



防御式编程

防御式编程Defensive Programming

- 为什么采用防御式编程
- 断言
- 错误处理技术
- 异常
- 隔离程序
- 辅助调试的代码
- 对防御式编程的防御式态度

可用、正确和优秀的代码

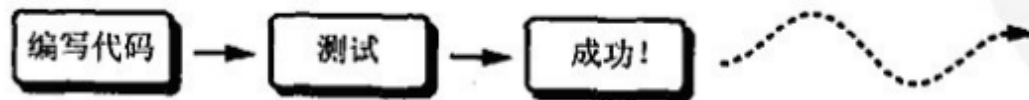
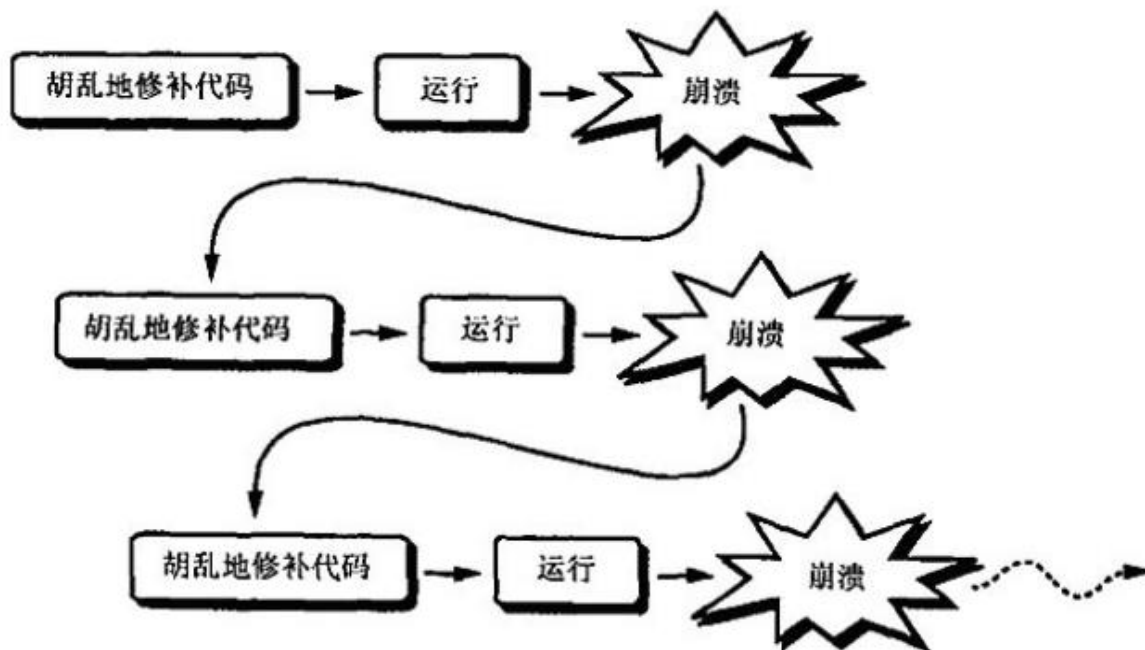
- 编写大多数情况下都能用（可用）的代码很容易
 - 提供意外输入会崩溃
- 正确的代码绝不会崩溃
 - 但所有可能输入集合很大，难以测试
 - 并非所有正确的代码都是优秀的代码
 - 可能逻辑难以理解，并几乎无法维护
- 优秀的代码是健壮的、高效的、当然也是正确的
 - 即便面对不常见输入，产品级代码也不会崩溃和产生错误结果

编程中的常见设想

- 这个函数“绝不会”被那样调用。传递给我的参数总是有效的
- 这段代码肯定会“一直”正常运行；它绝对不会产生错误
- 如果我把这个变量标记为“仅限内部使用”，就没有人会尝试访问这个变量

进行防御式编程时，不应该做任何设想！

你的软件开发过程是怎样的？



什么是防御式编程

- 通过预见到（或至少预先推测到）问题所在，断定代码中每个阶段可能出现的错误，并做出相应的防范措施，来防止类似意外的发生
 - 源于防御式驾驶
- 主要思想：
 - 子程序应该不因传入错误数据而被破坏，哪怕是由其他子程序产生的错误数据
 - 换句话说，要承认程序都会有问题，都会被修改

关于防御式编程

- 区别于检查错误
 - 防御性编程并不能排除所有的程序错误
- 区别于调试
 - 防御式编程是一种防卫方式，而不是补救方式
- 区别于测试
 - 测试不是防御式的，测试可以验证代码现在是正确的，但不保证在经历修改之后不会出错

防禦式编程技巧

- 使用好的编码风格和合理的设计
- 不要仓促地编写代码
- 不要相信任何人
- 编写清晰而非简洁的代码
- 不让其他人做不该做的修补工作
- 编译时打开所有警告开关
- 检查所有的返回值

防御式编程技巧-续

- 在声明位置初始化所有变量
- 尽可能推迟一些变量声明
- 使用安全的数据结构

```
char *unsafe_copy(const char *source)
{
    char *buffer = new char[10];
    strcpy(buffer, source);
    return buffer;
}
```

- 使用标准语言工具
- 审慎地进行强制转换

处理非法输入数据

- 好的程序应如何处理垃圾？
 - Garbage in, Garbage out?
- 检查所有来源于外部的数据的值
 - 文件、用户、网络或其他外部接口
 - 确保在允许范围内
- 检查子程序所有输入参数的值
 - ∵数据来自于其他子程序
- 决定如何处理错误的输入数据
 - It Depends!

断言Assertions

- 在开发期间使用的、让程序在运行时进行自检的代码
 - 作为对开发人员的警告
 - 通常是一个子程序或宏
 - 断言为真，表示程序运行正常；为假，表示代码中出现了意料之外的错误
- 构成（两个参数）
 - 布尔表达式：描述假设为真的情况
 - 显示的信息：断言为假时（不是必须的!）

举例

Java 示例:

```
assert denominator != 0 : "denominator is unexpectedly equal  
to 0.";
```

声明denominator不会
等于0



当第一个参数denominator != 0
为false时, 要打印的信息



断言的用途

- 输入参数或输出参数的取值处于预期的范围内
- 子程序开始（或结束）执行时，文件或流是处于打开（或关闭）的状态
- 子程序开始（或结束）执行时，文件或流的读写位置处于开头（或结尾）处
- 文件或流已用只读、只写或可读可写方式打开
- 仅用于输入的变量的值没有被子程序所修改

断言的用途-续

- 指针非空
- 传入子程序的数组或其他容器至少能容纳X个数据元素
- 表已初始化，存储着真实的数值
- 子程序开始（或结束）执行时，某个容器是空的（或满的）
- 一个经过高度优化的复杂子程序的运算结果和相对缓慢但代码清晰的子程序的运算结果相一致

什么时候使用断言

- 断言主要用于开发和维护阶段
 - 帮助查清相互矛盾的假定
 - 预料之外的情况
 - 传给子程序的错误数据等
- 生成产品代码时并不编译进去
 - 不希望用户看到产品代码中的断言信息
 - 同时避免降低系统性能


