Efficient Code Debugging

"Systems Skills in C and Unix"

CPL 课程助教 宁锐

2024 年 11 月 14 日

rning@smail.nju.edu.cn

南京大學

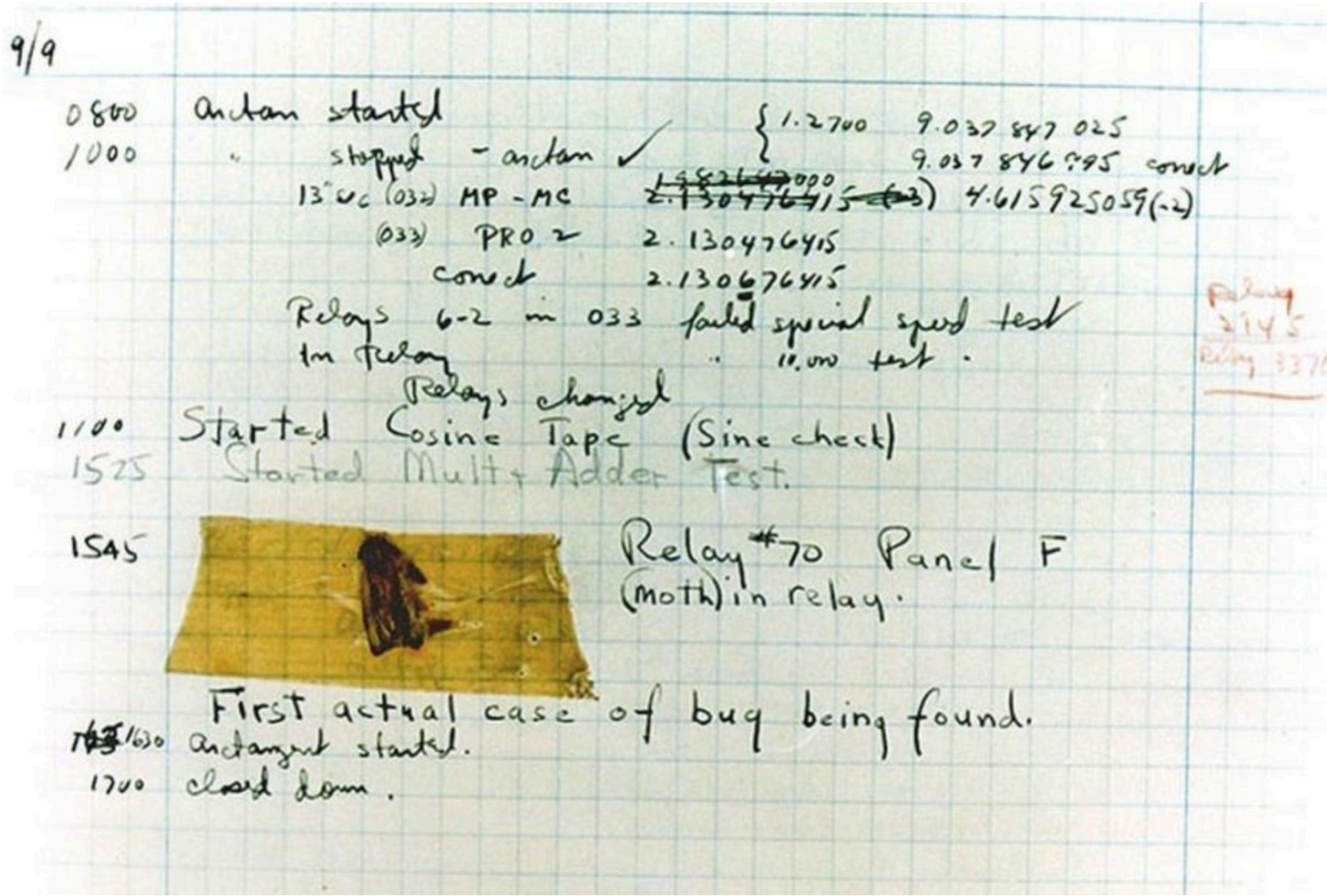
NANJING UNIVERSITY

CONTENTS

如何高效调试

1. 调试理论初探

1. 调试理论初探

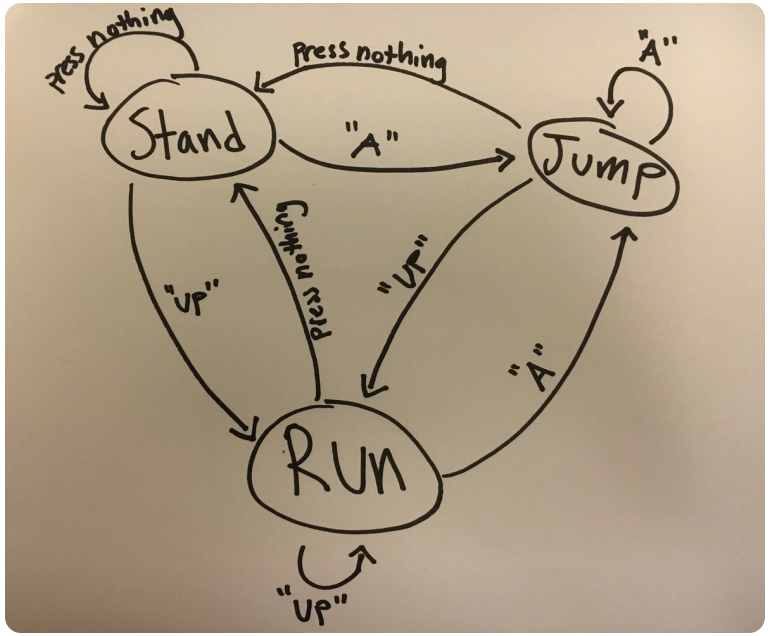

9/9
0800 Arctan started
1000 " stopped - arctan ✓
1.2700 9.037 847 025
9.037 846 795 correct
13"UC (032) MP - MC 4.615925059(-2)
(033) PRO 2 2.13047615
correct 2.13067615
Relays 6-2 in 033 failed special speed test
In relay " 11.00 test.
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Multy Adder Test.
1545
Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1700 closed down.

1. 调试理论初探

- **一切程序皆是有限状态机 (Finite State Machine)**
  - 局部变量和全局变量的取值构成了有限状态机的状态空间 (State Space)
  - 程序中的控制流语句构成了有限状态机的转移函数 (Transition Function)
