Lessons Learned From C/C++ Defects
从缺陷中学习 C/C++

淘宝广告技术部测试团队-北京搜索与广告算法测试组

版本信息

版本号	版本日期	主要修改点	备注
V0.1	2011-05-10	Initial	设计文档大框
V0.5	2011-09-13	收集 bug 并汇总整理,形成初稿	初稿完成
V1.0	2011-10-27	收集修改建议并整理包括格式	正式稿完成

序

C/C++是我自学的一门语言，那个年代，没有课程。当然一开始只是在 UNIX 下使用 cc 作为编译器，然后是 Quick C，Turbo C，Borland C/C++，gcc/g++，VS6，VS.NET。曾经多年把 C/C++当做吃饭的家伙，甚至给儿子写的个人网站都用她来写，明知她不是最方便的，但一直都是认为她是无所不能的，没有之一。

经历了多年与她一起的日日夜夜，将近使用了 15 年后，忽然有了一个想法，想把一直以来让我欢喜让我忧的 C/C++的伴侣写下来。这个伴侣遇到时很会让人头痛，甚至会很绝望，让人忍不住会骂一句“可恶的 C（++）”（但更多的时候是非常兴奋，搓着双手，面露喜色，甚至自告奋勇地说，又出现了，我来我来），这个就是著名的“Coredump”。这里引用一下当时写的前言：

“起因是在日常工作，包括自己有兴趣写一些 C、C++程序的时候，常常遇到各种各样的程序崩溃的情况。从一开始无从下手，根本不知道怎么回事，到后来结识了一位好朋友，Coredump，到后来与她成为至交好友。这个好友，绝对是一个冷酷的朋友，如果你不了解她，一定会手足无措，不知从何下手；一旦开始认识她，就会发现原因其中缊藏着很多奥妙，让你在不断探寻她的秘密中获得那种偷窥似的快感；当你成功地了解到她的一切，与她成为好友，日久生情，就变成你作为 C++程序员生涯中不可分割的一部分，一日不见，如隔三秋；更甚者，如果分开一段时间，忽然在某处突然相遇，那种与她又能在一起的刺激和兴奋，应验了久别胜新婚的感觉...

我的心愿是能够和 C++的程序员们一起，把我们每个人与 Coredump 的恩爱情仇记录下来，就象爱情纪实小说一样，让后来者从容面对 C++生命中这段必经的精彩之旅，让过来者会心一笑，心有戚戚，也就物有所值了。”

但是，起了个开头，限于精力和水平，只是开了一个头，离完成更是遥遥无期，至今心

里有一个结。

今天看到各位同学花了这么多的心思写了一本书出来，非常敬佩，同时很惭愧。不同的角度，不同的整理方法，但是带来的是对 C++程序员在质量提升、语言技巧方面的实用、有益的帮助。书中涉及的方面非常全面，基本全是实战经验，附有大量的代码可供参考、练习。入门程序员必然可以学习和了解到 C/C++的很多关键细节，通过学习并操练，少走很多弯路；中级程序员们可以温故知新，扩大知识面，深入理解，提升进阶；高级程序员会在欣赏 C/C++编程中的逻辑、科学和艺术的同时，回到细节中去，收获或多或少的心得体会；甚至，架构师和主管们，是不是从中可以找到不少面试题目？

当然，书中由于都来源于实际，在一个 C++老程序员的眼里，覆盖面还不是最广，还有很多的点没有涉及，如能改变所有人生活方式的网络编程方面，还有让人如痴如醉的 Template 等等。但从书的版本号来看，1.0.5 表明了我们还会继续丰富。相信工程师们，看到自己的案例和代码的加入，让版本号不断更新，内容愈加加丰厚详实，会非常自豪和兴奋。

对参与本书编写工作的同学、特别是辛哲同学，表示敬意。

用时下流行的一个词来帮助形容这本书，就是“干货”，C/C++编程领域的“干货”。

<完>

三多

2011.12.08 杭州

序：测试之美

这个序言可能与这本白皮书的内容无关。借用《测试之美》这本书的名字作为标题，是因为你只有热爱这份工作，才能感受到测试之美。我想也正是因为同学们对测试工作的热爱，催生了这本书的诞生。最近了解到一些初入测试行业的同学们对职业发展的困惑，我想分享一下我从事测试 11 年之后的一些感受：我比任何时候都热爱我的工作，也比任何时候都更加体会测试之美！

我刚开始作为测试工程师在微软中国研发中心工作的时候，也曾迷茫过，困惑过。我总是为自己发现不了更多有效的 Bug 而受挫。另一方面，像很多新人一样，我开始觉得测试是一件枯燥、没有前途的工作。我甚至在工作了半年之后寻找重新回去做开发的机会。我还是坚持了下来，并在此后的过程中慢慢体会到测试的乐趣。

一个钓鱼的初学者并不具备各种钓鱼技能和经验，只能在鱼多的池塘里垂钓。而有经验的钓鱼者会根据季节、水质、鱼的种类和大小等选择合适的鱼竿和鱼饵。测试人员通过精心的测试环境和数据准备发现 Bug 的过程，就好像垂钓者捕获一只大鱼一样让人兴奋。充分运用你的测试技能，享受发现隐藏很深 Bug 时的快感，这是测试的探索之美。

测试是技术与艺术的结合。测试=VERIFICATION+VALIDATION。VERIFICATION is "are we building the product RIGHT?"， VALIDATION is "are we building the RIGHT product?"。VERIFICATION 是“验证”产品满足设计的需求，而 VALIDATION 则是站在用户的角度“确认”产品是否满足客户的需求。VERIFICATION 是技术，VALIDATION 是艺术。技术追求精确、严谨，而艺术则洞彻如何通过完美细节取悦大众。测试人员通过自己的一点点努力逐步提升产品的用户体验，让最终用户受益，这是测试的艺术之美。

20 年前的 DOS 程序员被中断调用、调色盘、如何突破 640K 内存限制等问题搞得焦头烂额，而现在的程序员已经不再为此操心了。软件开发技术在日新月异，开发人员只有不断

学习新技术才能满足岗位的要求。软件的质量和复杂度对测试提出了更高的挑战。想想看，我们现有的测试技术与 20 年前有多大的区别？测试技术整体滞后于软件技术的大规模发展，测试作为研发流程的重要环节确实需要在技术上更多的创新和突破。提升测试的重用性，让用例变得更加灵活去适应需求和设计的变化；让测试变得可视化，测试环境能够真实模拟用户环境，研发过程和质量数据集中展示和透明；让测试虚拟化，任何人都可以方便地构造一套独立的测试环境并指派自动化的任务运行。这是我期待的测试的未来，这是测试的想象之美。

最后我想对测试的新人说几句：要热爱你的工作；要充满好奇心，乐于去探究事物如何运作；要质疑一切，包括权威；要善于学习；要驾驭你的产品。只有这样，你才能驰骋于测试的自由，享受测试之美！

爱因斯坦曾经说过他坚信宇宙的原则是简单而美丽的。我想测试的意义或者目的就是让你负责的产品变得简单而美丽！

让我们大家为此而努力吧！

谢谢!

元仲

2011.11.25

前 言

这是我们从年初开始计划写的一本书。写这本书的初衷是：

- 在测试人员的眼里，在不同的项目或产品中，不同的开发人员在 Coding 中重复着同样的错误，甚至同一个人经常会重复同样的错误。
- 将时间周期拉的更长一些再看，一个开发人员，从刚毕业参加工作到具备丰富编程经验，从一个新手到成为专家，在这个成长过程中，一个开发人员也在重复着前人走过的道路，重复着同样的问题。
- 测试人员在日常测试工作中积累了大量 Bug 方面的经验，这些 Bug 是非常有价值的，不要让这些有价值的东西沉睡在 Bugfree 里，总结出来会让更多人受益。
- 一本 Bug 总结书相信一定能帮助众多开发人员提高技能，避免重复前人已经犯过的错误，这样还可以提高软件产品质量，减少生产故障，并减轻测试人员的工作量。
- 另外，对提高测试人员技能相信也会有很大帮助。
- C/C++是淘宝所有核心业务最常用的编程语言，对编程技能要求较高，更容易产生严重的生产事故，一旦出现故障追查问题根源也比较困难，所以我们首先选择 C/C++语言写这本书。
- 本书将从具体问题/Bug 出发，通过一个个具体问题的具体分析，让读者从中受益。

为了写这本书， 我们专门成立了一个白皮书小组，利用工作之余时间将这本书收集整理完成。白皮书小组成员主要包括：从雪，茹冰，水媚，青丝，泰官，穆岚，石天，辛哲。小组的主要职责是保证这本书能顺利的以高质量产出，包括中间的一些收集督促整理编辑排版等工作。在收集 Bug 阶段，很多同学积极参与，提供 bug 的同学包括：水媚，从雪，菘露，楚倩，青丝，嘉曼，元玥，师喧，辛哲，岚珊，绮珠。在白皮书编写整个过程中，北京搜索与广告算法测试小组的所有同学都参与了本书的讨论，建议，审查，及其它各种帮助性

工作，人员包括: 楚倩，嘉曼，岚珊，穆岚，绮珠，青丝，茹冰，师暄，红灯，水媚，泰官，辛哲，元玥，绛仙，菁菁，拉图，蓝晨，云卿，凤茜，石天，若楠，沐风，岚裳，萧藤，歆玥，雨萱，亦子，朱藻。

最后说明一下，由于参与这本书编写的人在 C/C++编程方面的能力所限，书中一些内容可能会有纰漏或不正确的地方，希望发现者请及时告诉我们，我们再进行更正. 谢谢!

期望这本书能真真切切的帮助到他人。

辛哲 @北京搜索与广告算法测试组

目录

第1章 基础问题

本节主要介绍由基础的一些问题所引发的 bug，涉及到 C++编程的一些基本概念和知识点，包括：宏定义、类型转换（显式的和隐式）和运算符优先级等。

1.1 不加括号的宏定义会有意想不到的错误

- **代码示例：**

```
#define COEFF(X,Y) X/3600.0-Y
…
get_delimiter() {
    …
    result[i] = static_cast<int> (( COE (d+f,f)*i) / (f*d) + i);
    …
}
```

- **现象&后果：**

result[i]这一句完成对宏定义的调用，程序运行时会出现数据处理错误。

- **Bug 分析：**

宏替换本身只是文本替换，所以一般要对参数加上括号，这样处理表达式参数（即宏的参数是个算法表达式的时候）时不容易出错。

宏定义替换过程如下：
#define COEFF(X,Y) X/3600.0-Y，COEFF(6+5,4)这个调用会替换成 6+5/3600-4，这与宏定义本身的意图是不一致的。而#define COEFF(X,Y) (X)/3600.0-(Y)，COEFF(6+5,4)这个调用是被替换成(6+5)/3600-(4)，这才是宏定义本身所希望的。

- **正确代码：**

宏定义#define COEFF(X,Y) X/3600.0-Y

应修改成
#define COEFF(X,Y) ((X)/3600.0-(Y))

- **Bug 定位：**

这个 bug 是在实际运行的时候发现结果不对，到代码中加上调试输入，发现用宏来实现的内

联函数的值不正确，进而发现的这个问题。

- **编程建议：**

宏会使得我们看到的代码和编译器看到的代码不相同，因此导致各种异常行为。
各种编程规范中都有建议，应尽可能的少使用宏。
可以尽量的使用其他(如内敛函数、const)来代替宏(下述三点来自 google c++风格指南)：
1--用宏来实现内敛进而提高效率，可以用内敛函数来替代。
2--用宏来存储常量，可以用 const 变量带替代。
3--用宏来"缩短"长变量名，可以用引用来替代。
ps：宏定义的高级应用，可参见《c 语言宏的高级应用》。

1.2 运算符优先级的问题

- **代码示例：**

```
//To get 2*n+1
int func ( int n )
{
    return n<<1+1;
}
```

- **现象&后果：**

程序运行时会出现数据处理错误。

- **正确代码：**

使用左移 1 位来代替乘以 2 的运算是很好的办法。
但是 C++里认为+的优先级大于<< ，
因此上述结果变成 return n<<(1+1); 。

- **Bug 分析：**

code review 过程中发现。

- **编程建议：**

1.养成加括号的好习惯，可以尽量避免一些不必要的错误。

2.注意：运算符重载之后，其优先级也继承重载之前的。

3.优先级的问题，参见文章：

http://www.cppblog.com/aqazero/archive/2010/10/29/8284.html

1.3 尽量减少使用宏，因为即使加了括号也可能会出问题

- **代码示例：**

```
#define SQR(x) ((x) * (x))
```

- **现象&后果：**

宏定义的时候即使对参数加了括号，当宏的参数是某些表达式（如自加自减）的时候，宏替换会出错。导致算出来的结果错误。

- **Bug 分析：**

当调用如下语句时：

```
i=3;
SQR(++i);
```

将宏展开就是：

```
((++i) * (++i));
```

结果就是

```
4*5=20;
```

- **正确代码：**

这个问题是加再多层括号都无法解决的，是宏定义天生的弊端（预编译阶段替换）带来的。正确的做法是用 inline 函数代替宏：

```
inline long SQR(int x){ return (x * x); }
```

- **Bug 定位：**

code review 过程中发现。

- **编程建议：**

在使用宏定义时应注意的地方：

1.使用宏代码最大的缺点是容易出错，预处理器在复制宏代码时常常产生意想不到的边际效

应，因为预处理器不能进行类型安全检查，或者进行自动类型转换；
2.对于 C++ 而言，使用宏代码还有另一种缺点：无法操作类的私有数据成员。假如内联函数是成员函数，对象的地址（this）会被放在合适的地方，这也是预处理器办不到的。
3.C++ 语言的函数内联机制既具备宏代码的效率，又增加了安全性，而且可以自由操作类的数据成员。

