

计算机系统基础  
Programming Assignment

# About Debugging, Plagiarism, and Asking Questions

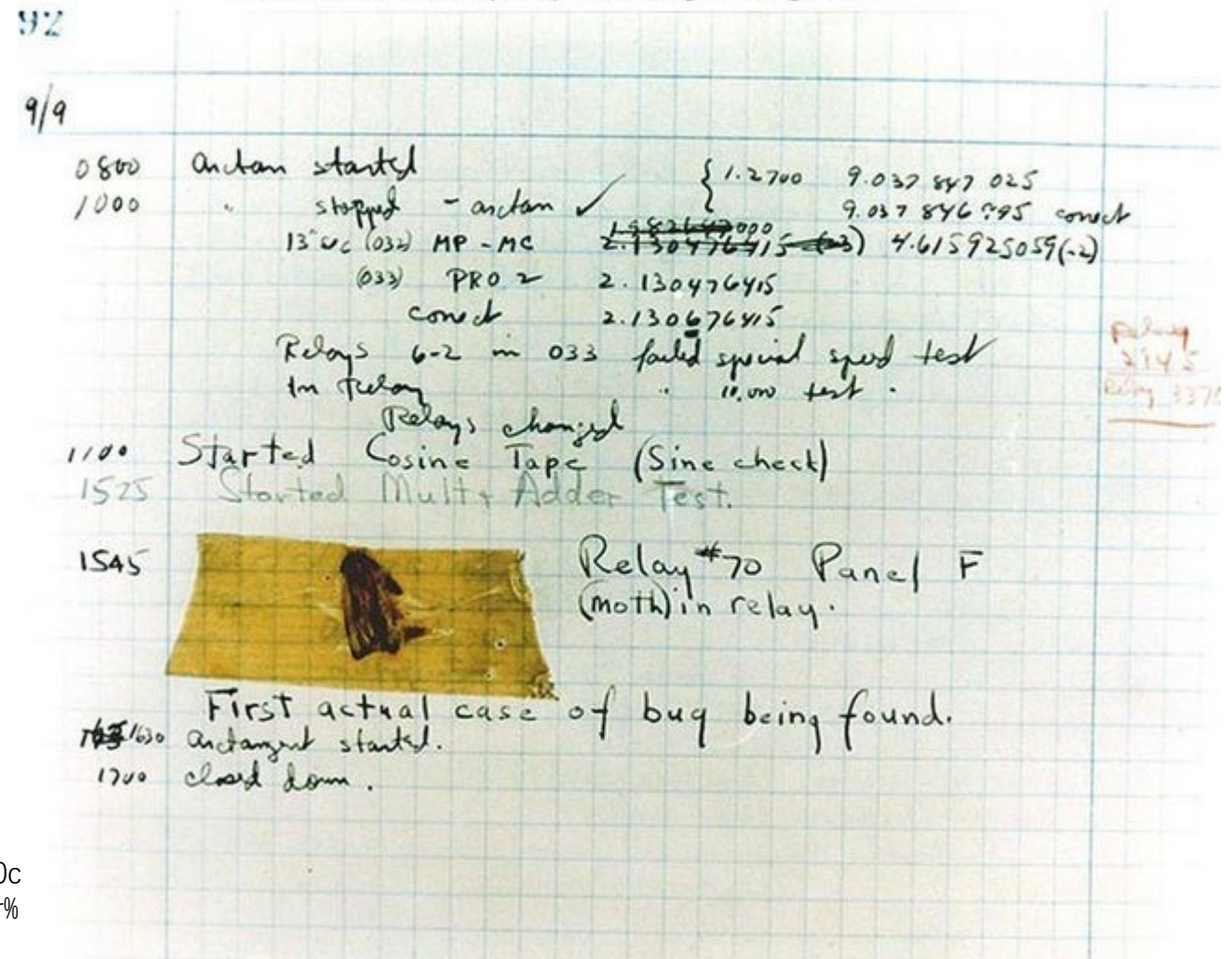
2020年11月5日

南京大学《计算机系统基础》课程组

# Sep 9, 1947 CE: World's First Computer Bug

On September 9, 1947, a team of computer scientists reported the world's first computer bug—a moth trapped in their computer at Harvard University [1].