建立自己的断言机制

- 常见的高级语言都支持断言
 - 如C++ 、 Java 、 C#和Microsoft Visual Basic 等
- 如果不直接支持断言也可以自己实现
 - J2SE 1.3及早期版本没有内建的断言机制
 - C++中标准的assert宏并不支持文本信息

建立自己的断言机制：C++

C++示例： 一个实现断言的宏

```
#define ASSERT( condition, message ) {  
  \ if ( ! (condition) ) ( \  
    LogError( "Assertion failed:  
      ", #condition, message ) ; \  
    exit( EXIT_FAILURE ) ; \  
}
```



断言的两个参数

使用指导

- 用错误处理代码来处理预期会发生的非正常情况，用断言来检查永远不该发生的情况
 - 错误处理：检查有害的输入数据
 - 断言：检查代码中的bug，可看作是可执行的注解
- 避免把需要执行的代码放到断言中
 - 当关闭断言功能时，编译器会将其排除在外

VB示例：

```
Debug.Assert( PerformAction() ) ' Couldn't perform action
```

约束

- 用断言来注解并验证前/后置条件
 - “契约式设计”的一种
 - Preconditions: 子程序或类的调用方代码在调用子程序或实例化对象之前要确保为真的属性
 - 调用方代码对所调用代码要承担的义务
 - Postconditions: 子程序或类在执行结束后要确保为真的属性
 - 子程序或类对调用方代码所承担的责任
 - Invariants: 当程序执行到达特定点（如循环中、方法调用等）时都保持为真的条件
 - 否则程序逻辑存在问题

```
Private Function Velocity ( _  
    ByVal latitude As Single, _  
    ByVal longitude As Single, _  
    ByVal elevation As Single _  
    ) As Single
```

**如果变量来源于系统外部，
就应该用错误处理代码来检
查和处理非法数据**

```
' Preconditions  
Debug.Assert ( -90 <= latitude And latitude <= 90 )  
Debug.Assert ( 0 <= longitude And longitude < 360 )  
Debug.Assert ( -500 <= elevation And elevation <= 75000 )  
...  
' Postconditions  
Debug.Assert ( 0 <= returnVelocity And returnVelocity <= 600 )  
  
' return value  
Velocity = returnVelocity  
End Function
```

- 对于高健壮性的代码，应该先使用断言再处理错误

```
Private Function Velocity ( _
```

```
.....
```

```
) As Single
```

```
' Preconditions
```

```
Debug.Assert ( -90 <= latitude And latitude <= 90 )
```

```
Debug.Assert ( 0 <= longitude And longitude < 360 )
```

```
Debug.Assert ( -500 <= elevation And elevation <= 75000 )
```

```
...
```

```
If ( latitude < -90 ) Then
```

```
    latitude = -90
```

```
Elseif ( latitude > 90 ) Then
```

```
    latitude = 90
```

```
End If
```

```
If ( longitude < 0 ) Then
```

```
    longitude = 0
```

```
Elseif ( longitude > 360 ) Then ...
```

前置条件-续

- **Do not assert** method specification

```
/**
 * Sets the refresh rate.
 * @param rate refresh rate, in frames per second.
 * @throws IllegalArgumentException if rate <= 0 or
 * rate > MAX_REFRESH_RATE.
 */
public void setRefreshRate( int rate )
{
    // Enforce specified precondition in public method
    if (rate <= 0 || rate > MAX_REFRESH_RATE)
        throw new IllegalArgumentException("Illegal rate: " + rate);
    setRefreshInterval(1000/rate);
}
```

前置条件-续

■ 可以断言非public方法的前置条件

```
/**
 * Sets the refresh interval (to a legal frame rate).
 * @param interval refresh interval in milliseconds.
 */
private void setRefreshInterval( int interval )
{
    // preconditions in nonpublic method
    assert interval > 0 && interval <= 1000/MAX_REFRESH_RATE : interval;
    ... // Set the refresh interval
}
```

不变式Invariants—续

- 永远都应该为真的条件
- 内部不变式
 - 程序运行到特定时刻应该为真的事实
 - 如： `assert x > 0`
- 控制流不变式
 - 断言不会被运行到的代码，如 `assert false: suit;`
 - 注意：将语句放置在编译器认为不会被运行到的地方会报错
- 类不变式
 - 指类对象作为有效的类成员必须满足的条件
 - 如： `assert person.age >= 0 && person.age < 150;`
 - 在即将从public方法和构造函数中返回时断言类的不变式

注意

- 断言是提高软件质量技术的有益辅助手段
 - 可看作是**可执行的**注释
 - 相比注释，能更主动地对程序中的假定作出说明
- 不能依赖断言来让代码正常工作
 - 最佳方式是在一开始不要在代码中引入错误
- 断言不能有任何副作用

```
int i = pullNumberFromThinAir();  
assert(i == 6);  
printf("i is %d\n", i);
```

Question: 应该进行多少约束校验?

- 每一行都设置一个校验?
 - 过多的约束校验, 可能是代码的逻辑变得不明确
- 可读性是衡量程序质量的最佳标准
 - 在主要函数中放置前置条件和后置条件, 并且在关键的循环中放置不变条件, 就已经足够了
 - 在修正错误的地方加入一条断言也是一个良好的习惯

补充：断言举例

```
public class AssertionDemo {  
    public static void main(String[] args) {  
        int i; int sum = 0;  
        for (i = 0; i < 10; i++) {  
            sum += i;  
        }  
        assert i == 10;  
        assert sum > 10 && sum < 5 * 10 : "sum is " + sum;  
    }  
}
```

- 另一种断言的用法是放在没有default处理的switch语句

```
switch (month) {  
    case 1: ... ; break;  
    case 2: ... ; break;  
    ...  
    case 12: ... ; break;  
    default: assert false : "Invalid  
month: " + month  
}
```

```
if (x < 0) {  
    ...  
}  
else if (x == 0) {  
    ...  
}  
else {  
    assert x > 0;  
    ...  
}
```

断言 vs. 错误处理

- 断言处理代码中不应发生的错误
- 错误处理处理那些预料中可能发生的错误

错误处理技术

- 错误可能而且必将发生
 - 错误是预先就知道的，区别于程序中的bug
 - 如想打开的数据库文件已被删除，磁盘空间已满…
- 用户错误
 - 提供错误输入，或试图进行荒谬操作
- 程序员错误
 - 是一种bug，由程序员引入的代码缺陷
- 意外情况
 - 网络连接失败，打印机墨水用光等

错误处理技术

- 错误处理技术用来处理那些预料中可能要发生的错误情况
 - 返回中立值（如数值返回0）
 - 换用下一个正确的数据
 - 返回与前次相同的数据
 - 换用最接近的合法值（如经度设置为 $(-180, 180)$ 之间）
 - 把警告信息记录到日志文件中
 - 用语言内建的异常机制抛出一个异常
 -