所以在 C++ 程序中，应该用内联函数取代所有宏代码，只有"断言 assert"例外，assert 是仅在 Debug 版本起作用的宏，它用于检查"不应该"发生的情况。

1.4 污染环境的宏定义

- **代码示例：**

```
#define hash_map __gnu_cxx::hash_map

…

hash_map<long,long> myhash;
```

- **现象&后果：**

若是别人 include 了你的头文件，但在自己程序又使用了自己定义的 hash_map，或者标准模板库中的 hash_map。使用了错误的库，程序就不能预期执行。

- **Bug 分析：**

由于宏替换是在预编译阶段就执行，所有出现 hash_map 的地方都会用__gnu_cxx::hash_map 来替换。因此即使你在你自己的头文件中这样包含了：#include <stl_hash_map.h>，编译器也不会去找你想要用的 STL hash_map，虽然它们在使用可能相差无几。但这是一个隐藏的定时炸弹，哪天你突然觉得 hash_map 达不到你要的需求，也许你想提升性能，也许你想扩展功能，这时候炸弹就引发了。

- **正确代码**

```
#define hash_map __gnu_cxx::hash_map

…

__gnu_cxx::hash_map <long, long> myhash;
```

- **Bug 定位：**

code review。

- **编程建议：**

1．别忘记在每个 cpp 文件中使用 using namespace xxx;

2．小心使用宏定义，尤其不要让你的宏名称与系统库撞车。

1.5 多行宏定义使用错误

- **代码示例：**

```
#define EXIT(info) std::cerr<<info<<std:endl;exit()

int function(int data)

{

    if(data<0)

        EXIT(“data is a negative number!”);

}
```

- **现象&后果：**

定义一个 EXIT 宏是用来当程序出错时发出一个出错信息然后退出。但是无论 data 是否为负数，程序都退出了。由于宏扩展错误，导致在不同的情况下都会执行相同的语句。

- **Bug 分析：**

EXIT 宏展开后，组成 2 个语句，代码变成：

```
if(data<0)

std::cerr<<“data is a negative number!”<<std:endl;exit();
```

正确的缩进以后，代码变成：

```
if(data<0)

    std::cerr<<“data is a negative number!”<<std:endl;

exit();
```

这样就可以看出程序为什么在不同条件下都会退出了。

- **正确代码：**

可以用 inline 函数来代替语句宏。

```
inline void EXIT (const char info[])
{
    std::cerr<<info<<std:endl;
exit();
}
```

值得注意的是用大括号包起来的方式：

```
#define EXIT(info) {std::cerr<<info<<std:endl;exit()}
```

也是有潜在危险的，如：

```
Int fuction(int data)
{
If(data < 0)
      EXIT("data is a negative number!");
else
      cout << data << endl;
}
```

在宏替换之后为：

```
int fuction(int data)
{
if(data < 0)
{
std::cerr<<info<<std:endl;
exit();
};
else
      cout << data << endl;
}
```

这时多余的分号会出现语法错误。

- **Bug 定位：**

功能测试

- **编程建议：**

宏扩展时，由于是直接的字符串替换，所以需要将扩展的字符串用花括号将其括起来，或者用 inline 函数来代替语句宏。

当宏定义是多行时，经典写法：

```
#define EXIT(info)
do {
std::cerr << info << std::endl;
exit();
} while(0)
```

用逗号来分隔多行代码，对逗号来说，其优先级最高：

```
#define EXIT(info) std::cerr << info << std::endl, exit();
```

1.6 Char 转为 int 时高位符号扩展的问题 💣💣

- **代码示例：**

```
static get_utili(const char *p)
{
    int util;
    …
    while (isspace((int)*p))    //跳过空格
        ++p;
    util = (int) *p++;
    …
}
```

- **现象&后果：**

当传入的参数 p 指向的内容为 0x9A、0XAB 等内容(最高位为 1)时，得到的 int 型变量 util 的

值将会出错，因为 char 会进行符号扩展，使得 0x9A(十进制的 154)变成了-102。会造成程序运行时的数据处理错误。

- **Bug 分析：**

char 符号扩展是与编译器相关的，但在 x86 平台上，对于任何主流的编译平台，char 总是进行符号扩展的。上述代码在将 char 型的*p 赋给 int 型变量 util 的时候，需要先进行 char 型到 unsigned char 型的转换，以避免按照 char 的最高位进行符号扩展。
上述出错代码的符号扩展过程如下：
因为要扩展的短数据类型为有符号数的 char x=10011100b（即 0x9A），因而在 int y=(int)x 时--进行符号扩展，即短数据类型的符号位填充到长数据类型的高字节位（比短数据类型多出的那一部分），则 y 的值为 11111111 10011100b(变成了十进制的-102)；但是，将要扩展的短数据类型变成无符号数后--unsigned char x=10011100b（即 0x9A）。在 int y=(int)x 时--进行扩展的时候是以零扩展，即用零来填充长数据类型的高字节位，则 y 的值应为 00000000 10011100b(十进制的 154)。

- **正确代码：**

util = (int) *p++;改成

util = (int)(unsigned char) *p++;

- **Bug 定位：**

该 bug 是在 code review 的过程中发现的。
char 符号扩展的问题，如果在测试时没有构造相应的 case，就会很难被发现。面对这类问题，细致的 code review 是必不可少的，不管是通过 code review 直接发现问题还是通过 review 来丰富相应 case 的构造，code review 都应该是一个不可缺少的环节。

- **编程建议：**

与此 bug 扩展的相关知识点的参考资料地址：
《编程卓越之道》的第一卷：深入理解计算机中，有一节很为详细的介绍了符号扩展、零扩展的相关内容，具体章节为 2.7 符号扩展，零扩展，以及缩减。

下载地址可参见

http://homepage.mac.com/randyhyde/webster.cs.ucr.edu/

www.writegreatcode.com/

如果必须要进行类型转换的话，建议用 c++标准的 static_cast<int> 。

1.7 int 转为 char 时的数据损失 💣💣

- **代码示例：**

```
char c;
while ((c = getchar()) != EOF) {
    putchar(c);
}
```

- **现象&后果：**

平台相关，在一些机器上运行十分正常，而在其他机器上可能死循环。这是测试的隐患。

- **Bug 分析：**

有些系统下会将 char 看成 unsigned char（可以通过 g++编译时加参数-funsigned-char 来模拟）。getchar()返回一个 int 型，int 型赋给 unsigned char c，这样当 getchar 返回 EOF（-1）时，转到 unsigned char 后的 ascii 值是 255。然后 c 和 EOF 比较，即 unsigned char 和 int 的比较，系统会将它们均专为 unsigned int 来比较，对前者是(unsigned int)255，对后者是(unsigned int)-1=232.虽然它们都表示-1，但 8 位的-1 和 32 位的-1 之间差距却甚大，永远不会相等，从而会造成死循环。

- **正确代码：**

把第一行定义改为 int c;

- **Bug 定位：**

code review 过程中发现。

- **编程建议：**

注意类型的截断和扩展。特别是 char, unsigned char, unsigned int, int, short 等之类的赋值，要尤为小心。

1.8 有符号的困惑 💣💣

- **代码示例：**

```
struct data{
	int flag:1;
	int other:31;
};
int main(){
	data test;
	test.flag = status();
	if (test.flag == 1)
	{
		……
	}
}
```

- **现象&后果：**

无论 status 的返回值是 1 还是其他值，都无法走到 if 分支里面。

- **Bug 分析：**

在结构体中设置了一位的标志位，只用一个 bit 来表示 int 时，这一位是用来表示符号位的。而带符号的 1 位数字只能是 0 或者-1。如果 stauts()返回值是 1，当 1 赋给 flag 时，由于 1 位长的字段不可能保存值 1 而失败。这时会出现溢出，就将 flag 设置为-1。所以 status 无论返回什么值给 flag，都不会是 1。这样就无法走到 if 分支。

- **正确代码：**

设置 1 位的标志位时，要用无符号型：

```
unsigned int flag:1;
```

- **Bug 定位：**

功能测试。

- **编程建议：**

总的来说，单字节字段一个是无符号型的，很容易发生溢出。

1.9 临时变量溢出 💣💣

- **代码示例：**

```
long multiply(int m,int n)
{
    long score;
    score=m*n;
    return score;
}
```

- **现象&后果：**

测试的过程中发现，当 m、n 分别取 1 亿（在 int 范围之内），带入以上公式溢出。导致程序运行时出现数据处理错误。

- **Bug 分析：**

score=m*n; 这行代码在执行时，m 和 n 相乘的结果会先存储在一个临时的 int 变量中，而后再赋值给 long 变量 score。两个 int 相乘的取值范围是 long 的范围，因此相乘是很容易溢出的。

- **正确代码：**

```
long multiply(int m,int n)
{
    long score;
    score=static_cast<long>(m) * static_cast<long> (n)
    return score;
}
```

- **Bug 定位：**

功能测试。

- **编程建议：**

对隐式的类型转换，一般来说向上是安全的，向下会出现数据截断丢失，对程序的正确性造

成影响。

从效率的观点来看，临时变量/对象的构造和释放是不必要的开销，如果可以，最好避免。

1.10 整除的精度问题 💣💣

- **代码示例：**

```
float result;

result=1/6;
```

- **现象&后果：**

程序计算出来的结果不是期待的值。

- **Bug 分析：**

1 和 6 都是整数，所以 1/6 是一个整除运算。这样语句 result=1/6 是执行 1 除以 6 的整除运算。整除运算会自动丢弃结果的小数部分，于是结果为 0。0 被转化为浮点数并赋值给 result。这样 result 就得不到期望的准确值。

- **正确代码：**

```
result=1.0/6.0
```

- **Bug 定位：**

功能测试

- **编程建议：**

要注意类型的自动转化问题。

1.11 浮点数比较的精度问题 💣💣

- **代码示例：**

```
void friend_bizns_t::_calc_data(float f, double d, int result[], int nums) {
  …
  if (f == expect_f && d == expect_d) {
        …
```

```
    return;
}
```

- **现象&后果：**

测试的过程中发现，上述代码中 if 条件为真的代码，不能一定被走到，即使构造的 case 会使得 f 和 expect_f 相等&&使得 d 和 expect_d 相等。这是因为浮点数的精度有限，不能准确的表示小数，故而不能精确的直接比较两个浮点数的大小。

- **Bug 分析：**

浮点数其表示精度的位数有限，于是不能准确的表示一个小数（IEEE 754 规定的单精度 float 数据类型的表示精度为 7 位有效数字，双精度 double 为 16 位有效数字），所以，在代码中对浮点数据类型使用== 、<= 、>=、 !=等运算符都是不正确的。
比如你在计算中做如下比较：if (result == expect_result)，这个结果几乎是永远不可能为真的。假如某一次比较这个结果为真了，那也是一次偶然的结果，因为这个比较结果是特别具有不稳定性的：数据、编译器、CPU 的微小改变就会使程序产生截然不同的结果。

于是浮点运算考虑误差时一般使用的是相对误差，并不是绝对误差。不同数量级浮点数据类型之间的运算所要考虑的误差的范围是不同的。

- **正确代码：**

```
if (f == expect_f&& d==expect_d)
```

应改为：

```
if (fabs(f - expect) < 0.00001 && fabs(d-expect_d) < 0.00001)
```

- **Bug 定位：**

实际跑程序的时候发现 if 为真的分支一直没有走到，于是在代码中加入调试信息，通过 test log 的输出，定位到是浮点数比较不当的问题。

浮点数的相关问题，其外在现象比较明显，测试时比较容易发现，但在定位问题的时候如果没有足够的对浮点数操作的敏锐度，会很难发现不符合预期输出的原因。

- **编程建议：**

与此 bug 扩展的相关知识点的参考资料地址。
浮点数除了上述比较的问题外，浮点运算过程中还需要考虑异常的发生，IEEE 754 规定的异常类型包括：非法运算、除零、溢出（上溢和下溢）以及运算不准确，glibc 关于浮点异常的描述：http://www.delorie.com/gnu/docs/glibc/libc_406.html

glibc 关于浮点异常类型处理的描述：

http://www.delorie.com/gnu/docs/glibc/libc_472.html。供参考。

1.12 小结

总结上述的一些问题，可以看出，对这样由于基础知识不清楚造成的 bug，如果编译阶段编译器并未提醒，在后面的测试中还是较难发现的，而且一旦复现，后果也比较严重。在每个逻辑业务的实现中都离不开这些基本元素，一错则牵动全局，因此是编程人员需要特别注意的。

基础的问题更接近底层，如编译器如何执行，系统如何调用，库函数如何实现，以及各个平台的差异等。而这些，对成为一名优秀的工程师是必不可少的装备。

第2章 编译问题

本节主要介绍编译环境中遇到的一些典型问题。编译相关的问题，和后续章节相比，相对简单，理解容易，是入门的学问，需要掌握，否则出现问题往往还耽误很多时间。

2.1 so 使用问题

- **代码示例：**

无。

- **现象&后果：**

导致最终返回给用户的数据错误。

- **Bug 分析：**

用错 so 版本,系统目录下有低版本的 so,集成编译时,用的是高版本的 so,但是在启动服务时,没有 export 集成版本 so 的 path,按 so 的加载顺序,就会先去加载系统目录下的低版本 so,从而导致最终的查询结果错误,可以使用 ldd 来查看:

```
ldd iquery
        libAliWS.so => /usr/lib/libAliWS.so (0x00002b093c48f000)
        libdl.so.2 => /lib64/libdl.so.2 (0x0000003242600000)
        libpthread.so.0 => /lib64/libpthread.so.0 (0x0000003242e00000)
        librt.so.1 => /lib64/librt.so.1 (0x0000003246200000)
        libstdc++.so.6 => /usr/lib64/libstdc++.so.6 (0x0000003246600000)
        libm.so.6 => /lib64/libm.so.6 (0x0000003242a00000)
        libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x0000003245e00000)
        libc.so.6 => /lib64/libc.so.6 (0x0000003242200000)
        /lib64/ld-linux-x86-64.so.2 (0x0000003241200000)
```