Photo # NH 96566-KN (Color) First Computer "Bug", 1947



[1] <https://www.nationalgeographic.org/thisday/sep9/worlds-first-computer-bug/#:~:text=On%20September%209%2C%201947%2C%20a%20team%20of%20c,computer,functioning%2C%20safety%2C%20and%20security%20of%20computer%20operating%20systems.>

# Bugs can cause nasty consequences

## A Collection of Well-Known Software Failures [2]

Software systems are pervasive in all aspects of society. From electronic voting to online shopping, a lot of things rely on software. And, unfortunately, there are a lot of software bugs.

### Table of contents

- [Economic Cost of Software Bugs](#)
- [Knight Capital's \\$440 million loss](#) \*\*\*\*
- [Microsoft Zune's New Year Crash](#) \*\*\*\*
- [Air-Traffic Control System in LA Airport](#) \*\*\*\*\*
- [Northeast Blackout](#) \*\*
- [NASA Mars Climate Orbiter](#) \*\*\*\*
- [Denver Airport Baggage-handling System](#) \*
- [Therac-25](#) \*
- [USS Yorktown Incident](#) \*\*\*\*
- [Ariane 5 Explosion](#) \*\*\*\*
- [A List of Security Bugs](#)

[2] <http://www.cse.psu.edu/~gxt29/bug/softwarebug.html>

# Debugging is often a difficult task

- **Classify bugs by the ease of repeating their failures [3]**
  - Bohrbugs
    - Bohrbugs, like the Bohr atom, are solid, easily detected by standard techniques, and hence boring.
  - Heisenbugs
    - But Heisenbugs may elude a bugcatcher for years of execution. Indeed, the bugcatcher may perturb the situation just enough to make the Heisenbug disappear. This is analogous to the Heisenberg Uncertainty Principle in Physics.

```
#include <stdio.h>

void func(int a, int b)
{
    printf("a = %d, b = %d\n", a, b);
}

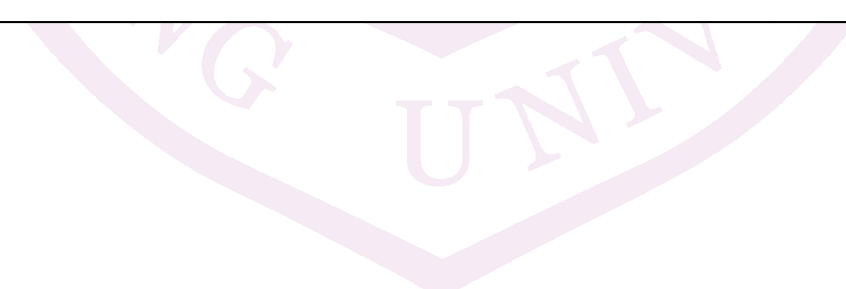
int main()
{
    int t = 5;
    func(t++, ++t);
    return 0;
}
```

```
#include <stdio.h>

int check(int x)
{
    printf("x = %d\n", x);
    return x;
}

void func(int a, int b)
{
    printf("a = %d, b = %d\n", a, b);
}

int main()
{
    int t = 5;
    func(check(t++), check(++t));
    return 0;
}
```



```
#include <stdio.h>

void func(int a, int b)
{
    printf("a = %d, b = %d\n", a, b);
}

int main()
{
    int t = 5;
    func(t++, ++t);
    return 0;
}
```

a = 6, b = 7

```
#include <stdio.h>

int check(int x)
{
    printf("x = %d\n", x);
    return x;
}

void func(int a, int b)
{
    printf("a = %d, b = %d\n", a, b);
}

int main()
{
    int t = 5;
    func(check(t++), check(++t));
    return 0;
}
```

x = 6

x = 6

a = 6, b = 6

# 发现Bug时的正常心路历程 (type-I)

第一阶段：**不可能是我的错!** 一定是框架代码、编译器、操作系统、虚拟机、CPU……里有bug!

第二阶段：嗯……似乎这里有一点<sub>小问题</sub>，但是不至于吧~

第三阶段：当初这代码怎么能跑起来的!!!! ? ? ? ?