错误处理技术—1

■ 返回中立值

- 继续执行操作并简单地返回一个没有危害的数值
 - 数值计算可以返回0
 - 字符串操作可以返回空字符串
 - 指针操作可以返回一个空指针

■ 换用下一个正确的数据

- 在处理数据流的时候，返回下一个正确的数据即可

错误处理技术—2

- 返回与上一次相同的数据
 - “重用上一次正确的结果”
 - 如windows系统崩溃后用上一次配置重新启动
- 换用最接近的合法值
 - 常出现在数值超出其正常设定的上下界的时候
 - Velocity的例子
- 把警告信息记录到日志文件中
 - 在使用这种方法的时候需要对错误信息进行标示，或者将警告信息单独存放，以便快速查询定位
 - 可以和其他技术结合使用

错误处理技术—3

▣ 返回一个错误码

- 当只有系统的某些部分处理错误，而其他部分则不在本地(局部)处理错误时
 - 设置一个状态变量的值
 - 用状态值作为函数的返回值
 - 用语言内建的异常机制抛出一个异常

▣ 调用错误处理子程序或对象

- 把错误处理集中在一个全局的错误处理子程序或对象
- 优点：能把错误处理的职责都集中到一起
- 代价：错误处理代码与整个程序紧密耦合

错误处理技术—4

- 显示出错消息
 - 当心：不要告诉系统的潜在攻击者太多东西
 - “程序应该只在没有什么可做的情况下才向用户报告错误”
- 用最妥当的方法在局部处理错误
 - 具体采用何种错误处理方法由具体程序员决定
 - 灵活度高，但系统的整体性能将无法对其正确性或可靠性的需求
- 关闭程序
 - 适用于人身安全攸关的应用程序

健壮性与正确性

- 处理错误最恰当的方式要根据出现错误的软件的类别而定
- 正确性 (correctness) 人身安全攸关的软件
 - 指软件按照需求正确执行任务的能力
 - 永不返回不准确的结果，哪怕不返回结果也比返回不准确的结果好
- 健壮性 (robustness) 消费类应用软件
 - 指软件对于规范要求以外的输入情况的处理能力
 - 要不断尝试采取某些措施，以保证软件可以持续地运转下去，哪怕有时做出一些不够准确的结果

一旦确定了某种方法，要确保始终如一地贯彻。

错误处理代码示例

1

```
void flowErrorHandling()
{
    if (operationOne())
    {
        if (operationTwo())
        {
            if (operationThree())
            { ... do more ...
            }
        }
    }
}
```

4

```
void gotoHell()
{
    if (!operationOne()) goto error;
    ... do something ...
    if (!operationTwo()) goto error;
    ... do something ...
    if (!operationThree()) goto error;
    ... do something ...
    return;
error:
    ... clean up after errors ...
}
```

2

```
void flattenedErrorHandling()
{
    bool ok = operationOne();
    if (ok)
        ok = operationTwo();
    if (ok)
        ok = operationThree();
    if (ok)
        .. do more ...
    if (!ok)
    {
        ...clean up after errors ...
    }
}
```

3

```
void shortCircuitErrorHandling()
{
    if (!operationOne()) return;
    ... do somethng ...
    if (!operationTwo()) return;
    ... do something ..
    if (!operationThree()) return;
    ... do something ...
}
```

5

```
void ExceptionalHandling()
{
    try {
        operationOne();
        operationTwo();
        operationThree();
    }
    catch( ... )
    {
        .. Cleanup after erros ...
    }
}
```

错误信息编写

- 当需要由用户清理错误时
 - 以用户期望的方式呈现信息
 - “磁盘空间：0KB”？
 - 确保用户能够理解错误信息
 - 不要呈现毫无意义的错误代码
 - “Error code 707E”？
 - 将后果严重的错误与仅仅是警告区分开来
 - 可使用“Error:”或图标
 - 提示用户每种选择可能的后果后再提出问题
 - “是否继续”？

异常

- 是把代码中的错误或异常事件传递给调用方代码的一种特殊手段
 - 当遇到意料之外的情况，但不知道该如何处理时，可以抛出一个异常
 - 同继承一样，谨慎使用可降低复杂度
- 基本结构
 - 子程序使用`throw`抛出一个异常对象
 - 被调用链上层其他子程序的`try-catch`语句捕获

表8-1 支持几种流行的编程语言的表达式

跟异常相关的特性	C++	Java	Visual Basic
支持 try-catch 语句	支持		
支持 try-catch-finally 语句	不支持		
能抛出什么	std::exception 对象或 std::exception 派生类的 对象; 对象指针; 对象引用; string 或 int 等数据类型	Exception 对象 或 Exception 派 生类的对象	Exception 对 象 或 Exception 派生类的对 象
未捕获的异常 所造成的影响	调用 std::unexpected() 函 数, 该函数在默认情况下将调 用 std::terminate(), 而这 一函数在默认情况下又将调 用 abort()	如果是一个“受检 异常 (checked exception)” 则终 止正在执行的线 程; 如果是“运 行时异常(runtime exception)”则不产 生任何影响	终止程序执 行
必须在类的接口中 定义可能会抛出的 异常	否	是	否
必须在类的接口中 定义可能会捕获的 异常	否	是	否

Q: 为什么C++没有
finally?

Trace a Program Execution

Suppose no exceptions in the statements

```
try {  
    statements;  
}  
catch (TheException ex) {  
    handling ex;  
}  
finally {  
    finalStatements;  
}
```

Next statement;

Trace a Program Execution

```
try {  
    statements;  
}  
catch (TheException ex) {  
    handling ex;  
}  
finally {  
    finalStatements;  
}
```

Next statement;



The final block is
always executed

Trace a Program Execution

```
try {  
    statements;  
}  
catch (TheException ex) {  
    handling ex;  
}  
finally {  
    finalStatements;  
}
```

Next statement;



Next statement in
the method is
executed

The diagram illustrates the flow of program execution. It starts with a try-catch-finally block. An orange arrow points from the end of the finally block to a box labeled 'Next statement;'. Another orange arrow points from this box to a rounded orange box labeled 'Next statement in the method is executed'.

