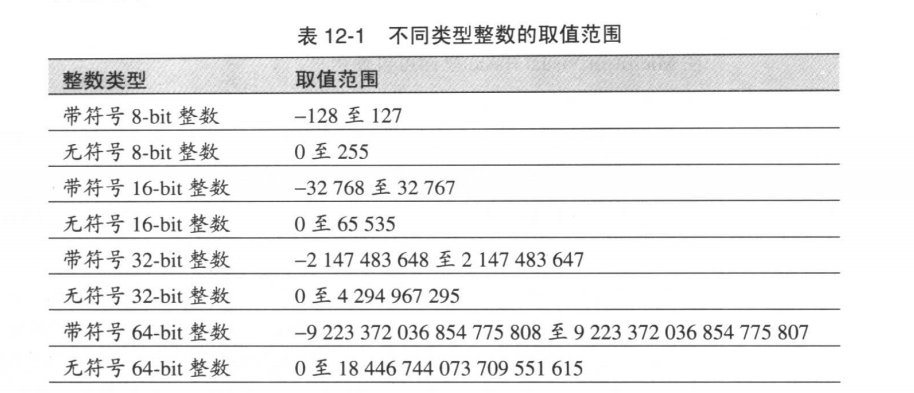
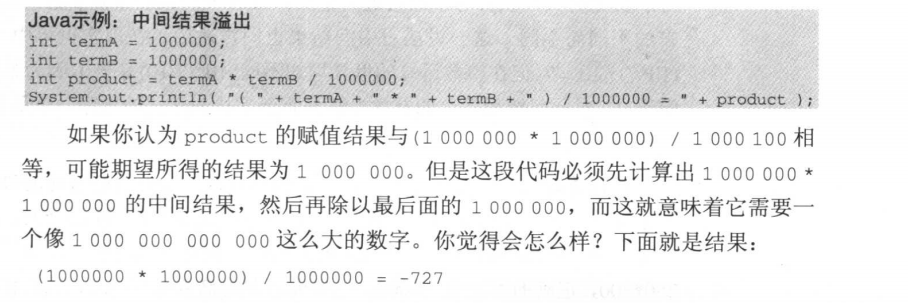
1. 防御式编程：
2. 程序的bug用assert断言
3. 一般性的错误处理分系统进行处理：如果是上线的APP，则需要追求稳定性，不能让其崩溃，当发生错误时可以给其一个适当的正常值，让其跑完，同时打印log日志文件；如果是医学类的诊断，则要求精确，如果发生错误时，不能给其一个适当值，应该让其直接崩溃。
4. 使用try-catch进行异常处理，注意：try-catch不能用于构造和析构函数内。
5. 过多的防御处理该如何解决：在自己程序内部尽量做到数据的无误，可以在使用自己内部的数据时适当的减少预防处理；当接受来自外部的数据时，一定要进行防御式处理。
6. 设置断言宏，使得断言触发时有打印信息。
7. 断言里面不应该有程序的逻辑代码，因为屏蔽断言时可能会影响到整个程序的运行。
8. 多学习C++11，这里有只能指针。
9. 最好别用一个智能指针赋值给另一个智能指针，容易出现值的循环。
10. 避免使用“神秘数值（magic number）”。神秘数值是在程序中出现的、没有经过解释的数值文字量（litertal number）。
11. 避免混合类型的比较：如果x是浮点数，i是整数，那么下面的测试 if(i == x) then ... 是不能保证可行的。在编译器设法弄清了应该用神秘类型去进行比较后，它会把其中一种类型转化为另一种，执行一些四舍五入运算后才能得出的结果。要是在这样的青黄下你的程序还能跑起来，那就算是你的运气了。
12. 注意编译器的警告。
13. 对于整数：
14. 检查整数的除法：10\*（7/10）= 0
15. 检查整数溢出：在做整数乘法或加法的时候，你要留心可能的最大整数。比如，如果你执行250\*300，正确答案是75000.但如果最大的整数是65535，那么你得到的答案可能会是9464，因为发生了整数溢出（75000-65535 = 9464），下面列出了整数类型的取值范围：