# 调试公理

机器永远是对的

未测试代码永远是错的





# 初学者发现Bug时的正常心路历程 (type-II)

第一阶段：啊……又出错了呢，呵呵



# 初学者的调试公理

是人就会犯错

Debug不是一件轻松的事情

踩遍所有坑，成就伟大程序员



# Debug是有一些方法的

- 认识bug
- 少写点bug
- 科学debug



# Debug是有一些方法的

- 认识bug
- 少写点bug
- 科学debug

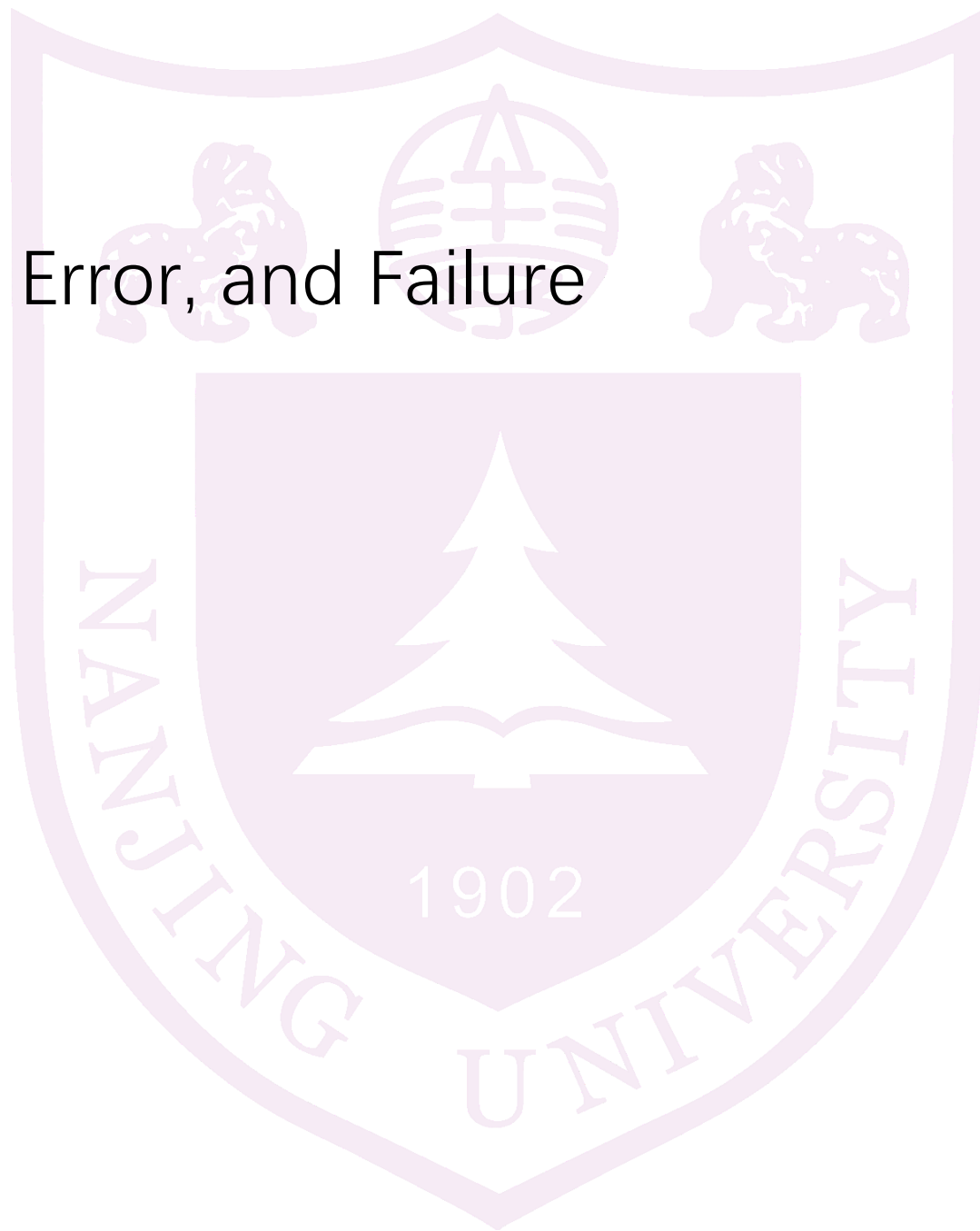


# 认识bug

三个基本术语：Fault (Defect), Error, and Failure

我们能看到的，叫failure

- Segmentation Fault
- Floating Point Exception
- Hit Bad Trap
- ...

