西安電子科技大學 XIDIAN UNIVERSITY | 程序设计竞赛实训基地 Programming Contest Training Base

Competitive Programming 101

席若尧

西安电子科技大学程序设计竞赛实训基地

2021 年 7 月 5 日

内容

ICPC 环境简介

程序的行为和编译优化

常见编程错误

调试技巧

第 1 节

ICPC 环境简介

基本认知

- 程序设计竞赛与竞技体育/电子竞技类似
- 决定比赛成绩的因素有：天赋、训练、运气
- 不想训练的话也有很多低水平比赛可以打 (校赛，省赛，CSP 等)
- 想好好打的话，要对这项比赛有基本的认同感
- 尽量避免自视高贵/发表暴论

工作环境

- Ubuntu GNU/Linux
  - 没有 root 权限
- GCC
- OpenJDK
- Python 3 or PyPy 3
- Vim, Emacs, Gedit, Code::Blocks 等

DOMJudge

- 提交代码
- 打印代码
- 看榜
- 提问

团队配合

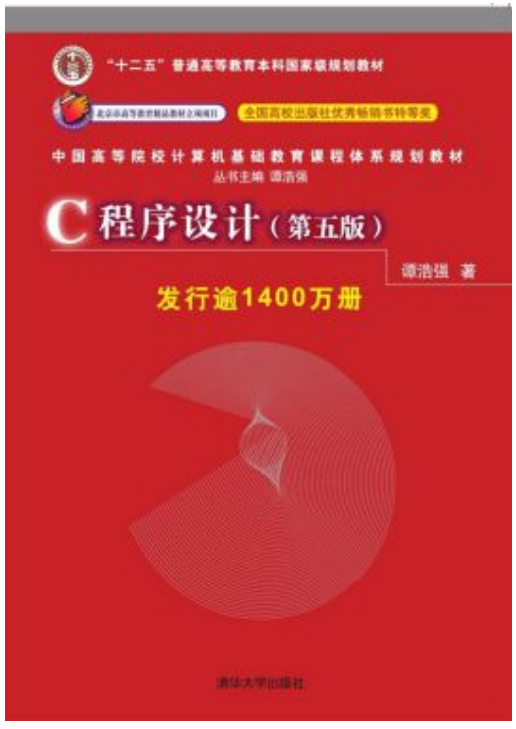
- 三人一机
- 实力 (训练量) 是配合的基础

第 2 节

程序的行为和编译优化

现状

- 本节内容应该是程序设计基础课程的一部分，但是懂的都懂



现状

- “你不确定的话就写个程序跑一下吧。”

现状

- “你不确定的话就写个程序跑一下吧。”
- 要是化学老师这么教课，实验室早炸上天了

现状

- “我打了 3 年 OI 还能不知道自己的程序啥行为？”

现状

- “我打了 3 年 OI 还能不知道自己的程序啥行为？”
- 众所周知 NOIP 不开 -O2

程序的行为

C 标准 (ISO/IEC 9899:2011 §5.1.2.3 p6) 规定，程序的**可观测行为**包括：

- 访问 volatile 变量
- 写入文件
- 操作交互式输入输出设备

实现 (在实践中由编译器和运行库组成) 只需要正确实现可观测行为。编译器可以在保证可观测行为不变的前提下，通过调整生成的机器代码，使得程序变得更小或更快。

例
Hello world

```cpp
#include <cstdio>

int main()
{
    std::printf("Hello, world.\n");
    return 0;
}
```

例
Hello World

生成的汇编代码根本没有 printf?

```
shell$ g++ hw.cc -O0 -S
shell$ grep -A10 'main:' hw.s
main:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movl    $.LC0, %edi
    call    puts
    movl    $0, %eax
```

例
取模

```
1  const int M = 1000000007;
2
3  int f(int a)
4  {
5      return a % M;
6  }
```

没有除法指令?

```
shell$ g++ mod.cc -O0 -S -fverbose-asm
shell$ grep -A10 'a % M' mod.s
# mod.cc:5:     return a % M;
    movl    -4(%rbp), %eax  # a, tmp84
    movslq  %eax, %rdx  # tmp84, tmp85
    imulq   $1152921497, %rdx, %rdx #, tmp85, tmp86
    shrq    $32, %rdx   #, tmp87
    sarl    $28, %edx   #, tmp88
    movl    %eax, %ecx  # tmp84, tmp89
    sarl    $31, %ecx   #, tmp89
    subl    %ecx, %edx  # tmp89, _2
    imull   $1000000007, %edx, %ecx #, _2, tmp90
    subl    %ecx, %eax  # tmp90, tmp84
```

例
取模 (误)

```
1  int M = 1000000007;
2
3  int f(int a)
4  {
5      return a % M;
6  }
```

例
取模 (误)

```
shell$ g++ mod_bad.cc -O0 -S -fverbose-asm
shell$ grep -A4 'a % M' mod_bad.s
# mod_bad.cc:5:     return a % M;
    movl    M(%rip), %ecx   # M, M.0_1
# mod_bad.cc:5:     return a % M;
    movl    -4(%rbp), %eax  # a, tmp85
    cltd
    idivl   %ecx    # M.0_1
    movl    %edx, %eax  # tmp86, _4
```