- **当我们已经知道 bug 的存在**
  - Segmentation Fault
  - Wrong Answer

怎么找到程序中的 bug？

```c
int main(void) {
    int a, b, c;
    scanf("%d%d%d", &a, &b, &c);
    if (a == 0) {
        printf("No final project for cpl\n");
    } else if (b + c % 3 == 0) {
        printf("No final exam for cpl\n");
    } else {
        printf("You failed the final exam\n");
    }
    return 0;
}
```



2. 调试器的基本操作

2. 调试器的基本操作

- **开始之前：** 你需要准备好VS Code，或者CLion(推荐)。如果你使用VS Code，请确保你的调试环境可以正常使用。
- **一些术语：**
  - 栈 (Stack)：所有局部变量存放的地方
  - 堆 (Heap)：使用malloc分配的内存所存放的地方
  - 栈帧 (Frame)：函数作用域中的所有变量（包含函数体内声明的所有变量和函数参数）
  - 断点 (Breakpoint)：调试过程中程序暂停的点
  - 步过 (Step Over)：执行当前所暂停的语句，并停在当前暂停语句的下一个语句
  - 步入 (Step Into)：对于函数调用语句，进入被调用函数。对于非函数调用语句，同步过
  - 步出 (Step Out/Finish)：立即执行完当前所在函数的所有剩余语句（类似continue）

