

程序设计实践与技巧（一） ——界面、风格和排错

北京大学信息科学技术学院

张 路

2017年10月

大纲

- ▶ 接口界面 (Interface)
- ▶ 编程风格 (Style)
- ▶ 软件排错 (Debugging)

界面的责任

- ▶ 为调用者提供服务和访问
- ▶ 向调用者隐藏实现细节
- ▶ 管理资源的分配、释放和共享
- ▶ 错误处理



界面的分类

▶ 程序界面

- 类和函数的接口

▶ 用户界面



程序界面:为别人用的库

► 考虑的问题

- 提供的服务
 - 统一方便又不过多过滥，基本操作正交，风格相同
- 信息隐藏
 - 哪些可见？哪些私用？访问方式？实现细节？
- 资源管理
 - 谁管理内存分配和释放？共享信息要拷贝吗？
- 错误处理
 - 谁检查错误？报告还是忽略？怎么报告？有何恢复性操作？
- 制定规范

程序界面设计原则(1)

——单一功能原则

▶ 什么是单一功能原则？

- 一个函数完成的功能要单一
- 一个类处理的事务范围要单一
- 例如，STL栈的pop, top （没有 ptop）

▶ 从提供的服务上分析

▶ 从代码的修改角度分析

▶ 参考：

- 《Head first 面向对象分析与设计》
- 《Head first 设计模式》



程序界面设计原则(2)

——开放/封闭原则

▶ 什么是开放/封闭原则？

- 一个类应该对扩展开放，对修改关闭。
- 对新的功能的扩充，应该通过增加新类实现，而不是修改已有类的代码

▶ 目的

- 降低程序各部分之间的耦合性，利于程序模块替换。
 - 使软件各部分便于单元测试。
 - 软件升级时可以只部署发生变化的部分，而不会影响其它部分。
-

程序界面设计原则(3)

——一致性和规范性原则

- ▶ 什么是一致性和规范性原则？
- ▶ 实际是设计风格问题
- ▶ 如STL容器提供了一致的界面，即使面对一个不熟悉的函数，预计应该如何使用它也会变得很容易



程序界面设计原则(4)

——完整、安全、高效、简短

- ▶ 完整：提供所需的基本的合理功能
- ▶ 安全：使用安全的算法，其次是效率
 - memcpy, memmove
 - memcpy: WORD对齐、cacheline对齐、预取，非常高效；但src、dest两个数组不能重叠
- ▶ 高效且简短：
 - 函数个数尽量少，功能不要重叠，达到“多快好省”

程序界面设计原则(5)

——保护性编程、DbC和Exception

- ▶ **保护性编程**：每个函数内部检查输入是否合法
- ▶ **Design by Contract (DbC)**：每个函数只保证在输入合法的情况下正确输出
- ▶ **Exception**：利用Exception机制可以做到两者折中

程序界面设计原则(6)

——依赖倒置

- ▶ 小粒度的复用 vs. 大粒度的复用
- ▶ 利用动态绑定

程序界面设计原则(7)