Trace a Program Execution

```
try {  
    statement1;  
    statement2;  
    statement3;  
}  
catch (Exception1 ex) {  
    handling ex;  
}  
finally {  
    finalStatements;  
}
```

Suppose an exception
of type `Exception1` is
thrown in `statement2`

Next statement;

Trace a Program Execution

```
try {  
    statement1;  
    statement2;  
    statement3;  
}  
catch (Exception1 ex) {  
    handling ex;  
}  
finally {  
    finalStatements;  
}
```

Next statement;



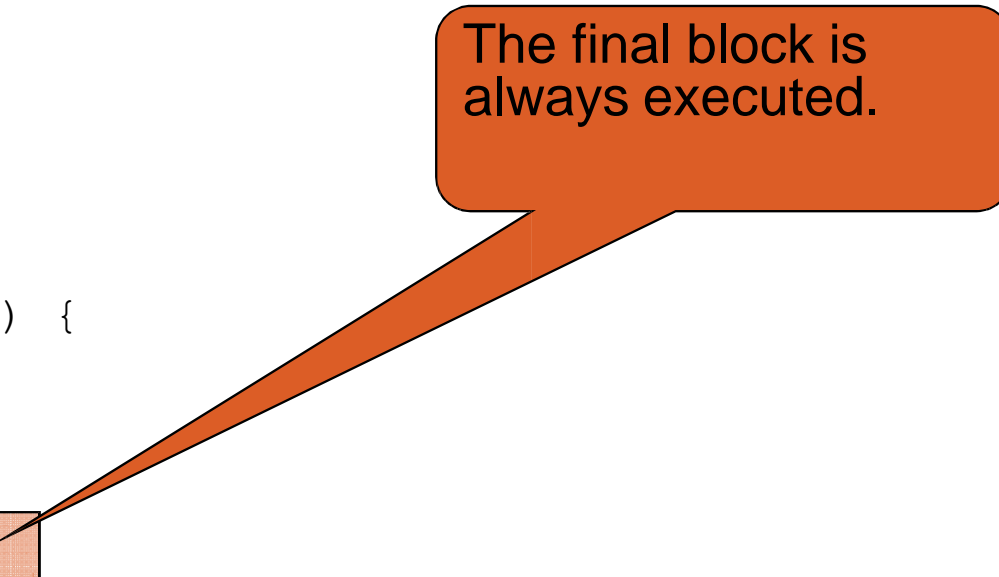
The exception is handled.

A diagram illustrating the execution of a try-catch-finally block. An orange callout box with the text "The exception is handled." has a pointer line extending from it to a small orange rectangular box containing the code "handling ex;". This box is located within the catch block of the provided code snippet.

Trace a Program Execution

```
try {  
    statement1;  
    statement2;  
    statement3;  
}  
catch (Exception1 ex) {  
    handling ex;  
}  
finally {  
    finalStatements;  
}
```

Next statement;



The final block is
always executed.

Trace a Program Execution

```
try {  
    statement1;  
    statement2;  
    statement3;  
}  
catch (Exception1 ex) {  
    handling ex;  
}  
finally {  
    finalStatements;  
}
```

Next statement;

The next statement
in the method is now
executed.



Trace a Program Execution

```
try {  
    statement1;  
    statement2;  
    statement3;  
}  
catch(Exception1 ex) {  
    handling ex;  
}  
catch(Exception2 ex) {  
    handling ex;  
    throw ex;  
}  
finally {  
    finalStatements;  
}
```

statement2 throws
an exception of type
Exception2.

Next statement;

Trace a Program Execution

```
try {  
    statement1;  
    statement2;  
    statement3;  
}  
catch(Exception1 ex) {  
    handling ex;  
}  
catch(Exception2 ex) {  
    handling ex;  
    throw ex;  
}  
finally {  
    finalStatements;  
}
```



Handling exception

Next statement;

Trace a Program Execution

```
try {  
    statement1;  
    statement2;  
    statement3;  
}  
catch(Exception1 ex) {  
    handling ex;  
}  
catch(Exception2 ex) {  
    handling ex;  
    throw ex;  
}  
finally {  
    finalStatements;  
}
```

Execute the final
block

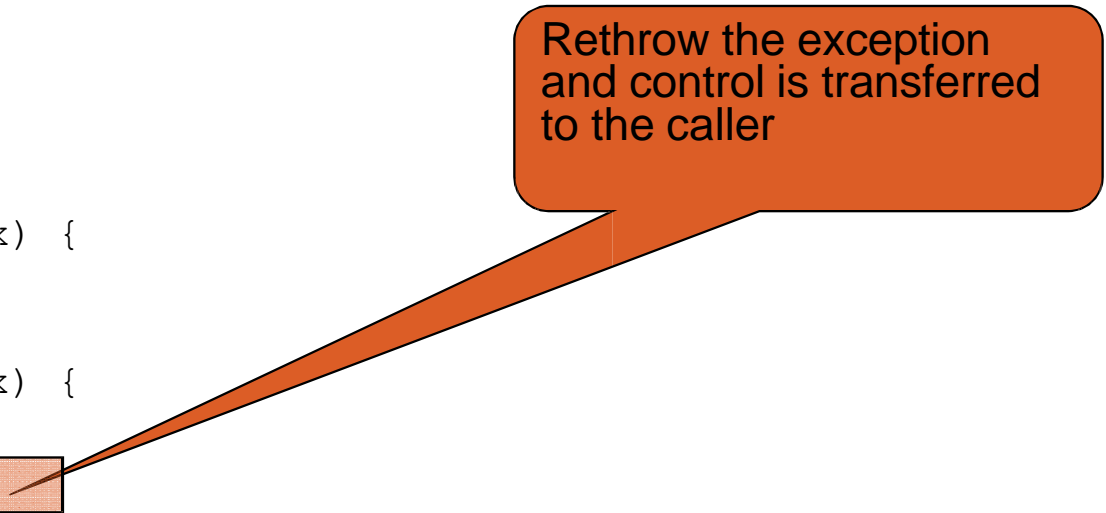
An orange arrow originates from the 'finalStatements;' line in the 'finally' block and points diagonally upwards and to the right towards the 'Execute the final block' callout box.

Next statement;

Trace a Program Execution

```
try {  
    statement1;  
    statement2;  
    statement3;  
}  
catch(Exception1 ex) {  
    handling ex;  
}  
catch(Exception2 ex) {  
    handling ex;  
    throw ex;  
}  
finally {  
    finalStatements;  
}
```

Next statement;



Rethrow the exception
and control is transferred
to the caller

使用建议-1

- 用异常通知程序的其他部分，发生了不可忽略的错误
 - 能提供一种无法被忽略的错误通知机制
 - 消除了错误向外扩散的可能性
- 只有真正例外的情况下才抛出异常
 - 仅在其他编码实践方法无法解决的情况下才使用异常（同断言类似，处理罕见且不该发生的情况）
 - 会增加复杂度：调用子程序的代码需要了解被调用代码中可能会抛出的异常，弱化了封装性

使用建议-2

- 不能用异常来推卸责任
 - 可以在局部处理就在局部处理掉
- 避免在构造函数和析构函数中抛出异常，除非你在同一地方把它们捕获
 - C++中，构造函数中的异常会造成资源泄露
- 在恰当的抽象层次抛出异常
 - 确保异常的抽象层次与子程序接口的抽象层次是一致的

举例

- 抛出的异常也是程序接口的一部分，和其他具体的数据类型一样

Java反例：抛出抽象层次不一致的异常的类

```
class Employee (  
    ...  
    public TaxId GetTaxId() throws EOFException {  
        ...  
    }  
}
```

子程序的调用方代码不是与Employee类的代码耦合，而是与比Employee类层次更低的抛出EOFException异常的代码耦合起来了

改进

- GetTaxId() 代码应抛回一个与其所在类的接口相一致的异常

Java示例:一个在一致的抽象层次上抛出异常的类

```
class Employee (  
    ...  
    public TaxId GetTaxId() throws EmployeeDataNotAvailable {  
        ...  
    }  
}
```