- **正确代码：**

export LD_LIBRARY_PATH=so 所有目录。

- **Bug 定位：**

运行前先用 ldd 检查下命令所依赖的 so 路径是否正确。

- **编程建议：**

so 使用前一定要先 export lib_path,或者把 export lib_path 写到.bash_profile 中。

2.2 链接包的加载问题

- **代码示例：**

在 k2 联调中出现过这样一个问题，在机器上安装了两个 bucket 包，链接包的名字都是 libbucket.a，结果编译是链接了错误的包，程序能正常运行，而这一块的代码没有做任何改变，所有不会关注这一块的逻辑。但联调却发现逻辑不正常。

- **现象&后果：**

功能逻辑错误，开发花费很长时间才找到问题所在。

- **Bug 分析：**

因为引擎和算法都有自己的 bucket 包，而且两个 bucket 包的包名都是 libbucket.a，调用接口也一样。结果在编译的时候优先链接了错误的包。

- **正确代码：**

每个包加上相应的前缀：libbucket.a 改为 xxx_libbcket.a 。

- **编程建议：**

良好的 lib 命名规范可以避免同类问题的发生。

2.3 使用命名空间来区分不同 cpp 中的同名类

- **代码示例：**

Makefile 中在设置动态链接库查找路径时，有两个基础库：algo_util 和 algo_common
-L/home/a/lib64 -lalgo_util -lalgo_common -liniparser -lpthread -ldl
这两个库中存在命名相同的类 log4cpp，且没有用 namespace 区分到两个不同的命名空间，导致在运行程序时，预期应当调用 algo_common 中的 log4cpp，而实际使用的却是 algo_util 中的 log4cpp。

- **现象&后果：**

程序运行时结果与预期不符。

- **Bug 分析：**

运行程序时，预期 log 输出与实际不一致，多次检查 algo-common 中的 log4cpp 都没发现问题，最后在 algo-util 库中发现了命名相同的一个类，才定位到问题。

- **正确代码：**

在我们自己的程序代码中增加命名空间：
using namespace K2;
检查命名空间中不能存在相同的类名。

- **Bug 定位：**

功能测试 。

- **编程建议：**

注意命名空间的使用，不能滥用，防止冲突。

2.4 小结

这里主要列举了动态链接包、命名空间等 4 个问题，场景都比较类似，相信读者从这几个 case 中可以有所借鉴。

第3章 库函数问题

本节介绍测试过程中常见的库函数问题。C++库函数可以降低程序员开发软件的难度，提高程序代码编写效率。使用库函数应清楚四个方面的内容：

1、 函数的功能及所能完成的操作

2、 参数的数目和顺序，以及每个参数的意义及类型

3、 返回值的意义及类型

4、 需要使用的包含文件

这是要正确使用库函数的必要条件。下面的十三个 bug，都是因为对库函数了解不清楚，错误使用而带来不好的影响。

3.1 snprintf 的 format 参数的问题

- **代码示例：**

```
int string_copy( const char*p, char* tmpbuf, int tlen)
{
    int ret = 0;
    ret = snprintf(tmpbuf, tlen, p);
    return ret;
}
```

- **现象&后果：**

如果 p 中存在格式化字符串，就会出现段错误。
例如 const char*p = "123456%s789";

某些条件下程序会出现段错误。

- **Bug 分析：**

使用 snprintf 常见的 bug，直接将字符串 p 作为 snprintf 中的 format 参数。snprintf 函数在处理参数时，发现有一个%s，把它当作 format 参数，接着继续寻找字符串指针，找不到指针，系统就会给一个随机数，然后访问到一个随机的内存地址，进行了非法访问。

- **正确代码：**

```
ret = snprintf(tmpbuf, tlen, p)
```

```
改成
ret = snprintf(tmpbuf, tlen, "%s", p)
```

- **Bug 定位：**

```
code review 过程中发现。
```

- **编程建议：**

```
snprintf()函数通常用来代替有安全隐患的 sprintf()函数，但是新手很容易用错。有关 snprintf()函数的信息，参见 C++ Reference 给出的 snprintf 的详细说明（http://www.cppreference.com/wiki/io/c/snprintf）。
```

3.2 snprintf 返回值的问题 💣💣

- **代码示例：**

```
int string_copy( const char*p, char* tmpbuf, int tlen)
{
    int ret = 0;
    ret = snprintf(tmpbuf, tlen, "%s", p);
    tmpbuf[ret] = '\0';
    return ret;
}
```

- **现象&后果：**

```
snprintf 返回的 ret 表示的是“预写入”的字符数。
当 p 的长度小于 tlen 时，ret 等于 strlen(tmpbuf)，等于 strlen(p)，这时 tmpbuf[ret] = '\0';是没有问题的
当 p 的长度大于 tlen 时，ret 大于 strlen(tmpbuf)，等于 strlen(p)，这时 tmpbuf[ret] = '\0';就写到了非法内存地址了，可能引发重大 bug。
```

- **Bug 分析：**

```
snprintf 返回的不是字符串长度，是预写入的字符串长度。如果把返回值当作字符串长度来用，就会出 bug，
比如这样：
const char*p = "1234567890";
tmpbuf[8];
ret = snprintf(tmpbuf, 8, "%s", p);
//这时 ret 返回值是 10，而不是 8
```

```
 tmpbuf[ret] = '\0';
//就是 tmpbuf[10] = '\0'，发生了写越界
```

预写入的意思是 ret = snprintf(tmpbuf, 8, "%s", p);当 p 的长度大于 8 的时候，返回值是 p 的长度，10，预写入长度也就是“打算要写 10 个字符”的意思。但实际在 tmpbuf 里只写了 8 个字符。

- **正确代码：**

```
tmpbuf[ret] = '\0';
```

改成

```
tmpbuf[tlen] = '\0';
```

- **Bug 定位：**

code review 过程中发现。

- **编程建议：**

1、这是新手很容易犯的错。snprintf()函数通常用来代替有安全隐患的 sprintf()函数。

有关 snprintf()函数的信息，参见 C++ Reference 给出的 snprintf 的详细说明（http://www.cppreference.com/wiki/io/c/snprintf）。

3.3 string c_str 使用问题 💣💣

- **代码示例：**

```
std::string local_str;

local_str = "abcd";

const char *p = local_str.c_str();

…

local_str.append(“efg”);
```

- **现象&后果：**

指针 p 不再指向 local_str。

存在一个野指针，后果可大可小。

- **Bug 分析：**

这是使用 string 的 c_str()常犯的错误，由于 c_str()返回的是一个常量指针，它所指向的地方是不会改变的。在没有改变 local_str 的值之前，指针 p 就是 local_str 的首地址，但当对 local_str 追加了一些值之后，因为 append 会先开辟一段新内存，然后再将原来的值拷贝过来，所以 local_str 的首地址已经是新的了，不再是原来 p 的指向。

- **正确代码：**

在将 local_str.c_str()赋值在 p 之后，不能去改变 local_str 。

- **Bug 定位：**

code review 过程中发现。

- **编程建议：**

注意最好使用 strcpy()函数等来操作方法 c_str()返回的指针。

比如：最好不要这样:

```
char* c;

string s="1234";

c = s.c_str();
```

c 最后指向的内容是垃圾，因为 s 对象被析构，其内容被处理，同时因为 s 对象的析构是在对指针 c 完成赋值操作之后进行的，故此处并没有报错误

正确的用法是：

```
char c[20];

string s="1234";

strcpy(c,s.c_str());
```

这样才不会出错，c_str()返回的是一个临时指针，不能对其进行操作。这样就不存在野指针的问题。

3.4 string 字符串拷贝函数 strcpy 的使用问题

- **代码示例：**

```
std::string buffer = "ab\0c";

char * str_array = new array[buffer.size ()];

strcpy(str_array, buffer.c_str());
```

- **现象&后果：**

拷贝的 str_array 与被拷贝的 buffer 不相同。

程序不如预期。

- **Bug 分析：**

字符串在内存中其实就是一个二进制流，对二进制流来说\0 也是一个正常的 01 序列，但这个\0 对 string 对象来说是一个特殊的字符，它用来标志字符串的终结符，因此对字符串函数 strcpy 等函数来说，其操作的范围，只是字符串的开始到它碰到的第一个\0。故而当我的字符串中间存在\0 时，\0 后面的字符可能就会被忽视了。正确做法是使用 memcpy，这个函数是对内存的拷贝，如前面说的，任何字符都只是 01 序列，\0 也没有什么可区别对待的。

- **正确代码：**

```
memcpy(str_array, str.c_str(), length);
```

- **Bug 定位：**

code review 过程中发现。

- **编程建议：**

注意：memcpy 中的 length 是 buffer 的实际长度，不等价于 str.size()，因为 string 长度也是以\0 为结束所作的计算。

使用 string 来作为二进制 buffer 的时候，一定要注意内容中包含\0。

3.5 string 赋值问题 💣💣

- **代码示例：**

```
void message_copy(const char* msg,    size_t    length,    string &str) {
    for( int i = 0; i < length; i++) {
        str[i] = msg[i];
    }
}

char message[10];
```

```
….

std::string str;

str.reserve(10);

message_copy(message, 10, str)；

if (str.empty()) {

      std::cout << “message wrong”;

}
```

- **现象&后果：**

这段程序接收一个 message，并判断 message 是否为空，若为空则打印“message wrong”。但不管你给 message 多长的串，程序总提示 message 空。程序不如预期。

- **Bug 分析：**

问题出在 message_copy 上，函数里面 message 的内容被逐个按字符拷贝给 string，确实 message 也拷贝到了 str 的内存空间中，我们可以通过在 message_copy 函数调用后打印 str.c_str()看出来。string 对象内部维持了一段连续空间，以及表示长度的 size。size 的值与实际长度并非总是一致的，像上面的情况，虽然改动了 str 的内存空间中赋了值，但并没有触发 size 的改变，empty()函数只是根据原有的 size 是否为 0 做了判断，而没有重新计算 str 所占内存长度，所以才会造成非预期结果。

- **正确代码：**

```
str.append(message, length);
```

- **Bug 定位：**

code review 过程中发现。

- **编程建议：**

在使用 string 的时候，尽量使用 string 内部提供的字符串函数来操作如字符串追加 append（+），字符串拷贝 memcopy 等，这些函数或者重载的操作符都会对存储的值与长度进行同步。

若使用 std::count << str;打印 str 的内容，也会是空，是因为操作符<<重载函数内部的实现是取 size 长度的进行输出。

3.6 字符串比较 💣💣

- **代码示例：**

```
#include<iostream>

int main(){

    char *str_a="test";

    char *str_b="test";

    if(str_a==str_b)

    {

        ……

    }

}
```

- **现象&后果：**

判断 2 个字符串相等，如果相等就执行括号内的语句。但是在实际运行中，是可能执行括号内的语句，也有可能不执行括号内的语句。

程序的执行不具备稳定性。

- **Bug 分析：**

在本程序中，本来期望比较的是字符串是否相等，但是实际上比较的是字符型指针。这样运算时对比的是两个指针的内容——即两个字符串所存放的地址。虽然理论上两个“test”完全相同，但完全有可能是存放在两个不同地方的相同字符串。如果这样，那两个指向字符串首地址的指针的值肯定不同。这样 if 判断的结果就是 false。但可能是考虑到节省空间的缘故，很多编译器都是如此：只生成了一个字符串“test”。让这两个指针都指向了同一个字符串，所以对比的时候自然值也是可能是相同的。这样判断起来就具备不确定性。所以如果要比较 2 个字符串，最好将字符串声明为 string 类型，这样比较起来就没有问题了。

在这里还有一种错误的用字符型错误的来比较字符串。例如：

```
char str_a[4]="test";

char str_b[4]="test";

if(str_a==str_b)

{
```

```
    ……
}
```

这里实际上比较的是字符型数组。虽然不再像上面指针那样对比的是指针值，但是比较的是两个数组的地址。数组有自己独立的空间，即使两个数组的内容完全相同，但它们的地址是肯定不同的。所以判断必定是 false。

- **正确代码：**

```
string str_a=”test”;
string str_b=”test”;
```

- **编程建议：**

采用 strcmp 等字符串库函数。

3.7 multiset 删除的误用 💣💣💣

- **代码示例：**

```
void delOne(std::multiset<int> &a,int x) { //在 a 中删除掉一个 x
    a.erase(x);
}
```

- **现象&后果：**

当集合中只有一个 x 时，是正常的，否则是不正常的，甚至有运行时错误。

可能出现运行时错误，上述代码的功能是把 a 中全部的 x 都删除掉了。

如果传给 erase 函数的是具体变量值（对象），则删除所有与之相同值的。如果传迭代器，则删除该迭代器位置的。

- **正确代码：**

函数体改为

```
std::multiset<int>::iterator it = a.find(x);
if (it != a.end()) {
    a.erase(it);    //如果要删除全部 x 值 这里可以写 a.erase(x);或者 a.erase(*it);
}
```

- **编程建议：**

注意 mutilset (multimap)与普通 map,set 容器的本质不同。

3.8 删除容器元素的陷阱 💣💣💣

- **代码示例：**

```
#include <vector>

vector a

vector::iterator b;

b=a.erase(b);
```

- **现象&后果：**

在 VS 下，编译是没有问题的，但是用 gcc 编译时，会出现编译错误。

在 gcc 下无法通过编译。

- **Bug 分析：**

在一个循环中删除容器元素的常用方法就是 b=a.erase(b)，这样可以避免迭代器陷阱，即避免 b 成为一个无效迭代器。在 vs 下，这样是可以通过编译的，但是在 gcc 下，会出现一个编译错误：即 erase 方法不能返回一个迭代器。所以赋值给迭代器 b 是错误的。为了解决这个问题，只有将这句写成 a.erase(b++)。这样实际的执行过程就是：