- 看上去简洁很多?
- 悲剧的是，这条除法指令比之前的一大堆加起来都慢。
- 实际做题时，因为忘加这个 const，可能引入 50% 至 200% 的时间代价。

例
strstr (Linux)

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  char s[1000001], p[1000001];
5
6  int main()
7  {
8      cin >> s >> p;
9      cout << (strstr(s, p) ? "YES" : "NO") << '\n';
10     return 0;
11 }
```

- 这题不用 KMP 能过?

例
strstr (Linux)

```
shell$ g++ strstr.cc -O2
shell$ python3 -c "print('a' * 999999 + 'c')" >
    strstr.in
shell$ python3 -c "print('a' * 500000 + 'c')" >>
    strstr.in
shell$ time ./a.out < strstr.in
YES

real    0m0.031s
user    0m0.030s
sys 0m0.001s
```

- 在 Linux 上跑得飞快
- Glibc 的 `strstr` 是时间 $\mathcal{O}(n+m)$，空间 $\mathcal{O}(1)$ 的高级算法，而且是高度优化的手写汇编代码

例

strlen (运气好)

```cpp
#include <bits/stdc++.h>
using namespace std;

char arr[1000001];

int main()
{
    uint64_t checksum = 0;
    cin >> arr;
    for (int i = 0; i < strlen(arr); i++)
        checksum = checksum * 1145141 + arr[i];
    cout << checksum << '\n';
    return 0;
}
```

- 目测是 $\mathcal{O}(n^2)$ 的，会 TLE?

例
strlen (运气好)

```
shell$ g++ strlen-1.cc -O2 -o strlen-1.exe
shell$ python3 -c "print('c' * 500000)" > strlen-1.
    in
shell$ time ./strlen-1.exe < strlen-1.in
11453095983971851168

real    0m0.008s
user    0m0.008s
sys 0m0.000s
```

- 然而跑得飞快

例
strlen (运气差)

```cpp
#include <bits/stdc++.h>
using namespace std;

char arr[1000001];

int main()
{
    uint64_t checksum = 0;
    cin >> arr;
    for (int i = 0; i < strlen(arr); i++) {
        arr[i] = tolower(arr[i]);
        checksum = checksum * 1145141 + arr[i];
    }
    cout << checksum << '\n';
    return 0;
}
```

- 就改个大小写，不会出大问题吧

例
strlen (运气差)

```
shell$ g++ strlen-2.cc -O2 -o strlen-2.exe
shell$ python3 -c "print('c' * 500000)" > strlen-2.
    in
shell$ time ./strlen-2.exe < strlen-2.in
11453095983971851168

real    0m3.024s
user    0m3.024s
sys 0m0.000s
```

- 人都没了
- 编译器的能力是有极限的，~~所以我不做编译器辣！~~

几类行为

- 标准定义的行为
- 实现定义的行为 (implementation-defined)
- 未指定行为 (unspecified)
- 未定义行为 (undefined)

实现定义的行为

标准要求实现作出一致的选择。

- 一个典型例子是 `rand` 函数生成随机数的规则 (包括但不限于 `RAND_MAX` 的值)，这在 2020 – 2021 ICPC 区域赛南京站产生了显著的影响。
- 另一个例子是 FFT 等场合常用的卡常技巧 `a += M & (a >> 31);`，它的目的是将 $[-M, M)$ 中的整数模 M (变为 $[0, M)$ 中的整数)。其正确性依赖于 GCC 规定负数右移按算术右移规则进行，如果更换编译器则可能出错。

未指定行为

标准未作说明，允许实现随意选择。

- 几乎所有程序都会涉及未指定行为，例如函数调用时，对其参数求值的顺序就是未指定的。
- 但是，如果你的 (比赛用) 程序的输出依赖于某个未指定行为，那么很有可能产生难以调试的 bug。

例

```
1  // read an int from stdin
2  // implementation omitted
3  int read(void);
4
5  struct Point
6  {
7      int x, y;
8      Point(int xx, int yy) : x(xx), y(yy) {}
9  };
10
11 int main()
12 {
13     Point p(read(), read());
14     return 0;
15 }
```

- 如果实现先对后一个 `read()` 求值怎么办?

修复