下面演示CLion中调试器的基本用法~

0-1 比特翻转（flip.c）

- 基本操作
- Memory View
- Expression Evaluation

3. Runtime Error 的调试思路

3. Runtime Error 的调试思路：为什么RE?

- **究竟什么是Runtime Error？**
  - 简而言之，当你的程序返回值不为0时，Online Judge会告诉你运行错误。
- **那么什么时候你的程序返回值不为0？**
  - main函数 return 非0值
  - Segmentation Fault: 非法内存访问
    - 程序向只读区域写入数据 (向字符串常量中写入数据)
    - 程序向不存在的区域写入或读取数据 (对空指针 NULL 解引用)

显而易见的，第一种情况是我们可以控制的，但是第二种情形往往难以发现。

3. Runtime Error 的调试思路：如何揪出Bug?

- **杀鸡焉用牛刀：GNU Debugger(GDB)**

  GDB会**自动终止**在异常发生的代码处!

```
gdb ./test
```

- **达摩克利斯之剑：Address Sanitizer**

```
gcc -o test -fsanitize=address -g test.c
```

  Address Sanitizer 是Google开发的一个C/C++内存错误检测工具，通过在编译时插入探测代码来检测:

  - 缓冲区溢出（`*-overflow`）: `stack-buffer-overflow`, `global-buffer-overflow` ...
  - 释放后使用 (`*-use-after-*`): `stack-use-after-return`, `use-after-poison` ...

下面演示GDB (CLion中)和ASan的基本用法~

数独检验（sudoku.c）

- SIGSEGV结束的程序
- 缓冲区溢出错误
- 释放后使用错误

3. Runtime Error 的调试思路：题外话

为什么不是所有的非法内存访问都会触发SIGSEGV？

例如下面这段程序，在绝大多数的电脑上这段程序都能成功运行并寿终正寝，几乎不会触发 Segmentation Fault。

```c
int main() {
    int arr[10];
    arr[10] = 0;
    return 0;
}
```

3. Runtime Error 的调试思路：题外话

为什么不是所有的非法内存访问都会触发SIGSEGV？

可以把内存看作是一座大房子（大小通常为`*GB`），每个房间（大小通常为`4K`）都有明确的用途（分配给用户/操作系统）和权限（只读/可读可写）。操作系统就像是这座房子的管家，负责管理房间的使用情况。编译器就像是房子里的装修队。

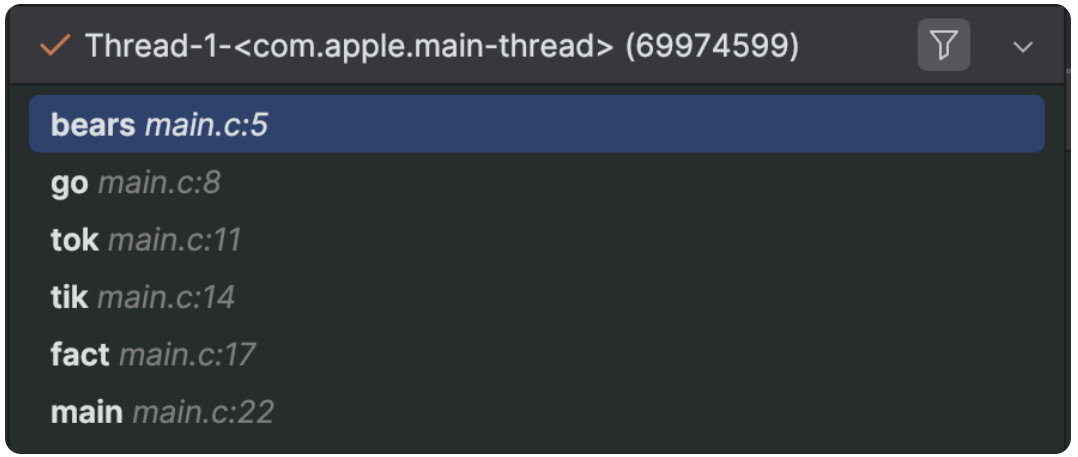
当你调用一个函数时，装修队（编译器）会在“栈”这个房间（栈不一定只有一个房间）里放入（push）一些数据。函数执行完后，这些数据会被移走（pop），栈指针（类似于房间内某个标记）会回退到之前的位置。这个栈指针可以看作装修队在房间里的一个分界线，指示当前“装修”到哪儿了。

