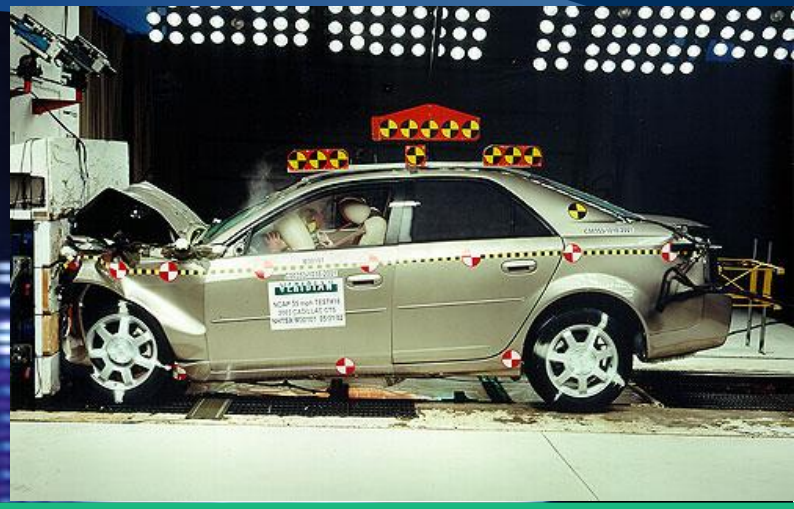
USTC

Chapter 26
Testing



王子磊 (Zilei Wang)
Email: zlwang@ustc.edu.cn
http://vim.ustc.edu.cn

Overview

- 本章介绍程序单元(如函数和类)的设计和测试，以到达程序正确性的目的
  - 讨论接口和测试用例的选择
  - 强调系统设计和从头测试的重要性以简化测试
  - 大致讨论了程序正确性和性能问题
- Contents
  - 正确性、证明和测试
  - 相关性
  - 系统测试
  - 测试GUI
  - 资源管理
  - 单元测试和系统测试
  - 找出不成立的假设
  - 测试的设计
  - 性能

正确性

- 关于程序的疑问
  - 你的程序是正确的吗？
  - 你有什么理由如此认为？
  - 你如何确认呢？
  - 为什么？
  - 你愿意乘坐使用这个代码的飞机吗？
- 你必须能够推断你的代码确切如此
  - 编程通常是非系统化的
  - 调试通常是非系统化的
  - 对于“你发现了最后一个bug”，你愿意赌什么？
- 一些有趣的相关疑问
  - 如果硬件不发生故障，程序能够永远运行吗？(可靠性)
  - 程序总是能够在合理的时间内给出结果吗？(性能)

证明

- 那么，为什么不像数学证明那样证明程序是正确的呢？
  - 通常，这是非常困难的或需要消耗太长的时间
  - 有时，证明也会出错！(即使是计算机证明或专家证明)
  - 计算机运算不是真正的数学运算
    - —— 记得近似和溢出错误(由于有限的精度)吗？
  - 那么，我们能够做什么
    - 遵从好的设计原则
    - 测试、测试、再测试！

测试

- 测试是“一种系统地查找错误的方法”
- 实际测试人员会使用很多工具
  - 单元测试框架
  - 静态代码分析工具
  - 故障植入工具
  - …
- 测试是一种需要高技巧且非常有价值的工作！
- “尽早测试、经常测试”
  - 当你写完一个函数或类时，考虑你如何测试它
  - 当你完成一个重要的修改时，重新测试(回归测试)
  - 在你移交之前(即使只有很小的修改)，重新测试

测试

- 一些有用的检测值(尤其是边界情况):
  - 空集
  - 小数据集
  - 大数据集
  - 极端分布的数据集
  - 末尾处可能发生问题的数据集
  - 重复元素的数据集
  - 奇数和偶数个元素的数据集
  - 由随机数组成的数据集

binary_search() 的原始测试

```
int a1[ ] = { 1,2,3,5,8,13,21 };
if (binary_search(a1,a1+sizeof(a1)/sizeof(*a1),1) == false)
    cout << "1 failed";
if (binary_search(a1,a1+sizeof(a1)/sizeof(*a1),5) == false)
    cout << "2 failed";
if (binary_search(a1,a1+sizeof(a1)/sizeof(*a1),8) == false)
    cout << "3 failed";
if (binary_search(a1,a1+sizeof(a1)/sizeof(*a1),21) == false)
    cout << "4 failed";
if (binary_search(a1,a1+sizeof(a1)/sizeof(*a1),-7) == true)
    cout << "5 failed";
if (binary_search(a1,a1+sizeof(a1)/sizeof(*a1),4) == true)
    cout << "6 failed";
if (binary_search(a1,a1+sizeof(a1)/sizeof(*a1),22) == true)
    cout << "7 failed";
```