```
1  // read an int from stdin
2  // implementation omitted
3  int read(void);
4
5  struct Point
6  {
7      int x, y;
8  };
9
10 int main()
11 {
12     Point p{read(), read()};
13     return 0;
14 }
```

- 语言标准要求初始化列表中的元素必须从左到右求值。

未定义行为

- 语法错误
- 语义错误
  - 违反约束条件 (constraint)：要求编译器将其视为编译错误
  - 其他：未定义行为，实现可以干任何事！

未定义行为

- “什么都可能发生”
- 在没有操作系统或操作系统欠缺安全机制时，甚至能够损毁硬件
- 在比赛中可能产生“本地测不出来 bug，但交上去就错”的问题

例
函数无返回值

- 一类十分值得注意的未定义行为是，在非 void 函数中，程序流程到达函数尾。
- 注意此时无论是否实际使用了该函数的返回值，只要程序执行到函数尾，都是未定义行为。
- 在编译器执行优化时，它可能假设“程序永远不会执行到该函数的尾部”，从而产生更加奇怪的行为。
- **老资格的 OI 选手**需要特别注意！

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  vector<int> G[114514];
5  int color[114514];
6  bool vis[114514];
7
8  int dfs(int u, int c)
9  {
10     color[u] = c;
11     vis[u] = true;
12     for (int v: G[u])
13         if (!vis[v])
14             dfs(v, !c);
15 }
```

关于未定义行为

某些人总是无视语言标准尝试使用各种未定义行为，然后到处说“这程序在我的电脑上能工作，为什么交上去就错？”对此，Roger Miller 讽刺道：

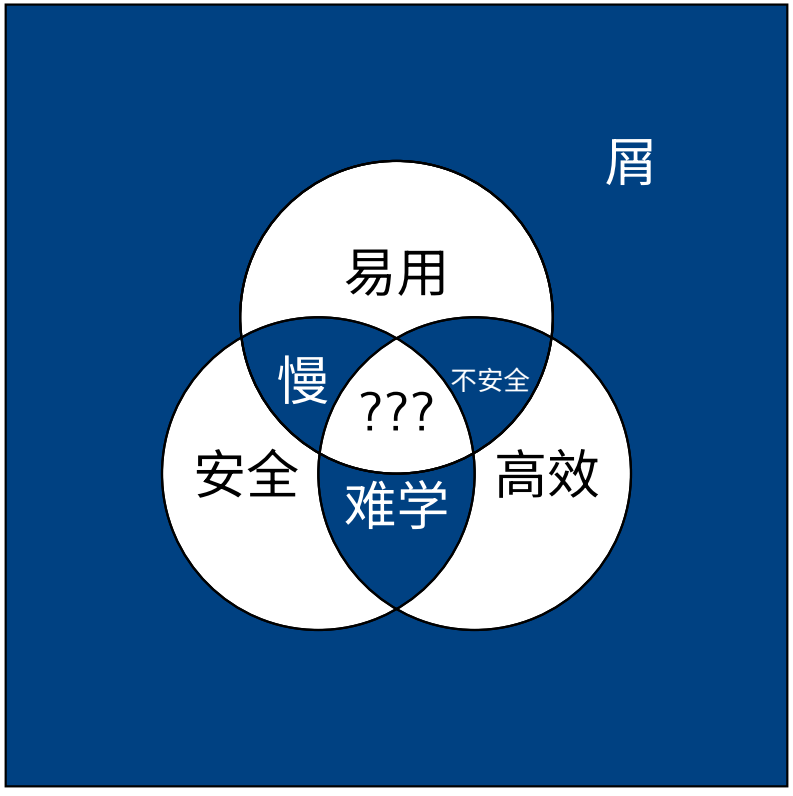
- 有人曾经告诉我，在打篮球的时候，你不能带球走。我找了个篮球试了一下，发现走得很好。那人显然根本不懂篮球。

关于未定义行为

某些人总是无视语言标准尝试使用各种未定义行为，然后到处说“这程序在我的电脑上能工作，为什么交上去就错？”对此，Roger Miller 讽刺道：

- 有人曾经告诉我，在打篮球的时候，你不能带球走。我找了个篮球试了一下，发现走得很好。那人显然根本不懂篮球。

没有银弹



第 3 节

常见编程错误

概述

- 在比赛和实际工作中，总是会不可避免地写出错误 (或有瑕疵) 的代码
- 在软件开发中，通常要为系统测试和调试预留 50% 以上的时间
- 本节主要介绍现场赛可用的测试和调试手段，其他 (如 Valgrind 等) 可以自行了解

评测结果

首先介绍评测系统 (主要是 DOMJudge) 可能返回的错误信息：

- COMPILER-ERROR
- TIMELIMIT
- RUN-ERROR
- OUTPUT-LIMIT
- WRONG-ANSWER (以及 NO-OUTPUT)

再次强调：未定义行为可能导致评测返回任何一种错误！

编译错误
COMPILER-ERROR

- 指你的程序无法编译成可执行文件
- 一般 OJ 都提供了查看编译器输出消息的方法，照着改即可
- 正式比赛要求选手机器和评测机完全一致，所以几乎不会出现
- 正式比赛 CE 不算罚时

时间超限
TIMELIMIT

- 你的程序跑得太慢了

运行错误 RUN-ERROR

- 你的程序没有以返回值 0 正常退出
- 一般是因为出现了未定义行为
- 也可能是不小心返回了一个非 0 值，比如压行压成了 `exit(printf("%d\n", x));`
- 现场赛使用 DOMJudge，内存超限也报告为运行错误