——使用Immutable Class

- ▶ 基本类型 vs. 自定义类型
- ▶ 举例: **String**
- ▶ 效率问题

用户界面设计

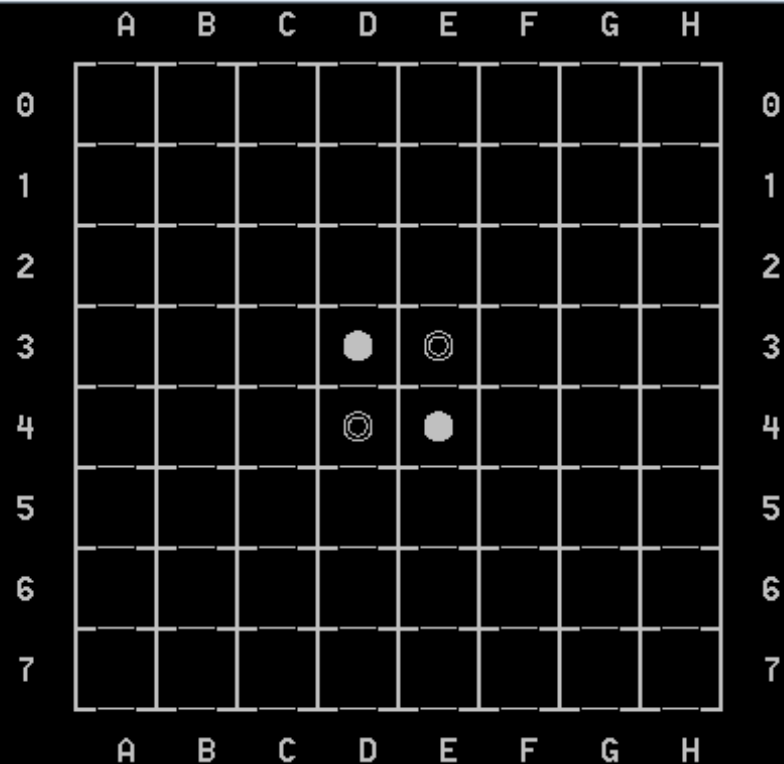
▶ 文本用户界面

- 提示信息的完整性

▶ 图形用户界面(GUI)

- 术语、单位、格式、字体、颜色统一和规范
- 显示内容的可拷贝性



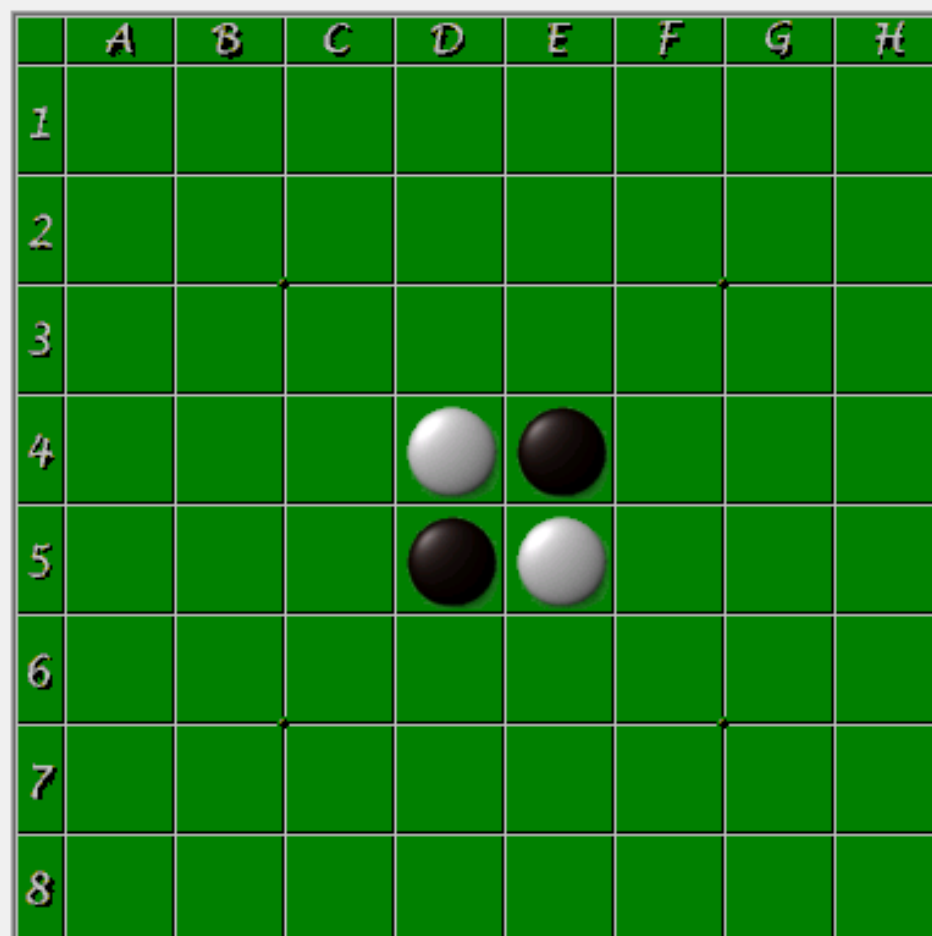


欢迎使用简易版黑白棋对战小程序 祝您玩得愉快 !!