1、iterator tmp=b;

2、b++;

3、erase(tmp);

这样就可以通过编译，同时又避免了迭代器陷阱。虽然在编程中，我们也可以直接把代码写成这样三行，这样就能严格按照这个顺序执行，消除了编译器的不确定性。但是，按照顺序执行，当 erase 了当前文章的迭代器后，下一个迭代器的位置也不能保证是在 erase 之前+1 的位置。这样就存在迭代器行为的不确定性。

- **正确代码：**

```
a.erase(b++);
```

- **编程建议：**

目前大部分编译器都能做到调用 erase 等函数都一定会返回一个迭代器，但是还是有些编译器没有做到，所以需要用这种方法来避免迭代器陷阱。

3.9 memcpy 使用的问题 💣💣💣

- **代码示例：**

```
char buffer[1024];
while( !getline(in_file , line) )
{
    memcpy(buffer , line.c_str() , line.size() );
    ……/*处理过程*/
}
```

- **现象&后果：**

程序运行过程中，发现 buffer 的内容和预期的不一样，会导致后面的处理过程面对错误的输入。

程序运行时有可能会出现 buffer 的内容错乱。

- **Bug 分析：**

最好在使用前调用 memset 函数，将 buffer 中所有位置置为 0，否则容易出现混乱。因为如果 memcpy 中设置拷贝的字节数小于 buffer 的字节数，这 buffer 尾部剩余的字节中的内容是不可预知的。这样 buffer 的内容就会出现错误，并不等于预期拷贝的字符串。

- **正确代码：**

```
char buffer[1024];
while( !getline(in_file , line) )
{
    memset(buffer , 0 , 1024)
    memcpy(buffer , line.c_str() , line.size() );
    ……/*处理过程*/
}
```

- **Bug 定位：**

该 bug 是在功能测试的过程中发现的。

- **编程建议：**

memcpy 的函数原型是：extern void *memcpy(void *destin, void *source, unsigned n);由 source 指向地址为起始地址的连续 n 个字节的数据复制到以 destin 指向地址为起始地址的空间内。如果目标数组 destin 本身已有数据，执行 memcpy（）后，将覆盖原有数据（最多覆盖 n）。所以，在调用 memcpy 之前需要将 destin 地址空间的所有位置置为 0。

memset 的函数原型是：void * memset (void *s,int c,size_t n)：将已开辟内存空间 s 的首 n 个字节的值设为值 c。通过 memset(buffer , 0 , 1024)，通过置 0 就可以避免目标地址的空间存在一些非法的字符。

3.10 strcpy 和 strncpy 使用不当 💣💣💣

- **代码示例：**

```
String line = “”;
char buffer[1024];
while(getline(in_file, line))
{
    /**line 长度不超过 1024*/
    strncpy(buffer , line.c_str() , line.size());
    ……
}
```

- **现象&后果：**

程序运行过程中，发现 buffer 的内容和预期的不一样，会导致后面的处理过程面对错误的输入。

这段代码会导致每次循环 buffer 的内容不正确。

- **Bug 分析：**

strcpy 复制结束时，会在末尾加’\0’，但 strncpy 则不会，需要程序在后面补’\0’。

- **正确代码：**

```
String line = “”;
char buffer[1024];
while(getline(in_file, line))
{
    /**line 长度不超过 1024*/
    strncpy(buffer , line.c_str() , line.size());
    buffer[ line.size() ] = ‘\0’;
```

```
    ……
}
```

- **Bug 定位：**

该 bug 是在功能测试的过程中发现的。

- **编程建议：**

需要注意 strcpy 和 strncpy 的差异性：两者的作用都是字串复制，但是 strcpy 的原型是：char *strcpy(char *dest, char *src);把 src 所指由'\0'结束的字符串复制到 dest 所指的数组中。

注意：当 src 串长度>dest 串长度时，程序仍会将整个 src 串复制到 dest 区域，可是 dest 数组已发生溢出。因此会导致 dest 栈空间溢出以致产生崩溃异常。如果不考虑 src 串的完整性，可以把 dest 数组最后一元素置为 NULL，从 dest 串长度处插入 NULL 截取字串。

而 strncpy 的原型是：char * strncpy(char *dest, char *src, size_t n);将字符串 src 中最多 n 个字符复制到字符数组 dest 中。strncpy 并不像 strcpy 一样遇到 NULL 才停止复制，而是等凑够 n 个字符才开始复制，返回指向 dest 的指针。

注意：1）src 串长度<=dest 串长度，这里的串长度包含串尾 NULL 字符。如果 n=(0, src 串长度)，src 的前 n 个字符复制到 dest 中。但是由于没有 NULL 字符，所以直接访问 dest 串会发生栈溢出的异常情况。如果 n = src 串长度，与 strcpy 一致。如果 n = dest 串长度，[0,src 串长度]处存放于 desk 字串，[src 串长度, dest 串长度]处存放 NULL。

2）src 串长度>dest 串长度。如果 n =dest 串长度，则 dest 串没有 NULL 字符，会导致输出会有乱码。如果不考虑 src 串复制完整性，可以将 dest 最后一字符置为 NULL。

3.11 sizeof 的问题 💣💣💣

- **代码示例：**

```
char a[20] = “0123456789”;
int length = sizeof(a);
```

- **现象&后果：**

取字符串长度错误，会导致功能错误。

- **Bug 分析：**

在代码中通过 sizeof 判断字符串的长度，实际上，sizeof 用来判断空间的大小，strlen 用来判断字符串的长度，遇到’\0’结束。这时候得到的字符串长度是 20，显然这个结果是错误的

- **正确代码：**

```
Int length = strlen(a);
```

- **Bug 定位：**

该 bug 是功能测试的过程中发现的。

- **编程建议：**

在获取长度时需要考虑清楚，如果是获取字符串长度需要使用 strlen 函数，如果是想获取数据结构的大小或者空间的大小就需要使用 sizeof 函数。这种错误往往都是不细心造成的。

3.12 STL: 迭代器中 erase 循环删除的问题 💣💣💣

- **代码示例：**

```
std::vector<int> int_vec;

int_vec.push_back(1);

int_vec.push_back(2);

int_vec.push_back(2);

int_vec.push_back(3);



for (std::vector::iterator it = int_vec.begin(); it != int_vec.end(); ++it)

{

    if (*it == 2) {

        int_vec.erase(it);

    }

}
```

- **现象&后果：**

上面代码的目的是，使用迭代器循环删除 vector 中所有的 2，但你会发现并不是所有的 2 都被删除掉了。

- **Bug 分析：**

```
iterator erase(iterator _P)
{
      copy(_P + 1, end(), _P);
      _Destroy(_Last - 1, _Last);
      --_Last;
      return (_P);
 }
```

这是 erase 的源码，可以看出，erase 一个元素即_P 所指向位置时，会把这个元素后面的所有元素往前挪一个位置，而迭代器_P 虽然位置未变化，但它指向的地址已经有一个新元素挪过来了，所以对整个容器来说，其实已经自动指向了下一个元素。而在我们的代码中，erase 之后循环中还会对++it，致使漏掉了一个元素的检查，因此在整个循环执行完之后，vector 中依然有你不愿意见到的元素。

- **正确代码：**

```
for (std::vectorr<int>::iterator it = int_vec.begin(); it != int_vec.end();)
{
      if (*it == 2) {
   int_vec.erase(it);
   continue;
    }
    it++;
}
```

- **编程建议：**

无。

3.13 小结

库函数问题往往都是比较基础的问题，要避免这样的问题，除了在日常工作中多学习研究库函数说明，多实践，多实验，还需要在编程的过程中保持细心与耐心，不错用，不乱用。

C++库函数最好的说明文档就是：http://www.cplusplus.com/reference/，多看看一定会收益良多。

第4章 逻辑问题

本节介绍测试过程中常见的逻辑问题。包括逻辑分支的遗漏、逻辑细节的考虑不周、被遗漏的判定条件、潜在的死循环，等各种情况。代码中的策略，应考虑到逻辑的各种分支情况，下面的几个 bug，都因逻辑考虑不周而带来较为严重的后果。

4.1 本应该被进行的判断

- **代码示例：**

```
for(int i =0; i< 10; i++){
      fc_index->result[i].click = buff[i].click;
      fc_index->result[i].cost = buff[i].cost;
 }
```

- **现象&后果：**

crm 上客户把默认出价置为 0 后，看不到点击和消耗。

- **Bug 分析：**

未对默认出价可能为 0 的点进行判断。

- **正确代码：**

```
for(int i =0; i< 10; i++){
    if(buff[i].click == 0 || buff[i].cost == 0){
            fc_index->result[i].click = 1;
            fc_index->result[i].cost = 10;
     }
    else{
            fc_index->result[i].click = buff[i].click;
            fc_index->result[i].cost = buff[i].cost;
            }
}
```

- **Bug 定位：**

功能测试：

crm 伪登录，修改默认出价，正常值都可以显示，唯有把默认出价的可能拉到左边边界值 0 时，出现空白，由此猜测对该种情况，是由于代码里不够健壮

- **编程建议：**

每一个循环，都要有容错的判断，而不只是正常功能。

4.2 不应该出现的负数

- **代码示例：**

```
int finalscore = sumscore == 0? 0: (score * 10 / sumscore);
return finalscore;
```

- **现象&后果：**

推荐结果中相关度一列出现负数，不符合业务逻辑。

- **Bug 分析：**

相关性计算中，会进行减分，因此当完全不相关，又有减分的情况，最终分数会为负数，不符合业务逻辑。应该判断如果最终分数 < 0，则返回 0。

- **正确代码：**

```
int finalscore = sumscore
if (sumsore > 0) {
    finalscore==score * 10 / sumscore;
}
else
    finalscore==0;
LogDebug("[query end:] query(%s) score(%d) sumscore(%d) finalscore(%d)",
query.query.c_str(), score, sumscore, finalscore);
return finalscore;
```

- **Bug 定位：**

请求关键词推荐服务，发现推荐出的词有相关性为负的结果，猜测在相关性计算上面特殊情况。

- **编程建议：**

对于业务相关的计算，除了保证计算公式正确，还要注意对计算结果进行有效范围的判断；对于取值范围的边界值也要作为单元测试点。

4.3 被遗漏的逻辑分支

- **代码示例：**

```
bool monitor_file_t::get_file_time(stat fstat_ , time_t *p_time) {
…
//这个函数里实现的功能是，获取文件时间：如果文件不存在或者不是标准文件，则返回 false，否则获取文件时间，返回 true
…

bool monitor_file_t::is_update(stat fstat_) {
        time_t now;
        //if 文件更新，返回 true
        if (get_file_time(fstat_ , &now)) {
                if (now != _tv_last) {
                        _tv_last = now;
                        return true;
                }
        }
        //if 文件未更新，返回 false
        UB_LOG_ERROR("monitor file not update.\n");
        return false;
}
```

- **现象&后果：**

程序不会带来功能上的严重后果，但逻辑考虑的不全面:当文件不规范或者不存在的时候，函数 monitor_file_t 也会认为文件没有更新，给出 ERROR LOG“monitor file not update”，但事实 ERROR LOG 应是 monitor file not exist OR not a regular file。这个函数对错误逻辑的处理是不完善的，应增加相应的 LOG 输出。

- **Bug 分析：**

get_file_time 函数返回了两个信息，一个是获取文件时间成功与否的标识，一个是文件的更新时间。is_update 函数在调用它的时候，采用的逻辑是：当 get_file_time 函数带回了文件的更新时间且和上一轮获取的更新时间不相同时，(对应代码：if (get_file_time(fstat_ , &now)) { if (now != _tv_last)这两句)，则认为文件已更新，其他情况就都当作一种情况，认为文件没有更新。这个从逻辑是上合并了不可合并的错误处理情况。

is_update 应该增加如下逻辑：get_file_time 函数的返回值为 true 的情况下，如果 now != _tv_last，则认为文件更新了，否则认为文件没更新，输出相应 log。当 get_file_time 函数的

返回值为 false 的情况下，则是文件不存在或者文件不规范，输出相应 log。

- **正确代码：**

```
bool monitor_file_t::is_update(stat fstat_) 这一函数应该修改为：
bool monitor_file_t::is_update(stat fstat_) {
        time_t now;
        if (get_file_time(fstat_,&now)) {
                if (now != _tv_last) {
                        _tv_last = now;
                        return true;
                }
                else {
                        UB_LOG_ERROR("monitor file not update.\n");
                        return false;
                }
        }
        UB_LOG_ERROR("monitor file not exist OR not a regular file.\n");
        return false;
}
```

- **Bug 定位：**

实际测试的时候有这样的 case 覆盖，发现不管是文件未存在还是文件未更新，给出的 log 输出都为文件未更新。到代码中查看发现是逻辑上的分类遗漏。

- **编程建议：**

调用函数时要明确函数的接口后正确使用函数，特别要注意函数的异常处理逻辑。

4.4 没有判断 ret 值

- **代码示例：**

```
ret = m_kwrecmTdbm.store(key, val, m_kwrecmTdbmExpire, 0);

// m_kwrecmTdbmExpire 为 cache 过期时间

Log::Error("kwrecmCacheStore: {%s} failed.", key.c_str());
```

- **现象&后果：**

调用 m_kwrecmTdbm.store() 后，当 ret 不为 0 时，没有进行处理。

- **Bug 分析：**

m_kwrecmTdbm.store() 返回 ret 值包含了错误码，需要进行错误处理。

- **正确代码：**

```
ret = m_kwrecmTdbm.store(key, val, m_kwrecmTdbmExpire, 0);
if(ret != 0)
{
	Log::Error("kwrecmCacheStore: {%s} failed.", key.c_str());
}
Log::Debug("kwrecmCacheStore: {%s} {%s}", key.c_str(), val.c_str())
```