输出超限
OUTPUT-LIMIT

- 你输出了太多东西，超过了题目的限制 (DOMJudge 默认 8 MB)
- 可能是陷入了带输出的死循环
- 也可能是忘了删除调试输出 :)
- 对于某些题目只是 WA 的一种表现形式
  - 例如：如果有解输出一个 $[0, 9]$ 中的整数，如果无解输出 `"a very very long error message"`，那么如果把很多有解的情况判断成了无解就可能输出超限

答案错误
WRONG-ANSWER

- 你的程序输出和标准答案不一致，或者被 SPJ 判断为错误
- 不同评测系统对于空白字符 (特别是行末空格和文末回车) 的处理不一致，如果空白字符有区别可能返回 AC、WA、PE 等不同结果。
- DOMJudge 没有 PE，无 SPJ 的情况下会忽略行末空格和文末回车，但对于空白字符的其他差异会直接返回 WA。建议写输出时遵循一般的规范，每行都以 "\n" 结束，行末不要有多余的空格
- 如果完全没有输出，一些评测系统 (包括 DOMJudge) 会返回 "NO-OUTPUT"

编程错误

我们已经了解了评测结果，下面讨论它们产生的原因，即程序设计竞赛中常见的编程错误。

什么是栈溢出

- 函数调用的过程中会把一些数据 (返回地址，局部变量，部分参数) 放入系统的栈空间中，调用结束后再弹出。
- 为了更灵活地为堆和共享库分配内存，Linux 默认将栈空间限制在 8MB。
- 如果栈的大小越过 8MB，内核会发送 SIGSEGV 信号杀死进程。
- Windows 默认栈空间限制为 1MB。

栈溢出的表现

- RUN-ERROR
- 段错误
- segmentation fault (core dumped)

预防和排查

- 绝对不要写死递归
- 一些评测系统会调整系统栈空间限制，因此可以大胆使用栈
  - DOMJudge 的当前版本不限制栈空间，建议区域赛热身赛进行测试
  - Codeforces 调整到了 256MB
  - 然而大多数在线评测系统都没有调整
- 可以将本机调成不限制栈空间：`ulimit -s unlimited`
  - 只对当前终端开出来的进程有效
  - 这会导致死递归难以排查，甚至卡死系统，可以选择折中地把栈调大一点：`ulimit -s 131072` (单位 KB)
- 对于递归，本机测试开与评测时相同的优化 (例如 `-O2`)，防止误判 (将其他错误当成栈溢出，或者相反)
- 不要将巨大的数组或结构体声明为自动局部变量或按值传递的参数
  - 可以开全局或者静态变量，通过指针/引用传递
- 使用非递归写法，或者显式开栈模拟递归

汇编开栈

暗黑魔法，使用时自负风险。

```
#include <bits/stdc++.h>
__attribute__((aligned(32)))
char _s_t_a_c_k_[128 << 20];
int main()
{
    __asm volatile (
#ifdef __x86_64__
        "movq %0, %%rsp"
#else
        "movl %0, %%esp"
#endif
        ::"r"(_s_t_a_c_k_+(128 << 20)):
    );

    fclose(stdout);
    _Exit(0);
}
```

整数溢出

以下属于未定义行为：

- 带符号整数算术运算溢出
- 移位位数超过整数位数
- 使用 `scanf` 读取数字时，输入超过格式化字符串指定的表示范围
- 浮点数转化为整数时，其值超过了整数类型能表示的范围

以下行为可能导致出人意料的结果：

- 无符号整数算术运算溢出 (丢弃高位)
- 使用 `cin` 读取数字时，输入超过变量的表示范围 (读入错误的值，并设定 `failbit`)
- 整数类型互相转化时高位被丢弃
- 浮点数转化为整数时小数部分被截断

整数溢出

以下属于未定义行为：

- 带符号整数算术运算溢出
- 移位位数超过整数位数
- 使用 `scanf` 读取数字时，输入超过格式化字符串指定的表示范围
- 浮点数转化为整数时，其值超过了整数类型能表示的范围
  - 欧洲的 Ariane 5 型运载火箭首飞即因为代码中 64 位浮点值向 16 位整数转换时发生此类错误而坠毁

以下行为可能导致出人意料的结果：

- 无符号整数算术运算溢出 (丢弃高位)
- 使用 `cin` 读取数字时，输入超过变量的表示范围 (读入错误的值，并设定 `failbit`)
- 整数类型互相转化时高位被丢弃
- 浮点数转化为整数时小数部分被截断

整数溢出的后果

- 大部分会 WA，可能会 RE 甚至 TLE。
- 在测试时你的程序可能莫名其妙输出负数。

```
int a, b, ans = 0;
scanf("%d%d", &a, &b);
for (int i = a; i <= b; i++)
    ans += foo(i);
printf("%d\n", ans);
return 0;
```

