* 在异常情况下表现良好的程序之所以如此，是因为它们被设计成这样，而不是偶然发生的。异常安全的程序不是偶然创建的。如果一个程序在设计时没有考虑异常，那么它在异常情况下的表现良好的几率，与一个在设计时没有考虑多线程的程序在多线程环境下的表现良好的几率差不多：几乎为零。
* **设计难度增加**：
  + 编写行为符合预期的构造函数和析构函数变得更加困难。
  + 必须确保程序在异常发生时不会突然终止。
* 异常安全程序必须从一开始就被设计为能够处理异常，而不是事后修补。
* 文本

  描述已自动生成

**如何应对不可避免的异常？**

虽然无法避免所有异常，但可以通过以下方式减少异常的影响：

* **编写异常安全的代码**：
  + 使用 RAII 和智能指针管理资源，确保资源在异常发生时也能被正确释放。
* **捕获并处理异常**：
  + 在可能抛出异常的代码周围使用 try-catch 块，捕获并处理异常。
* **提供清晰的错误信息**：
  + 在捕获异常时，记录错误信息以便调试和修复。
* **设计健壮的程序**：
  + 考虑边界条件和异常情况，确保程序在异常发生时仍能保持一致性。
* **测试和验证**：
  + 通过单元测试、集成测试和压力测试，发现并修复潜在的异常问题

第9条：使用析构函数来防止资源泄漏。

通过遵守资源应该封装在对象内部的规则，通常可以避免出现异常时的资源泄漏。但是，如果在获取资源的过程中抛出异常，例如，当你在一个资源获取类的构造函数中时？如果在自动销毁此类资源的过程中引发异常，会发生什么情况？构造函数和析构函数不需要特殊的技术吗？

第10条：防止构造函数中的资源泄漏。

文本

描述已自动生成

必须设计构造函数，使它们能够在自己之后进行清理。

在构造过程中处理异常的可能性可能很棘手，但是auto\_ptr（和类似auto\_ptr的类）可以消除大部分的麻烦。

第11条：防止异常离开析构函数。

第一种是对象在“正常”条件下被销毁，例如当它超出作用域或被显式删除时。第二种是对象在异常传播的栈展开（stack-unwinding）过程中被异常处理机制销毁。

**1. 什么是调用栈（Call Stack）？**

调用栈是程序运行时用于管理函数调用和返回的一种数据结构。每当一个函数被调用时，它的局部变量、参数和返回地址等信息会被压入调用栈；当函数返回时，这些信息会从栈中弹出

**2. 什么是栈展开（Stack Unwinding）？**

当异常被抛出时，C++ 的异常处理机制会从当前函数开始，沿着调用栈向上查找匹配的 catch 块。在这个过程中，**栈展开**会发生，即：

* 从抛出异常的函数开始，逐层退出调用栈中的函数。
* 在退出每一层函数时，**销毁该函数中的局部对象**（调用它们的析构函数）

文本

描述已自动生成

因此，我们有两个很好的理由来阻止异常从析构函数中传播出来。首先，它防止在异常传播的堆栈展开部分调用终止。其次，它有助于确保析构函数总是完成它们应该完成的所有事情。

第12条：理解抛出异常与传递参数或调用虚函数的区别。

当你调用一个函数时，控制权最终会返回到调用位置（除非函数未能返回），但当你抛出异常时，控制权不会返回到抛出位置。

1. **函数参数传递 vs 异常传递**：
   * 在函数调用中，参数可以通过值、引用或指针传递。如果通过引用传递（如 operator>> 中的 w），则不会发生复制操作，直接操作原始对象。
   * 在异常抛出时，无论异常是通过值还是引用捕获，抛出的对象都会被复制。这是因为抛出异常后，控制权不会返回到抛出点，原始对象会在离开作用域时被销毁。如果不复制，catch 子句将接收到一个已经被销毁的对象，这是无效的。
2. 异常不能通过指针捕获，因为指针捕获会导致类型不匹配。异常对象是通过值或引用传递的副本，而不是原始对象的指针。

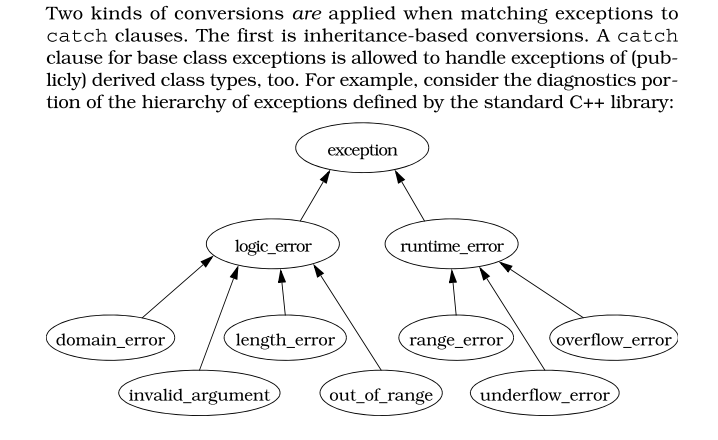


意味着即使异常是通过引用捕获的，catch块也不可能修改localWidget;它只能修改localWidget的副本。

文本

低可信度描述已自动生成

异常应该是罕见的情况



当你调用虚函数时，调用的函数是与调用对象的动态类型最接近的类中的函数。可以说，虚函数采用“最佳匹配”算法，而异常处理遵循“首次匹配”策略。如果派生类的 catch 子句位于基类的 catch 子句之后，编译器可能会发出警告（有些编译器甚至会报错，因为这样的代码在C++中曾经是非法的），但最好的做法是预先避免这种情况：永远不要将基类的 catch 子句放在派生类的 catch 子句之前。

第13项：通过引用捕获异常。

多麽令人愉快的事件的汇合！如果你通过引用捕获，你就避开了关于对象删除的问题，如果你这样做，如果你不这样做，你就会被诅咒;你避免了切片异常对象;你保留了捕获标准异常的能力;你限制了异常对象需要被复制的次数。那你还在等什么？通过引用捕获异常！

条款14：明智地使用异常规范。

unexpected 的默认行为是调用 terminate，而 terminate 的默认行为是调用 abort，因此违反异常规范的程序的默认行为是停止运行。活动堆栈帧中的局部变量不会被销毁，因为 abort 会关闭程序执行而不执行此类清理。

保持对异常规范的平衡是很重要的。它们提供了关于函数可能引发的异常类型的优秀文档，并且对于违反异常规范的情况非常严重，以至于需要立即终止程序的情况，它们默认提供了这种行为。同时，编译器只对它们进行部分检查，很容易在无意中违反。此外，它们还可以防止高级异常处理程序处理意外异常，即使它们知道如何处理。在这种情况下，异常规范是一个可以明智地应用的工具。在将它们添加到函数中之前，请考虑它们赋予软件的行为是否确实是您想要的行为。

项目15：了解异常处理的成本。

若要将例外状况相关的成本降到最低，请在可行的情况下，在不支援例外状况的情况下进行编译;将try区块和例外状况规格的使用限制在您确实需要的位置;并且只在真正例外的情况下掷回例外状况。