- **Bug 定位：**

code review 发现调用 m_kwrecmTdbm.store() 的 ret 值没有进行判断 。

- **编程建议：**

要对函数调用的 ret 值的错误码进行异常处理并打印 EOORLOG，以增强程序的健壮性。

4.5 永不变化的逻辑值

- **代码示例：**

```
float MaxSoc = -1;
for (int i=0,i<values;i++)
 {
	score = TextPair3[i].score;
	row = TextPair3[i].row;
	float fstaticScore    = score;
	if (fabs(fstaticScore - MaxSoc)>0) {
		MaxRow = row;
	}
	…
}
```

- **现象&后果：**

测试过程中发现，该段代码永远是把最后一条记录的 row 当作最大的 MaxRow 来输出，导致 MaxRow 选取错误，程序得不到正常结果。

- **Bug 分析：**

出错的地方是在 if 语句的逻辑中，if 语句用来实现对最大 static score 记录的选取，判断如下：如果当前记录的 staticScore 大于当前最大分，则把 MaxRow 置为当前记录的 row。这里忽略的一点是，没有把最大分随之一并更新，最大分一直都停留在初始的-1 状态，进而导致所有的结果与之比较都大于之，这样就导致了不管哪条结果最大，总是输出最后一条结果为最大 row 的情况。

- **正确代码：**

if (fabs(fstaticScore - MaxSoc)>0) 语句处做如下修改：

```
if (fabs(fstaticScore - MaxSoc)>0) {
        MaxRow = row;
        MaxSoc = fstaticScore
  }
```

- **Bug 定位：**

测试时比较容易覆盖到的逻辑，只要 case 中覆盖到非最后一条结果的静态分(static score)最大这样的情况即可发现。

- **编程建议：**

在遍历集合统计最值时，要注意最值变量的更新和每遍历到一条记录时判断是否要更新要一致，即更新的最值必须在判断是否更新的条件中。

4.6 脏数据的引入 💣💣

- **代码示例：**

```
bool pack_get_item(const char *pline, int &type, pack_item_t & item_){//函数说明
…
}

bool get_russ(const char *pline) {
        pack_item_t item_;
        …
        pack_get_item(pline, &type1, &item_);
        …
```

```
    pack_get_item(pline, &type2, &item_);
    …
}
```

- **现象&后果：**

程序运行时会出现数据处理错误：构造 case，使得函数 pack_get_item 在 type 参数为 type1 的时候成功，在 type 参数为 type2 的时候失败。这时第二次运行函数 pack_get_item 后得到的 item_中的数据就是脏数据。因为第二次调用 pack_get_item 函数没有成功，item_中的数据还是第一次调用 pack_get_item 后的数据。

- **Bug 分析：**

上述函数的结构大致如下：函数 get_russ 调用函数 pack_get_item 二次，调用 pack_get_item 一次后，后续会用引用 item_参数做一系列动作，而后调用其第二次。
这一过程存在的问题是：当函数 get_russ 第一次调用 pack_get_item 成功，而第二次调用 pack_get_item 失败，因为没有对第二次调用 pack_get_item 函数的返回值进行判断，也没有对 item_参数进行归 0，导致 item_在进入到第二次 pack_get_item 函数调用的时候里面还包含着第一次被调用后存入的结果，而使得在第二次调用 pack_get_item 函数失败的时候，第一次调用函数 pack_get_item 的 item_结果，会作为脏数据引入到第二次调用 pack_get_item 函数后的后续操作中。
这是典型的引入脏数据的问题。Dev 在设计函数 pack_get_item 的时候，已经赋予了其 bool 型的返回值，而在函数 get_russ 中使用函数 pack_get_item，却没有对其返回值进行判断。导致第二次调用函数 pack_get_item 后的 item_因为包含上一次的数据而有效。

- **正确代码：**

```
pack_get_item(pline, &type1, &item_);
 加上对其返回值的判断
bool b=pack_get_item(pline, &type1, &item_);
if (b ==false){
  …
}
```

- **Bug 定位：**

code review 的阶段发现没有对函数的返回值进行判断，就在测试的时候构造了第二次调用 pack_get_item 函数失败的 case。

- **编程建议：**

1.当调用函数传入引用或指针函数时，在使用前注意验证传入参数的赋值正确性，可从以下

几点：

（1）若函数有返回值代表正确处理或错误码，则调用后需要根据返回值正确和错误分开处理。

（2）若函数无错误码，可先对传入参数初始化某个值，调用后如果值仍为初始值则处理有错，如值符合预期则继续下一步处理。

2.同一个函数中多次同一个变量时，注意使用前要初始化，避免引入上一次或未初始化的脏数据。

4.7 潜在的数组越界 💣💣

- **代码示例：**

```
int index = 0;

_div[index] = pp;

while (*pp && index < sep_col_sizes) {

    if (seperator == *pp) {

    *pp = '\0';

    _div[++index] = pp + 1;

    }

    pp++;

}
```

- **现象&后果：**

当 div 数组的 length 小于 sep_col_size，上述代码会出现数组越界，会带来内存问题。一旦触发上述代码的 bug，会带来 core。

- **Bug 分析：**

这里只对一行文本中存储的内容过多导致包含列的数目超过了规定的大小进行了判断，而没有对 index 值太小，即一行存储的内容不全异常情况进行判断和处理。需要补充对存储内容不全的情况的判断。

这个例子给我们的警示是，在我们阅读代码的时候，应避免对代码的逻辑带着走。不应该仅仅的陷于代码的实现当中。review code 的过程中不仅仅是一个了解代码功能的过程，更是一个对软件的设计思路和实现逻辑 review 的过程。

结合本例，应注意到，此处代码完成是的对一行文本的解析，文本是否符合默认的字段要求，这也是一个需要 check 的检查点。

- **正确代码：**

```
int index = 0;

_div[index] = pp;

int len = sizeof(div);

if ( len < sep_col_sizes ) {

    ...//做异常情况的处理

}

else {

    while (*pp && index < sep_col_sizes) {

        if (seperator == *pp) {

            *pp = '\0';

            _div[++index] = pp + 1;

        }

        pp++;

    }

}
```

- **Bug 定位：**

实际测试的时候设计了 div 数组长度不满 sep_col_size 的情况，覆盖到了上述情况。

- **编程建议：**

数组遍历下标越界会出 core，所以遍历数组时在满足算法逻辑的前一定要保证数组长度是数组遍历的一个结束点，或在结束条件中包含。

4.8 试图产生的指针很可能不存在 💣💣

- **代码示例：**

```
void func(const int* pInt, size_t size);
```

```
…

std::vector<int> vecInt;

…// 对 vecInt 进行操作

func( &vecInt[0], vecInt.size() );
```

- **现象&后果：**

当 func 调用时，如果 vector 对象指针仍为空指针，程序会 core dump 。

- **Bug 分析：**

当 vector 中没有元素，&vecInt 试图产生一个不存在的指针，而当在函数中操作这样一个指针，只要进行取值操作就会造成程序的奔溃。

- **正确代码：**

```
if (!vecInt.empty()) {

    func(&vecInt[0], vecInt.size());

}
```

或者

```
int func(const int* pInt, size_t size)

{

    if (pInt==NULL) return -1;

    …

}
```

- **Bug 定位：**

设计 vecInt 在调用 func 时为空的 case，进行功能测试或单元测试 。

- **编程建议：**

对于指针的操控前需要判断指针不为 NULL，特别是数组或 vector 容器复杂的初始化逻辑中，需要考虑到是否有未被赋值的逻辑；对于函数中的 C 指针参数，使用前需要 NULL 判断，以避免错误调用。

此 bug 参考自《effective STL》

http://read.pudn.com/downloads108/ebook/448291/%E7%BB%8F%E5%85%B8C%2B%2B%E5%9

B%BE%E4%B9%A6%E4%B8%8B%E8%BD%BD1(14%E6%9C%AC)/Effective%20STL-revised.pdf 。

4.9 虚假截断 💣💣

- **代码示例：**

```
// buildNZipIndex 为建立非压缩索引的函数，返回索引的条数

int buildNZipIndex(…);



//超长限制：当测试数据的个数＞ 2 ^ 6 需要进行截断
const uint32_t MAX_IDX2_FD_BIT              = 6;
const uint32_t MAX_IDX2_DOC_NUM             = (1<<MAX_IDX2_NUM_BIT);


具体函数调用如下：

//useLen 为上游倒排链的实际长度，doc_count 为截断后的实际 doc 数, pind1->len 为倒排链长



uint32_t useLen = 0;
ret = buildNZipIndex(&base_num, pbuilder->raw_buf, pbuilder->raw_buf_size, useLen);
if (unlikely(ret<=0)) {
      TERR("build uncompressed index failed. ret %d, line sign=%llu", ret, pind1->term_sign);
      return -1;
}
if((uint32_t)ret > MAX_IDX2_DOC_NUM) {
      doc_count = MAX_IDX2_DOC_NUM;
      TERR("idx2 docnum over bit level, true=%d, limit=%d, line sign=%llu",    ret, doc_count,
pind1->term_sign);
} else {
      doc_count = ret;
}
pind1->len = useLen;
```

- **现象&后果：**

上游的二级倒排链长度超过限制时，超长部分也能索引成功，而且 docsnum 却显示为截断后的。

- **Bug 分析：**

当二级倒排链链超长后，只对 doc_count 做了截断操作，没对实际的长度 useLen 做截断操作，在截断后误把 useLen 赋给了链表长度，应该赋值为 doc_count。

- **正确代码：**

```
uint32_t useLen = 0;
ret = buildNZipIndex(&base_num, pbuilder->raw_buf, pbuilder->raw_buf_size, useLen);
if (unlikely(ret<=0)) {
    TERR("build uncompressed index failed. ret %d, line sign=%llu", ret, pind1->term_sign);
    return -1;
}
if((uint32_t)ret > MAX_IDX2_DOC_NUM) {
    doc_count = MAX_IDX2_DOC_NUM;
    TERR("idx2 docnum over bit level, true=%d, limit=%d, line sign=%llu",  ret, doc_count,
pind1->term_sign);
} else {
   doc_count = ret;
}
pind1->len = doc_count;
```

- **Bug 定位：**

功能测试设计属于边界值测试的 case，超过程序设置范围后，处理得有些矛盾。

- **编程建议：**

当函数里相近意义的变量较多时，使用时一定要小心区分。

4.10 潜在的死循环 💣💣

- **代码示例：**

```
int32_t IndexNoneOccParse::next(char*& outbuf, uint32_t& docId)
{
    const char* pre = _begin;
    char* cur = NULL;
    // read doc-id
    do {
        docId = strtoul(pre, &cur, 10);
```

```
            if (unlikely(pre == cur) || cur >= _end) { //  解析到行尾，退出循环
            docId = _preDocId;
            return 0;
            }
            pre = cur;

            // read occ
            strtoul(pre, &cur, 10);
            if (unlikely(pre == cur) || cur > _end) {
                  return -1;
            }
            pre = cur;
     } while(_node == docId);
    _begin = cur;
    _node = docId;
    outbuf = (char*)&_node;

    _preDocId = docId;
    return sizeof(_node);
```

- **现象&后果：**

查询索引数据可能会死循环。

- **Bug 分析：**

程序默认是上游数据正确是有序的，没对上游数据无序的情况做判断，结果建立索引时导致索引乱序，在查询索引时当 docid 高位压缩后重复时，查询进入死循环。

- **正确代码：**

```
int32_t IndexNoneOccParse::next(char*& outbuf, uint32_t& docId)
{
      const char* pre = _begin;
      char* cur = NULL;

      // read doc-id
      do {
            docId = strtoul(pre, &cur, 10);
            if (unlikely(pre == cur) || cur >= _end) { //  解析到行尾，退出循环
                  docId = _preDocId;
```

```
                return 0;
            }
            pre = cur;
            // read occ
            strtoul(pre, &cur, 10);
            if (unlikely(pre == cur) || cur > _end) {
                return -1;
            }
            pre = cur;
        } while(_node == docId);

        _begin = cur;
        _node = docId;
        outbuf = (char*)&_node;

        if(docId < _preDocId) {
            return -1;
        }
        _preDocId = docId;
        return sizeof(_node);
}
```

- **Bug 定位：**

索引查询性能时发现，用大量的 term 做查询，发现有些 term 返回的 doc 无序。

- **编程建议：**

不要太相信上游数据的正确性，一定要对数据错误异常进行判断，以保证程序的健壮性。

4.11 带来 core dump 的函数声明 💣💣💣

- **代码示例：**

```
obj. method1(inrt array[], unsigned    int size)
…
obj.method1(array, -1)
```

- **现象&后果：**

调用 method1 的时候传入的参数为-1 的时候，会引发 core dump。

- **Bug 分析：**

method1(inrt array[], unsigned int size) 在函数声明的时候直接把 size 设置为无符号的整形，而实际调用 method1 的时候又不能保证传入函数的参数一定是正整数，当传入的参数为负数的时候，会发生从 int 到 unsigned int 的强制转换。-1 转换 unsigned int 型的时候，将会被转换成一个超大的正整数(64 位平台即转换为 2^32-1)，传入 method1 函数将会出现数组越界，进而 core dump。
这时，即使 method1 函数中有 size 值是否为正的判断，也是于事无补。因为参数 size 在传入 method1 函数的时候就已经被转换为正整形了。结果这一问题的方法是因为把 method1 函数的 size 参数声明为 int 型，而在函数中加上对 size 是否为正数的判断。

- **正确代码：**

```
obj. method1(inrt array[], int size) )
//obj. method1 函数中加上对 size 是否为正数的判断
…bj.method1(array, -1))
```