3. Runtime Error 的调试思路：题外话

为什么不是所有的非法内存访问都会触发 SIGSEGV？

随着函数调用的嵌套/局部变量的增多，栈的大小需要增长，管家（操作系统）发现之后就会一次性分配给施工队（编译器）一个完整的房间（4K页面），但你只用到了部分空间，比如说房间的前半部分。

这时，房间的后半部分虽然你没有申请使用，但它依然在房子里，并且管家没有明确把它锁起来（实际上也无法上锁）。

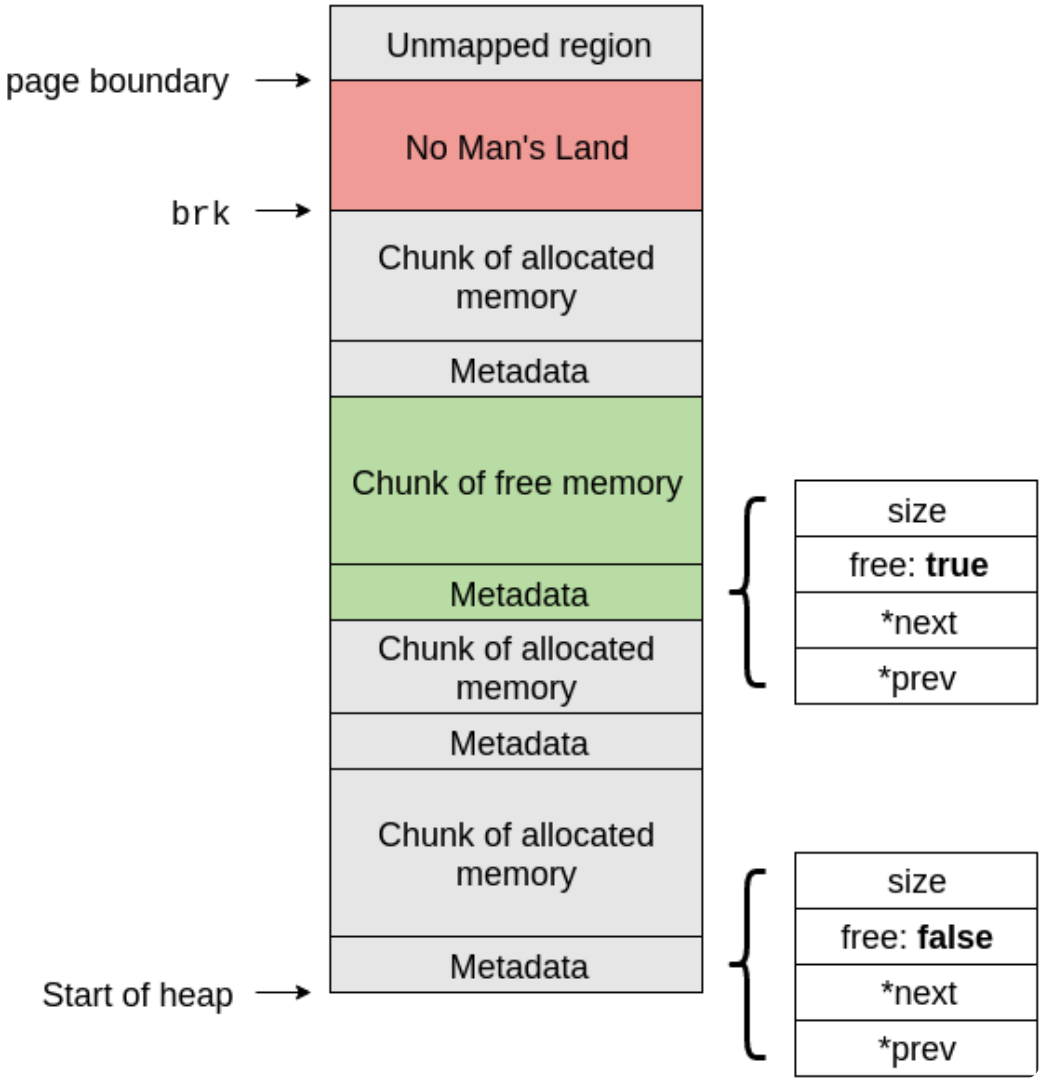


3. Runtime Error 的调试思路：题外话

为什么不是所有的非法内存访问都会触发 SIGSEGV？

如果你误闯入这块未锁的区域（比如访问 栈指针和`4K`页面边界之间的内存），管家并不会马上过来阻止你，因为这块区域虽然理论上你不应该用，但实际上它还是在房子范围内，所以不会触发报警（SIGSEGV）。

只有当你跑到房子外面（完全未映射的内存区域）或者去一个明确上了锁的地方（向写保护的区域写入数据/从不可读的地址读入数据），管家才会报警（触发 SIGSEGV）。



4. Time Limit Exceeded 的调试思路

4. Time Limit Exceeded 的调试思路：为什么TLE?

- 代码逻辑中存在死循环
- 代码效率低下

4. Time Limit Exceeded 的调试思路：如何揪出Bug?

- **杀鸡焉用牛刀：GNU Debugger(GDB)**

  巧用Debugger的`continue`和`pause`功能定位代码中的死循环。

```
gdb ./test
```

- **调试不仅仅是调试：Google Performance Tools (gperftools)**

  利用gperftools生成流图分析代码中函数/每一行的开销。

```
# Compile with libprofiler
gcc -g -Wl,--no-as-needed,-lprofiler,--as-needed -o example example.c
# Run with some specific env var
LD_LIBRARY_PATH=/usr/local/lib/ CPUPROFILE=./example.prof ./example