**在设计上需要增加EmployeeDataNotAvailable异常类型，
并将更底层的EOFException异常映射为该异常类型**

使用建议-3

- 在异常消息中加入关于导致异常发生的全部信息
 - 确保消息中含有为理解异常抛出原因所需的信息，如数组下标错误
- 避免使用空的catch语句
 - 意味着try或catch里的代码不对

```
try {  
    ...  
    // lots of code  
    ...
```

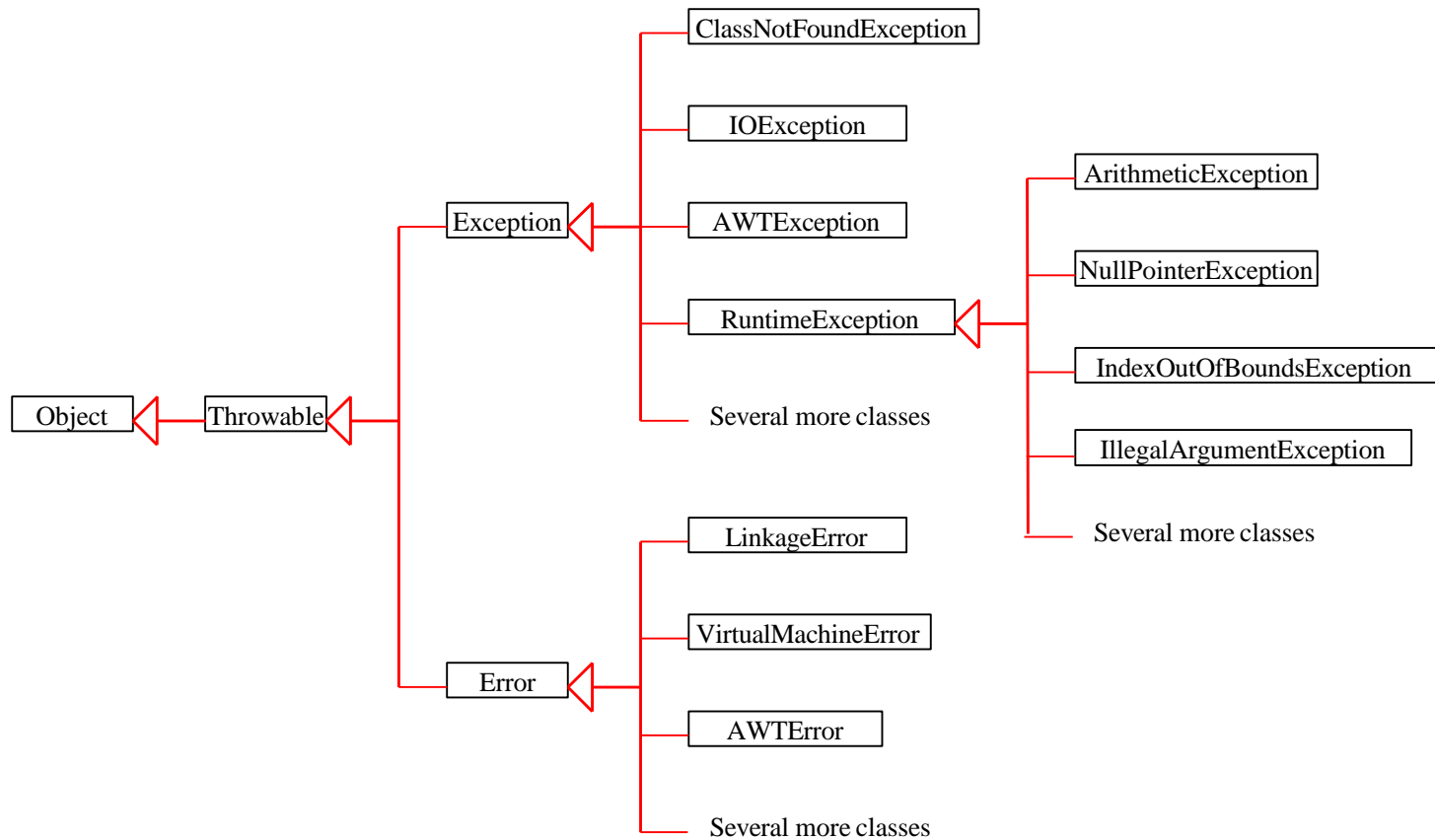
如确实无法表现为调用方抽象层次上的异常

```
} catch ( AnException exception ) {  
    LogError("Unexpected exception");  
}
```