重复代码、无法区分测试用例、信息有限！

一个更好的测试(仍是原始的)

将变量值放入到一个数据文件中，如格式为：

{ 27 7 { 1 2 3 5 8 13 21} 0 }

表示

{test_number value {sequence} result}

i.e., #27 号测试，调用我们的 binary_search 来在序列{ 1 2 3 5 8 13 21}中查找值7，并检测结果是否为 0 (代表 false，没有找到)

现在，可以相对容易地写出很多测试用例，或者编写一个程序来产生具有随机值的测试用例文件

相关性

基本上，我们期望每个函数都

- 有定义良好的输入
- 有定义良好的结果
  - 包括对输入参数的任何修改
  - 确定时间后结束(无限循环)
- 与任何非显式输入对象没有相关性
  - 实际情况下很难做到
- 使用不超过可提供的资源且使用是合适的
  - e.g., 时间、内存、网络带宽、文件、锁

相关性

在下面这个无意义的函数中你能看出多少相关性？

```
int do_dependent(int a, int& b) // messy function
                                // undisciplined dependencies
{
    int val ;
    cin>>val;
    vec[val] += 10;
    cout << a;
    b++;
    return b;
}
```

更复杂的函数怎么办呢？ 局部性

资源管理

在下面这个无意义的函数中，哪些资源(内存、文件等)是获取后一直没有释放的？

```
void do_resources1(int a, int b, const char* s)    // messy function
                                                   // undisciplined resource use
{
    FILE* f = fopen(s,"r");  // open file (C style)
    int* p = new int[a];                  // allocate some memory
    if (b<=0) throw Bad_arg();            // maybe throw an exception
    int* q = new int[b];                  // allocate some more memory
    delete[ ] p;                          // deallocate the memory pointed to by p
}
```

在3/4句出现异常时，p将泄漏

更好的资源管理

```
// less messy function
void do_resources2(int a, int b, const string& s)
{
    istream is(s.c_str(),"r");          // open file
    vector<int>v1(a);                   // create vector (owning memory)
    if (b<=0) throw Bad_arg();          // maybe throw an exception
    vector<int> v2(b);                  // create another vector (owning memory)
}
```

do_resources2() 会发生泄漏吗？

循环

大多数错误出现在两端，即第一次和最后一次

你能找出下面代码的3个问题吗？ 4? 5?

```
int do_loop(vector<int>& vec) // messy function
                              // undisciplined loop
{
    int i;  // ??
    int sum;   //??
    while(i<=vec.size()) sum+=v[i];   // ??
    return sum;
}
```

缓冲区溢出

- 缓冲区溢出实际上是一种特殊的循环错误
  - 存储超过容量的字节到数组中
  - 这里“额外的字节”存储到哪里了？(很可能不是合适的地方)
- 是病毒作者和黑客的首要攻击工具
- 一些易受攻击的函数(最好避免使用)：
  - **gets, scanf** *// these are the worst: avoid!*
  - **sprintf**
  - **strcat**
  - **strcpy**
  - **…**

缓冲区溢出

- 不要把“避免不安全的函数”作为护身符
  - 理解它们错误之处，且不要写出等价的代码
  - 不安全的函数 (e.g. **strcpy()**) 也是有用的
    - 如果你真正想拷贝一个 0 结尾的字符串，你不会比 **strcpy()** 做得更好 – 这时需要确保你使用的字符串不会溢出

```
char buf[MAX];
char* read_line()        // harmless? Mostly harmless? Avoid like the plague?
{
    int i = 0;
    char ch;
    while (cin.get(ch) && ch!='\n')  buf(i++)=ch;
    buf[i+1]=0;
    return buf;
}
```

缓冲区溢出

- 不要把“避免不安全的函数”作为护身符
  - 理解它们错误之处，且不要写出等价的代码
  - 编写简单安全的代码

```
string buf;
getline(cin,buf);        // buf expands to hold the newline terminated input
```

分支

- 在 if 和 switch 语句中
  - 所有的分支都覆盖了吗?
  - 每个条件采取了正确的相关行动了吗?
- 消息嵌套的 if 和 switch 语句
  - 编译器会忽略你的缩进
  - 你每嵌套一次，就需要处理所有可能的情况
    - 每一层都要乘上分支数 (不是加法呀)
- 对 switch 语句
  - 记得 **default** case 并在每种情况下 **break**
    - 除非你真的想“穿过去”