- **Bug 定位：**

程序运行到对 method1 调用的地方会出现 core dump，通过这个 core 定位到问题所在。

- **编程建议：**

C 语言中的基本数据类型不安全，无构造和析构函数，容易产生内存泄露，而且基本数据类型间编译器会做隐性的类型转换；建议：

了解编译器的隐性类型转换规则；

函数声明中要求正数的参数，不要用 Unsigned 类型，应该使用有符号类型如 int 并在函数体中对参数范围进行判断；

对于大的工程，建议使用自定义的安全类型。

4.12 小结

逻辑分支控制着程序的执行流程，根据不同的输入产生了对应输出，其中包含了正确和异常的逻辑处理，实现了程序功能、业务逻辑等。

本章大多数 bug 涉及程序层面的逻辑导致程序健壮性不足，业务相关的 bug 影响程序的业务功能，下面从输入到输出的可能流程给出了编程时的注意事项。

1. 输入数据要进行必要的正确性检查，如有序化数据有序化检查。
2. 变量使用正确：初始化和释放正确，避免错用类似命名变量。
3. 变量的特征值处理：如除 0，指针为空，数组下标越界，字符串空串。
4. 函数调用要区分对待不同的返回值和异常分支。

5. 输出数据：要进行数值和业务相关的范围有效性检查。

第5章 文件处理

本节主要介绍 C++对文件操作时出现的几个常见典型问题，并对其分析与总结。

5.1 写日志文件没有调用 fflush

- **代码示例：**

```
//..fp 是一个 FILE*类型，日志文件句柄
//do 1 …
fprintf(fp, “1 done\n”);
// do 2…
fprintf(fp,” 2 done\n”);
实例代码
#include <stdio.h>
int main()
{
        int* p = NULL;
        FILE *fp;
        fp = fopen("111","w");
        fprintf(fp,"%s","hello world");
        //fflush(fp);
        //*p = 10;          //如果在这里程序异常退出了.不加 fflush 的话,log 不会写入.
        fclose(fp);
        return 0;
}
```

- **现象&后果：**

如果程序中途出现了运行时错误，则日志文件很可能不是全部想要的信息

程序本来想记录日志，在操作后记录一些信息，以便调试。然而，这样的日志文件却有可能不完整，当程序中途出现运行时错误时。

- **Bug 分析：**

当输出终端是文件时，只有当缓冲区满的时候，才回真正输出缓冲区的内容，并清空缓冲区。这和输出到屏幕不同，当输出到屏幕时，除了缓冲区满外，遇到'\n'会自动清空缓冲区，另外读入内容也会清空缓冲区。当写入到文件时，往往需要显示调用 fflush();

- **正确代码：**

在每个 fprintf 后面，加一行 fflush(fp);

- **Bug 定位：**

code review 过程中发现。

- **编程建议：**

显示调用 fflush 来清空输出缓冲区是一个好习惯。这有助于日志文件的形成，以便于调试。

5.2 对同一文件同时读写 💣💣

- **代码示例：**

```
FILE *fp = fopen( filename,  “r+”);
struct record rec;
while (fread( (char *) &rec,  sizeof(rec),  1,  fp) == 1) {
    /*  操作 rec*/
    if (/*需要重新写入*/) {
        fseek(fp,  -(long) sizeof(rec) ,  1);
        fwrite( (char *) &rec,  sizeof(rec) , 1 , fp);
    }
}
```

- **现象&后果：**

当重新写入 rec 值时，调用一次 fwrite 函数。紧接着又在 while 里面调用了一次 fread。对文

件又读又写，可能产生运行时错误。

- **Bug 分析：**

为了保证和过去文件不能同时读写的兼容性，在两次读写操作之间加入 fseek 函数才可以。因此在上面那个 fwrite 之后，需要加入一个 fseek 函数的调用。

fseek 的解释:

重定位流(数据流/文件)上的文件内部位置指针

int fseek(FILE *stream, long offset, int fromwhere);

函数设置文件指针 stream 的位置。如果执行成功，stream 将指向以 fromwhere（偏移起始位置：文件头 0，当前位置 1，文件尾 2）为基准，偏移 offset（指针偏移量）个字节的位置。如果执行失败(比如 offset 超过文件自身大小)，则不改变 stream 指向的位置。

成功，返回 0，否则返回其他值。

上面的代码中, fseek(fp, -(long) sizeof(rec) , 1);将位置指针向左偏移了 sizeof(rec)，再进行读 fread 的时候，从此位置开始，正常逻辑应该是在原有位置上继续读，所以需要加 fseek(fp, (long) sizeof(rec) , 1)将位置指针回到原处。

- **正确代码：**

在 if 函数的语句块最后加入一句 fseek(fp, (long) sizeof(rec), 1);

- **Bug 定位：**

code review 过程中发现。

- **编程建议：**

C 语言头文件 stdio.h 的文件操作还是很方便的，可能比 fstream 要好用一些。建议在对文件同时读写的时候，如果情况允许，建立一个临时文件。否则，一定要在两次读写之间加上 fseek 的函数调用，改变文件的状态。

fseek 函数一般用于二进制文件，因为文本文件要发生字符转换，计算位置时往往会发生混乱。

5.3 小结

C++文件处理函数的深入了解是避免出现上述问题的根本，特别是文件读写，以及二进制文件的处理。

第6章 内存使用

用 C 或 C++写程序，需要更多地关注内存，这不仅仅是因为内存的分配是否合理直接影响着程序的效率和性能，更为主要的是，当我们操作内存的时候一不小心就会出现问题，而且很多时候，这些问题都是不易发觉的，比如内存泄漏，比如悬挂指针。本章将从内存管理、内存泄漏、内存回收等方面来探讨 C++内存管理问题。

6.1 数组定义和值初始化形式混淆

- **代码示例**

```
int *ip = new int(12);
for (int i = 0; i < 12; i++) {
    ip[i] = i;
}
delete [] ip;
```

- **现象&结果**

```
产生运行时错误。Message 可能如下：
glibc detected *** free(): invalid next size (fast)
```

- **Bug 分析**

int *ip=new int(12)代表用 12 来初始化 ip 所指定的内存的变量值。

New 返回的指针 ip 是 int 类型，不是一个数组指针，赋值的时候，采用数组的方式，造成越界访问内存，并且在结束的时候用指针数组 delete,造成程序崩溃。

要把小括号改写成中括号。

- **正确代码**

```
int *ip = new int[12];
for (int i = 0; i < 12; i++) {
    ip[i] = i;
```

```
}

delete [] ip;
```

- **编程建议**

注意申请内存，尤其是连续内存与初始化的区别。最好的内存申请方法是：不去申请内存而使用标准模板库。例如，使用 vector 代替数组，但这样做付出的事效率上的代价。

6.2 数组传参时的 sizeof

- **代码示例**

```
void copy(int a[12],int b[12]) {
    memcpy(a,b,sizeof(a));
}
void del(int a[12]) {
    memset(a,0,sizeof(a));
}
```

- **现象&结果**

内存中的内容与设想不符。在未使用后面内存内容的时候表现正常，运行时出现意想不到的情况。

- **Bug 分析**

memset 和 memcpy 第三个参数是拷贝的空间大小。数组传参时，无法预知数组长度。所以 sizeof(参数)名就相当与 sizeof(type*)，大概只相当于一个 int 的大小，与初衷（全部初始化或 copy）不符。

- **正确代码**

```
void copy(int a[12],int b[12],int len) {
    memcpy(a,b,sizeof(int)*len);
}
void del(int a[12],int len) {
    memset(a,0,sizeof(int)*len);
```

```
}
```

- **编程建议**

数组传参数时，连通数组长度一起传入是一个好主意。或者用 std::vector 代替数组可以避免不必要的麻烦。

6.3 临时对象的生存期

- **代码示例**

```
class String {
    public: //…
    ~string() {
        delete [] s_;
    }
    friend String operator+(const String &, const String &);
    operator const char *() const {
        return s_;
    }
    private:
        char *s_;
};
//…
String s1, s2;
const char *p = s1 + s2;
printf("%s\n",p);
```

- **现象&结果**

在测试时，往往可能一直正常工作。但有时会出错，特别是在多线程时，会出现诡异的错误。

- **Bug 分析**

在倒数第二行，p=s1 + s2 时，生成了一个临时对象，而当赋给 p 后，该对象随之析构。所以

p 指向了一块非法内存。在中间没有别的其他操作时，那块非法内存只是被打了个“未使用”的标签，内容还没有被改变，所以可能输出正常的值。但是，没有任何方法阻止那块内容的改变，所以在插入一些其他操作后，可能输出的是随机值。

- **正确代码**

在倒数第二行上面加一行 String temp = t1 + t2; //显示给出临时对象，

然后倒数第二行改为：const char *p = temp; 。

- **编程建议**

在重构代码时要十分小心，注意临时变量的生存期。

6.4 第二次调用会覆盖第一次调用结果所占用的内存

- **代码示例**

```
char *tmp(void)
{
    static char result[30] ;
    static int flag=0;
    ++flag;
    std::sprintf(result,”%d”,flag);
    return(result);
}
char *a=tmp();
char *b=tmp();
```

- **现象&结果**

该程序本来打算获取不同的 result 字符串，但是 2 次调用之后返回的是同样的值。第二次调用会覆盖第一次调用结果所占用的内存。

- **Bug 分析**

在程序中，虽然我们定义了 a，b 这 2 个指针，但是在赋值的过程中，这 2 个指针都指向了一个相同的变量：result。第一次调用 tmp()时 a->result =”1”，第二次调用 tmp()时 b->result =”2”。但是因为 a->result 也是指向 result，所以就变成了 a->result =”2” 。

- **正确代码**

```
char a[30];
strcpy(a,tmp());
char b[30];
strcpy(b,tmp());
```

- **编程建议**

解决这个问题的方法是在每次调用后复制这个字符串，而不能仅仅声明一个指针指向它，或者是用字符数组来保存这个该字符串。

6.5 带来不确定返回值的内存越界

- **代码示例**

```
class A {
    public:
        A() {}
        ~A() {}
    void alloc(MemPool& cPool) {
        _buf = NEW_VEC(&cPool, int32_t, 2);
        _buf[0] = 0xffffffff;
    }
    private:
        int32_t* _buf;
};

class B {
    public:
        B() {}
        ~B() {};
    void alloc(MemPool& cPool) {
        _buf = NEW_VEC(&cPool, char, 1);
        strcpy(_buf, "cat_id_path");
    }
    void print() {
        fprintf(stderr, "%s\n", _buf);
```

```
    }
    private:
    char* _buf;
};

int main()
{
    A a;
    B b;
    MemPool cPool;

    b.alloc(cPool);
    b.print();
    a.alloc(cPool);
    b.print();
    return 0;
```

- **现象&结果**

查询错误。

- **Bug 分析**

_buf = NEW_VEC(&cPool, char, 1);
strcpy(_buf, "cat_id_path");分配的内存不够，赋值之后，再没有其他的内存操作时候，还不会报错。但是当再次有分配内存的操作时候，就会覆盖刚才的_buf 的后面部分。导致读取_buf，得不到正确的结果。

- **正确代码**

```
class A {
    public:
        A() {}
        ~A() {}
    void alloc(MemPool& cPool) {
        _buf = NEW_VEC(&cPool, int32_t, 2);
        _buf[0] = 0xffffffff;
    }
    private:
    int32_t* _buf;
};
```

```
class B {
	public:
		B() {}
		~B() {};
	void alloc(MemPool& cPool) {
	_buf = NEW_VEC(&cPool, char, 20);
	strcpy(_buf, "cat_id_path");
	}
	void print() {
		fprintf(stderr, "%s\n", _buf);
	}
	private:
		char* _buf;
};

int main()
{
	A a;
	B b;
	MemPool cPool;

	b.alloc(cPool);
	b.print();
	a.alloc(cPool);
	b.print();
	return 0;
}
```

6.6 非地址引用变量分配的内存

- **代码示例**

```
virtual int doStatistic(search::SearchResult *pSearchResult,StatisticResultItem *pStatResItem);
   {
	if(pStatResItem == NULL)
	pStatResItem = NEW(pMemPool, StatisticResultItem);
	pStatResItem->nCount = _statMap.size();
	StatisticHeader *pStatHeader = NEW(pMemPool, StatisticHeader);
```

```
    pStatHeader->nVersion = STATISTIC_VERSION;
    pStatHeader->nCount = _pStatParam->nCount;
    pStatHeader->szFieldName = _pStatParam->szFieldName;
    pStatHeader->szSumFieldName = _pStatParam->szSumFieldName;
    pStatResItem->pStatHeader = pStatHeader;
    pStatResItem->ppStatItems = NEW_VEC(pMemPool, StatisticItem*, pStatResItem->nCount);

}
```

- **现象&结果**

返回值不正确,查询错误。

- **Bug 分析**

*pStatResIte 作为 func 的参数，而且 func 内对指针重新 new 了一个内存地址，并进行了一系列操作。
外部在调用该 func 时，pStatResIte 并没返回 func 内部操作后的值，原因是该变量在 func 中是做为传值参数存在,而非地址引用。

- **正确代码**

```
virtual int doStatistic(search::SearchResult *pSearchResult, StatisticResultItem *&pStatResItem);
```

6.7 有关变量的作用域

- **代码示例**

```
char *s = (char*)qi->getParam("distcnt");
    if(!s) {
        char *s = NEW_VEC(mempool, char, 2*dimension);
        if(!s) {
            return -1;
        }
        for(int i =0; i<2*dimension-2; ++i) {
            s[i] = '0';
            ++i;
            s[i] = ':';
        }
        s[2*dimension-2] = '0';
        s[2*dimension-1] = '\0';
    }
```