1. 选择落子
2. 新开始
3. 读盘
4. 存盘
0. 结束

请输入0-5中的某个数字：请按任意键继续 . . .

设置(S) 选项(O) 帮助(H)



黑棋：2 白棋：2 你先走！

大纲

- ▶ 接口界面
- ▶ 编程风格
- ▶ 软件排错


```

int main() {
    int qq, i, j, tot, pre;
    scanf("%d", &qq);
    while(--qq>=0){

memset(stock,0,sizeof(stock));
    scanf("%d%d",&side,&n);
    max=0;
    min=0;
    tot=0;
    for(i=0;i<n;++i){
        scanf("%d",&j);
        ++stock[j];
        if(j>max) max=j;
        if(j<min) min=j;
        tot+=j*j;
        ...
    }
}
}

```

```

bool Case::Cutable(COORDINATE leftTop, int piece){
    ...
}
void main() {
    int caseNum;
    cin>>caseNum;
    Case cutCake;
    while ((caseNum--)>0){
        cutCake.Input();
        cutCake.Output()
    }
}

```

什么是程序设计风格？

▶ 程序设计风格 (Programming Style)

编写程序时使用的规则集合

▶ 规则示例：

变量命名方式：qq? caseNum?

语句排版等：{} 对齐方式？语句缩进？

注释：提高代码可读性和可维护性
等等

为什么需要注意风格？

- ▶ 阅读代码方便
- ▶ 自己的代码适合别人学习
- ▶ 大型程序的维护与修改
- ▶ 利于团队合作

额外的工作量为了减少工作量！

培养良好的代码风格

我们必须：

- ▶ 现在处于学习阶段，不能太懒；
- ▶ 团队合作要求统一的风格；
- ▶ 从最切身的利益讲：有利于助教或者老师批改；

所以我们的代码需要风格！

程序设计风格

- ▶ 命名
- ▶ 语句
- ▶ 注释
- ▶ 文档

命名

- ▶ 命名是程序风格中最重要的部分，也是初学者最容易忽略的部分。
- ▶ 一个好的变量命名应该满足：
 - 词能达意： `COORDINATE` , `leftTop`
 - 表明身份：变量？函数？全局？局部？常量？类？宏？...
 - 存储类型： `int`？ `float`？ `char`？

命名

- ▶ 词能达意——标识符应当直观，可望文知意

如“获得字符串的长度”的函数，下面两个名称哪个更好？

hdzfcdd()

getLength()

请尽量使用英文，而不是拼音（或拼音简写）来命名

命名

- ▶ 词能达意——标识符应当直观，可望文知意



命名

► 表明身份

- `variable`: 局部变量
- `g_variable`: 全局变量
- `m_variable`: 成员变量
- `doSomething()`: 函数
- `CONSTANT_VARIABLE`: 常量

命名

▶ 存储类型

- `Char chGrade`
- `BOOL bEnable`
- `Int nLength`
- `WORD wPos`
- `LONG lOffset`

▶ 匈牙利命名法

变量名 = 身份 + 类型 + 对象描述

匈牙利命名法的缺点：

- ▶ 忽视了用抽象数据类型作为基本类型
- ▶ 另外，如果改变机器字长、变量类型等，整体都要改
- ▶ 现在有了很好的IDE工具，如：VC, SourceInsight等. 选中变量，会自动提示
- ▶ 代码书写习惯比强制使用更重要
- ▶ 系统性、整体性、读性，分类清楚有注释！

注意

1. 标识符的长度应当符合“**min-length && max-information**”原则
2. 命名规则尽量与所采用的**操作系统或开发工具的风格**保持一致。
Windows: AddChild
Unix: add_child
3. 用正确的反义词组命名具有互斥意义的变量或相反动作的函数等
如minvalue, maxvalue()

注意

4. 尽量避免名字中出现数字编号，如 Value1, Value2 等，除非逻辑上的确需要编号
5. 要用到某个常数时，最好设置一个**常量**来代替这个数字
6. 尽量不要用全局或文件范围变量。但是允许采用全局范围内的类型定义（包括类定义）

