

OI Wiki (Beta)

OI Wiki 项目组

2026 年 10 月 4 日

目录

1 简介 1
1.1 Getting Started 2
1.2 关于本项目 2
1.3 如何参与 3
1.4 OI Wiki 不是什么 7
1.5 格式手册 9
1.6 数学符号表 34
1.7 F.A.Q. 45
1.8 镜像站列表 50
1.9 致谢 51

2 比赛相关 53
2.1 比赛相关简介 54
2.2 赛事 54
2.2.1 OI 赛事与赛制 54
2.2.2 ICPC/CCPC 赛事与赛制 61
2.3 题型 62
2.3.1 题型概述 62
2.3.2 交互题 66
2.4 学习路线 78
2.5 学习资源 82
2.6 技巧 89
2.6.1 读入、输出优化 89
2.6.2 分段打表 96
2.6.3 常见错误 96
2.6.4 常见技巧 106
2.7 出题 109

3 工具软件 123
3.1 工具软件简介 124
3.2 代码编辑工具 124
3.2.1 Vim 124
3.2.2 Emacs 128
3.2.3 VS Code 133
3.2.4 Atom 142
3.2.5 Eclipse 143
3.2.6 Notepad++ 146
3.2.7 Kate 153
3.2.8 Dev-C++ 157
3.2.9 CLion 167
3.2.10 Geany 174
3.2.11 Xcode 175
3.2.12 GUIDE 186
3.2.13 Sublime Text 188

3.2.14 CP Editor 198
3.2.15 Code::Blocks 205
3.3 评测工具 209
3.3.1 评测工具简介 209
3.3.2 Arbiter 210
3.3.3 Cena 216
3.3.4 CCR Plus 216
3.3.5 Lemon 217
3.4 命令行 218
3.5 命令行编译与调试 223
3.6 编译器 227
3.7 WSL (Windows 10) 234
3.8 Special Judge 247
3.9 Testlib 255
3.9.1 Testlib 简介 255
3.9.2 通用 256
3.9.3 Generator 259
3.9.4 Validator 262
3.9.5 Interactor 263
3.9.6 Checker 264
3.10 Polygon 269
3.11 OJ 工具 273
3.12 LaTeX 入门 277
3.13 Git 294

4 语言基础 307
4.1 语言基础简介 308
4.2 C++ 基础 308
4.2.1 Hello, World! 308
4.2.2 C++ 语法基础 309
4.2.3 变量 314
4.2.4 运算 325
4.2.5 流程控制语句 331
4.2.5.1 分支 331
4.2.5.2 循环 334
4.2.6 高级数据类型 338
4.2.6.1 数组 338
4.2.6.2 结构体 340
4.2.6.3 联合体 341
4.2.6.4 指针 342
4.2.7 函数 349
4.2.8 文件操作 351
4.3 C++ 标准库 354
4.3.1 C++ 标准库简介 354
4.3.2 STL 容器 356
4.3.2.1 STL 容器简介 356
4.3.2.2 迭代器 357

4.3.2.3 序列式容器 359
4.3.2.4 关联式容器 364
4.3.2.5 无序关联式容器 368
4.3.2.6 容器适配器 370
4.3.3 STL 算法 372
4.3.4 bitset 375
4.3.5 string 388
4.3.6 pair 390
4.4 C++ 进阶 393
4.4.1 类 393
4.4.2 命名空间 399
4.4.3 值类别 402
4.4.4 重载运算符 406
4.4.5 引用 410
4.4.6 常量 413
4.4.7 新版 C++ 特性 418
4.4.8 Lambda 表达式 428
4.4.9 pb_ds 436
4.4.9.1 pb_ds 简介 436
4.4.9.2 堆 437
4.4.9.3 平衡树 440
4.4.10 编译优化 442
4.5 C++ 与其他常用语言的区别 456
4.6 Pascal 转 C++ 急救 458
4.7 Python 速成 472
4.8 Java 速成 494
4.9 Java 进阶 498

5 算法基础 529
5.1 算法基础简介 530
5.2 复杂度 530
5.2.1 复杂度简介 530
5.2.2 均摊复杂度 535
5.3 枚举 538
5.4 模拟 541
5.5 递归 & 分治 543
5.6 贪心 549
5.7 排序 553
5.7.1 排序简介 553
5.7.2 选择排序 554
5.7.3 冒泡排序 557
5.7.4 插入排序 558
5.7.5 计数排序 560
5.7.6 基数排序 564
5.7.7 快速排序 571
5.7.8 归并排序 578

5.7.9 堆排序 582
5.7.10 桶排序 584
5.7.11 希尔排序 586
5.7.12 锦标赛排序 591
5.7.13 Tim 排序 595
5.7.14 排序相关 STL 599
5.7.15 排序应用 602
5.8 前缀和 & 差分 603
5.9 二分 615
5.10 倍增 631
5.11 构造 634

6 搜索 639
6.1 搜索部分简介 640
6.2 DFS（搜索） 640
6.3 BFS（搜索） 643
6.4 双向搜索 646
6.5 启发式搜索 651
6.6 A* 653
6.7 迭代加深搜索 656
6.8 IDA* 656
6.9 回溯法 660
6.10 Dancing Links 663
6.11 Alpha–Beta 剪枝 686
6.12 优化 694

7 动态规划 699
7.1 动态规划部分简介 700
7.2 动态规划基础 700
7.3 记忆化搜索 706
7.4 背包 DP 710
7.4.1 背包 DP 基础 710
7.4.2 背包 DP 进阶 715
7.5 区间 DP 720
7.6 DAG 上的 DP 721
7.7 树形 DP 725
7.8 状压 DP 730
7.9 数位 DP 733
7.10 插头 DP 742
7.11 计数 DP 773
7.12 动态 DP 775
7.13 概率 DP 780
7.14 DP 套 DP 787
7.15 DP 优化 796
7.15.1 DP 优化简介 796
7.15.2 单调队列/单调栈优化 800

7.15.3 斜率优化 804
7.15.4 四边形不等式优化 807
7.15.5 Slope Trick 优化 822
7.15.6 WQS 二分 838
7.15.7 状态设计优化 873

8 字符串 875
8.1 字符串部分简介 876
8.2 字符串基础 876
8.3 标准库 877
8.4 字符串匹配 878
8.5 字符串哈希 880
8.6 字典树 (Trie) 889
8.7 前缀函数与 KMP 算法 903
8.8 Boyer–Moore 算法 915
8.9 Z 函数（扩展 KMP） 933
8.10 AC 自动机 937
8.11 后缀数组 (SA) 957
8.11.1 后缀数组简介 957
8.11.2 最优原地后缀排序算法 971
8.12 后缀自动机 (SAM) 985
8.13 后缀平衡树 1007
8.14 广义后缀自动机 1015
8.15 后缀树 1024
8.16 Manacher 1035
8.17 回文树 1039
8.18 序列自动机 1050
8.19 最小表示法 1054
8.20 Lyndon 分解 1056
8.21 Main–Lorentz 算法 1059