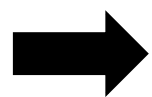


# 认识bug：从Fault到Failure

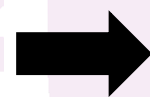
- 一个Bug是怎样最终导致程序出错的？

*从程序中的Fault到Failure的传播过程*

有一个Fault  
也就是Bug



运行时导致  
了一个Error



最后导致了  
程序的  
Failure

ALU中sub的标志位设置逻辑错误

cmp指令的结果错误，jcc指令跳转到了错误的分支

Hit Bad trap  
Segmentation Fault

# Debug是有一些方法的

- 认识bug
- 少写点bug
- 科学debug



## 少写点bug（纯个人经验）

- 良好的编程风范
- 保持随手检查和测试的习惯





# 少写点bug：良好的编程风范

- 其一：KISS

- Keep It Simple and Stupid

```
us = -32768;  
i = us;  
printf("i = %d, 0x%08x\n", i, i);
```

Smart

```
us = -32768;  
ui = us;  
printf("ui = %d, 0x%08x\n", ui, ui);
```

```
us = -32768;  
i = sign_ext(us, 16);  
printf("i = %d, 0x%08x\n", i, i);
```

Stupid



```
us = -32768;  
ui = zero_ext(us, 16);  
printf("ui = %d, 0x%08x\n", ui, ui);
```

sign\_ext()函数我们单独测试过

用笨（Stupid）但是保险的方法来写代码

# 少写点bug：良好的编程风范

- 其一：KISS

- Keep It Simple and Stupid

普通实现

精简实现



会出现  
大量相  
似的重  
复代码

```
make_instr_func(mov_r2rm_v) {  
    OPERAND r, rm;  
    // 指定操作数长度  
    rm.data_size = r.data_size = data_size;  
    int len = 1;  
    // 操作数寻址  
    len += modrm_r_rm(eip + 1, &r, &rm);  
    // 执行mov操作  
    operand_read(&r);  
    rm.val = r.val;  
    operand_write(&rm);  
    // 返回操作数长度  
    return len;  
}
```

```
#include "cpu/instr.h"  
  
static void instr_execute_2op() {  
    operand_read(&opr_src);  
    opr_dest.val = opr_src.val;  
    operand_write(&opr_dest);  
}
```

`make_instr_impl_2op(mov, r, rm, v)`

[nemu/src/cpu/instr/mov.c](#)

同样指  
令同样  
的逻辑

一行对应  
一条指令  
的实现

代码要尽量简洁（Simple的其中一种解读）

# 少写点bug：良好的编程风范

- 其二：好看吗？好看就是好代码！
  - 统一的排版
    - 统一的缩进
    - 一行只写一句
    - 一个函数不超过一屏
  - 函数/变量名命名方式
    - 下划线方式：mov\_r2rm\_v, instr\_execute\_2op, operand\_write
    - 驼峰命名法：movR2rmV, instrExecute2op, operandWrite
    - 不要混用，例如，约定函数采用下划线法，局部变量采用驼峰
    - 起有意义的名字
  - 写注释

# 少写点bug：保持随手检查和测试的习惯

- 其三：墨菲定律：有可能出错的事情，就会出错

```
void set_CF_add(uint32_t result, uint32_t src, size_t data_size) {  
    result = sign_ext(result & (0xFFFFFFFF >> (32 - data_size)), data_size);  
    src = sign_ext(src & (0xFFFFFFFF >> (32 - data_size)), data_size);  
    cpu.eflags.CF = result < src;  
}
```

## 想得到的地方都设置检查点

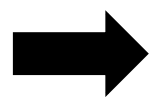
```
void vaddr_write(vaddr_t vaddr, uint8_t sreg, size_t len, uint32_t data) {  
    assert(len == 1 || len == 2 || len == 4);  
    laddr_write(vaddr, len, data);  
}
```