注意

7. 程序中不要出现仅靠大小写区分的相似的标识符
8. 程序中不要出现标识符完全相同的局部变量和全局变量，尽管两者的作用域不同而不会发生语法错误，但会使人误解
9. 全局函数的名字应当使用“动词”或“动词+名词（动宾词组）”。类的成员函数应当只使用“动词”，被省略掉的名词就是对象本身。

如 `DrawBox()` //全局函数
 `box_Draw()` //类的成员函数

注意

10. 变量的名字应当使用“名词”或“形容词+名词”，
如 `float value;`
`float oldvalue;`
`float newvalue;`

史上最糟糕的两个变量名data、total

- ▶ <http://petdance.com/2012/04/the-worlds-two-worst-variable-names/>
- ▶ `total = price * qty;`
- ▶ `total2 = total - discount;`
- ▶ `total2 += total2 * taxrate;`
- ▶ `total3 = purchase_order_value + available_credit;`
- ▶ `if (total2 < total3) {`
- ▶ `print "You can't afford this order.";`
- ▶ `}`

语句

► 语句的风格主要包括：

缩进：

空格： `int b = a + c;` vs `int b=a+c;`

空行：

- 两类元素（类、函数等）定义之间增加空行
- 逻辑上独立的代码片段前后用空行
- 连续的两个多行定义之间用空行隔开
- 多行定义和其他代码之间应该用空行隔开

不要滥用空格

- ▶ 不要在单目运算符和操作对象间加空格，如：

`sizeof (str);` => `sizeof(str);`

`(int) a+b` => `(int)a+b`

- ▶ 不要在引用操作符.、->、[]前后加空格

`a [i]` => `a[i]`

`pNode-> leftTree!=0` =>

`pNode->leftTree=0`

语句

- ▶ 少用具有二义性或者很难理解的语句:

`i+++i;`

`stra[i++] = strb[i++] = ' ';`

- ▶ 少用多用途的复合表达式

`d=(a=b+c)+r` //既求a值又求d值

- ▶ 不要把程序中的复杂表达式与“真正的数学表达式”混淆

`a<b<c`是真正的数学表达式,

而程序中的逻辑表达式应该是`a<b && b<c`

语句

▶ 不要吝啬括号

```
while((c=getchar())!=EOF)
```

括号的用途：

- (1) 显式确定运算的优先级
- (2) 提高代码的可理解性，减少误解
- (3) 减少错误

语句

▶ 大括号（建议前一种）

<code>if (condition)</code>	<code>while (condition)</code>
<code>{</code>	<code>{</code>
<code>...</code>	<code>...</code>
<code>}</code>	<code>}</code>

V. S.

<code>if (condition) {</code>	<code>while (condition) {</code>
<code>...</code>	<code>...</code>
<code>}</code>	<code>}</code>

注意

1. { } 之内的代码块在 ‘{’ 右边数格处左对齐。
2. 代码行最大长度宜控制在70 至80 个字符以内。

注意

1. 长表达式拆分

- ① 在低优先级操作符处拆分成新行
- ② 操作符放在新行之首（以便突出操作符）
- ③ 拆分出的新行要进行适当的缩进，使排版整齐，语句可读。

if语句（与零值比较）

► 布尔型变量与零值比较

- 正确的风格

- `if (value) // 表示value为真`
- `if (!value) // 表示value为假`

- 不应将布尔型变量直接与`true`、`false`, 或者1、0 比较, 即不要写成

- `if (value == true)`
- `if (value == 1)`
- `if (value==false)`
- `if (value==0)`

if语句（与零值比较）

▶ 整型变量与零值比较

- 应当将整型变量用“==”或“!=”直接与0比较。
 - `if (value == 0)`
 - `if (value != 0)`
- 不应模仿布尔变量的风格而写成
 - `if (value)` // 会让人误解 `value` 是布尔变量
 - `if (!value)`

if语句（与零值比较）

▶ 浮点变量与零值比较

- 不可将浮点变量用“==”或“!=”与数字0比较。
- `if (x==0.0)` 是隐含错误的比较！
- `if ((x>=-EPSINON) && (x<=EPSINON))`, 其中EPSINON是允许的误差（即精度）