- **现象&结果**

Coredump 。

- **Bug 分析**

原因是指针为空,指针 S 在{}外定义，初始值为空（getparam 返回空），在{}中重定义了指针，并对指针做了操作，但作用域只在{}内，当{}外使用指值 S 时，由于为空，所以就 coredump 了。

- **正确代码**

```
char *s = (char*)qi->getParam("distcnt");
    if(!s) {
        s = NEW_VEC(mempool, char, 2*dimension);
        if(!s) {
            return -1;
        }
        for(int i =0; i<2*dimension-2; ++i) {
            s[i] = '0';
            ++i;
            s[i] = ':';
        }
        s[2*dimension-2] = '0';
        s[2*dimension-1] = '\0';
    }
```

6.8 指针释放的问题

- **代码示例**

```
Int function_a()
{
    Module* A = new Module();
    ……
    function_b(A);
    function_c(A);
}
Int function_b(Module* A)
{
```

```
    Module* B = new Module();
    …..
    /**释放资源**/
delete A;
A = NULL;
    delete B;
    B = NULL;
}
```

- **现象&结果**

在 function_b 中同时释放 A 和 B，这样，调用 function_c 的时候，会发生 coredump 。

- **Bug 分析**

当模块 a 调用模块 b 的指针时，由于析构模块 a,同时将模块 b 的指针释放，此时模块 b 已经被释放掉，外部模块再调用模块 b，则会发生 coredump；模块 b 中的所有资源将会变成内存垃圾。

- **正确代码**

```
Int function_a()
{
    Module* A = new Module();
    ……
    function_b(A);
    function_c(A);
    delete A;
    A = NULL;
}
Int function_b(Module* A)
{
    Module* B = new Module();
    …..
    /**释放资源**/
    delete B;
    B = NULL;
}
```

- **编程建议**

自己负责产生的内存空间，自己负责释放。

6.9 指针在参数传递过程中的问题

- **代码示例**

```
function_a()
{
    Int* num = 10;
    function_b(num);
    …..
}
function_b(int* num)
{
    Int*   buf =0;
    ……/*计算 buf 的值*/
    num = buf;
    printf(“%d” , *num);
}
```

- **现象&结果**

经常代码中有这样的问题，将一个对象的指针传给另一个模块，需要改变指针所指向的内容，但实际上却改变了这个指针；编译不会出现任何问题，但是这个指针可能已经指向了一个危险的地方。

- **Bug 分析**

这段代码编译不会出现问题，甚至在 function_b 中打印 num 数值的时候，运算结果正确，但实际上，代码已经改变了 num 的指针，在 function_a 中，num 已经指向一个错误的位置。

- **正确代码**

```
function_a()
{
    Int* num = 10;
    function_b(num);
    …..
}
function_b(int* num)
{
    Int*   buf =0;
    ……/*计算 buf 的值*/
```

```
    *num = *buf;
    printf(“%d” , *num);
}
```

- **编程建议**

函数中传指针或引用参数，要注意修改是指针本身还是指针的内容，若不希望改指针本身，建议加 const 声明如：function_b(int * const num)。

6.10 内存释放问题 💣💣

- **代码示例**

```
void convertRankInfo()
{
RankInfo* rank = new RankInfo;
……/*对 rank 进行处理*/…
}
```

- **现象&结果**

申请的空间没有释放会造成内存溢出，编译器也不会报告问题。

- **Bug 分析**

此代码编译和运行都不存在任何问题，但在服务的生存期，此函数每执行一次，就会有一个空间申请，虽然每次浪费的空间很小，短期看不出任何内存变化，但服务运行几天后，就会出现内存耗光的现象。

- **正确代码**

```
void convertRankInfo()
{
    RankInfo* rank = new RankInfo;
    ……/*对 rank 进行处理*/…
    if(rank != NULL)
    {
        delete rank;
        rank = NULL;
    }
}
```

- **编程建议**

函数内动态分配的内存空间如果无外部指针，应在函数退出时释放空间。

6.11 不成对的 new 与 delete

- **代码示例**

```
char *buff = new char[reslen];
char *utfbuff = new char[reslen];
…
…
delete  buff;
delete  utfbuff;
```

- **现象&结果**

这段代码没有引发运行时的错误，但本身是存在隐患的。如果后续对*(buff+1)进行了误使用。便使用了本应该已经释放了的内存空间

- **Bug 分析**

其实对于基本类型如 char int 等，delete 还是 delete []是一样的，都能够释放掉内存。
但是对于自定义的类，比如 string *str = new string[10]，用 delete str 和 delete [] str 的区别是 delete str 只对 str[0]调用了析构函数，而 delete []str 则对 str 数组里的每个元素都调用了析构函数。这样如果后续对*(str+1)进行了使用，就会使用到你本来打算已经释放掉的内存空间了。
也就是说，当仅仅是 delete str 的时候，str+1，str+2,…等就成为了本应"在野"的"执政"指针。

- **正确代码**

```
char *buff = new char[reslen];
char *utfbuff = new char[reslen];;
…
…
delete [] buff;
delete [] utfbuff;
```

- **Bug 定位**

因为这个问题不会带来运行时的错误，所以在测试执行的过程中不易发现。该问题是在 code review 时发现的。这种问题，只有在 code review 过程中有着相关的敏锐度，才会抓出类似这种不见得会在运行时出错的问题。

- **编程建议**

一般来说，new[]一定要和 delete[]对应。除非你能接受后果。

6.12 函数中途退出忘记释放内存 💣💣

- **代码示例**

```
void func(char* in, int inlen)
{
	char *p = new char[20];
	if ( inlen < 20 ) {
		return 0;
	}
	strncpy(p, 20, in);
	…
	delete [] p;
	return 1;
}
```

- **现象&结果**

若 inlen<20,则函数中途退出，而未释放内存，导致内存泄露。

- **Bug 分析**

常见的错误！函数内 new 了一块内存，在函数中间退出，这时忘记释放内存，导致了内存泄漏。

- **正确代码**

```
在程序退出前添加释放内存语句 delete [] p;
if ( inlen < 20 ) {
		delete [] p;
		return 0;
	}
```

- **Bug 定位**

code review 过程中发现。

6.13 只 delete 了第一层指针，没有 delete 其包含的指针元素 💣💣

- **代码示例**

```
{
        m_pDatReaders.destroy();
        m_pCounts.destroy();
        if(m_ppSlvIndexTravel) delete[] m_ppSlvIndexTravel;
        m_ppSlvIndexTravel = NULL;
        nited = false;
}
```

- **现象&结果**

```
内存泄漏。
```

- **Bug 分析**

泄漏点在 include/build/SlvAttributeReader.h destroy()函数中，m_ppSlvIndexTravel 是二维指针，destroy 只 delete 了 ppSlvIndexTravel，没有 delete 其包含的指针元素。
执行 delete 操作的时候，有必要严格判断一下。

- **正确代码**

```
{
    m_pDatReaders.destroy();
    m_pCounts.destroy();

    if(m_ppSlvIndexTravel)
    {
            for (int i = 0; i < m_nSlvSegmentCount; i++)
             {
                    if (NULL != m_ppSlvIndexTravel[i])
                    {
                            delete m_ppSlvIndexTravel[i];
                     }
              }
             delete[] m_ppSlvIndexTravel;
      }
     m_ppSlvIndexTravel = NULL;
     inited = false;
  }
```

6.14 小结

内存管理是 C++最令人切齿痛恨的问题，也是 C++最有争议的问题，C++高手从中获得

了更好的性能，更大的自由，C++菜鸟的收获则是一遍一遍的检查代码和对 C++的痛恨，但内存管理在 C++中无处不在，内存泄漏几乎在每个 C++程序中都会发生，因此要想成为 C++高手，内存管理一关是必须要过的，除非放弃 C++，转到 Java 或者.NET，他们的内存管理基本是自动的，当然你也放弃了自由和对内存的支配权，还放弃了 C++超绝的性能。

第7章 多线程问题

本节主要介绍 C++多线程编程时出现的几个常见典型问题.并对其分析与总结。

7.1 多线程共享一个函数内的静态变量的问题 💣💣

- **代码示例：**

```
int base::func(int n)
{
	static int pre_n = 0;
	if (n>pre_n) {
		printf("new n is larger");
		pre_n = n;
		return n;
	}else{
		printf("old n is larger");
		return pre_n;
	}
	return n;
}
```

- **现象&后果：**

程序逻辑出现混乱，线程存在安全隐患。

- **Bug 分析：**

static 静态变量的两个特性:记忆性和全局性。

所谓”记忆性”是指在两次函数调用时，在第二次调用进入时，能保持第一次调用退出时的值 。下面是示例代码：

```
#include <iostream>

using namespace std;

void staticLocalVar()

{
```

```
        static int a = 0;
        cout<<"a="<<a<<endl;
        ++a;
}
int main()
{
        staticLocalVar(); //output a=0
        staticLocalVar(); //output a=1
        return 0;
}
```

所谓全局性，是指全局性和唯一性. 普通的 local 变量的存储空间分配在 stack 上，因此每次调用函数时，分配的空间都可能不一样，而 static 具有全局唯一性的特点，每次调用时，都指向同一块内存.这也就导致了另外一个问题,”不可重入性”.在多线程和递归设计上,需要特别注意.

下面是示例代码

分析在多线程情况下的不安全性.(为方便描述，标上行号)

```
① const char * IpToStr(UINT32 IpAddr)
② {
③ static char strBuff[16]; // static 局部变量，用于返回地址有效
④ const unsigned char *pChIP = (const unsigned char *)&IpAddr;
⑤ sprintf(strBuff, "%u.%u.%u.%u", pChIP[0], pChIP[1], pChIP[2], pChIP[3]);
⑥ return strBuff;
⑦ }
```

假设现在有两个线程 A,B 运行期间都需要调用 IpToStr()函数，将 32 位的 IP 地址转换成点分 10 进制的字符串形式. 现 A 先获得执行机会，执行 IpToStr()，传入的参数是 0x0B090A0A，顺序执行完应该返回的指针存储区内容是:”10.10.9.11”，现执行到⑥时，失去执行权，调度到 B 线程执行，B 线程传入的参数是 0xA8A8A8C0，执行至⑦，静态存储区的内容是 192.168.168.168. 当再调度到 A 执行时，从⑥继续执行，由于 strBuff 的全局唯一性，内容已经被 B 线程冲掉，此时返回的将是 192.168.168.168 字符串，不再是 10.10.9.11 字符串.

使用 static 变量可以在函数里保存上次的处理结果。但是注意，如果是在多线程环境中，多线程会共用同一个 static 变量。

也就是说，线程 1 的处理结果，可能被线程 2 当成上次的处理结果。即：

如果多线程环境下使用静态变量，那么可能这个静态变量就被多个线程来回修改，就不是线程安全的。如：A 线程的状态会被 B 线程读取到。

- **正确代码：**

解决方法是，为了线程安全，使用类成员变量保存函数上次处理结果。

```
class base
{
    …
    int m_pre_n;
}
int base::func(int n)
{
      if (n>pre_n) {
            printf("new n is larger");
            m_pre_n = n;
            return n;
      }else{
            printf("old n is larger");
            return pre_n;
      }
      return n;
}
int base::reset()
{
      m_pre_n=0;
}
```

同时，别忘了初始化赋值，而且重置线程数据时需要重新初始化赋值。

- **Bug 定位：**

code review 发现 static 变量且为多进程调用，则有线程共享的问题。

- **编程建议：**

关于多线程间共享同一个静态变量的文章，可以参见：http://www.cppblog.com/suiaiguo/archive/2009/07/23/90940.html 。

7.2 string 线程安全问题 💣💣

- **代码示例：**

```
void* thread_a(void* ptr)

{

        ((std::string*)ptr)->append("apple");

}

void* thread_b(void* ptr)

{

        ((std::string*)ptr)->append("orange");

}

…

pthread_t pid;

std::string global_str;

pthread_create(&pid, NULL, thread_a, &global_str);

pthread_create(&pid, NULL, thread_b, &global_str);
```

- **现象&后果：**

程序的目的是两个线程循环地向一个公共变量中添加字符串”apple”和”orange”，不在乎 apple 和 orange 出现的顺序和多少。某些时候，公共变量 global_str 会有”apporangele”的子串，甚至会出现 Segmentation fault。

- **Bug 分析：**

使用 gdb 查看产生的 core 文件，可以看到程序 core 在了 memcopy。先看一下 append 函数的实现原理，append 主要有两步，首先会先开辟一段内存，然后再将旧的拷贝过来，因此 append 并非是原子的，当然也不是线程安全的。当两个线程的冲突发生在拷贝阶段，共享变量中会出现非预期的结果，如果冲突发生在开辟内存阶段，就会出现段错误。

- **正确代码：**

对共享变量的写操作加上锁

如下:

```
pthread_mutex_t      mutex;

void* thread_a(void* ptr)

{
```

```
            pthread_mutex_lock(&mutex)
            ((std::string*)ptr)->append("apple");
            pthread_mutex_unlock(&mutex);
}
void* thread_b(void* ptr)
{
pthread_mutex_lock(&mutex)
            ((std::string*)ptr)->append("orange");
            pthread_mutex_unlock(&mutex);
}
…
pthread_t pid;
std::string global_str;
pthread_create(&pid, NULL, thread_a, &global_str);
pthread_create(&pid, NULL, thread_b, &global_str);
```

- **Bug 定位：**

code review 发现多线程操控全局数据时的是非原子性操作。

- **编程建议：**

多线程编程时，如果存在多个线程并行访问同一变量时,需要考虑对其的访问保护问题。

7.3 小结

使用多线程可以把占据长时间的任务放到后台执行,并且能够使程序的运行速度更快,但是资源的共享也会带来线程安全问题.线程安全大都是由全局变量及静态变量引起的.多线程的应用中,需要考虑不同线程之间的数据同步和防止死锁。