# 少写点bug：保持随手检查和测试的习惯

- 其三：墨菲定律：有可能出错的事情，就会出错

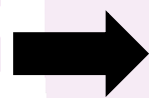
从程序中的Fault到Failure的传播过程

有一个Fault  
也就是Bug



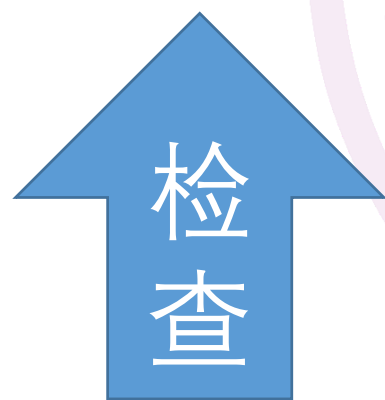
快

运行时导致  
了一个Error



慢

最后导致了  
程序的  
Failure



让Error立即变成Failure  
被我们看到

# 少写点bug：保持随手检查和测试的习惯

- 其三：墨菲定律：有可能出错的事情，就会出错

```
uint32_t hw_mem_read(paddr_t paddr, size_t len)
{
    uint32_t ret = 0;
    memcpy(&ret, hw_mem + paddr, len);
    return ret;
}
```

检查过paddr和len的合法性吗？

```
void hw_mem_write(paddr_t paddr, size_t len, uint32_t data)
{
    memcpy(hw_mem + paddr, &data, len);
}
```

# 少写点bug：保持随手检查和测试的习惯

- 其四：全覆盖的测试是不现实的，但不意味着可以不测

## 浮点运算的单元测试

测试边界条件

```
float input[] = { p_zero.fval, n_zero.fval, p_inf.fval, n_inf.fval, denorm_1.fval, denorm_2.fval,
big_1.fval, big_2.fval, p_nan.fval, n_nan.fval, denorm_3.fval, small_1.fval, small_2.fval,
10000000, 1.2, 1.1, 1, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1, -0.1, -0.2, -0.3, -0.4, -0.5, -0.6,
-0.7, -0.8, -0.9, -1, -10000000};
```

