

附录 A 编码检查表

本附录给出的问题列表，总结了本书的所有观点。使用本表的最好办法是花两周时间评审一下你的设计和编码实现。先花几分钟时间看一看列表，一旦熟悉了这些问题，就可以灵活自如地按它写代码了。此时，就可以把表放在一边了。

一般问题

- 你是否为程序建立了 DEBUG 版本？
- 你是否将发现的错误及时改正了？
- 你是否坚持彻底测试代码，即使耽误了进度也在所不惜？
- 你是否依靠测试组为你测试代码？
- 你是否知道编码的优先顺序？
- 你的编译程序是否有可选的各种警告？

关于将更改归并到主程序

- 你是否将编译程序的警告（包括可选的）都处理了？
- 你的代码是否未用 Lint
- 你的代码进行了单元测试吗？
- 你是否逐步通过了每一条编码路径以观察数据流？
- 你是否逐步通过了汇编语言层次上的所有关键代码？
- 是否清理过了任何代码？如果是，修改处经过彻底测试了吗？
- 文档是否指出了使用你的代码有危险之处？
- 程序维护人员是否能够理解你的代码？

每当实现了一个函数或子系统之时

- 是否用断言证实了函数参数的有效性？
- 代码中是否有未定义的或者无意义的代码？
- 代码能否创建未定义的数据？
- 有没有难以理解的断言？对它们作解释了没有？
- 你在代码中是否作过任何假设？
- 是否使用断言警告可能出现的非常情况？
- 是否作过防御性程序设计？代码是否隐藏了错误？
- 是否用第二个算法来验证第一个算法？
- 是否有可用于确认代码或数据的启动（startup）检查？

- 代码是否包含了随机行为？能消除这些行为吗？
- 你的代码若产生了无用信息，你是否在 DEBUG 代码中也把它们置为无用信息？
- 代码中是否有稀奇古怪的行为？
- 若代码是子系统的一部分，那么你是否建立了一个子系统测试？
- 在你的设计和代码中是否有任意情况？
- 即使程序员不感到需要，你也作完整性检查吗？
- 你是否因为排错程序太大或太慢，而将有价值的 DEBUG 测试抛置一边？
- 是否使用了不可移植的数据类型？
- 代码中是否有变量或表达式产生上溢或下溢？
- 是否准确地实现了你的设计？还是非常近似地实现了你的设计？
- 代码是否不止一次地解同一个问题？
- 是否企图消除代码中的每一个 if 语句？
- 是否用过嵌套？：运算符？
- 是否已将专用代码孤立出来？
- 是否用到了有风险的语言惯用语？
- 是否不必要地将不同类型的运算符混用？
- 是否调用了返回错误的函数？你能消除这种调用吗？
- 是否引用了尚未分配的存储空间？
- 是否引用已经释放了的存储空间？
- 是否不必要地多用了输出缓冲存储？
- 是否向静态或全局缓冲区传送了数据？
- 你的函数是否依赖于另一个函数的内部细节？
- 是否使用了怪异的或有疑问的 C 惯用语？
- 在代码中是否有挤在一行的毛病？
- 代码有不必要的灵活性吗？你能消除它们吗？
- 你的代码是经过多次“试着”求解的结果吗？
- 函数是否小并容易测试？

每当设计了一个函数或子系统后

- 此特征是否符合产品的市场策略？
- 错误代码是否作为正常返回值的特殊情况而隐藏起来？
- 是否评审了你的界面，它能保证难于出现误操作吗？
- 是否具有多用途且面面俱到的函数？
- 你是否有太灵活的（空空洞洞的）函数参数？
- 当你的函数不再需要时，它是否返回一个错误条件？
- 在调用点你的函数是出易读？

- 你的函数是否有布尔量输入？

修改错误之时

- 错误无法消失，是否能找到错误的根源？
- 是修改了错误的真正根源，还是仅仅修改了错误的症状？