if语句（与零值比较）

▶ 指针变量与零值比较

- 应当将指针变量用“==”或“!=”与NULL比较。

- `if (p == NULL)`

- `// p 与NULL 显式比较，强调p 是指针变量`

- `if (p != NULL)`

- 不良风格

- `if (p == 0)` // 容易让人误解p 是整型变量

- `if (p != 0)`

- `if (p)` // 容易让人误解p 是布尔变量

- `if (!p)`

注意

1. `switch`不要忘记`break`和最后的`default` 分支。

2. 参数命名要恰当，顺序要合理。

- `void StringCopy(char *str1, char *str2);`
- 修改参数名: `strSource` 和`strDestination`。
- 顺序呢? 先目的参数, 后源参数
- 如果写成这样:

```
void StringCopy(char *strSource, char  
*strDestination);
```

注释

- ▶ 初学者往往注重完成特定的算法功能，而忽视了注释以及代码可读性、可维护性
- ▶ 注释应当是编码的一部分。没有注释，编码不算完整。写出好的注释如同写出好的代码，需要经验积累与素养
- ▶ 注释要清晰、简洁，并且有价值。

注释

► 注释通常用于

- 版本和版权声明

- 版权信息
- 文件名称, 标识符, 摘要
- 当前版本号, 作者/修改者, 完成日期
- 版本历史信息

- 函数接口说明

- 重要的代码行或段落提示

```
/*  
* Copyright(c) 2001,上海贝尔有限公司网络应用事业部  
* All rights reserved.  
*  
* 文件名称: filename.h  
* 文件标识: 见配置管理计划书  
* 摘要: 简要描述文件的内容  
*  
* 当前版本: 1.1  
* 作者: 输入作者 (或修改者) 名字  
* 完成日期: 2001年7月20日  
*  
* 取代版本: 1.0  
* 原作者: 输入原作者 (或修改者) 名字  
* 完成日期: 2001年5月20日  
*/
```


注释的规范

1. 使用`//`，因为`/*... */`不支持嵌套注释。
2. 长注释应和代码分在不同的行。
3. 如果注释掉大段代码，请使用`‘//’`注释代码，不要使用`‘/*...*/’`。用`‘/*...*/’`作注释可能会导致嵌套注释，当被注释掉的代码块很大时更容易出现这种情况，这样可能导致注释掉的区域不是我们想要的范围。

-
- **确保所有注释（随代码）及时更新。**一定要牢记注释是代码的一部分，所以修改代码时，相应的注释也要改。没有及时更新的注释会误导代码阅读和维护，甚至产生严重的副作用。
 - 应当准确、易懂，防止注释有二义性。错误的注释不但无益反而有害。
 - 尽量避免在注释中使用缩写，特别是不常用缩写。
 - 当代码比较长，特别是有多重嵌套时，应当在一些段落的结束处加注释，便于阅读。


0

// 写这段代码的时候
// 只有上帝和我知道它是干嘛的
// 现在，只有上帝知道




1

// 虽然我也不想这样做，
// 但是有个讨厌的家伙就要我这样做，
// 迟早会证明，他是错的。
// 2011-12-21



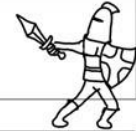
2

/* 如果你能弄清楚这一段，求
求你务必发邮件至aaa@bbb.com
告诉我，不胜感激 */



3

/*勇士，乐观地接收这个现实吧！
2011-11-23*/



4

// 这个不知道是谁写的
// 看起来没用
// 但我不敢删

5

// 去他妈的。就这样发布吧。




6

/* 魔法。勿动！ */




7

/* 致终于来到这里的勇敢的人：
你是被上帝选中的人，英勇的、不辞劳苦的、不眠不休地来修改我们这最棘手的代码的编程骑士。你，我们的救世主、人中之龙，我要对你说：永远不要放弃，永远不要对自己失望，永远不要逃走。永远不要哭泣，永远不要说再见。永远不要说谎来伤害自己。*/