9 数学 1065
9.1 数学部分简介 1066
9.2 布尔代数 1066
9.3 数字系统 1073
9.3.1 数字系统简介 1073
9.3.2 进位制 1073
9.3.3 平衡三进制 1081
9.3.4 格雷码 1083
9.4 位操作 1085
9.5 二进制集合操作 1093
9.6 高精度计算 1096
9.7 快速幂 1117
9.8 置换和排列 1124
9.9 弧度制与坐标系 1135
9.10 复数 1139

9.11 数论 1146
9.11.1 数论基础 1146
9.11.2 模算术简介 1155
9.11.3 素数 1165
9.11.4 最大公约数 1174
9.11.5 欧拉函数 1182
9.11.6 筛法 1185
9.11.7 分解质因数 1195
9.11.8 裴蜀定理 & 一次不定方程 1204
9.11.9 费马小定理 & 欧拉定理 1209
9.11.10 模逆元 1215
9.11.11 线性同余方程 1219
9.11.12 中国剩余定理 1222
9.11.13 升幂引理 1226
9.11.14 阶乘取模 1228
9.11.15 卢卡斯定理 1237
9.11.16 同余方程 1244
9.11.17 二次剩余 1249
9.11.18 阶 & 原根 1261
9.11.19 离散对数 1272
9.11.20 高次剩余 & 单位根 1277
9.11.21 数论分块 1300
9.11.22 狄利克雷卷积 1308
9.11.23 莫比乌斯反演 1317
9.11.24 狄利克雷双曲线法 & 杜教筛 1339
9.11.25 素数计数 1354
9.11.26 Powerful Number 筛 1379
9.11.27 Min_25 筛 1389
9.11.28 洲阁筛 1394
9.11.29 类欧几里得算法 1396
9.11.30 连分数 1417
9.11.31 Stern–Brocot 树与 Farey 序列 1468
9.11.32 二次域 1481
9.11.33 Pell 方程 1489
9.12 多项式与生成函数 1505
9.12.1 多项式与生成函数简介 1505
9.12.2 代数基本定理 1510
9.12.3 快速傅里叶变换 1513
9.12.4 快速数论变换 1525
9.12.5 快速沃尔什变换 1529
9.12.6 Chirp Z 变换 1536
9.12.7 多项式牛顿迭代 1540
9.12.8 多项式多点求值|快速插值 1544
9.12.9 多项式初等函数 1545
9.12.10 常系数齐次线性递推 1558
9.12.11 多项式平移|连续点值平移 1561

9.12.12 符号化方法 1566
9.12.13 Lagrange 反演 1573
9.12.14 形式幂级数复合|复合逆 1575
9.12.15 普通生成函数 1587
9.12.16 指数生成函数 1592
9.13 组合数学 1598
9.13.1 排列组合 1598
9.13.2 抽屉原理 1604
9.13.3 容斥原理 1604
9.13.4 斐波那契数列 1615
9.13.5 错位排列 1621
9.13.6 卡特兰数 1623
9.13.7 斯特林数 1631
9.13.8 贝尔数 1647
9.13.9 伯努利数 1649
9.13.10 Entringer Number 1654
9.13.11 Eulerian Number 1657
9.13.12 分拆数 1658
9.13.13 范德蒙德卷积 1665
9.13.14 Pólya 计数 1666
9.13.15 图论计数 1679
9.14 线性代数 1692
9.14.1 线性代数简介 1692
9.14.2 向量 1693
9.14.3 内积和外积 1697
9.14.4 矩阵 1702
9.14.5 初等变换 1711
9.14.6 行列式 1715
9.14.7 线性空间 1717
9.14.8 线性基 1724
9.14.9 线性映射 1741
9.14.10 特征多项式 1745
9.14.11 对角化 1752
9.14.12 Jordan 标准型 1755
9.15 线性规划 1760
9.15.1 线性规划基础 1760
9.15.2 单纯形法 1773
9.16 抽象代数 1791
9.16.1 基本概念 1791
9.16.2 群论 1798
9.16.3 环论 1816
9.16.4 域论 1835
9.16.5 Schreier–Sims 算法 1863
9.17 概率论 1875
9.17.1 基本概念 1875
9.17.2 条件概率与独立性 1877

9.17.3 随机变量 1879
9.17.4 随机变量的数字特征 1880
9.17.5 概率不等式 1884
9.18 博弈论 1887
9.18.1 博弈论简介 1887
9.18.2 公平组合游戏 1889
9.18.3 零和游戏 1910
9.18.4 非公平组合游戏 1916
9.19 数值算法 1916
9.19.1 插值 1916
9.19.2 数值积分 1927
9.19.3 高斯消元 1930
9.19.4 牛顿迭代法 1935
9.20 序理论 1937
9.21 杨氏矩阵 1946
9.22 拟阵 1951
9.23 Berlekamp–Massey 算法 1957

10 数据结构 1963
10.1 数据结构部分简介 1964
10.2 栈 1964
10.3 队列 1966
10.4 链表 1974
10.5 哈希表 1981
10.6 并查集 1985
10.6.1 并查集 1985
10.6.2 并查集复杂度 2007
10.7 堆 2011
10.7.1 堆简介 2011
10.7.2 二叉堆 2012
10.7.3 配对堆 2016
10.7.4 左偏树 2021
10.8 块状数据结构 2033
10.8.1 分块思想 2033
10.8.2 块状数组 2036
10.8.3 块状链表 2040
10.8.4 树分块 2043
10.8.5 Sqrt Tree 2047
10.9 单调栈 2053
10.10 单调队列 2054
10.11 ST 表 2058
10.12 树状数组 2064
10.13 线段树 2084
10.13.1 线段树基础 2084
10.13.2 线段树合并 & 分裂 2159
10.13.3 李超线段树 2166

10.13.4 猫树 2173
10.13.5 区间最值操作 & 区间历史最值 2174
10.13.6 Kinetic Tournament Tree 2189
10.14 划分树 2194
10.15 二叉搜索树 & 平衡树 2199
10.15.1 二叉搜索树 & 平衡树 2199
10.15.2 Treap 2207
10.15.3 Splay 树 2240
10.15.4 WBLT 2258
10.15.5 替罪羊树 2287
10.15.6 笛卡尔树 2297
10.15.7 Size Balanced Tree 2301
10.15.8 AVL 树 2327
10.15.9 红黑树 2348
10.15.10 左偏红黑树 2383
10.15.11 AA 树 2397
10.16 跳表 2402
10.17 可持久化数据结构 2409
10.17.1 可持久化数据结构简介 2409
10.17.2 可持久化线段树 2410
10.17.3 可持久化块状数组 2415
10.17.4 可持久化平衡树 2415
10.17.5 可持久化字典树 2423
10.17.6 可持久化可并堆 2425
10.18 树套树 2426
10.18.1 线段树套线段树 2426
10.18.2 平衡树套线段树 2427
10.18.3 线段树套平衡树 2427
10.18.4 树状数组套权值线段树 2429
10.18.5 分块套树状数组 2433
10.19 K-D Tree 2445
10.20 动态树 2456
10.20.1 Link Cut Tree 2456
10.20.2 全局平衡二叉树 2484
10.20.3 Euler Tour Tree 2492
10.20.4 Top Tree 2518
10.21 析合树 2553
10.22 PQ 树 2562
10.23 手指树 2573
10.24 霍夫曼树 2577