预防和排查

- 开始编码前预先考虑好输入、中间结果、最终结果的可能范围
- 该取模的取
- 该开 64 位和 128 位整数的开
- 该转型的转：`111 * a * b`
- 该换语言的换
- 该写高精度的写
- 打开编译器相关警告选项
- 编造数据测试是否有溢出
- 如果测试过程中发现溢出，打开运行时检查工具
- 如果确实需要溢出，使用无符号整数
- 反对盲目蛮干

无效指针

- 未初始化的指针
  - 和一般未初始化值一样，使用就是未定义行为
- 空指针
  - 只能用作哨兵值 (sentinel)
- 指向数组最后一个元素“之后一个元素”的指针
  - 可以构造出来，例如 `sort(a, a+n);`
  - 可以用于偏移运算，例如 `int *p = a+n; a[-2];`
  - 不能解引用，否则引发未定义行为
- 越出数组界限的指针
  - 对不指向数组中元素的指针进行偏移运算是未定义的
  - 对指向数组中元素的指针进行偏移运算，越出数组范围 (结果不指向同一数组中的元素，或该数组最后一个元素“之后的一个元素”)，行为是未定义的

无效指针的危害

- RE、WA、TLE、MLE 都有可能
- 可能严重干扰调试器

预防和排查

- 数组开到足够大
- 对指针和数组下标进行必要检查
  - `for (; j < n && a[j] < b[i]; j++) foo(j);`
- 不要乱用指针
- 如果本地测试出现问题，怀疑无效指针，可以打开相关运行时检查

无效的迭代器

- 和指针一样，迭代器也不能越界
- 此外，在一些 (可能意想不到的) 情况下，迭代器会失效变成非法的

一个例子

我们希望删掉一个 std::set 里面所有的偶数:

```
#include <set>
#include <cstdio>
using namespace std;

int main()
{
    set<int> S = {4,-1,5,8,-2,-5};
    for (int x: S)
        if (x % 2 == 0)
            S.erase(x);
    for (int x: S)
        printf("%d\n", x);
}
```

然后就炸了

```
shell$ g++ set_invalidate.cc
shell$ ./a.out
Segmentation fault (core dumped)
```

哪里有迭代器?

根据语言标准，循环

```
for (int x: S)
    if (x % 2 == 0)
        S.erase(x);
```

等价于

```
for (auto __begin = S.begin(),
          __end = S.end();
          __begin != __end;
          ++__begin) {
    int x = *__begin;
    if (x % 2 == 0)
        S.erase(x);
}
```

哪里有迭代器?

根据语言标准，循环

```
for (int x: S)
    if (x % 2 == 0)
        S.erase(x);
```

等价于

```
for (auto __begin = S.begin(),
          __end = S.end();
          __begin != __end;
          ++__begin) {
    int x = *__begin;
    if (x % 2 == 0)
        S.erase(x);
}
```

- 删掉一个偶数以后，指向它的迭代器 __begin 就失效了，然后再使用这个迭代器就会发生未定义行为

正确的写法

先自增，再删除：

```
for (auto it = S.begin(); it != S.end(); )
    if (*it % 2 == 0)
        S.erase(it++); // or: it = S.erase(it);
    else
        it++;
```

另一个例子

你可能认为不删除就没问题了，然而……

```
#include <vector>
#include <cstdio>
using namespace std;

int main()
{
    vector<int> v = {1,2,3};
    for (int i: v)
        if (i > 0)
            v.push_back(-i);
    for (int i: v)
        printf("%d\n", i);
}
```

又爆炸了

```
shell$ g++ vec_invalidate.cc
shell$ ./a.out
1
2
3
-1
-19669008
```

- 什么鬼?

又爆炸了

```
shell$ g++ vec_invalidate.cc
shell$ ./a.out
1
2
3
-1
-19669008
```

- 什么鬼?
- 这是因为 vector 在插入元素时，可能需要重新分配一段更大的内存，并搬移原有元素，从而导致之前的迭代器失效

又爆炸了

```
shell$ g++ vec_invalidate.cc
shell$ ./a.out
1
2
3
-1
-19669008
```

- 什么鬼?
- 这是因为 vector 在插入元素时，可能需要重新分配一段更大的内存，并搬移原有元素，从而导致之前的迭代器失效
- 改用下标就行了

防范和排查

- 不要滥用迭代器
- 使用 range-based for 循环时，最好不要对被迭代的容器进行插入或删除操作
- 在 set、map、multiset、multimap、list 等中进行删除操作时，可以使用 `c.erase(it++)` 的写法
  - 对于 C++11 以上，还支持 `it = c.erase(it)` 的写法
- 如果怀疑使用了无效迭代器，可以打开 C++ 标准库的运行时检查

超时的原因

- 算法假了
- 算法是真的，但某个细节没考虑，导致高次时间复杂度
- 常数太大

超时的原因

- 算法假了
  - 典型代表：暴力字符串匹配、keduoli 树
- 算法是真的，但某个细节没考虑，导致高次时间复杂度
  - 典型代表：`for(int i = 0; i < strlen(s); i++)`
  - `memset`
- 常数太大
  - 典型代表：`endl` 或者忘了用 `cin.tie(0)`、`valarray`