8

// 亲爱的维护者：
// 如果你尝试了对这段程序进行“优化”，
// 并认识到这种企图是大错特错，
// 请增加下面这个计数器的个数，
// 用于对后来人进行警告：
// 人们总共在这里浪费了 = 39h



原载于stackoverflow
翻译改编自《外刊IT评论》

文档

- ▶ 文档是一个良好程序不可分割的一部分。一个程序能否被广泛使用很大程度长觉得于它的文档的质量。
- ▶ 文档应该包含：
 - 问题背景；
 - 问题分析；
 - 问题求解；
 - 问题的结果分析；
 - 程序的性能；
 - 最后的总结；

文档示例

2. 行优先搜索，判断可否放下只需检查一行↵

每次要放下新的正方形时，必须判断当前位置是否可以放。不可以放有两种情况。一种是**越界**，用一条语句判断就可以了：↵

```
if (x + i > side || y + i > side) return false;↵
```

(x,y)是当前要填的小格，i是选的正方形，side是目标边长。↵

另一种情况是新的正方形会否与原来已放下正方形重叠。这里只需要检查一行。这是因为我们的枚举是按行优先的。每次找到要放入的格子是所有为空的格子里面，行数最小的，如果有多个行数同样最小，就是列数最小那个。容易证明，在前一种越界情况已排除的前提下，边长为i的正方形能够放入以(x,y)为左上角的区域，当且仅当，(x,y+j) $0 \leq j < i$ 这i个格子为空。我们用下面的代码判定：↵

```
for (j = 0; j < i; ++j)↵
```

```
    if (map[x][y + i])↵
```

```
        return false;↵
```

(x,y)是当前要填的小格，i是选的正方形。↵

需要注意的文档风格

- ▶ 张贴大量代码，甚至全部代码，而没有说明

解题代码如下：↵

```
#include<iostream>↵
```

```
using namespace std;↵
```

```
↵
```

```
void cutCakes( int k);    //切第 k 块蛋糕↵
```

```
int lie[42] = { 0 }, sum[ 11 ] = { 0 }; //用 lie[42]存放模拟的蛋糕状况↵
```

```
bool isOk = false;                //sum 存放大小为角标大小的蛋糕块数↵
```

```
int s = 0 , n = 0;                //s 指输入的蛋糕尺寸 , n 指 n 个人分 n 块蛋糕↵
```

```
↵
```

```
void cutCakes(int k)              ↵
```

```
{↵
```

```
    int i = 0 , j = 0 , x = 1 ;↵
```

```
    ↵
```

```
    if( k == n + 1 )    // n 块蛋糕全摆好 , 成功↵
```

```
    {↵
```

```
        isOk=true;↵
```

总结

- ▶ 命名
- ▶ 语句
- ▶ 注释
- ▶ 文档

说明

- ▶ 一般说来，程序设计的风格只有“相对更好”，没有绝对的严格规范。这里给出的**建议**只是推荐使用
- ▶ 这里主要讨论程序设计的风格，至于编写更漂亮和高效的代码，请参考附录的书籍

大纲

- ▶ 接口界面
- ▶ 编程风格
- ▶ 软件排错

code monkey之歌



错误(bug)类型

- ▶ 编译错误 (**Compiling error**)
- ▶ 链接错误 (**Link error**)
- ▶ 异常错误
 - ▶ **Runtime error**
- ▶ 逻辑错误 (算法错误)
 - ▶ 程序运行结果不对



如何减少错误发生？

- ▶ 追求简单优雅的程序设计结构
- ▶ 减少程序各部分耦合
- ▶ 完全了解函数的输入输出要求后再使用
- ▶ 避免使用不安全的语法
 - ▶ 少用pointer多用reference
 - ▶ 取消goto
- ▶ 一些**notes**
 - ▶ 尽量不用全局变量
 - ▶ 注意变量初始化
 - ▶ 尽量使用**const**
 - ▶ 详尽的注释

没有哪种方法能防止你犯错误!!!



