比较为相等的一个独立对象。另见半正规类型。

- **要求 (requirement)** : (1) 对程序或程序的一部分的预期行为的描述；(2) 对函数或模板对其参数所作出的假设的描述。
- **资源 (resource)** : 获取而得的并随后必须被释放的事物，比如文件句柄，锁，或者内存。另见句柄，所有者。
- **舍入 (rounding)** : 将一个值转换为某个较不精确类型的数学上最接近的值。
- **RTTI**: 运行时类型信息 (Run-Time Type Information) 。 ???
- **作用域 (scope)** : 程序文本 (源代码) 的区域，在其中可以对一个名字进行涉指。
- **半正规 (semiregular)** : 可复制的（也包括可移动的）且可默认构造的具体类型（参见 `std::semiregular` 概念）。复制的结果是一个与原对象具有相同的值的独立类型。半正规类型的行为与像 `int` 这样内建类型大致相似，但可能没有 `==` 运算符。另见 **正规类型**。
- **序列 (sequence)** : 可以以线性的顺序访问的一组元素。
- **软件 (software)** : 代码片段及其关联数据的集合；通常可以和程序互换运用。
- **源代码 (source code)** : 由程序员所生产的代码，（原则上）可以被其他程序员阅读。
- **源文件 (source file)** : 包含源代码的文件。
- **规范 (specification)** : 对代码片段应当做什么的描述。
- **标准 (standard)** : 由官方承认的对某事物的定义，比如编程语言。
- **状态 (state)** : 一组值。
- **STL**: 标准库中的容器，迭代器，以及算法部分。
- **字符串 (string)** : 字符的序列。
- **风格 (style)** : 旨在统一语言功能特征的使用的一组编程技巧；有时候以非常限定的方式来仅代表诸如命名和代码展现等的低层次规则。
- **子类型 (subtype)** : 派生类型；一个类型具有另一个类型的所有（可能更多）的性质。
- **超类型 (supertype)** : 基类型；一个类型具有另一个类型的性质的子集。
- **系统 (system)** : (1) 用以在计算机上实施某种任务的一个或一组程序；(2) 对“操作系统”的简称，即计算机的基本执行环境及工具。
- **TS**: [技术规范](#)。技术规范所处理的是仍处于技术开发之中的工作，或者是认为这项工作以后可能会被同意采纳为国际标准，但并不会立即处理。技术规范的出版是为了其立即可用，也是为了提供一种获得反馈的方法。其目标是最终能够被转化并重新作为国际标准来出版。
- **模板 (template)** : 由一个或多个的类型或（编译时）值进行参数化的类或函数；支持泛型编程的基本 C++ 语言构造。
- **测试 (testing)** : 系统化地查找程序中的错误。
- **权衡 (trade-off)** : 对多个设计和实现准则进行平衡的结果。
- **截断 (truncation)** : 从一个类型转换为另一个无法精确表示被转换的值的类型时发生的信息损失。
- **类型 (type)** : 为一个对象定义了一组可能的值和一组操作的事物。
- **未初始化的 (uninitialized)** : 对象在初始化之前的（未定义的）状态。
- **单元 (unit)** : (1) 为值赋予含义的一种标准度量（例如，距离单位 km）；(2) 较大的整体中的一个可区分的（比如命名的）部分。
- **用例 (use case)** : 程序的某个特定（通常简化的）使用，以测试其功能并演示其目的。
- **值 (value)** : 根据某个类型所解释的一组内存中的位。
- **值类型 (value type)** : 一些人用这个术语来表示正规或半正规类型。
- **变量 (variable)** : 给定类型的具名对象；除非未初始化否则包含一个值。
- **虚函数 (virtual function)** : 可在派生类中进行覆盖的成员函数。
- **字 (word)** : 计算机中内存的基本单元，通常是用以持有一个整数的单元。

To-do: 未分类的规则原型

这是我们的未完成列表。

以下各条目最终将成为规则或者规则的一部分。

或者，我们也会决定不需要做出改动并将条目移除。

- 禁止远距离友元关系
- 应不应该处理物理设计（文件里有什么）和大规模设计（程序库，程序库的组合）？
- 命名空间
- 避免在全局作用域中使用 using 指令（但允许如 std 或其他的“基础”命名空间（如 experimental））
- 命名空间应当有什么粒度？是（如 Sutter/Alexandrescu 所定义的）所有被设计为一同工作或者一同发布的类和函数，还是应该更窄或是更宽？
- 应该用内联命名空间吗（比如 `std::literals::*_literals`）？
- 避免隐式转换
- Const 成员函数应当是线程安全的.....aka, 但我并不想真的改掉变量，只是在第一次调用它的时候向它赋一个值.....argh
- 始终初始化变量，为成员变量使用初始化列表。
- 无论谁编写了接受或返回 `void*` 的公开接口，都应该上火刑。我曾经好多年都以它作为自己的个人喜好来着。 :)
- 尽可能应用 `const`：成员函数，变量，以及 `const_iterators`
- 使用 `auto`
- `(size)` vs. `{initializers}` vs. `{Extent{size}}`
- 不要过度抽象
- 不要沿着调用栈向下传递指针
- 通过函数底部退出
- 应当提供在多态之间进行选择的指导方针吗？是的。经典的（虚函数，引用语义） vs. Sean Parent 风格（值语义，类型擦除，类似 `std::function`） vs. CRTP/静态的？也许还需要 vs. 标签派发？
- 我们的指导方针是否应当在构造函数或析构函数中禁止进行虚函数调用？是的。许多人都禁止了，虽然我觉得这是 C++ 的一大优势 ??? -保留意见（D 走向 Java 之路太让我失望了）。有好的例子吗？
- 在 lambda 方面，在算法调用和其他回调场景中什么因素会影响决定使用 lambda 还是（局部？）类？
- 讨论一下 `std::bind`，Stephen T. Lavavej 对它有太多批评，使我开始觉得它是不是真的会在未来消失掉。应该建议以 lambda 代替它吗？
- 怎么处理泄漏的临时变量？：`p = (s1 + s2).c_str();`
- 指针和迭代器的失效会导致悬挂指针：

```

void bad()
{
    int* p = new int[700];
    int* q = &p[7];
    delete p;

    vector<int> v(700);
    int* q2 = &v[7];
    v.resize(900);

    // ... 使用 q 和 q2 ...
}