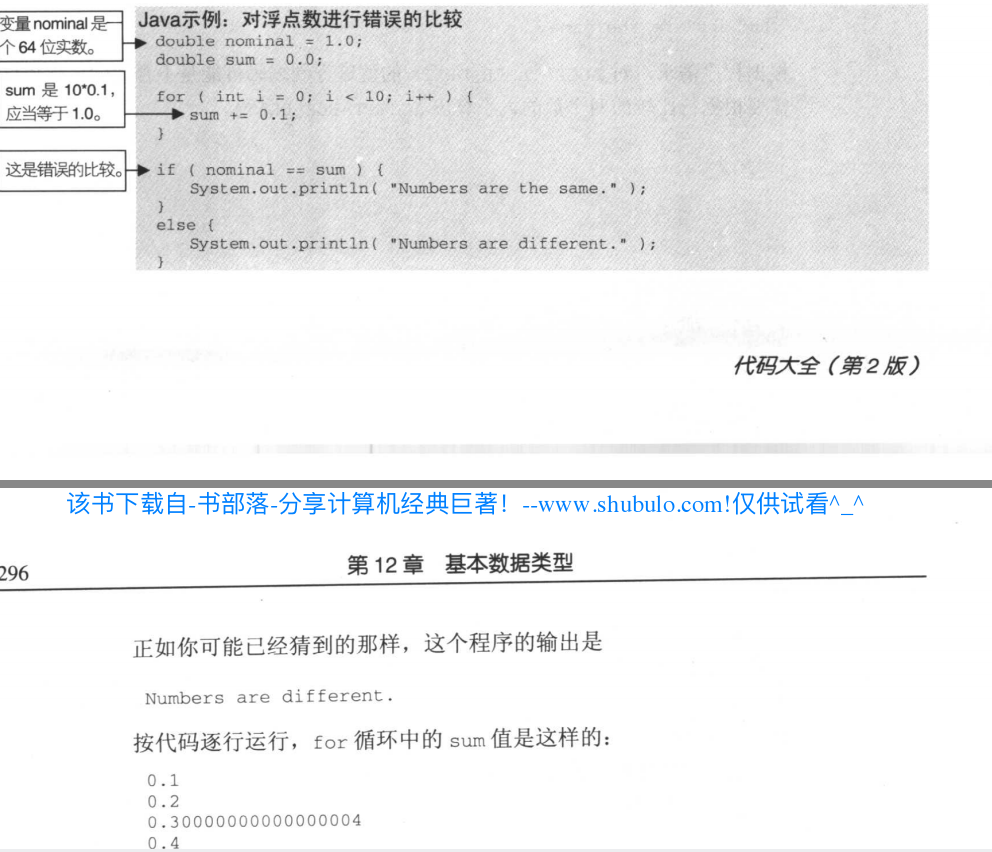
c. 检查中间结果溢出：实例如下



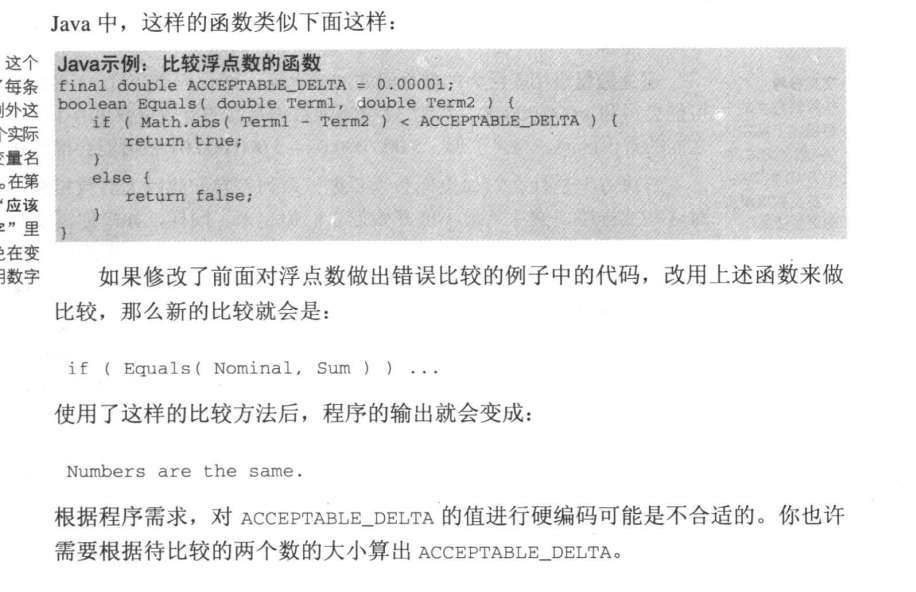
1. 对于浮点数：
2. 避免数量级相差巨大的数之间的加减法运算：32位浮点变量，1000000.00+0.1可能会得到1000000.00，因为32位不能给你足够的有效位数包容1000000和0.1之间的数值区间。与之类似，5000000.02-5000000.01很可能会得到0.0。

解决方案：先对这些数排序，然后从最小值开始把它们加起来。同样，如果你需要对无穷数列进行求和，那么久从最小的值开始——从本质上来说，是要做逆向的求和运算。这样做并不能消除舍入问题，但是能使者问题的影响减少到最低限度。