11 图论 2583
11.1 图论部分简介 2584
11.2 图论相关概念 2584
11.3 图的存储 2591
11.4 DFS（图论） 2598

11.5 BFS（图论） 2602
11.6 树上问题 2607
11.6.1 树基础 2607
11.6.2 树的直径 2618
11.6.3 树的中心 2623
11.6.4 树的重心 2625
11.6.5 最近公共祖先 2632
11.6.6 树链剖分 2644
11.6.7 树上启发式合并 2661
11.6.8 虚树 2669
11.6.9 树分治 2678
11.6.10 动态树分治 2689
11.6.11 AHU 算法 2698
11.6.12 树哈希 2703
11.6.13 树上随机游走 2711
11.7 有向无环图 2713
11.8 拓扑排序 2715
11.9 最短路问题 2721
11.9.1 最短路 2721
11.9.2 差分约束 2733
11.9.3 k 短路 2736
11.9.4 同余最短路 2745
11.10 生成树问题 2750
11.10.1 最小生成树 2750
11.10.2 最小树形图 2773
11.10.3 最小直径生成树 2778
11.11 斯坦纳树 2783
11.12 拆点 2790
11.13 连通性相关 2792
11.13.1 强连通分量 2792
11.13.2 双连通分量 2799
11.13.3 割点和桥 2809
11.13.4 圆方树 2815
11.13.5 点/边连通度 2829
11.14 环计数问题 2831
11.15 最小环 2840
11.16 2-SAT 2850
11.17 欧拉图 2857
11.18 哈密顿图 2862
11.19 二分图 2863
11.20 平面图 2866
11.21 弦图 2876
11.22 图的着色 2882
11.23 网络流 2892
11.23.1 网络流简介 2892
11.23.2 最大流 2893

11.23.3 最小割 2918
11.23.4 费用流 2921
11.23.5 上下界网络流 2928
11.23.6 Stoer–Wagner 算法 2931
11.24 图的匹配 2935
11.24.1 图匹配 2935
11.24.2 二分图最大匹配 2945
11.24.3 二分图最大权匹配 2969
11.24.4 一般图最大匹配 2982
11.24.5 一般图最大权匹配 2998
11.24.6 稳定匹配 3012
11.25 Prüfer 序列 3019
11.26 矩阵树定理 3026
11.27 LGV 引理 3038
11.28 最大团搜索算法 3042
11.29 支配树 3047
11.30 图上随机游走 3061

12 计算几何 3067
12.1 计算几何部分简介 3068
12.2 二维计算几何基础 3068
12.3 三维计算几何基础 3073
12.4 距离 3075
12.5 Pick 定理 3084
12.6 三角剖分 3086
12.7 凸包 3099
12.8 扫描线 3111
12.9 旋转卡壳 3122
12.10 半平面交 3127
12.11 平面最近点对 3132
12.12 随机增量法 3138
12.13 最小圆覆盖 3139
12.14 反演变换 3146

13 杂项 3155
13.1 杂项简介 3156
13.2 离散化 3156
13.3 双指针 3157
13.4 离线算法 3161
13.4.1 离线算法简介 3161
13.4.2 CDQ 分治 3162
13.4.3 整体二分 3185
13.4.4 莫队算法 3195
13.4.4.1 莫队算法简介 3195
13.4.4.2 普通莫队算法 3195
13.4.4.3 带修改莫队 3201

13.4.4.4 树上莫队 3205
13.4.4.5 回滚莫队 3214
13.4.4.6 二维莫队 3217
13.4.4.7 莫队二次离线 3222
13.4.4.8 莫队配合 bitset 3229
13.5 分数规划 3231
13.6 随机化 3238
13.6.1 随机函数 3238
13.6.2 随机化技巧 3248
13.6.3 爬山算法 3260
13.6.4 模拟退火 3264
13.7 悬线法 3266
13.8 有限状态自动机 3270
13.9 计算理论基础 3292
13.10 字节顺序 3299
13.11 约瑟夫问题 3300
13.12 表达式求值 3302
13.13 在一台机器上规划任务 3308
13.14 主元素问题 3309
13.15 Garsia–Wachs 算法 3312
13.16 15-puzzle 3314
13.17 Kahan 求和 3316
13.18 珂朵莉树/颜色段均摊 3318
13.19 空间优化简介 3324

14 专题 3327
14.1 RMQ 3328
14.2 并查集应用 3332
14.3 括号序列 3335
14.4 线段树与离线询问 3337

第一章

简介

1.1 Getting Started

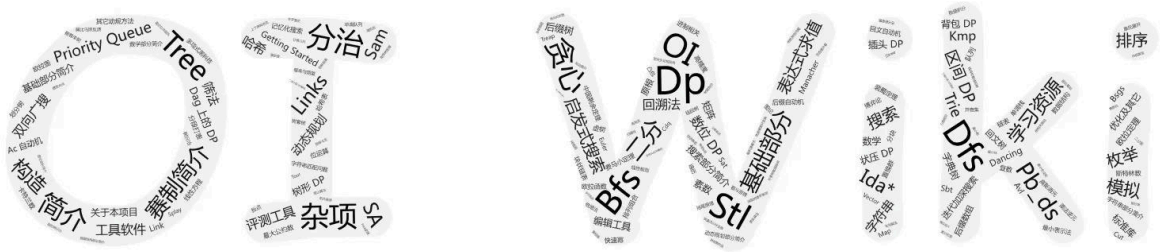
欢迎来到 OI Wiki!

Watch 201

[1]

Stars 27k

[2]



[3]

OI（Olympiad in Informatics，信息学奥林匹克竞赛）在中国起源于 1984 年，是五大高中学科竞赛之一.

ICPC（International Collegiate Programming Contest，国际大学生程序设计竞赛）由 ICPC 基金会（ICPC Foundation）举办，是最具影响力的大学生计算机竞赛. 由于以前 ACM 赞助这个竞赛，也有很多人习惯叫它 ACM 竞赛.

OI Wiki 致力于成为一个免费开放且持续更新的 **编程竞赛 (competitive programming)** 知识整合站点，大家可以在这里获取与竞赛相关的、有趣又实用的知识. 我们为大家准备了竞赛中的基础知识、常见题型、解题思路以及常用工具等内容，帮助大家更快速深入地学习编程竞赛中涉及到的知识.

本项目受 CTF Wiki[4] 的启发，在编写过程中参考了诸多资料，在此一并致谢.

外部参考资料

[1] GitHub watchers

[2] GitHub stars

[3] Word Art

[4] CTF Wiki

1.2 关于本项目

关于本项目

OI Wiki 致力于成为一个免费开放且持续更新的 **编程竞赛 (competitive programming)** 知识整合站点.

交流方式

本项目主要使用 Issues[1]/QQ[2]/Telegram[3] 进行交流沟通.