```

- LSP
- 私有继承 vs/and 成员
- 避免静态类成员变量（竞争条件，几乎就是全局变量）
- 使用 RAII 锁定保护 (`lock_guard`, `unique_lock`, `shared_lock`)，绝不直接调用 `mutex.lock` 和 `mutex.unlock` (RAII)
- 优先使用非递归锁（它们通常用作不良情况的变通手段，有开销）
- 联结 (join) 你的每个线程！（因为如果没被联结或脱离 (detach) 的话，析构函数会调用 `std::terminate`.....有什么好理由来脱离线程吗？）-- ??? 支持库该不该为 `std::thread` 提供一个 RAII 包装呢？
- 当必须同时获取两个或更多的互斥体时，应当使用 `std::lock` (或者别的死锁免除算法？)
- 当使用 `condition_variable` 时，始终用一个互斥体来保护它（在互斥体外面设置原子 `bool` 的值的做法是错误的！），并对条件变量自身使用同一个互斥体。
- 绝不对 `std::atomic<user-defined-struct>` 使用 `atomic_compare_exchange_strong` (填充位中的区别会造成影响，而在循环中使用 `compare_exchange_weak` 则能够归于稳定的填充位)
- 单独的 `shared_future` 对象不是线程安全的：两个线程不能等待同一个 `shared_future` 对象（它们可以等待指代相同共享状态的 `shared_future` 的副本）
- 单独的 `shared_ptr` 对象不是线程安全的：不同的线程可以调用指代相同共享对象的不同 `shared_ptr` 的非 `const` 成员函数，但当一个线程访问一个 `shared_ptr` 对象时，另一个线程不能调用相同 `shared_ptr` 对象的非 `const` 成员函数（如果确实需要，考虑代之以 `atomic_shared_ptr`）
- 算术相关规则

参考文献

- [Abrahams01]: D. Abrahams. [Exception-Safety in Generic Components](#).
- [Alexandrescu01]: A. Alexandrescu. Modern C++ Design (Addison-Wesley, 2001).
- [C++03]: ISO/IEC 14882:2003(E), Programming Languages — C++ (updated ISO and ANSI C++ Standard including the contents of (C++98) plus errata corrections).
- [Cargill92]: T. Cargill. C++ Programming Style (Addison-Wesley, 1992).

- [Cline99]: M. Cline, G. Lomow, and M. Girou. C++ FAQs (2nd Edition) (Addison-Wesley, 1999).
- [Dewhurst03]: S. Dewhurst. C++ Gotchas (Addison-Wesley, 2003).
- [Henricson97]: M. Henricson and E. Nyquist. Industrial Strength C++ (Prentice Hall, 1997).
- [Koenig97]: A. Koenig and B. Moo. Ruminations on C++ (Addison-Wesley, 1997).
- [Lakos96]: J. Lakos. Large-Scale C++ Software Design (Addison-Wesley, 1996).
- [Meyers96]: S. Meyers. More Effective C++ (Addison-Wesley, 1996).
- [Meyers97]: S. Meyers. Effective C++ (2nd Edition) (Addison-Wesley, 1997).
- [Meyers01]: S. Meyers. Effective STL (Addison-Wesley, 2001).
- [Meyers05]: S. Meyers. Effective C++ (3rd Edition) (Addison-Wesley, 2005).
- [Meyers15]: S. Meyers. Effective Modern C++ (O'Reilly, 2015).
- [Murray93]: R. Murray. C++ Strategies and Tactics (Addison-Wesley, 1993).
- [Stroustrup94]: B. Stroustrup. The Design and Evolution of C++ (Addison-Wesley, 1994).
- [Stroustrup00]: B. Stroustrup. The C++ Programming Language (Special 3rd Edition) (Addison-Wesley, 2000).
- [Stroustrup05]: B. Stroustrup. [A rationale for semantically enhanced library languages](#).
- [Stroustrup13]: B. Stroustrup. [The C++ Programming Language \(4th Edition\)](#). Addison Wesley 2013.
- [Stroustrup14]: B. Stroustrup. [A Tour of C++](#). Addison Wesley 2014.
-
- [Stroustrup15]: B. Stroustrup, Herb Sutter, and G. Dos Reis: [A brief introduction to C++'s model for type- and resource-safety](#).
- [SuttHysl04b]: H. Sutter and J. Hyslop. [Collecting Shared Objects](#) (C/C++ Users Journal, 22(8), August 2004).
- [SuttAlex05]: H. Sutter and A. Alexandrescu. C++ Coding Standards. Addison-Wesley 2005.
- [Sutter00]: H. Sutter. Exceptional C++ (Addison-Wesley, 2000).
- [Sutter02]: H. Sutter. More Exceptional C++ (Addison-Wesley, 2002).
- [Sutter04]: H. Sutter. Exceptional C++ Style (Addison-Wesley, 2004).

- [Taligent94]: Taligent's Guide to Designing Programs (Addison-Wesley, 1994).