预防和排查

- 做好时间复杂度分析
  - 几乎一定要分析最坏情况
  - “期望时间复杂度”几乎没有用
- 使用工具寻找可能存在的高次复杂度和大常数
- 卡常数

精度问题

众所周知，浮点数的精度是有限的。

- `float`：23 位
- `double`：52 位
- `long double`：可能是 52 位或 64 位

预防

- float 这种东西就别用了吧……
- 如果要输出的东西本身就是一个浮点值 (如长度、面积)，那么可以放心使用 double，但要注意控制精度，防止出现数值稳定性问题
  - 一般会有 SPJ
  - 减少中间步骤
  - 防止出现奇异值
  - 如果没有 SPJ 的话需要调一调 eps ……
  - 比如 1.285 四舍五入保留小数点后两位，不加 eps 输出会暴毙
- 如果有比较逻辑，要输出 YES/NO 或方案数，尽量避免浮点数
  - x >= y?
  - 32 位机器上“薛定谔的精度”
  - 避免除法
  - 如果一定要用除法，写分数类

特殊浮点值

- INF：无限大量 (有正负)
- 0 (有正负)
- NaN

- 某次比赛某出题人写了这样的 SPJ，有什么问题?

```
1  // a and b are outputed by the contestant
2  // ans is provided by the jury
3  int check(int a, int b, double ans)
4  {
5      double t = (double) a / (double) b;
6      if (fabs(t - ans) > 1e-9)
7          return WA;
8      return AC;
9  }
```

- 某次比赛某出题人写了这样的 SPJ，有什么问题?

```
1 // a and b are outputed by the contestant
2 // ans is provided by the jury
3 int check(int a, int b, double ans)
4 {
5     double t = (double) a / (double) b;
6     if (fabs(t - ans) > 1e-9)
7         return WA;
8     return AC;
9 }
```

- 选手只要输出 $a = b = 0$ 就 AC 辣!

排查

- 有时候很难排查出最早在哪里算出了 INF 或者 NaN
- 在程序开始时加一行：`feenableexcept(FE_ALL_EXCEPT);` 就能让你的程序在算出 INF 或者 NaN 的时候直接崩溃，然后用调试器就能看具体是哪一行了

例子

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    feenableexcept(FE_ALL_EXCEPT);
    double a, b;
    cin >> a >> b;
    cout << sqrt(a / b) << '\n';
    return 0;
}
```

结果

```
shell$ g++ feenableexcept.cc -g
shell$ echo "1 1"  | ./a.out
1
shell$ echo "1 0"  | ./a.out
Floating point exception (core dumped)
shell$ echo "0 0"  | ./a.out
Floating point exception (core dumped)
shell$ echo "-1 1" | ./a.out
Floating point exception (core dumped)
```

浮点异常

- floating point exception (core dumped)
- 如果没开 feenableexcept，则和浮点数并没有什么关系
- 意味着出现了除以 0 或者模 0
- 直接用调试器看在哪里除了 0 就行了

误用 STL

- 提供给 sort 或者 set 的比较函数是假的?
- 在没排序的数组上执行 lower_bound 等二分查找函数?
- 可以打开 C++ 运行库的运行时检查，来寻找可能的这种错误

调试技巧

- 下面介绍一些比赛/训练时可能有用的调试技巧

编译警告

建议使用的警告选项：

- `-Wall -Wextra`
- `-Wshadow`：防止局部变量不小心遮盖其他变量
- `-Wformat=2`：防止 `printf/ scanf` 写错
- `-Wconversion`：防止意外的类型转换

某些情况下有用的警告选项：

- `-Wstack-usage=1`：看栈空间使用情况

例子

```
1  struct mat
2  {
3      int a[4][4];
4      mat operator*(const mat &rhs) const
5      {
6          mat m;
7          for (int i = 0; i < 4; i++)
8              for (int j = 0; j < 4; j++) {
9                  m.a[i][j] = 0;
10                 for (int k = 0; k < 4; k++)
11                     m.a[i][j] += a[i][k] * rhs.a[k][j];
12             }
13     }
14 };
```

错误

编译器立刻说出我忘了写 return 语句:

```
shell$ g++ -c -Wall -Wextra matrix_bug.cc
matrix_bug.cc: In member function 'mat mat::
    operator*(const mat&) const':
matrix_bug.cc:13:9: warning: no return statement in
     function returning non-void [-Wreturn-type]
    13 |         }
       |         ^
```

另一个例子

```
1  #include <iostream>
2  using namespace std;
3
4  const int M = 998244353;
5  int pow_mod(int a, int b); // impl skipped
6
7  int main()
8  {
9      long long a, b;
10     cin >> a >> b;
11     pow_mod(a, b);
12 }
```

编译警告