Telegram 群组链接为 @OI_wiki[4]，QQ 群号码为 588793226[5]，欢迎加入.

Note

原则上来说，上述群组是 OI Wiki **讨论群**，所以请尽量不要在群组中发表过多与 OI Wiki 无关的内容.

项目方针

- OI Wiki 不是什么→1.4@p21

外部参考资料

[1] Issues 

[2] QQ 

[3] Telegram 

[4] @OI_wiki 

[5] 588793226 

1.3 如何参与

在文章开始之前，OI Wiki 项目组全体成员十分欢迎您为本项目贡献页面. 正因为有了上百位像您一样的人，才有了 OI Wiki 的今天!

这篇文章将主要叙述参与 OI Wiki 编写的写作过程. 请您在撰稿或者修正 Wiki 页面以前，仔细阅读以下内容，以帮助您完成更高质量的内容.

贡献指南

请您在编辑前查看 OI Wiki 贡献指南[1] 和 项目方针→1.2@p16，以更好地和社区贡献者进行合作、交流.

参与协作

Warning

在开始编写一段内容之前，请查阅 Issues[2]，确认没有别人在做相同的工作之后，开个 新 issue[3] 记录待编写的内容.

Tip

在 Issues 中也有很多待修复/解决的问题，尤其是我们的迭代计划（Iteration Plan）. 从这里获取任务是一个很好的开始!

为了保证条目内容的专业性和准确性，我们建议您在编辑前先考虑以下几点:

1. **选择您熟悉的领域**: 请优先编辑那些与您的专业知识、学习背景或兴趣爱好相关的条目. 这有助于您创作出高质量的内容.
2. **谨慎对待新领域**: 如果您对某个主题还处于初学阶段或不太了解，建议您先通过阅读、学习加深理解，待有一定把握后再动手编辑.
3. **查阅相关资料**: 为条目添加内容或进行修订时，建议您先查阅权威文献和资料，确保信息准确无误. 也欢迎您在页面评论区或我们的社区提出问题，与其他编者交流讨论.

我们珍惜每位贡献者的热情和付出，也理解大家的专业水平不尽相同. 让我们携手合作，共同呵护这个知识的乐园，用准确、专业的内容去帮助更多读者. 期待您的贡献! 在这里引用维基百科的一句话:

> 不要害怕编辑，勇于更新页面! [1]

在 GitHub 上编辑

参与 OI Wiki 的编写 **需要** 一个 GitHub 账号（可以前往 GitHub 的账号注册页面[5] 页面注册），但 **不需要** 高超的 GitHub 技巧，即使你是一名新手，只要按照下面所述的步骤操作，也能够 **非常出色** 地完成编辑.

> **Tip**
>
> 在你的更改被合并到 OI Wiki 的主仓库之前，你对 OI Wiki 的内容所作出的修改均不会出现在 OI Wiki 的主站上，所以无需担心你的修改会破坏 OI Wiki 上正在显示的内容.
>
> 如果还是不放心，可以查看 GitHub 的官方教程[6].

编辑单个页面内的内容

1. 在 OI Wiki 上找到对应页面；
2. 点击正文右上方（目录左侧）的 **「编辑此页」**（edit）按钮，在确认您已经阅读了本页面和 格式手册→1.5@p23 后点击按钮根据提示跳转到 GitHub 进行编辑；
3. 在编辑框内编写你想修改的内容. 请注意，在修改和接下来的提交过程中，请 **关闭您的自动翻译软件**，因为它可能产生不必要的麻烦（例如您修改的文件有时会被其错误改名，从而影响目录结构）；
4. 编写完成后滚动到页面下方，按照本文中 commit 信息格式规范→1@p15 填写 commit 信息，之后点击 **Propose changes** 按钮提交修改. 点击按钮后，GitHub 会自动帮你创建一份 OI Wiki 仓库的分支，并将你的提交添加到这个分支仓库.
5. GitHub 会自动跳转到你的分支仓库的页面，此时页面上方会显示一个绿色的 **Create pull request** 按钮，点击后 GitHub 会跳转到一个创建 Pull Request 页面. 向下滚动检查自己所作出的修改没有错误后，按照本文中 Pull Request 信息格式规范→1@p15 一节中的规范书写 Pull Request 信息，然后点击页面上的绿色的 **Create pull request** 按钮创建 Pull Request.
6. 不出意外的话，你的 Pull Request 就顺利提交到仓库，等待管理员审核并合并到主仓库中即可.

在等待合并的时间里，你可以给他人的 Pull Request 提意见、点赞或者点踩. 如果有新消息，会在网页右上角出现提示，并附有邮件提醒（取决于个人设置中配置的通知方式）.

编辑多个页面内的内容

如果你需要同时编辑互相无关联的多个页面的内容，请按照上方的 编辑单个页面内的内容→1@p15 一节一次修改所有页面.

1. 打开 OI-Wiki/OI-Wiki[7] 仓库，点击键盘上的 `.` 按钮（或者将 URL 中的 `github.com` 更改为 `github.dev`）[2]，进入 GitHub 的网页版 VS Code 编辑器；
2. 在编辑器中作出对页面源文件的更改，可以使用页面右上方的预览按钮（或按下 `Ctrl+K` `V` 快捷键）在右侧打开预览界面；
3. 修改完成后使用左侧的 Source Control 选项卡，并按照本文中 commit 信息格式规范→1@p15 填写 commit 信息并提交，提交时会提示是否创建此仓库的分支，点击绿色的 **Fork Repository** 按钮即可.
4. 提交后会在网页上方的中央弹出一个提示框，在第一次的提示框内填写标题，第二次的提示框内填写此提交要提交到的仓库内分支名称，之后右下角会弹出一个提示框，内容类似于 `Created Pull Request #1 for OI-Wiki/OI-Wiki.`，点击蓝字链接即可查看该 Pull Request.

向 Pull Request 追加更改

1. 打开 OI-Wiki 的 Pull Request 列表[9]，找到您提交的 Pull Request 并点击.
2. Pull Request 页面的标题下方将会有一段例如 `<您的 ID> wants to merge x commits into OI-wiki:master from <您的 ID>:patch-1` 的文字，点击 `<您的 ID>:patch-1` 部分.

---

[1] 维基百科：新手入门/编辑[4]

[2] Web-based editor - GitHub Codespaces - GitHub Docs[8]

3. 您应该会被重定向到您的分支仓库中，而且文件列表左上角的分支名称是你提交 Pull Request 的分支名称（在本示例中应为 `patch-1`）.
4. 进行您需要的更改.
    - 如果您需要编辑单个文件或多个互相无关联的页面的内容，请直接找到你要的文件并进行更改，更改完成后滚动到页面下方，按照本文中 commit 信息格式规范→1@p15 填写 commit 信息，之后点击 **Commit changes** 按钮提交修改.
    - 如果您需要编辑多个文件，点击键盘上的 `.` 按钮（或者将 URL 中的 `github.com` 更改为 `github.dev`）[3]，进入 GitHub 的网页版 VS Code 编辑器并作出更改. 然后使用左侧的 Source Control 选项卡，并按照本文中 commit 信息格式规范→1@p15 填写 commit 信息并提交修改.
5. 这时你的更改会被自动追加在您的 Pull Request 中.