系统测试

- 首先做单元测试，然后集成各单元，直到我们获取整个系统为止
  - 理想情况下，系统各部分应该(尽可能地)能够独立测试
  - 理想情况下，应该采用可重复的方式
- 如何测试基于 GUI 的应用程序？
  - 控制的反转使 GUI 测试比较困难
  - 人类行为是完全不可重复的
    - 时间、遗忘、厌烦等
    - 但有时人工仍然是需要的(只有人类才能“看和感觉”)
  - 用测试脚本模拟用户输入
    - 在这种方式，一个测试脚本能够代替很多的人类测试
  - 一个优秀的分层应用程序在各层之间具有定义良好的接口
  - 允许应用程序在不同 GUI 系统之间移植
    - GUI 通常作为一种锁定机制来使用

测试类

- 它是单元测试的一种
  - 但大多数类对象是有状态的
  - 类通常依赖于各成员函数的交互
- 基类必须与它的派生类一起进行组合测试
  - 虚函数
  - 构造/初始化是多个类的共同责任
  - 私有数据实际是有用的
- 以我们定义的Shape 类为例
  - Shape 有若干函数
  - Shape 具有可变状态 (我们能够添加点、修改颜色等)，也就是说，一个函数的结果能够影响其他函数的行为
  - Shape 有虚函数，也就是说， Shape 的行为依赖于继承自它的派生类
  - Shape 不是一个算法 (why? 状态)
  - 对 Shape 的改变会影响屏幕显示 (因而可能仍需要人工测试？)

找出不支持的假设

- 比如，非法输入参数
  - 它应该从不会发生，但实际上可能会
- 在每次调用前或函数开始处进行检查
  - 依赖于哪些代码我们能够修改
  - *e.g.*, **sqrt** 首先检查它的输入参数是一个非负值
- 它有可能是困难/有问题的
  - 考虑 **binary_search(a,b,v);** // is v in [a:b)
    - 对前向迭代器（如一个 **list**），我们不能使用 **a<b** —— 没有 < 操作
    - 对随机访问迭代器，我们不能检查 **a** 和 **b** 是否属于相同的序列
    - 唯一漂亮的解决方案是引用一个运行时检测库 (非代码静态检查)
  - 扫描整个序列以验证它是顺序的，但这比实际要做的二分搜索工作量还要大
    - binary_search() 的目标比线性搜索更快
- 有时，仅仅在“调试/测试模式”下
  - 在最终产品代码中只留下可负担的测试

测试设计

- 使用定义良好的接口
  - 从而你能够为这些接口编写测试
  - 定义不变式、前置和后置条件
- 有一种用文本表达操作的方式
  - 从而，它们能够存储、分析和重现
- 在调用和被调用代码处嵌入对未检查假定的测试
  - 在系统测试之前获取错误参数
- 最小化相关性，并保持相关性的显式化
  - 以更容易解释代码
- 有一个清晰的资源管理策略

尽量最小化调试！

性能

- 它足够高效了吗?
  - 注意，不是“它尽可能的高效了吗？”
  - 计算机是很快的
    - 你需要百万级的操作才能够察觉到 (不使用工具)
  - 重复访问持久化数据 (硬盘上) 可能是可察觉的
  - 重复访问web可能是可察觉的
- 计时“有趣的”测试用例
  - *e.g.*, 使用 **time** 或 **clock( )**
  - 重复次数大于三次；应该有 ±10% 的可信度

性能

- 下面代码有什么问题？

```
for (int i=0; i<strlen(s); ++i) {
        // do something with s[i]  // ?? — long
}
```

- 它是网络消息日志分析器的一部分
  - 用于具有成千上万行长记录的日志文件

使用 clock( )

```
int n = 10000000;                          // repeat do_somenting() n times
clock_t t1 = clock();
if (t1 = = clock_t(-1)) {                  // clock_t(-1) means "clock() didn't work"
      cerr << "sorry, no clock\n";
      exit(1);
}
for (int i = 0; i<n; i++) do_something();  // timing loop
clock_t t2 = clock();
if (t2 = = clock_t(-1)) {
      cerr << "sorry, clock overflow\n";
      exit(2);
}
cout << "do_something() " << n << " times took "
      << double(t2-t1)/CLOCKS_PER_SEC << " seconds "        // scale result
      << " (measurement granularity: "
      << CLOCKS_PER_SEC << " of a second)\n";
```