尽量提高覆盖率

```
for (i = 0; i < 10000000; i++)
{
    ...
}
```

不能保证排除bug，但可以增强代码健壮性

# 少写点bug：保持随手检查和测试的习惯

- 其四：全覆盖的测试是不现实的，但不意味着可以不测

## • 测试的要点

- 构造测试输入：应当包含边界情况并提高覆盖
- 判断标准：
  - 1) 和‘黄金版本’比较：alu和fpu的测试
  - 2) 检查性质：三角函数的测试

如：给定任意  $0 \leq a \leq b \leq \pi/2$ ，必有  $\sin(a) \leq \sin(b)$



# 少写点bug：保持随手检查和测试的习惯

- 其四：全覆盖的测试是不现实的，但不意味着可以不测

PA 2-1的测试用例覆盖率极低，没有针对各指令的单元测试，请大家积极贡献测试用例

## • 测试的要点

- 构造测试输入：应当包含边界情况并提高覆盖
- 判断标准：
  - 1) 和‘黄金版本’比较：alu和fpu的测试
  - 2) 检查性质：三角函数的测试

如：给定任意  $0 \leq a \leq b \leq \pi/2$ ，必有  $\sin(a) \leq \sin(b)$

# Debug是有一些方法的

- 认识bug
- 少写点bug
- 科学debug

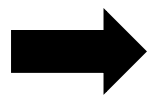


# Debug

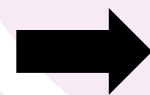
- 复现错误 (Failure)
- 定位缺陷 (Fault)
- 运用工具

*从程序中的Fault到Failure的传播过程*

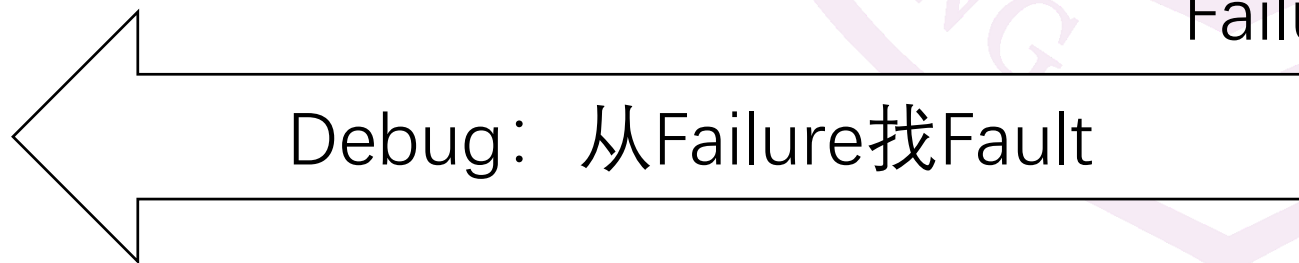
有一个Fault  
也就是Bug



运行时导致  
了一个Error



最后导致了  
程序的  
Failure



# Debug: 复现错误 (Failure)

PA中主要都是

- **Classify bugs by the ease of repeating their failures [3]**
  - **Bohrbugs**
    - Bohrbugs, like the Bohr atom, are solid, easily detected by standard techniques, and hence boring.
  - **Heisenbugs**
    - But Heisenbugs may elude a bugcatcher for years of execution. Indeed, the bugcatcher may perturb the situation just enough to make the Heisenbug disappear. This is analogous to the Heisenberg Uncertainty Principle in Physics.

# Debug: 复现错误 (Failure)

```
float input[] = { p_zero.fval, n_zero.fval, p_inf.fval, n_inf.fval,  
denorm_1.fval, denorm_2.fval, big_1.fval, big_2.fval, p_nan.fval,  
n_nan.fval, denorm_3.fval, small_1.fval, small_2.fval, 10000000, 1.2,  
1.1, 1, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1, -0.1, -0.2, -0.3, -0.4, -  
0.5, -0.6, -0.7, -0.8, -0.9, -1, -100000000};
```

```
for (i = 0; i < 10000000; i++)  
{  
    ...  
}
```

复现+缩小范围：在众多的测试输入中，找到能够触发failure的**那一个**，或者少数几个测试输入

# Debug: 定位缺陷 (Fault)

- 从那一个触发failure的测试输入入手
  - 火眼金睛法：
    - 人肉推理，看代码模拟代码执行过程和程序内部状态的改变，发现Error产生的位置
  - 单步/断点调试法：
    - 逐条/段分析/检查执行过程，发现Error产生的位置
  - 根据发现的Error，推理代码的逻辑错误

# Debug: 运用工具

- printf大法
- assert大法
- gdb大法

```
#include <stdio.h>

void func(int a, int b)
{
    printf("a = %d, b = %d\n", a, b);
}

int main()
{
    int t = 5;
    func(t++, ++t);
    return 0;
}
```

a = 6, b = 7

```
#include <stdio.h>

int check(int x)
{
    printf("x = %d\n", x);
    return x;
}

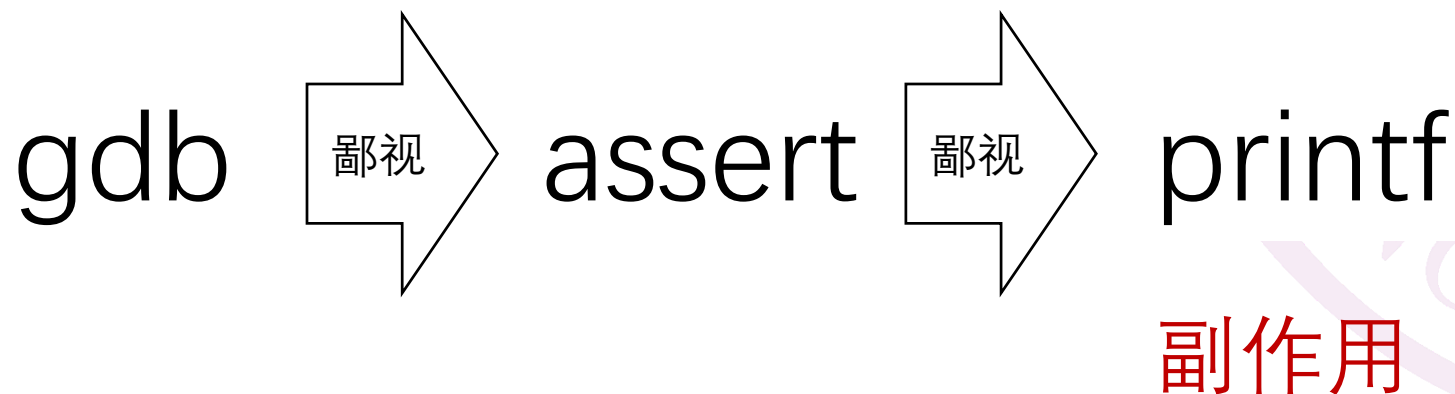
void func(int a, int b)
{
    printf("a = %d, b = %d\n", a, b);
}

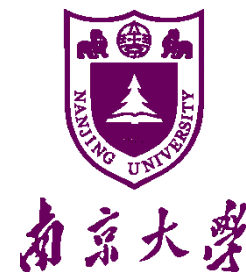
int main()
{
    int t = 5;
    func(check(t++), check(++t));
    return 0;
}
```