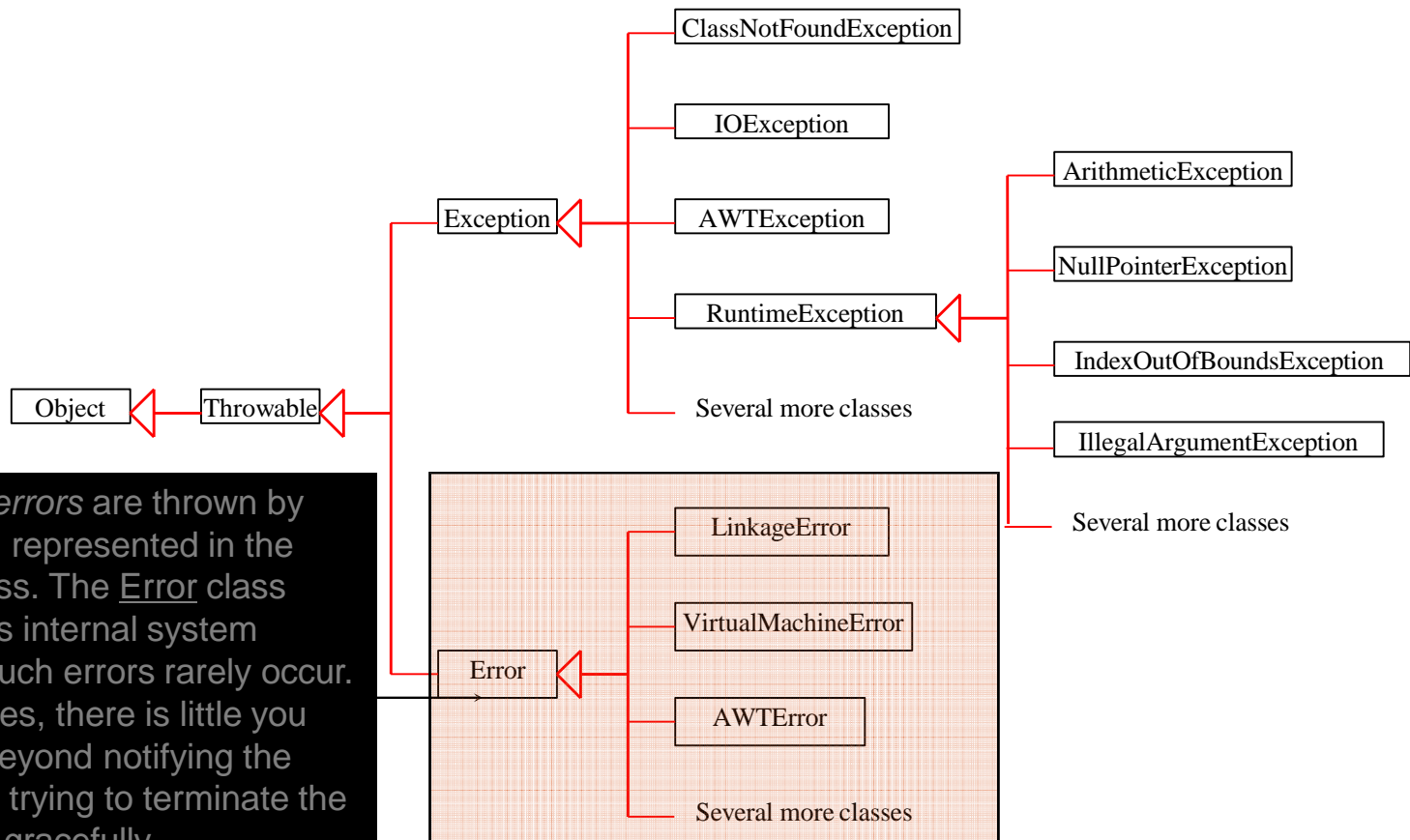

使用建议-4

- 了解所用函数库可能抛出的异常
 - 未能捕获由函数库抛出的异常可导致程序崩溃
 - 可通过编写原型代码来演练该函数库
- 把项目中对异常的使用标准化
 - 当抛出多种类型的异常，如对象、数据及指针的话，则考虑建立一个标准
 - 创建项目的特定异常类：构造自己的异常类继承体系
 - 规定哪些异常在局部处理，哪些不在局部处理
 - 规定是否可以在构造或析构函数处理异常
 - 确定是否要使用集中的异常报告机制