第8章 异常处理

本节主要介绍 C++多线程异常处理的几个常见典型问题，并对其分析与总结。

8.1 异常引发的异常 💣💣

- **代码示例：**

```
class a{
      private:
      int count；
      public：
      a(void):count(0){};
      set_count(int value){ count = value; }
        ~a(void) {
              if(count!=0)
              throw(problem(“error”));
      }
};
void main(){
      a  a1;
      a1.set_count(1);
      …
      throw(problem(“error”));
      …
}
```

- **现象&后果：**

当走到异常分支时，程序并不能正确处理这个异常，反而会 coredump。

- **Bug 分析：**

这里的问题是嵌套产生了 2 个异常：当 main 函数执行到错误分支时会抛出一个异常，这时程序就由异常代码控制，它负责销毁所有的局部变量，包括变量 a。当 a 被销毁时，会调用析构函数~a。而析构函数中也会抛出一个异常。这时，程序就会调用 terminate()函数。

- **正确代码：**

```
~a(void) {
    if(count!=0)
    Cout<<"error"<<endl;
}
```

- **Bug 定位：**

code review 过程中发现。

- **编程建议：**

如果想捕获第二个异常，并解决其他相似的问题，可以使用标准的 set_terminate 函数去创建一个专门负责处理意料外的异常。

也可以加 try catch 自己捕获自己的异常。比如

```
        ~a(void)
          {
            try{
                  if(count!=0)
                  throw(problem(“error”))
                }
            catch(…){}
}
```

在有两种情况下会调用析构函数。第一种是在正常情况下删除一个对象，例如对象超出了作用域或被显式地 delete。第二种是异常传递的堆栈辗转开解（stack-unwinding）过程中，由异常处理系统删除一个对象。

现在假设在异常被激活的情况下（比如某个地方抛出了一个 MyException）调用某个析构函数，而这个析构函数又抛出了一个异常（比如叫 AnotherException），那么现在应该怎么办？是忽略这个新的 exception 继续找能够 catch MyException 的地方呢还是扔掉 MyException 开始找能够处理 AnotherException 的地方呢？

因此 C++语言担保，当处于这一点时，会调用 terminate()来杀死进程。所以，如果在析构函数中抛出异常的话就不要指望谁能 catch 了。

禁止异常传递到析构函数外有两个原因，第一能够在异常转递的堆栈辗转开解（stack-unwinding）的过程中，防止 terminate 被调用。第二它能帮助确保析构函数总能完成我们希望它做的所有事情。

http://hi.baidu.com/elliott_hdu/blog/item/50cf341f0b30caf0e0fe0b25.html

http://blog.csdn.net/zdl1016/article/details/4204245

8.2 小结

程序编写人员总是希望自己所编写的程序都是正确无误的,而且运行结果也是完全正确的,但是这几乎是不可能的,智者千虑必有一失,因此程序编写这不仅要考虑程序没有错误的理想情况,更要考虑程序存在错误时的情况,应该能够尽快发现错误消除错误,这就是 C++的异常处理机制. 如何正确使用异常机制是非常重要的。

第9章 性能问题

本节主要介绍了编程中的一些细节引起的性能问题，问题虽小，影响却很大，下面我们就来一一看下吧。

9.1 strlen 用作循环条件

- **代码示例：**

```
const char *p = "1234567890";

for( i=0; i<strlen(p); i++)

{
    …
}
```

- **现象&后果：**

时间复杂度本来应该是 O(n)，但是这样写就成了 O(n^2)，当字符串 p 较长时，时间浪费非常严重。

- **Bug 分析：**

for(i=0; i<strlen(p); i++)，每次循环的时候都要计算一次 strlen(p)

1
1<strlen(p)
2
2<strlen(p)
3
3<strlen(p)
…
显然，如果字符串长一万，那就要循环一万次，算一万次 strlen，而算一次 strlen 需要一万次加法，就是一万乘以一万次加法，这样的代码是非常低效的，因此，应使用一个变量存储 strlen(p)的值来作循环判定条件。

- **正确代码：**

```
int len = strlen(p);
```

```
for(int i=0; i<len; i++)
```

- **Bug 定位：**

code review 过程中发现。

- **编程建议：**

提高代码的执行效率：尽量减少循环判断语句的复杂度。

9.2 STL 中 list 容器慎用 size 💣💣

- **代码示例：**

```
list<int> lst;

…

for (int i = 0; i < lst.size(); i++) {

    …

}
```

- **现象&后果：**

STL 中 list 的复杂度是 O(n)，而在每次循环中都去做 i < lst.size()的判断，所造成的复杂度将是 O(n2)，对性能是极大的损伤，尤其是在 list 中的元素非常多的情况下。

- **Bug 分析：**

gcc list 的 size()的实现如下：

```
size_type size() const {

    size_type __result = 0;
    distance(begin(), end(), __result);
    return __result;
}
```

可以看到，它是用 distance(begin()， end())来计算的。而 std::distance 的实现中，按照 iterator 类型的不同，实现的方式也不同。list 的 iterator，是属于双向 iterator，而非随机 iterator，因此，在 std::distance()中使用了一个循环来计算值。也就是说在 gcc 的 stl 库中，每次调用 list::size()函数,它都会从头到尾遍历一遍。

- **正确代码：**

```
int len = lst.size();
for(int i=0; i<len; i++)
```

- **Bug 定位：**

code review 过程中发现。

- **编程建议：**

与上一 case 系同类问题，循环越多影响越大。

与此类似的另外一种用法也要注意：

```
list<int> lst;
…
while(lst.size() > 0) {
    …
```

}这里我们可以用 while(!lst.empty())来代码，因为 empty()的复杂度只是 O(1)。

9.3 频繁的 new、delete 操作 💣💣

- **代码示例：**

```
_parser_t::content_t * _parser_t::get_parse_cont(const char *pline, int lineno) {
        //分配内存
    content_t *p_cont = new content_t();
    if (NULL != p_cont) {
        p_cont->pcontent = pline;
        p_cont->lineno = lineno;
    }
    return p_cont;
}
```

- **现象&后果：**

频繁的 new 和 delete 小块的内存会带来内存碎片的问题，造成不必要的内存浪费。

- **Bug 分析：**

这是一个性能问题。上述代码用来按行 get 监控文件的 parse_cont,当监控文件行数过大时，调用 get_parse_cont 函数的那个函数就会频繁的调用它，进而产生频繁的 new 和 delete 操

作。频繁的 new 和 delete 小块的内存会带来内存碎片的问题。使用 memorypool 来处理该种情况。

- **正确代码：**

持续的分配小内存，易产生内存碎片，应使用 memorypool。

- **Bug 定位：**

code review 过程中发现。

- **编程建议：**

Memorypool 相关资料可参考：

C++ Memory Pool http://www.codeproject.com/KB/cpp/MemoryPool.aspx

C++ Memory Pool 汉译版 http://blog.csdn.net/060/archive/2006/10/08/1326025.aspx Object Pooling for Generic C++ classes http://www.codeproject.com/KB/cpp/objpool.aspx

9.4 std::vector clear && 内存回收 💣💣

- **代码示例：**

```
std::vector<int> vec;

for(int i = 0; i <1000; i ++) {

    vec.push_back(i);

}

vec.clear();
```

- **现象&后果：**

在使用过程中， 程序占有的内存往往不降或者越来越多，导致浪费系统资源。

- **Bug 分析：**

可以在clear()之后先看一下vector的大小:vec.size()以及vector所占用的空间:vec. capacity()，发现它们分别是 0 和 1024。可以看到虽然 vec 的元素个数为 0 了，但是所占的内存空间依然为 clear 之前的样子。

clear()函数做的事情只是把元素全部删除掉，但并没有去释放元素所占的内存。std::vector 的内存如果增长上去，通过 clear 无法保证释放内存。所有空间只会在 vector 析构的时候才会回收。

- **正确代码：**

```
vec.swap(vector<int>());
```

- **Bug 定位：**

对应用程序进行压力测试时，会发现程序占用的内存逐渐上升；当压力停止后一段时间，内存平稳并且不会下降。

- **编程建议：**

虽然 swap 可以强行释放内存空间，但服务程序建议不要使用 vector。因为对于服务程序来说，内部的变量，除非是临时的，否则都不能轻易去 swap。

可以考虑使用 deque，在内存上它不同于 vector 的只增不减。不过，deque 根据编译器的不同,也不全部会自动回收内存，这一点也是需要注意的。

vector 在内存的动态分配上是 2 的阶乘递增的，如你的 vector 里面已经占用了 1024 个空间，现在再往里面插入一个元素，系统就会给你分配 2048 个空间；然后你再删除一个元素，虽然你实际只占了 1024，但系统并不会将你浪费的 1024 回收回来。举个实际的例子，一个服务程序里面用 vector 来存在在线的用户，在用户高峰的情况下，你的用户量可能达到了 10 亿，它占用了你上 1G 的内存，而后到达低谷，在线用户可能只有几百个，你会惊奇地发现，你的进程占用的内存依然这么大。

9.5 小结

从上面几个例子，我们可以从两方面进行下总结。一方面是对我们使用的函数、方法的复杂度要有一定的了解。令一方面是编程思维，频繁调用的代码，一定要足够简介高效，否则一定会对性能造成影响，反之，提高频繁调用的代码的执行效率，能大大的优化程序的性能。

第10章 跨平台问题

本节主要介绍 C++代码跨平台的几个常见典型问题，对其分析与总结。

10.1 std::string 类的查找返回值 💣💣

- **代码示例：**

```
bool Find(const string &s,const string &t) {
    return s.find(t)>=0;
}
```

- **现象&后果：**

大多数编译器下，可以正常运行。但是与平台相关。如果测试平台与开发平台一致，很难通过黑盒测试发现问题。

- **Bug 分析：**

因为命名 string::npos 是一个宏定义，未必是-1。尽管大多数情况下它是-1。类似的还有指针空类型 NULL 的定义，未必就是 0。

```
static    const size_type    npos = static_cast <size_type> (-1);
```

- **正确代码：**

改为 return s.find(t) != string::npos;

- **Bug 定位：**

code review 过程中发现。

- **编程建议：**

不要依赖于命名空间、头文件中的宏定义。

10.2 NULL 的问题 💣💣

- **代码示例：**

```
void doIt( char * );

void doIt( void * );

doIt( NULL ); // platform-dependent or ambiguous
```

- **现象&后果：**

代码在不同平台上移植可能出错。

- **Bug 分析：**

NULL 的定义在不同的平台上可能是不同的，如：

```
#define NULL ((char *)0)

#define NULL ((void *)0)
```

另外使用 NULL 必须包含标准的头文件.因此使用 NULL 可能带来代码移植的问题。

- **正确代码：**

用 0 代替 NULL 。

- **Bug 定位：**

code review 过程中发现。

- **编程建议：**

参考文档：《C++编程指南》 第六章 表达式和语句 ，

网址链接：http://ir.hit.edu.cn/~car/programming/rup/manuals/cpp/cpp.htm 。

10.3 不必要的类型转换 💣💣

- **代码示例：**

```
char *buffer;

char *curr_pos;

int length;

...

while( (*(curr_pos++) != 0x0a) &&

((UINT)curr_pos - (UINT)buffer < (UINT)length) );
```

- **现象&后果：**

从 64 位机上迁移到 32 位机上，算法达不到预期，不具有平台可迁移性。

- **Bug 分析：**

这段代码的目的是将定位一行的末尾即回车符 0x0a，这段代码不会处理比 INT_MAX 还长的字符串因为 length 变量是 int 型的。假设我们的输入在 int 长度范围内，现在的问题是在 64 位机上 buffer 和 curr_pos 的范围可能会指到 4G 之外。这里将 curr_pos 和 buffer 指针显式转换为 UINT，转换时会将高位的 4 个字节移除掉，而这些很有可能是一些有意义的位。

- **正确代码：**

```
while(curr_pos – buffer < length && *curr_pos != \n')

    curr_pos++;
```

- **Bug 定位：**

code review 过程中发现。

- **编程建议：**

32 位机和 64 位机上对某些基础类型的长度是有区别的，如 long 类型在 32 位机上是 4 个字节，而在 64 位机上是 8 个字节。变化最大的其实是指针，64-bit 机上 cpu 可寻址的范围是 32-bit 的两倍，相应一个内存地址的长度也成了 2 倍，故而指针也变了。

10.4 不同编译器对函数参数计算的顺序不同 💣💣

- **代码示例：**

```
dowith(int x)

{

    …

}

X=0;

dowith(x,++x);
```

- **现象&后果：**

在不同的编译器下，上述函数的调用的可能是 dowith(0,0)，也可能是 dowith(1,1)。代码与编译器（平台）相关，可以移植性变差。

- **Bug 分析：**

相关标准并没有规定，函数参数的计算顺序。所以，我们的代码不应该依赖于编译器计算参数的顺序。

比如:

```
main()
{
int i=8;
printf("%d\n%d\n%d\n%d\n",++i,--i,i++,i--);
}
```

如按照从右至左的顺序求值。运行结果应为：

8

7

7

8

如对 printf 语句中的++i，--i，i++，i--从左至右求值，结果应为：

9

8

8

9

- **正确代码：**

根据需要。先++x，或者先调用 dowith(x,x)。

- **Bug 定位：**

code review 过程中发现。

- **编程建议：**

建议不要在函数调用的时候改变作为参数变量的值。

10.5 小结

C/C++的编译是分为两个部分的，即编译和连接:源文件经编译后生成目标文件，然后再连接为可执行文件，通常情况下 C++程序不适合跨平台.但是通过良好的设计和编写规范代码，跨平台是可以实现的。

特别感谢

最后再次特别感谢一下白皮书小组的几个成员：穆岚，茹冰，水媚，青丝，泰官，从雪。他们几个人为白皮书最终面世付出了日常工作之外的很多时间和精力。

<EOF>