x = 6

x = 6

a = 6, b = 6





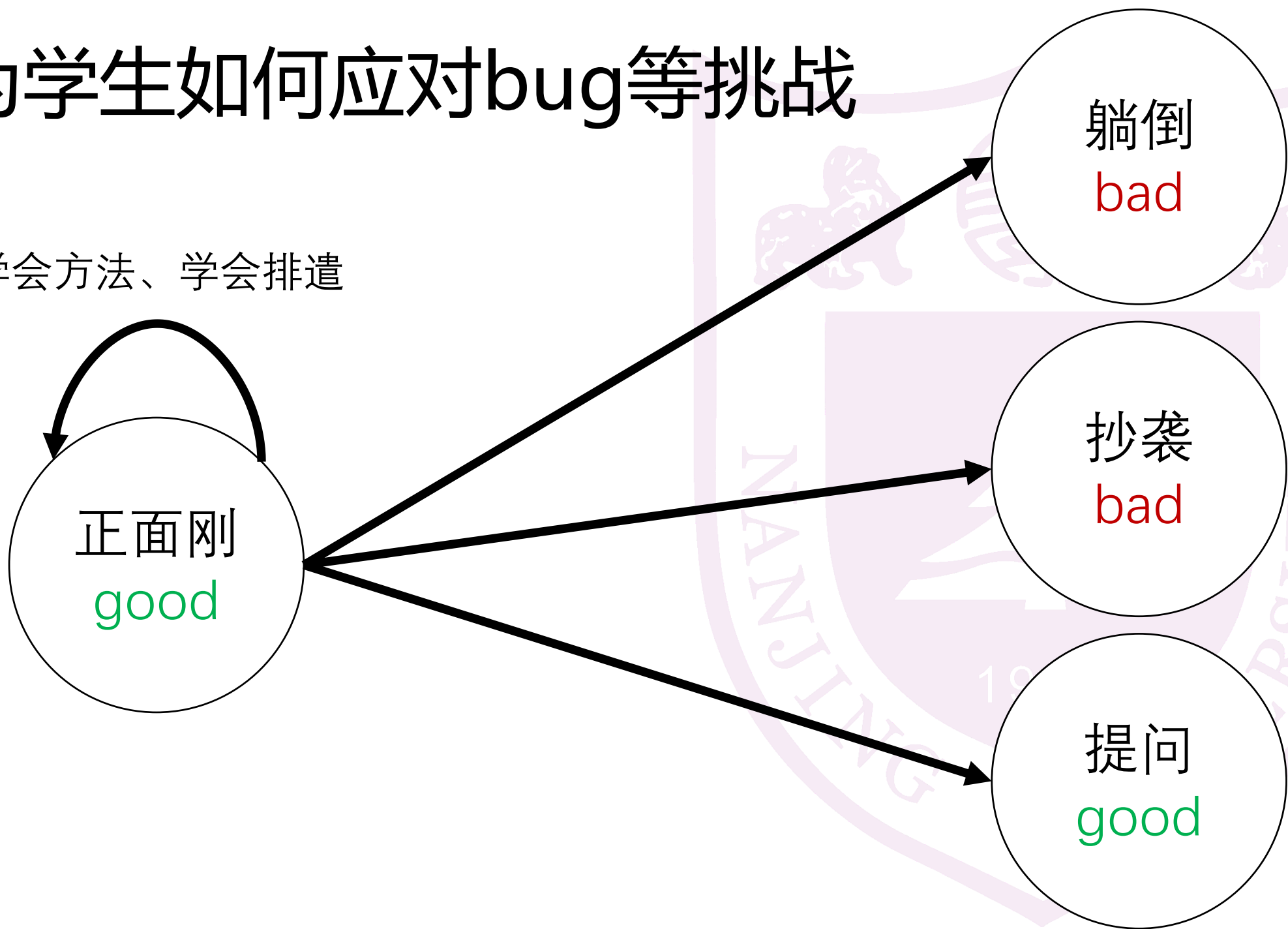
Bug是程序员一生之敌

踩遍所有坑， 成就伟大程序员



# 作为学生如何应对bug等挑战

学会方法、学会排遣



# 什么叫抄袭

## 7.2 Code Clone Types

There are basically two kinds of similarities between two code fragments. Two code fragments can be similar based on the similarity of their program text or they can be similar in their functionalities without being textually similar. The first kind of clones are often the result of copying a code fragment and then pasting to another location. In this section, we consider clone types based on the kind of similarity two code fragments can have:

- Textual Similarity: Based on the textual similarity we distinguish the following types of clones [35, 34, 153]:

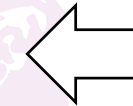
**Type I:** Identical code fragments except for variations in whitespace (may be also variations in layout) and comments.

**Type II:** Structurally/syntactically identical fragments except for variations in identifiers, literals, types, layout and comments.

**Type III:** Copied fragments with further modifications. Statements can be changed, added or removed in addition to variations in identifiers, literals, types, layout and comments.

- Functional Similarity: If the functionalities of the two code fragments are identical or similar i.e., they have similar pre and post conditions, we call them semantic clones [142, 156, 184, 60] and referred as *Type IV* clones.