Exception Classes



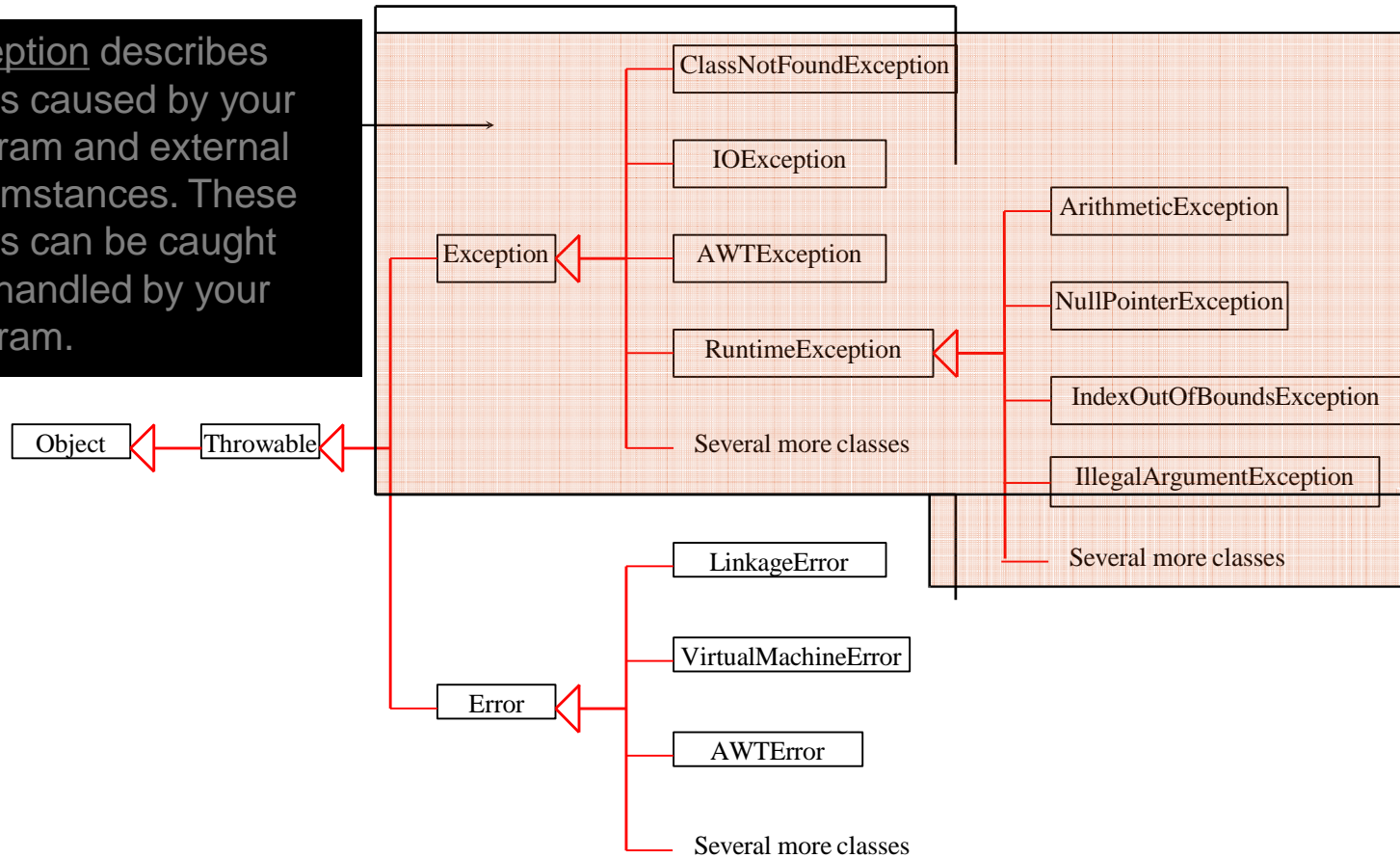
System Errors



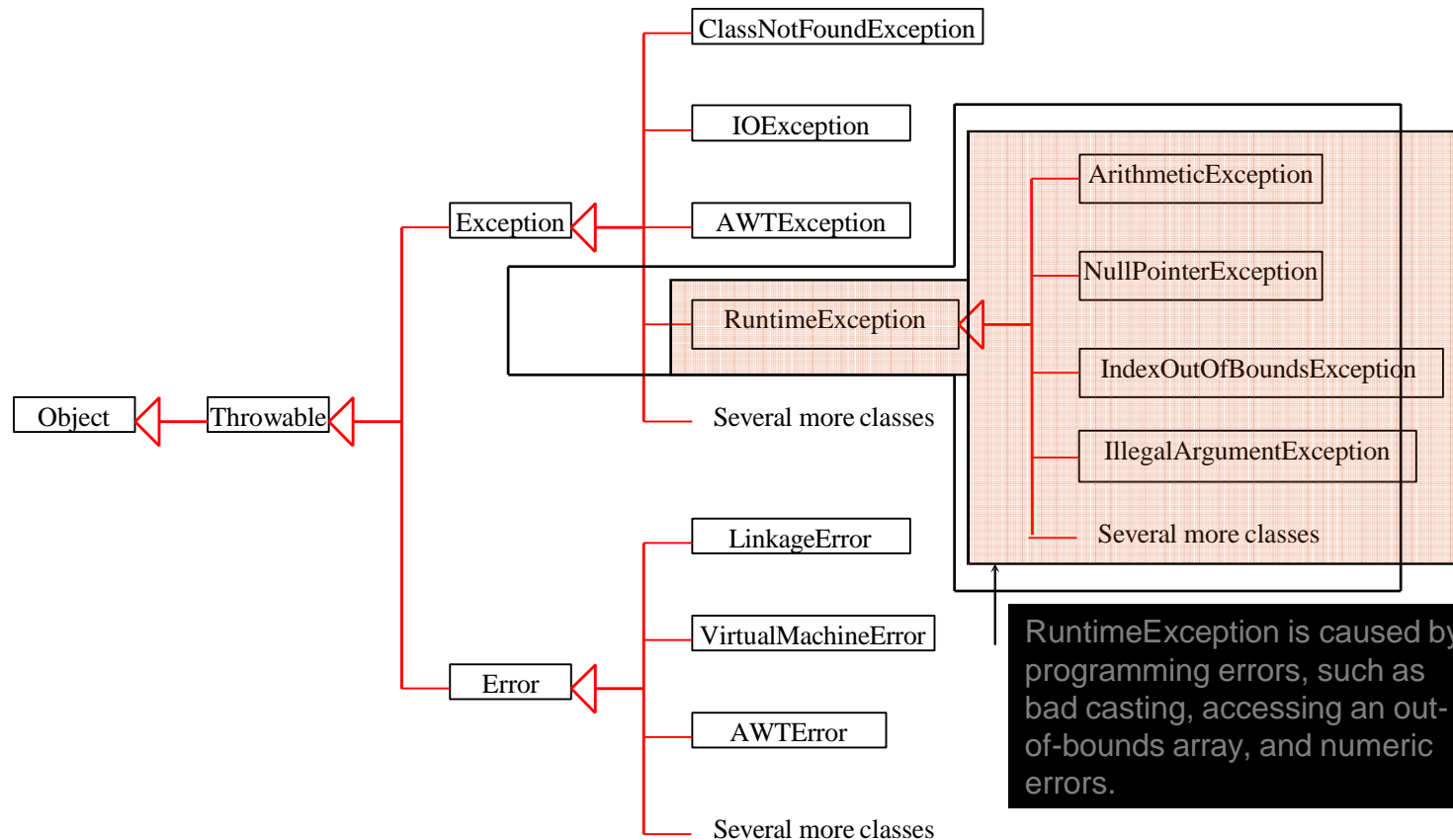
System errors are thrown by JVM and represented in the `Error` class. The `Error` class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.

Exceptions

Exception describes errors caused by your program and external circumstances. These errors can be caught and handled by your program.



Runtime Exceptions



■ 考虑创建一个集中的异常报告机制

Visual Basic示例:集中的异常报告机制 (定义部分)

```
Sub ReportException( _  
    ByVal className , _  
    ByVal thisException As Exception _  
)  
    Dim message As String  
    Dim caption As String  
    message =  
        "Exception: " & thisException .Message &  
        "Class : " & className & ControlChars .CrLf &  
        "Routine: " & thisException . TargetSite . Name & ControlChars .CrLf  
    caption = "Exception"  
    MessageBox . Show( message , caption , MessageBoxButtons . OK, _  
        MessageBoxIcon . Exclamation )  
  
End Sub
```

集中的异常报告机制

Visual Basic示例：集中的异常报告机制（调用部分）

Try

...

Catch exceptionObject As Exception

ReportException(CLASS_NAME , exceptionObject)

End Try

此处的异常报告只是一个非常简单的情况，实际开发中可以根据异常处理的需要开发或简或繁的代码

使用建议-5

■ 考虑异常的替换方案

- 不要因为编程语言提供了异常而使用它
 - “深入一种语言去编程”
- 应自始至终考虑各种各样的错误处理机制
- 思考，是否真的需要处理异常

```
try {  
    System.out.println(refVar.toString());  
}  
catch (NullPointerException ex) {  
    System.out.println("refVar is null");  
    if (refVar != null)  
        System.out.println(refVar.toString());  
    else  
        System.out.println("refVar is null");  
}
```

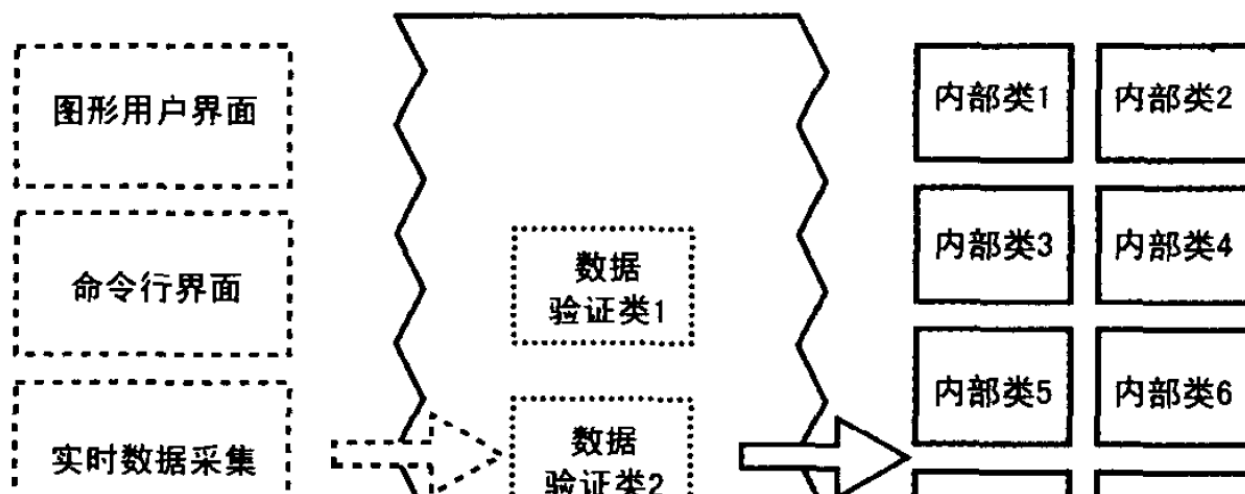

通过隔栏包容错误

■ 隔栏Barricade

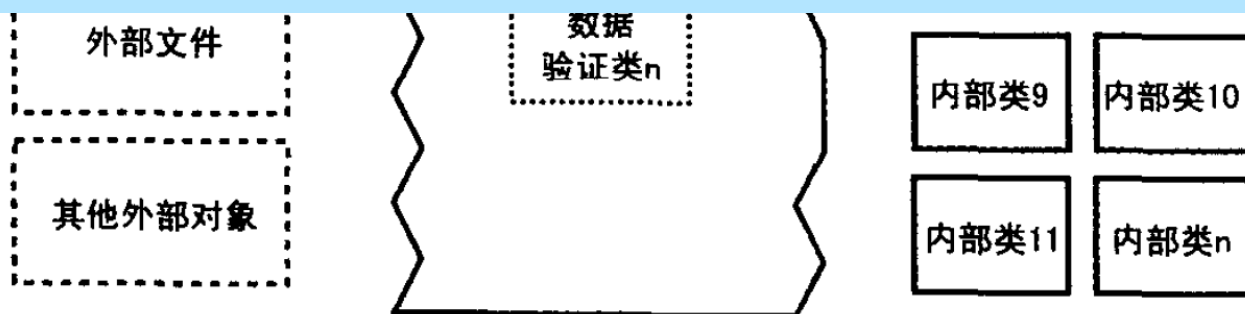
- 以防御式编程为目的而进行隔离的一种方法
- 把某些接口选定为“安全”区域的边界，对穿越安全区域边界的数据进行合法性校验
- 隔栏可以将检验工作集中在特定的模块中，从而降低其它部分采用防御式编程的成本

■ 隐喻：

- 船体外壳上装备的隔离舱
- 手术室的消毒处理



也可以在类层次采用类似方法。How?