重视排错

- ▶ 排错时间至少和写程序一样长
- ▶ 正常运行的程序不是没有错误,只是还没发现错误
- ▶ 尽量少借助排错系统
- ▶ 多进程多线程分布式系统要靠自己的经验和推理能力排错
- ▶ 找错误就像猜谜一样,可以是件愉快的事情



有线索，简单错误

- ▶ 大部分编译和链接错误

- ▶ 重视warning

- ▶ 检查最近的改动

- ▶ 取得堆栈轨迹和变量值，确定出错位置



简单代码容易出错

▶ **if(flag=1)**

▶ **if(x&y)**

▶ **if(...);**

▶ **int k;**

...;

cout<<a[k];

for (i=0; i<10; i++)

for (j=0;j<10;i++)

▣ **char str[10];**
str[1]='a';
str[2]='b' ;str[3]='c';
cout<<str<<endl;

▣ **int a[20];**
for (int i=0; i<20;
i++)
a[i] = a[i+1];

改正简单错误

- ▶ 读程序,而不是马上改程序
- ▶ 休息一下,有时你看到的代码实际是自己的意愿,不是实际写出来的东西
- ▶ 改正一个错误后,不要急着编译运行,看看是否在别处也有类似错误



无线索、难办的错误

- ▶ 把错误弄成可以重现的
- ▶ 对代码分而治之
- ▶ 加入打印语句，使搜索局部化
- ▶ 写自检测代码，错误排除后注释掉它或用排错选项控制它
- ▶ 研究错误的统计特征
- ▶ 检查宏定义的函数



不可重现的错误

- ▶ 检查日志和调试输出，确定大概出错位置，对多线程以及长时间运行的服务器程序尤其有用
- ▶ 检查变量初始化
- ▶ 当程序出现不可理解的异常时，多半是某个地方有内存越界，或存储分配错误
- ▶ 养成习惯：
 - ▶ 动态分配内存后，检查是否分配成功
 - ▶ 释放内存后把指针指向**NULL**，释放前检查指针是否为**NULL**
- ▶ <http://www.yolinux.com/TUTORIALS/C++MemoryCorruptionAndMemoryLeaks.htm>
|



最后的办法

- ▶ 单步跟踪
- ▶ 把你的代码解释给别人，甚至是一只玩具熊
- ▶ 检查外部环境
- ▶ **MSDN, Google**



最后最后的办法

- ▶ 引用的库错了
- ▶ 标准库错了
- ▶ 移植问题
- ▶ 硬件故障



编程环境中的调试技巧

- ▶ 肉眼检查

如: `for(i=n;i>=1;i++)`

- ▶ 注释——增 / 删注释

- ▶ 设置断点

- ▶ 执行跟踪

- ▶ 条件语句、循环内部

- ▶ 各种监视窗口

- ▶ 调试辅助代码

- ▶ 循环、递归、`printf()`、`getchar()`



针对POJ的调试

- ▶ 程序在执行时卡住无法输出结果，或输出非常混乱，全屏乱码
 - ▶ 对 数组和循环 做 断点+单步跟踪
- ▶ 程序不会卡死，但测试数据通不过
 - ▶ 首先检查算法、然后用插入辅助代码和单步跟踪
 - ▶ 检查变量混淆、数组大小、逻辑错误、初始清零
- ▶ 测试数据通过、提交wrong answer
 - ▶ 算法细节疏漏、测试数据不够、次优解
- ▶ 测试数据通过，提交 runtime error —— 细节！
 - ▶ 最可能是数组下标越界，new/delete，野指针等
- ▶ Time Limit Exceeded:
 - ▶ 算法不过关(也可能实现时有问题...), 算法不够优化。

参考文献

- ▶ 高质量**C++/C** 编程指南 林锐 博士
- ▶ 程序设计实践 Brian W. Kernighan, Rob Pike.
- ▶ C++ Coding Standard, Andrei Alexandrescu, Herb Sutter
- ▶ Effective C++

谢谢大家！

- ▶ 教材：张铭 赵海燕 王腾蛟 宋国杰，《数据结构与算法实验教程》，国家十一五规划教材，高教社 2011 年 1 月

国家精品课“数据结构与算法”

<http://www.jpk.pku.edu.cn/pkujpk/course/sjg/>