使用 Git 在本地进行编辑

> **⚠ Warning**
>
> 对于一般用户，我们更推荐使用上方所述的 GitHub 的 Web 编辑器进行编辑.

虽然大多数情况下您可以直接在 GitHub 上进行编辑，但对于一些较为特殊的情况（如需要使用 GPG 签名），我们更推荐使用 Git 在本地进行编辑.

大致流程如下：

1. 将主仓库 Fork 到自己的仓库中；
2. 将 Fork 后的分支仓库克隆（clone）到本地；
3. 在本地进行修改后提交（commit）这些更改；
4. 将这些更改推送（push）到你克隆的分支仓库；
5. 提交 Pull Request 至主仓库.

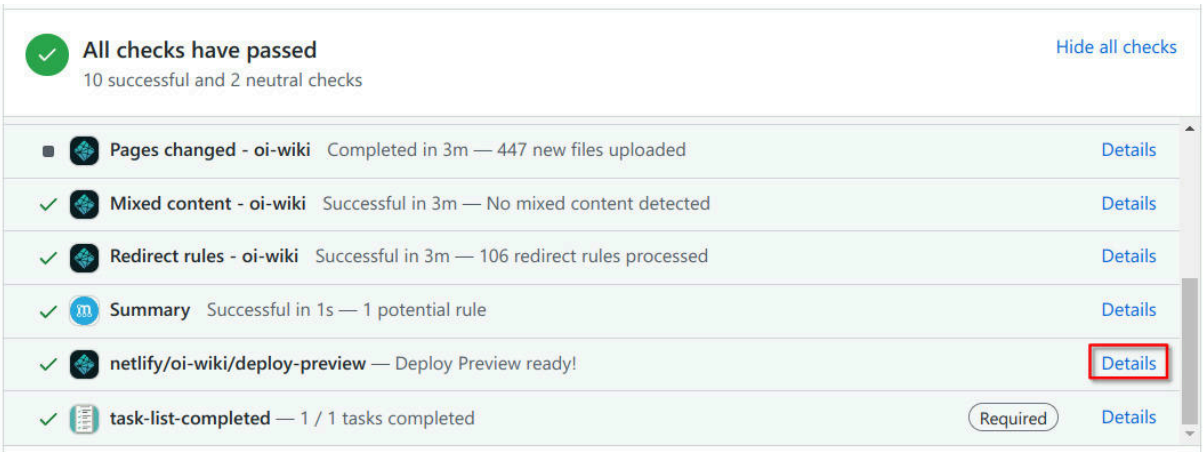
详细的操作方式可以参考 Git→3.13@p308 页面.

向 Pull Request 追加更改

在 clone 下来的本地分支仓库中继续进行修改，并提交（commit）以及推送（push）这些更改即可. 你的更改会被自动追加在 Pull Request 中.

在构建的网页中预览变更

在 Pull Request 页面下方可以找到测试页面，点击 netlify/oi-wiki/deploy-preview 一项的 Details 链接（如下图），可以进入自动构建的，由您变更后的页面供您预览.



[3] Web-based editor - GitHub Codespaces - GitHub Docs[10]

对于目录和引用的变更

通常情况下，如果您需要添加一个新页面，或者修改已有页面在目录中的链接，您就需要对 `mkdocs.yml`[11] 文件作出改动.

添加新页面可以参考既有的格式. 但除非是进行重构或修正名词, 否则 **我们不建议对既有页面的引用链接进行修改**，Pull Requests 中不必要的修改也将被驳回.

如果您坚持要修改链接，请注意更新 author 字段和重定向文件.

author 字段

GitHub API 在文件目录变更后不能跟踪统计, 所以我们在文件头手动维护了一个作者列表来解决这个问题. author 字段位于整个 Markdown 文件的开头，形如 `author: Ir1d, cjsoft`，相邻两个 ID 之间用逗号加空格隔开. 这里的 ID 是 GitHub 的用户名，即 GitHub profile 的地址（例如 https://github.com/Ir1d[12] 中的 `Ir1d`）.

修改链接时，需要将当前页面中的 contributors 逐一填入 author 字段.

重定向文件

在修改链接时，为了避免在站外引用时出现死链，需要修改重定向文件.

`_redirects`[13] 文件用于生成 netlify 的配置[14] 和 用于跳转的文件[15].

每一行表示一个重定向规则，分别写跳转的起点和终点的 url（不包含域名）:

```
/path/to/src /path/to/desc
```

注：所有跳转均为 301 跳转，只有在修改目录中 url 造成死链的时候需要修改.

Commit 信息格式规范

对于提交时需要填写的 commit 信息，请遵守以下几点基本要求:

1. commit 摘要请简要描述这一次 commit 改动的内容. 注意 commit 摘要的长度不要超过 50 字符，超出的部分会自动置于正文中.
2. 如果需要进一步描述本次 commit 内容，请在正文中详细说明.

对于 commit 摘要，推荐按照如下格式书写:

```
<修改类型>(<文件名>): <修改的内容>
```

修改类型分为如下几类:

- `feat`: 用于添加内容的情况.
- `fix`: 用于修正现有内容错误的情况.
- `refactor`: 用于对一个页面进行重构（较大规模的更改）的情况.
- `revert`: 用于回退之前更改的情况.

Pull Request 信息格式规范

对于 Pull Request，请遵守以下几点要求:

1. 标题请写明本次 PR 的目的（做了 **什么** 工作，修复了 **什么** 问题）.
2. 内容请简要叙述修改的内容. 如果修复了一个 issue 的问题，请在内容中添加 `fix #xxxx` 字段，其中 `xxxx` 代表 issue 的编号.
3. 请您仔细阅读 贡献指南[16] 和 社区公约[17]，并在同意后勾选 PR 模板中的框，表示您同意了以上指南和公约.

对于 Pull Request 的标题，推荐使用如下格式书写:

```
<修改类型>(<文件名>): <修改的内容> (<对应 issue 的编号>)
```

修改类型分为如下几类：

- `feat`：用于添加内容的情况.
- `fix`：用于修正现有内容错误的情况.
- `refactor`：用于对一个页面进行重构（较大规模的更改）的情况.
- `revert`：用于回退之前更改的情况.