# Generate flow graph
google-pprof -pdf -line example example.prof > example.pdf
```

下面演示GDB (CLion) 和gperftools的基本用法~

- 定位死循环
- 分析CPU时间开销

5. Memory Limit Exceeded 的调试思路

5. Memory Limit Exceeded 的调试思路：为什么MLE?

- 代码使用过多内存
- malloc的内存在使用结束后没有正确释放

5. Memory Limit Exceeded 的调试思路: 如何揪出Bug?

代码使用过多内存

下面这段代码使用多少内存?

```
uint32_t arr[1024 * 1024];
int main() {
  memset(arr, 0xff, sizeof arr);
  return 0;
}
```

5. Memory Limit Exceeded 的调试思路: 如何揪出Bug?

malloc的内存在使用结束后没有正确释放

```c
int main() {
  void* addr = malloc(4096);
  // free(addr);
}
```

- 达摩克利斯之剑：Leak Sanitizer

  ```
  gcc -o test -fsanitize=leak -g test.c
  ```

  LSan 是一个运行时内存泄漏检测器，它可以与 ASan 结合使用以同时获得检测内存访问错误和内存泄漏的能力，它也可以单独使用。（在MacOS上ASan和LSan合用似乎无法正常运行？）

下面演示LSan的基本用法~

6. Wrong Answer 的调试思路

6. Wrong Answer 的调试思路

一些小技巧（建立在你已经找到了错误例子的前提下！）

- 巧用`freopen`, 不用多次输入同样数据
- 巧用`watchpoint`, 观察特定位置的数据何时被哪条语句修改
- 巧用`conditional breakpoint`, 让循环/函数调用停在特定的条件下
- 巧用`ignore/until`, 替代多次Step Over
- 重视CLion中的每一行黄色波浪线Warning