```
shell$ g++ -c -Wall -Wextra -Wconversion
    pow_mod_bug.cc
pow_mod_bug.cc: In function 'int main()':
pow_mod_bug.cc:11:17: warning: conversion from '
    long long int' to 'int' may change value [-
    Wconversion]
    11 |          pow_mod(a, b);
       |                  ^
pow_mod_bug.cc:11:20: warning: conversion from '
    long long int' to 'int' may change value [-
    Wconversion]
    11 |          pow_mod(a, b);
       |                    ^
```

可以看出是抄快速幂板子忘了改参数类型。

运行期检查

编译警告虽然很有用，但由于停机问题是不可解的，不可能在编译期找出所有错误，这就需要使用运行期检查。下面介绍一些有用的，可以在区域赛使用的运行期检查工具：

- Undefined Behavior Sanitizer
- Address Sanitizer
- Libstdc++ Debug Mode

UBSan

使用编译选项 -fsanitize=undefined -g 开启，用于寻找未定义行为。
我们用一个初学者经常写出来的程序演示一下：

```cpp
#include <iostream>
using namespace std;

const int M = 998244353;

int main()
{
    int a, b;
    cin >> a >> b;
    long long c = a * b % M;
    cout << c << '\n';
    return 0;
}
```

测试结果

```
shell$ g++ overflow.cc -fsanitize=undefined -g
shell$ echo "100000 100000" | ./a.out
overflow.cc:10:18: runtime error: signed integer
    overflow: 100000 * 100000 cannot be represented
     in type 'int'
411821055
```

局限性

UBSan 虽然也能检测到一些越界访问的情况 (毕竟这种情况也属于未定义行为)，但对于稍微复杂一些的越界 (涉及动态内存分配或者一些库函数) 就无能为力了。例如：

```cpp
#include <cstdio>
using namespace std;

int main()
{
    char buf[5];
    scanf("%s", buf);
    printf("hello, %s\n", buf);
}
```

隐蔽的越界

这个程序就算开了 UBSan 也能正常运行，甚至在越界不多时，仍输出正确答案：

```
shell$ g++ scanf_bound.cc -fsanitize=undefined
shell$ echo wang9897 | ./a.out
hello, wang9897
```

然而交上去就自闭了……

ASan!

这时就需要更专业的 Address Sanitizer 了，我们使用 `-fsanitize=address -g` 启用它：

```
shell$ g++ scanf_bound.cc -fsanitize=address -g
shell$ echo wang9897 | ./a.out
=================================================================

==12384==ERROR: AddressSanitizer: stack-buffer-
    overflow on address 0x7ffe1de53355 at pc 0
    x7ff259a0dbe5 bp 0x7ffe1de531f0 sp 0
    x7ffe1de529a0
```

……

```
    #3 0x40122c in main /home/xry111/work/cp101/
       code/scanf_bound.cc:7
```

编译时使用 -D_GLIBCXX_DEBUG 打开调试模式。此时 libstdc++ 会插入两项运行时检查：

- 迭代器安全性检查
- 算法前提条件检查

例子

我们用前面那个错误使用 vector 的迭代器的程序演示一下：

```
#include <vector>
#include <cstdio>
using namespace std;

int main()
{
    vector<int> v = {1,2,3};
    for (int i: v)
        if (i > 0)
            v.push_back(-i);
    for (int i: v)
        printf("%d\n", i);
}
```

运行结果

```
shell$ g++ vec_invalidate.cc -D_GLIBCXX_DEBUG=1 -g
shell$ ./a.out
/usr/include/c++/11.1.0/debug/safe_iterator.h:330:
In function:
    __gnu_debug::_Safe_iterator<_Iterator,
        _Sequence, _Category>&
    __gnu_debug::_Safe_iterator<_Iterator,
        _Sequence,
    _Category>::operator++() [with _Iterator =
    __gnu_cxx::__normal_iterator<int*, std::
        __cxx1998::vector<int,
    std::allocator<int> > >; _Sequence = std::
        __debug::vector<int>;
    _Category = std::forward_iterator_tag]

Error: attempt to increment a singular iterator.
```

另一个例子

如果忘了排序就二分查找：

```
#include <algorithm>
#include <iostream>
using namespace std;

int main()
{
    int arr[] = { 3, 1, 2 };
    cout << binary_search(arr, arr+3, 2) << endl;
}
```

运行结果

```
shell$ g++ bsearch_buggy.cc -D_GLIBCXX_DEBUG
shell$ ./a.out
/usr/include/c++/11.1.0/bits/stl_algo.h:2249:
In function:
    bool std::binary_search(_FIter, _FIter, const
        _Tp&) [with _FIter = int*;
    _Tp = int]

Error: elements in iterator range [__first, __last)
    are not partitioned by
the value __val.
```

对拍

如果某题要求时间复杂度 $\mathcal{O}(nlogn)$，你写好以后交上去 WA 了，又找不到错，可以写个 $\mathcal{O}(n^2)$ 的暴力，然后用 $n = 1000$ 左右的随机数据去检验。

- 前提是你暴力能写对
- 要小心，有时候随机数据并不能拍出所有 bug
- 数据范围可以放小一些 (比如 $n = 10$)
- (训练时) 如果你有别人 (或者自己) 已经 AC 的程序也可以用来拍

造数据

下面给出一个简单的数据生成器：