示例：

- `fix(ds/persistent-seg): 修改代码注释使描述更清晰`
- `fix: tools/judger/index 不在目录中 (#3709)`
- `feat(math/poly/fft): better proof`
- `refactor(ds/stack): 整理页面内容`

协作流程

1. 在收到一个新的 Pull Request 之后，GitHub 会给 reviewer 发送邮件；
2. 与此同时，在 GitHub Actions[18] 和 Netlify[19] 上会运行两组测试，它们会把进度同步在 PR 页面的下方．GitHub Actions 主要用来确认 PR 中内容的修改不会影响到网站构建的进程；Netlify 用来把 PR 中的更新构建出来，方便 reviewer 审核（在测试完成后点击 Details 可以了解更多）；
3. reviewer 可能会发现问题，并提出 `review` 或 `suggested changes`（建议更改，显示为灰色图标）/ `requested changes`（强制更改，显示为红色图标，只会在 reviewer 拥有 repo 写权限时出现）．一般来说，reviewer 也会附上建议和需要进行的更改，在这时，您将会需要继续向 Pull Request 追加其他更改．更改的方法可以参考 `在 GitHub 上编辑` 或者 `使用 Git 在本地进行编辑` 部分的 `向 Pull Request 追加更改` 部分.
4. 在足够多 reviewer 投票通过一个 PR 之后，这个 PR 才可以合并到 master 分支中；
5. 在合并到 master 分支之后，GitHub Actions 会重新构建一遍网站内容，并更新到 gh-pages 分支；
6. 这时服务器才会拉取 gh-pages 分支的更新，并重新部署最新版本的内容.

参考资料与注释

[1] OI Wiki 贡献指南

[2] Issues

[3] 新 issue

[4] 维基百科：新手入门/编辑

[5] GitHub 的账号注册页面

[6] GitHub 的官方教程

[7] OI-Wiki/OI-Wiki

[8] Web-based editor - GitHub Codespaces - GitHub Docs

[9] OI-Wiki 的 Pull Request 列表

[10] Web-based editor - GitHub Codespaces - GitHub Docs

[11] `mkdocs.yml`

[12] https://github.com/Ir1d

[13] `_redirects`

[14] netlify 的配置

[15] 用于跳转的文件

[16] 贡献指南

[17] 社区公约

[18] GitHub Actions

[19] Netlify

1.4 OI Wiki 不是什么

Authors: abc1763613206, HeRaNO, NachtgeistW, r-value, Tiphereth-A, wlbksy, YZircon, 0zu-cc, real01bit

> ⚠ **注意**
>
> 作为项目方针的一部分，本页面十分重要，每个贡献者都应确保您的贡献满足如下条件.

OI Wiki 不是发表原创研究的场所

作为一个 Wiki，OI Wiki 不是发表 原创研究[1]（如 **新理论及解法、原创观点、自创定义或词语** 等）的场所. 例如：

- 您发现了某题目的非常规做法，若您不能证明该做法已经被应用于其他题目中，则 **不应** 在 OI Wiki 中开设单独的界面.
- 您提出了新的算法或数据结构，若您不能证明该内容已经被用于解决编程竞赛中的某一类问题，则 **不应** 将其提交至 OI Wiki.

OI Wiki 不是新闻的收集处

作为泛中文为主语境之下、以编程竞赛相关内容为主的知识整合站点，OI Wiki 侧重于提供 **稳定沉淀并已取得广泛认可** 的信息.

换言之，除非是权威机构发布的信息（如中国计算机学会发布的赛制更新），您所贡献的内容应当是已经经过检验沉淀，获得了广泛认可，且在一段时间内不会产生时效性问题的信息. 例如：

- 您发现 X 博士在某个位置发布了新的算法，若您想将其加入 OI Wiki 中，此时您应该观察其是否能获得广泛认可（如可作为正式比赛中的泛用优秀解法），且与 OI Wiki 中现存的算法有一定的区分性.
- 您发现 OI Wiki 中提到的某软件或人物出现了舆情问题，此时请您关注评论区等的相关公告，切勿重复开 issue 等说明问题. OI Wiki 的目的在于记载 **可长久流传的信息**，而不是 **带有时效性的临时信息**.

OI Wiki 不是档案馆

OI Wiki 主站不会收录各类 **文献、课件、讲义、说明书** 等资料，若您想提交有关编程竞赛的资料，请移步至 OI-wiki/libs[2].

OI Wiki 不是宣传工具

OI Wiki 是一个知识整合站点，**不是** 演讲台、论坛、宣传工具等，因此在添加或修改条目时，请勿：

- 发表观点或评论：OI Wiki 的内容必须 **客观中立**，如果您想要发表与 OI Wiki 有关的观点与评论，请移步至评论区或 issue 页面. 如果您想要发表的观点与评论与 OI Wiki 无关，请移步至个人博客或论坛；
- 作任何形式的宣传行为：出于知识收录需要，OI Wiki 可以接受对「在编程竞赛领域已经 **广为人知** 的网站或软件」的介绍. 除此之外，OI Wiki 不会接受任何非赞助商提供的任何宣传性质的内容.

OI Wiki 不是权威机构

作为一个主要依靠用户贡献的社区项目，OI Wiki 不具有权威性，不应也没有能力作为一个权威机构. OI Wiki 可以作为学习编程竞赛相关知识的参考，而不是标准教科书. 作为一个社区维护的参考站点，您不应将 OI Wiki 用作最终的权威标准（如作为编程竞赛的「考纲」），也不应盲目采信所有 OI Wiki 的内容. OI Wiki 不对由使用 OI Wiki 的内容而产生的任何后果负责.

OI Wiki 不是个人博客

OI Wiki 基于「Wiki」一词，本质上是依靠用户 **协同编辑内容** 的社区．这就意味着，您所提供的内容应当 **尽量减轻个人色彩**，服务于整个 Wiki，关注您所贡献的相关词条的结构．切勿以撰写个人博客文章的思维对待 Wiki 中的条目．例如：

- 您 **不应** 将 OI Wiki 作为您个人博客的导航站点；
- 您 **不应** 在词条的描述中加入条目历史、个人吐槽、冷笑话等无关内容．

以上例子的反面均为撰写个人博客时的常见思维，而这些在提倡客观中立的 OI Wiki 中是不受欢迎的．

OI Wiki 不是译名标准委员会

OI Wiki 的目标群体不是历史学家及语言学家，因此在外文名词的翻译中，OI Wiki 倾向于使用中文语境下 **已经获得了广泛认同** 的译文，即使它们可能是有瑕疵的．

在译文已经获得了广泛熟知认同的基础上，盲目因「正确性」而修改现有的译文只会引发更大的混淆与混乱，与之引发的文字游戏也会影响沟通效率．

OI Wiki 不是百科全书

OI Wiki 收录的内容应限定在「已经被应用于编程竞赛」的计算机科学、数学等领域的部分知识．其他与编程竞赛无关的领域或知识均不适合收录于 OI Wiki．

例如，如下的领域或知识 **不适合** 收录于 OI Wiki：

- 历史，艺术等无关领域；
- TBN 矩阵[3] 等虽然从属于有关领域但目前不能应用于编程竞赛的知识；
- PID 控制[4]、有限元法[5] 等目前不能应用于编程竞赛的领域算法；
- 深度学习[6]、强化学习[7] 等目前不能应用于编程竞赛的通用算法．

OI Wiki 不是编程语言的文档和学习指南

OI Wiki 收录的算法相较于代码实现，应更关心算法本身．算法的具体实现仅作为一种更加细致的理解或者实现提示，而不是给具体的语言学习者以方便．如果您想要学习某种语言，您应该阅读该语言的官方文档等资料．