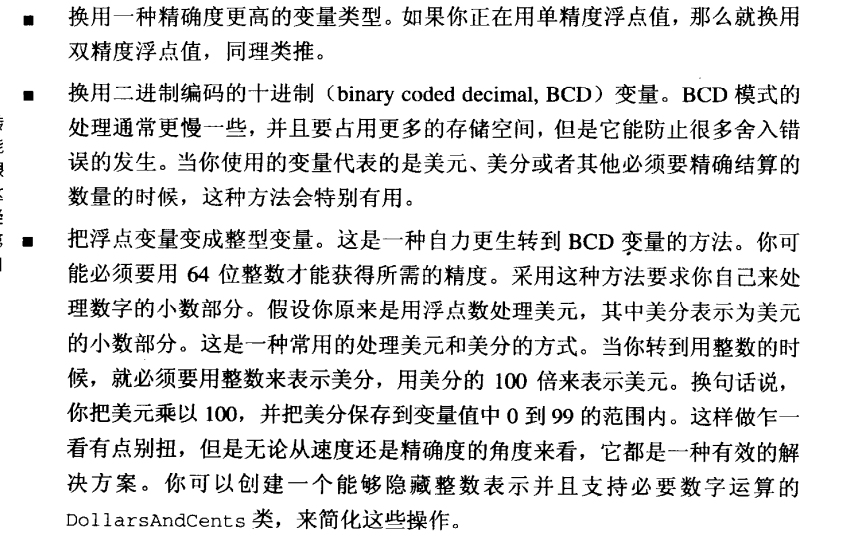
1. 避免等量判断。很多应该相等的浮点数值并不一定相等。这里的根本问题是，用两种不同方法求同一数值，结果不一定总得到同一个值。举例来说，10个0.1加起来很少会等于1.0。



解决方案：一种有效的方法是先确定可接受的精度范围，然后用布尔函数判断值是否足够接近。实例如下：



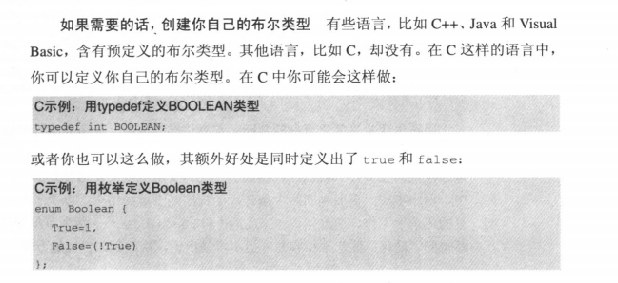
1. 处理舍入误差问题：由于舍入误差而导致的错误与由于数字之间数量级相差太大而导致的错误并无二致。问题相同，解决技术也相同。除此之外，下面列出一些专门用于解决舍入问题的常见方案：



1. 检查语言和函数库对特定数据类型的支持。
2. 字符和字符串：
3. 避免使用神秘字符和神秘字符串
4. 避免off-by-one错误：由于子字符串的下标索引方式几乎与数组相同，因此要避免因为读写操作超出了字符串末尾而导致的off-by-one（偏差一）错误。
5. 了解你的语言和开发环境是如何支持Unicode的
6. 在程序生命期中尽早决定国际化/本地化策略
7. 如果你知道只需要支持一种文字语言，请考虑使用ISO8859字符集
8. C语言中的字符串：
9. 注意字符串指针和字符数组之间的差异：字符数组是以‘\0’为结束符的。字符串指针只是保存一个一直，没有将字符串内容拷贝过来。
10. 把C-style字符串的长度声明为CONSTANT+1，这1是用来保存空结束符（位于字符串的最后取值为0的字节），在余下的代码里使用CONSTANT来指代字符串的长度。
11. 用null初始化字符串以避免没有结束符的字符串。C通过查找空结束符，即字符串末尾取值为0的字节，来判断字符串的末尾。不管你认为字符串有多长，只要C没有找到空结束符，它就认为字符串还没有结束。如果你忘记在字符串的最后放一个空值，字符串操作的结果可能就会与你预想的不一样。

我们可以使用两种方法避免没有结束符的字符串：

1. char name[MAX\_NAME\_LENGTH+1] = {0}; //声明一个数组，并初始化为0
2. 在你动态分配字符串的时候，使用calloc()而不是malloc()来把它初始化为0。malloc()只分配内存，并不执行初始化！
3. 用字符数组取代C中的指针
4. 用strncpy()取代strcpy()以避免没有结束符的字符串。
5. 布尔变量：