```
from random import randint, seed
seed(int(input()))
n = randint(1, 10)
print(n)
print(" ".join([str(randint(1, 100)) for i in range(n)]))
```

- 用 Python 是因为写起来短，而且随机数生成不会像 C 的 rand 那样呈现 implementation-defined 行为
- 读入种子是为了保证可重复性，只要记录下生成拍出错误的数据的种子，就能随时重新生成这组数据

简单的对拍器

```
#!/bin/sh

for ((i=0;;i++)) do
    echo $i | python3 generator.py > data-$i.in
    ./brute < data-$i.in > data-$i.ans
    if ! ./solution < data-$i.in > data-$i.out; then
        echo "runtime error on test $i"
        break
    fi
    if diff data-$i.ans data-$i.out; then
        rm data-$i.{in,out,ans}
    else
        echo "wrong answer on test $i"
        break
    fi
    echo "test $i ok"
done
```

白盒调试

如果我们用上面的方法发现了错误，但不知道为什么，就可能需要分析程序的执行过程……

输出中间结果

调试宏：

```
#include <bits/stdc++.h>
using namespace std;

#define DBG(x) \
(void)(cerr << "L" << __LINE__ \
            << ": " << #x << " = " \
            << (x) << endl)

int main()
{
    int something = 233;
    DBG(something);
}
```

输出的效果：L12: something = 233

断言

<cassert> 提供了 assert 宏，可以用来确保前提条件的成立。

- 例如，BSGS 算法要求 a 和 M 互质，我们可以在自己的 BSGS 模板里面加上 `assert(gcd(a, M) == 1);`
- 如果使用 BSGS 时不满足这个条件，程序就会报错退出
- 同时，它也可以在敲模板的时候提醒你 BSGS 只能用于这种情况
- 提交代码时可以在程序开头加 `#define NDEBUG`，关闭所有断言，以避免不必要的运行时间

使用调试器

建议直接使用 GDB 的命令行，Code::Blocks 的调试非常难用。调试时需要打开编译选项 -g，建议禁用优化。GDB 的常用命令有：

- b (breakpoint) 行号/函数名
- r (run) [< 输入文件名]
- n (next)
- s (step)
- c (continue)
- p (print) 表达式
- d (disp) 表达式
- cond (condition) 断点编号 表达式
- bt (backtrace)
- fr (frame) 栈帧编号

性能分析工具

- gcov/`-ftest-coverage -fprofile-arcs`：代码覆盖率检测，可以看代码中每一行被执行的次数
- gprof/`-pg`：代码剖析，可以看函数执行时间占总时间的百分比

例子

BAPC 2018 E 题的一份 TLE 代码。我们先造一个大数据试一下：

```
shell$ g++ bapc2018e.cc -O2
shell$ time ./a.out < bapc2018e-data.txt
965105033

real    0m0.709s
user    0m0.707s
sys 0m0.001s
```

可以看到跑得很慢。

使用 gprof

```
shell$ g++ bapc2018e.cc -O2 -fno-inline -pg
shell$ ./a.out < bapc2018e-data.txt
965105033
shell$ gprof
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/
    call  name
 96.00      0.82     0.82     1999     0.41
    0.41  pow_mod(int, int)
```

使用 gcov I

```
shell$ rm *.gc* -f
shell$ g++ bapc2018e.cc -o cov.exe -O2 -ftest-
    coverage -fprofile-arcs
shell$ ./cov.exe < bapc2018e-data.txt
965105033
shell$ gcov cov-bapc2018e > /dev/null
shell$ head bapc2018e.cc.gcov -n20
        -:    0:Source:bapc2018e.cc
        -:    0:Graph:cov-bapc2018e.gcno
        -:    0:Data:cov-bapc2018e.gcda
        -:    0:Runs:1
        -:    1:#include <bits/stdc++.h>
        -:    2:using namespace std;
        -:    3:
        -:    4:const int M = 1'0000'0000'9;
        -:    5:
124124000:    6:int pow_mod(int a, int x)
```

使用 gcov II

```
        -:    7:{
124124000:    8:    if (!x)
        -:    9:        return 1;
120120000:   10:    long long t = pow_mod(a, x>>1);
120120000:   11:    t = t * t % M;
120120000:   12:    if (x&1)
 64062000:   13:        t = t * a % M;
120120000:   14:    return t;
        -:   15:}
        -:   16:
```

结论

- 两种方法都能发现快速幂使用了过多时间
- gprof 输出的是时间，但只能精确到函数
- gcov 精确到行，但只能输出调用次数

求助他人（鸭？）

- 找别人帮忙调程序几乎一定会破坏友谊
- 如果一定要找，把代码的可读性弄得好一点……
- 小黄鸭调试法

课后学习

- 作业
- https://icpc.xidian.wiki/cce

西安电子科技大学 XIDIAN UNIVERSITY | 程序设计竞赛实训基地 Programming Contest Training Base

GL and HF!