出于知识收录需要，OI Wiki 可以收录编程竞赛常用语言的简单使用指南．除此之外，OI Wiki 不会收录诸如「某编程语言的某标准库里某函数的实现细节等」与编程竞赛和算法关系不大的内容．

外部参考资料

[1] 原创研究

[2] OI-wiki/libs

[3] TBN 矩阵

[4] PID 控制

[5] 有限元法

[6] 深度学习

[7] 强化学习

1.5 格式手册

在文章开始之前，OI Wiki 项目组全体成员十分欢迎您为本项目贡献页面．正因为有了上百位像您一样的人，才有了 OI Wiki 的今天！

本页面将列出在 OI Wiki 编写过程时推荐使用的格式规范与编辑方针．请您在撰稿或者修正 Wiki 页面以前，仔细阅读以下内容，以帮助您完成更高质量的内容．

如果您已迫不及待，想要快速上手，建议先阅读 太长不看版→1@p15 与 图片举例→1@p15 的章节．

Changelog

注意：只记录和写作、审阅等相关的改动，不记录格式修正等改动．

时间	主要内容	相关 Issue/Pull Request 链接
2026-02-22	完善引号使用的相关规定	#6793[1]
2026-01-07	要求使用全角句点替代中文句号	#6746[2]
2025-08-10	添加格式手册的格式要求；代码：补充片段代码相关要求	#6412[3]
2025-08-10	添加 Changelog 与 TL;DR	#6409[4]
2024-10-08	代码：为适应全平台测试完善了格式要求	#5912[5]，#5924[6]
2024-03-26	引用 OJ 题目链接时使用原链接，而不是镜像链接	#5482[7]
2023-10-09	主题插件：新增选项卡[1]的格式要求	#5152[9]
2023-07-23	对于工具类内容的下载安装等内容，要求引用官方文档	#5023[10]
2023-04-15	补充引号的使用规范	#4792[11]
2023-03-28	LaTeX：数学符号表	#4587[12]
2023-03-02	补充全半角标点与连接号的使用规范	#4726[13]
2022-12-13	主题插件：移除嵌套折叠框的阴影样式要求	#4500[14]
2022-08-09	引用内链的某一节内容时，使用中文标题	#4057[15]
2022-06-12	完善目录更改的相关要求[2]	#4043[16]
2021-09-09	主题插件：补充折叠框相关要求	#3517[17]
2021-09-03	LaTeX：`\Leftrightarrow` → `\iff`	#3499[18]
2021-08-18	代码：新增例题代码的格式要求	#3447[19]
2021-08-12	图片：动图优先使用 APNG 格式	#3422[20]
2021-06-29	图片：建议同时提交源文件	#3255[21]
2021-05-29	代码：删除大括号不换行的要求，补充可读性要求	#3197[22]
2021-03-15	站点维护：规范 Pull Request 的合并方式[3]	#3061[23]
2021-02-01	LaTeX：`\lt` → `<`，`\gt` → `>`	#2950[24]
2021-01-27	建议在 互联网档案馆[25] 保存外链备份	#2918[26]
2020-09-19	站点维护：Commit Message 与 Pull Request 标题的书写要求[4]	#2744[27]
2020-10-18	图片：优先使用 SVG 格式	#2215[28]
2020-08-05	LaTeX：新增多字母变量的格式要求	#2502[29]
2020-07-28	LaTeX：`cases` 环境禁止超过两列	#2466[30]
2020-07-24	LaTeX：`{n \choose m}` → `\dbinom{n}{m}`	#2442[31]
2020-07-20	Markdown：禁用删除线语法	#2422[32]

[1]选项卡：参见 Content tabs[8]．
[2]移至 如何贡献→1.3@p17．
[3]该规范写入了 <未找到引用 OIwikidocseditlanding> 并发布了公告，并未写入本文档．
[4]移至 如何贡献→1.3@p17．
[5]折叠框：参见 Collapsible Blocks[33]，有时我们也用「Details 语法」指代该语法，因其从功能上与 HTML 中的 `<details>` 元素[34] 功能一致．

2020-07-19	主题插件：要求保留折叠框[5]中空行的缩进空格；LaTeX：追加对数学公式的格式要求	#2412[35]
2020-07-11	最初版本	#2350[36]

太长不看版

为方便初次阅读本文档的用户，本节列举该手册中的若干重点事项：

- 文件存储：
  - 使用小写文件名，以 `-` 代替空格．详见 SAVE-1→1@p15.
  - 不要插入外链图片．详见 SAVE-2→1@p15.
  - 图片尽可能使用 SVG 格式，只应使用 SVG 1.1 标准．详见 SAVE-3→1@p15.
  - 动图应使用 SVG 或 APNG 格式．详见 SAVE-4→1@p15.
  - 有源文件的图片建议同时提交源文件．详见 SAVE-5→1@p15.
  - 插入外链时建议同时插入快照链接．详见 SAVE-6→1@p15.
  - 不要以插入外链的方式插入内链．详见 SAVE-7→1@p15.
- 标点符号：
  - 规范使用标点符号．在每句话的末尾添加 **句号**．详见 PUNC-1→1@p15 至 PUNC-7→1@p15.
  - 注意区分连接号（hyphen、en dash、em dash）．详见 PUNC-8→1@p15.
- Markdown 语法与主题扩展语法：
  - 只应使用二级、三级、四级标题．不要使用标题替代加粗．不要在标题写 LaTeX 公式．详见 LINT-1→1@p15、MDFM-1→1@p15、CONT-4→1@p15、CONT-9→1@p15.
  - 使用折叠框[6]语法和选项卡[7]语法时，须保持内部缩进一致，**对空行也是如此．不要漏掉** 空行的空格缩进．详见 LINT-6→1@p15、MDFM-6→1@p15.
  - 不要使用删除线 `~~foo~~` 语法．详见 LINT-3→1@p15.
  - 行间公式应写作

    ```
    $$
    a^{2}=b^{2}+c^{2}
    $$
    ```

    而不是 `$$a^{2}=b^{2}+c^{2}$$`．详见 LINT-5→1@p15.
  - 使用折叠框而不是块引用（Blockquotes）．详见 MDFM-5→1@p15.
  - 代码块只应使用 ``` 语法，且须标注语言．详见 LINT-7→1@p15、MDFM-3→1@p15.
- LaTeX 公式：
  - 不应与 数学符号表→1.6@p48 相冲突．详见 MATH-1.1→1@p15.
  - 注意字体的使用，详见 MATH-1.2→1@p15、MATH-1.15→1@p15、MATH-2.6→1@p15、MATH-2.7→1@p15.
  - 不要滥用 LaTeX 公式．详见 MATH-1.14→1@p15.
  - 不要 LaTeX 公式里使用程序设计语言的表示方式．（如：不要使用 $a == b$、$a << 1$、$a\%b$.）不要使用中括号连缀（$a[i][j]$）．详见 MATH-1.9→1@p15、MATH-1.10→1@p15.
- 代码：
  - 尽可能简洁易懂，避免压行等不良习惯．尽可能保证可读性，突出算法思想．详见 CONT-10→1@p15.
  - 不推荐直接把代码插入 Markdown 文档中．详见 CODE-1.1→1@p15、CODE-1.2→1@p15.