1. Enumerated Types(枚举类型)：
2. 用枚举类型来提高可读性
3. 用枚举类型来提高可靠性
4. 用枚举类型来简化修改
5. 将枚举类型作为布尔变量的替换方案：例如有时候一个状态有成功、警告、失败时
6. 检查非法数值：在if或者case语句中测试枚举类型时，务必记得检查非法值
7. 定义出枚举的第一项和最后一项，以便用于循环边界
8. 把枚举类型的第一个元素留作非法值
9. 明确定义项目代码编写标准中第一个和最后一个元素的使用规则，并且在使用时保持一致
10. 警惕给枚举元素明确赋值而带来的失误：给枚举值赋值不是连续时要注意
11. Array（数组）：
12. 确认所有的数组下标都没有超出数组的边界
13. 考虑用容器来取代数组，或者将数组作为顺序化结构处理：建议用于不要随机的访问数组，只能顺序的访问
14. 检查数组的边界点
15. 如果数组是多维的，确认下标的使用顺序是正确的
16. 在C中结合ARRAY\_LENGTH()宏来使用数组：

#define ARRAY\_LENGTH(x) (sizeof(x)/sizeof[0])

1. 创建自定义数据类型的指导原则：
2. 给所创建的类型去功能导向的名字
3. 要提防创建了引用预定义类型的类型名
4. 避免使用预定义类型
5. 不要重定义一个预定义的类型
6. 定义替代类型以便于移植
7. 考虑创建一个类而不是使用typedef：简单的typedef对隐藏变量的底层类型信息是有很大的帮助的。然而，在一些情况下，你可能会需要定义类所能获得的那些额外的灵活度和控制力。
8. struct（结构体）：
9. 用结构体来明确数据关系
10. 用结构体简化对数据块的操作
11. 用结构体来简化参数列表
12. 用结构体来减少维护
13. C++指针：
14. 理解指针和引用之间的区别
15. 把指针用于“按引用传递”参数，把const引用用于“按值传递”参数
16. 使用auto\_ptr
17. 灵活运用智能指针
18. C指针：
19. 使用显式指针类型而不是默认类型
20. 避免强制类型转换
21. 遵循参数传递的星号规则：也就是一级指针只能通过修改本身\*self来改变自己
22. 在内存分配中使用sizeof()确定变量的大小
23. 只有到万不得已时使用全局数据：
24. 首先把每一个变量设置为局部的，仅当需要时才把变量设置为全局的
25. 区分全局变量和类变量
26. 使用访问器子程序
27. 如何使用访问器子程序
28. 要求所有的代码通过访问器子程序来寻去数据
29. 不要把你所有的全局数据都扔在一处
30. 用锁定来控制对全局变量的访问
31. 在你的访问器程序里构建一个抽象层
32. 使得对一项数据的所有访问都发生在同一个抽象层上
33. 如何降低使用全局数据的风险
34. 创建一种命名规范来突出全局变量
35. 为全部的全局变量创建一份注释良好的清单
36. 不要用全局变量来存放中间结果
37. 不要把所有的数据都放在一个大对象中并到处传递
38. 堆区和栈区：
39. 栈区：局部变量、const局部常量，栈区向地址减小的方向增长，内存是连续的，先进后出。
40. 堆区：全局变量、new、malloc、alloc，链式结构，内存地址不连续。
41. 使用递归（recursion）：留心栈溢出、不要用递归去计算阶乘或斐波那契数列。
42. 二分查找法：又称“折半查找法”，对排序好的数组，每次取这个数与数组中间的数进行比较，复杂度是0（log n），如：设数组为a[n]，查找的数x，
43. 如果x == a[n/2]，则返回n/2;
44. 如果x < a[n/2]，则在a[0]到a[n/2 - 1]中进行查找；
45. 如果x > a[n/2]，则在a[n/2 + 1]到a[n-1]中进行查找；

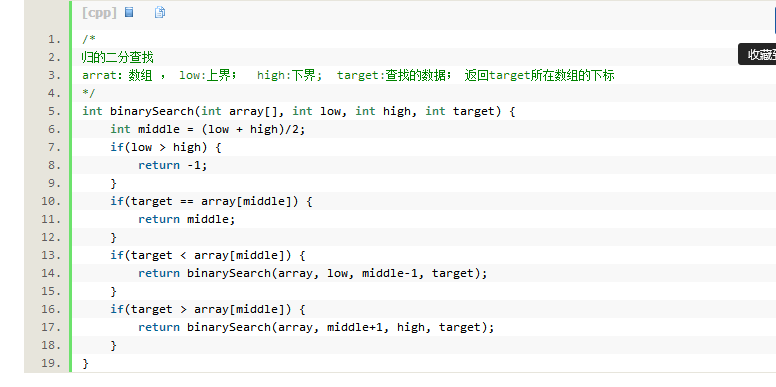
优点：比较次数较少，查找速度快，平均性能好

缺点：要求待查表为有序表，且插入删除困难

条件：查找的数组必须为有效数组

实现方式：

1. 递归：



1. 非递归（循环）：