**Type IV:** Two or more code fragments that perform the same computation but implemented through different syntactic variants.



形式上表现出克隆  
+ 未给出引用和出处  
+ 主观故意

我们检查抄袭

- 代码文本特征
- 编码行为特征

我们认定抄袭

- 客观证据
- 教师助教委员会认定
- 喝茶访谈认定

# 作为学生，你还有老师和助教

- 提问的艺术

- [https://github.com/ryanhanwu/How-To-Ask-Questions-The-Smart-Way/blob/master/README-zh\\_CN.md](https://github.com/ryanhanwu/How-To-Ask-Questions-The-Smart-Way/blob/master/README-zh_CN.md)

- 我们不愿意或无法回答的问题

- 没有经过思考或尝试就来问问题
  - 把我们当作human debugger或者试探参考实现
  - 表述不清的问题
    - 如，没有上下文直接问：“我为什么hit bad trap？”
  - 资料里已经讲清楚但没有看资料

# 作为学生，你还有老师和助教

- 提问的艺术

- [https://github.com/ryanhanwu/How-To-Ask-Questions-The-Smart-Way/blob/master/README-zh\\_CN.md](https://github.com/ryanhanwu/How-To-Ask-Questions-The-Smart-Way/blob/master/README-zh_CN.md)

- 我们愿意回答的问题

- 经过思考和努力后，在能够清晰表述问题，并提供一定分析和尝试结果的基础上所提出来的问题
  - 任何其他有趣的问题

# 作为学生，你还有老师和助教

- 但你也无需太担心
  - 如果问的问题不够友好，我们会回复：“这个问题我无法回答”，并给出简短的理由，并且不存在小本本来记录
  - 如果实在讲不清楚，我们也不会抛弃任何人



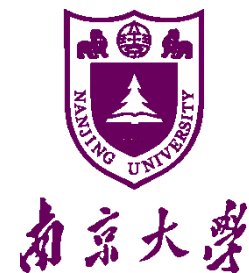


不要抄袭，不要提供给别人抄袭  
切实努力过再提问

作为学生，没有比学会知识更重要的事情

不懂就问，不理就催





Happy Hacking O\_o