下面演示GDB (CLion) 的上述进阶用法~

7. 如何不用调试写出正确的代码

7. 如何不用调试写出正确的代码

我们先看一个著名的笑话：

> 三个程序员被要求穿过一片田地，到达另一侧的房子。
> 菜鸟程序员目测了一下之间很短的距离，说：“不远！我只要十分钟。”
> 资深程序员看了一眼田地，想了一会，说：“我应该能在一天内过去。”菜鸟程序员很惊讶。
> 大神程序员看了一眼田地，说：“看起来要十分钟，但我觉得十五分钟应该够了。”资深程序员冷笑了一声。
> 菜鸟程序员出发了，但只过了一会，地雷爆炸了，炸出了巨大的洞。这下他必须偏移预定的路线，原路返回，反复尝试穿过田地。最后他花了两天到达目的地，到的时候颤颤发抖，还受了伤。
> 资深程序员一出发就匍匐前进，仔细地拍打地面，寻找地雷，只有在安全的时候才前进。他在一天的时间内小心谨慎地缓慢爬过了这片地，只触发了几个地雷。
> 大神程序员出发之后径直穿过了田地，十分果断。他只用了十分钟就到了另一边。
> “你是怎么做到的？”另外两个人问道，“那些地雷怎么没有伤到你？”
> “很简单，”他回答道，“我最初就没有埋地雷。”

7. 如何不用调试写出正确的代码

一般来说，我们写程序，都是这样的步骤：

1. 写出程序
2. 提交测试
3. 通过测试查找错误，回到 1 修补，如果没发生问题就假装程序写好了

7. 如何不用调试写出正确的代码

但是，为什么我们要劳心劳力的去debug？我们不把错误放进程序，不就好了？

或者，我们更进一步，为什么要写程序？把程序种子埋土里，然后过几天等程序长大了，就收掉，不就行了？

你先别笑，这的确是编写复杂程序的一个方法。

7. 如何不用调试写出正确的代码

更具体的说，当我们程序运行到一半，假设我们随机修改一下变量的值，很明显程序不会没问题的继续运行下去。

比如说，如果我们在写一个冒泡排序，当我们运行的时候，把数组下标最小的值和下标最大的值交换了一下，那接下来排序结果不一定正确。

```
for (int i = 0; i < n; i++) {
  // Find index of least element in [i, n)
  int min_idx = i;
  for (int j = i; j < n; j++)
    if (arr[j] < arr[min_idx])
      min_idx = j;

  // Swap arr[i] and arr[min_idx]
  swap(arr + i, arr + min_idx);
}
```

7. 如何不用调试写出正确的代码

也就是说，我们的程序需要遵守一定的规则(invariant)。对于上面的例子来说，我们需要遵守的invariant是，每次进入循环前，在循环体中的i指示的数组元素之前的所有元素都是正确排列的（相对于排好序的结果）。

所以，我们写程序的时候，要保证这些invariant的正确性。但是，我们可以反过来，从这些invariant，推导出一个程序！

7. 如何不用调试写出正确的代码

更具体的说，我们

1. 写下程序输入跟输出符合的条件，一般来说，输入的条件并不等价于输出的条件，所以这时候，这两个条件之间，是有一个空缺的
2. 根据这些规则，猜出一小段代码，这些代码会把这些空缺变得更小，然后我们不停进行这个操作，直到空缺消失。这时候，程序就编写完成了。

这时候，由于invariant给予了程序很大的限制，只要我们指定了invariant，剩下的程序，往往就没有什么‘选择’，只有一个写法了，于是写起来就很简单！这个方法，叫做stepwise refinement。

下面演示如何利用Invariant解决问题~

- 魔法阵 (quick-sort.c)

欢迎交流～

https://ryani.org/resources/debugging.pdf