假定此处的数据是
肮脏且不可信的。

这些类要负责清理数据。
它们构成了隔栏。

这些类可以假定数据
都是干净且可信的。

- 在输入数据时将其转换为恰当的类型
 - 输入数据通常是字符串或数字
 - 有时映射为“是”“否”等布尔变量，有时是枚举变量
 - 程序中长时间传递类型不明的数据，会增加程序的复杂度和崩溃的可能性
 - 应该在输入数据后立即将其转换到恰当的类型

隔栏vs.断言

- 隔栏的使用使断言和错误处理有了清晰的区分
- 隔栏部分包含了“脏数据”
 - 隔栏外部的程序应使用错误处理技术，在那里对数据做的任何假定都是不安全的
- 通过隔离部分之后的是“干净数据”
 - 隔栏内部的程序就应使用断言技术，因为传进来的数据应该已在通过隔栏时被清理过了
 - 隔栏内子程序出现错误数据，就是程序里的问题

辅助调试的代码

- 防御式编程的另一重要方面是使用调试助手（辅助调试的代码），从而帮助快速检测错误
- 误区
 - 产品级软件的种种限制也适用于开发中的软件
 - 如性能、资源
- 正确的认识
 - 应该在开发期间牺牲一些速度和资源，从而换取可以让开发更顺畅的内置工具
- 应该**尽早地**引入辅助调试的代码

进攻式编程Offensive Programming

- 进攻式编程是主动暴露可能出现错误的态度
 - 在开发阶段让它显现出来，而在产品代码运行时让它能够自我恢复
- 常用的进攻式编程方法
 - 确保断言语句使程序终止运行
 - 完全填充分配到的所有内存
 - 完全填充已分配到的所有文件或流
 - 确保每一个case 语句中的default分支或else分支都能产生严重错误（如终止程序）
 - 在删除一个对象之前把它填满垃圾数据
 - 让程序把错误日志文件用电子邮件发给你

移除辅助调试代码的方法

- 辅助调试的代码会使软件的体积变得庞大且速度变慢
- 使用类似ant和make这样的版本控制工具和make工具
 - 在开发模式下，你可以让make工具把所有的调试代码都包含进来一起编译
 - 在产品模式下，又可以让make工具把那些你不希望包含在商用版本中的调试代码排除在外

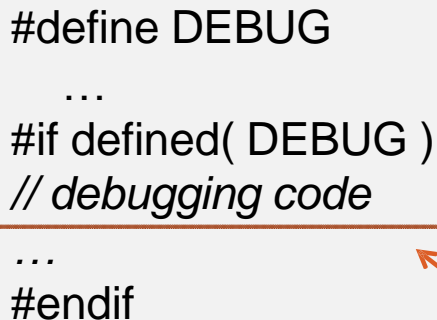
移除辅助调试代码的方法—2

■ 使用内置的预处理器

- 以用编译器开关来包含或排除调试用的代码

C++示例:直接使用预处理器来控制调试用的代码

```
#define DEBUG
...
#if defined( DEBUG )
// debugging code
...
#endif
```



所有使用辅助调试代码的地方都通过预处理器判断

使用内置的预处理器

■ 另一种更简约的形式

- 写一个能与预处理器指令同时使用的宏

```
#define DEBUG
#if defined( DEBUG )
#define DebugCode (code-fragment) {code-fragment}
#else
#define DebugCode( code_fragment )
#endif
...
DebugCode(
    statement 1;
    statement 2;
    ...
    statement n;
);
...
```

移除辅助调试代码的方法—3

- 使用调试存根debugging stubs
 - 开发代码中调用某子程序进行调试检查，产品代码中使用桩子程序替换它

C++示例：一段使用调试stub的子程序

```
void DoSomething(  
    SOME_TYPE *pointer ;  
    ...  
) {  
  
    // check parameters passed in  
    CheckPointer( pointer ) ;  
    ...  
}
```

在开发阶段，CheckPointer()
子程序会对传入的指针进行
全面检查

这行代码将调用检查指针的子程序

使用调试存根

C++示例:在开发阶段检查指针的子程序

```
void CheckPointer( void *pointer ) {  
    // 执行第1项检查—可能是检查它不为NULL  
    // 执行第2项检查—可能是检查它的地址是合法的  
    // 执行第3项检查—可能是检查它所指向的数据完好无损  
    ...  
    // 执行第n项检查— ...  
}
```

C++示例:在产品代码中检查指针的子程序

```
void CheckPointer( void *pointer ) {  
    // no code, just return to caller  
}
```



产品代码中该保留多少防御式代码？

- 开发阶段
 - 显示出的错误越多越好，引入很多防御代码
- 发布阶段
 - 希望错误尽可能偃旗息鼓，不要影响代码执行

产品代码中该保留多少防御式代码？

- 保留那些检查重要错误的代码
- 去掉检查细微错误的代码
- 去掉可以导致程序硬性崩溃的代码
 - 用户无法保存必要数据
- 保留可以让程序稳妥地崩溃的代码
 - 用于诊断潜在的严重错误
- 为你的技术支持人员记录错误信息
- 确认留在代码中的错误消息是友好的
 - “发生了内部错误”，可联系XX

如何对待防御式编程

- 过度的防御式编程也会引起问题
 - 程序会变得臃肿而缓慢
 - 增加了软件的复杂度
 - 防御式代码同样会有缺陷
- 正确的态度
 - 采用防御的姿态
 - 考虑好什么地方进行防御
 - 因地制宜地调整防御式编程的优先级

练习

- 修改以下代码，使之变得更“好”

```
char *unsafe_copy(const char *source)
{
    char *buffer = new char[10];
    strcpy(buffer, source);
    return buffer;
}
```

```
int safercopy(int from_len, char *from, int to_len, char *to)
{
    assert(from != NULL && to != NULL && "from and to can't be NULL");
    int i = 0;
    int max = from_len > to_len - 1 ? to_len - 1 : from_len;

    // to_len must have at least 1 byte
    if(from_len < 0 || to_len <= 0) return -1;

    for(i = 0; i < max; i++) {
        to[i] = from[i];
    }

    to[to_len - 1] = '\0';

    return i;
}
```