[6]折叠框：参见 Collapsible Blocks[37]，有时我们也用「Details 语法」指代该语法，因其从功能上与 HTML 中的 `<details>` 元素[38] 功能一致.

[7]选项卡：参见 Content tabs[39].

对本文档的格式要求

- FREQ-1：修订格式手册的条目时需同时补充 Changelog．若只是修正格式，则无需补充 Changelog．
- FREQ-2：除 太长不看版→1@p15 一节外，格式手册的条目都需要有不重复的编号，编号需要匹配正则表达式 `(?<category>[A-Z]{4})-(?<id>[1-9][0-9]*(?:\.[1-9][0-9]*)*)`，其中 `category` 应具有直观的含义．说明文字不需要有编号．
- FREQ-3：太长不看版→1@p15 的条目必须来自格式手册其他章节的内容，且需在末尾引用对应的条目编号．
- FREQ-4：条目的编号一旦确定就不应更改．如果确需更改（如删除、合并条目），则应用类似「已废止」、「迁移至 XXXX-id」的文字注明．

贡献文档要求

当你打算贡献某部分的内容时，你应该尽量熟悉以下三部分：

- 文档存储的格式
- 文档的合理性
- remark-lint 和 LATEX 公式的格式要求

文档引用与存储的格式

- SAVE-1：**文件名请务必都小写，以 - 分割．** 例如：`file-name.md`．
- SAVE-2：请务必确保文档中引用的 **外链** 图片已经全部转存到了 **本库内** 对应的 `images` 文件夹中（防止触发某些网站的防盗链），建议处理成 `MD 文档名称 + 编号` 的形式（可参考已有文档中图片的处理方式）．例如：本篇文档的文件名称为 format，则文档中引用的第一张图片的名字为 `format1.png`．
- SAVE-3：推荐使用 SVG 格式的图片[8]，以获取较好的清晰度和缩放效果．由于 OI Wiki 各组件对 SVG 标准的兼容性不同，所以您的图片应基于 SVG 1.1[41] 标准．
- SAVE-4：动图如果无法或者不会制作 SVG 格式的，则推荐使用 APNG 格式[9] 的文件．Windows 用户可使用 ScreenToGif[43] 录制，Linux 用户可使用 Peek[44] 录制，注意需要在设置里调整为录制 APNG．其他情况则推荐先制作为 MP4 等视频文件再转换为 APNG，如果使用 ffmpeg 则可以使用 `ffmpeg -i filename.mp4 -f apng filename.apng -plays 0` 转换．[10]
- SAVE-5：同时具有源文件和导出图像的图片（例如 JPG 文件与 PSD 文件或者 SVG 图像与 TikZ TeX 源代码），建议将源文件以与图片相同的文件名保存于同一目录下．
- SAVE-6：请确保您的文档中的引用链接的稳定性．**不推荐** 引用 **自建** 服务中的资源（如自建 OJ 里的题目）．建议在添加时同时将该外链存于互联网档案馆[11]，以防无法替代的链接失效．
- SAVE-7：站内链接请去掉网站域名，并且使用相对路径链接对应 `.md` 文件．例如，在本页面（`intro/format`）中链接杂项简介（`misc`），应使用 `[杂项简介](../misc/index.md)`．可以在链接中添加 hash 来链接到某一节，例如 `[Pull Request 信息格式规范](./htc.md#pull-request-信息格式规范)`→1.3@p17，hash 的值可以通过位于每个标题右侧的按钮或者位于网页右侧的目录中的链接得到．

文档的合理性

合理性，指所编写的 **内容** 必须具有如下的特性：

- STRC-1：由浅入深，内容的难度应该具有渐进性．
- STRC-2：逻辑性．
  - STRC-2.1：对于算法或数学概念类内容的撰写应该尽量包含以下的内容：
    1. 原理：说明该内容对应的原理；

[8] SVG|MDN[40]
[9] APNG[42]
[10] OI-wiki/OI-wiki#3422[45]
[11] Save Page in Internet Archive[46]

2. 例子：给出 1 ~ 2 个典型的例子；
3. 题目：在该标题下，**只需要给出题目名字和题目链接**. 对于算法类题目，题目链接 OJ 的优先级为：原 OJ（国外 OJ 要求国内可流畅访问）> UOJ > LOJ > 洛谷.

示例页面：IDA*→6.8@p670

- STRC-2.2：对于工具类内容的撰写应该尽量包含以下的内容：
  1. 简介：阐明该工具的背景与用途.
  2. 配置方式：详细给出配置环境与使用的过程，下载与安装方法建议尽量引用官方文档.

  示例页面：WSL (Windows 10)→3.7@p248

除现有内容质量较低的情况外，建议尽量从 **补充** 的角度来做贡献，而非采取直接覆盖的方式. 如果拿不准主意，可以参考 关于本项目的交流方式→1.2@p16 一节，与 OI Wiki 项目组联系.

文档的基本格式要求

Remark-lint 的格式要求

remark-lint[47] 可以自动给项目内文件统一风格. OI Wiki 现在启用的配置文件托管在 .remarkrc[48].

在配置过程中 OI Wiki 项目组也遇到了一些 remark-lint 不能很好处理的问题，所以请严格按照下列要求编辑文档：

- LINT-1：不要使用如 `<h1>` 或者 `# 标题` 的一级标题.
- LINT-2：标题要空一个英文半角空格，例如：`##␣简介`.
- LINT-3：由于 remark-lint 不能很好地处理删除线，因此请不要使用删除线语法（不使用删除线语法的另外一个原因是，删除线划去的内容大多为「抖机灵」性质，对读者理解帮助不大，不符合下面的「文本内容的格式要求」中 对内容表述的要求→1@p15）.
- LINT-4：列表：
  - LINT-4.1：列表前要有空行，新开一段.
  - LINT-4.2：使用有序列表（如 `1.␣例子`）时，点号后要有空格.
- LINT-5：行间公式前后各要有一行空行，否则会被当做是行内公式.
- LINT-6：使用 `???` 或 `!!!` 开头的 Details 语法时，每一行要包括在 Details 语法的文本框的文本，开头必须至少有 4 个空格.

即使是空行，也必须保持与其他行一致的缩进. 请不要使用编辑器的自动裁剪行末空格功能.

✓ **示例**

下面的代码中用 `␣` 表示空格 ␣.

```
???+ warning
␣␣␣␣请记得在文本前面添加 4 个空格. 其他的语法还是与 Markdown 语法一致.
␣␣␣␣
␣␣␣␣不添加 4 个空格的话，文本就不会出现在 Details 文本框里了.
␣␣␣␣
␣␣␣␣这个`???`是什么的问题会在 [下文](#MDFM-5) 解答.
```

⚠ **Warning**

请记得在文本前面添加 4 个空格. 其他的语法还是与 Markdown 语法一致.

不添加 4 个空格的话，文本就不会出现在 Details 文本框里了.

这个 `???` 是什么的问题会在 下文→1@p15 解答.

- LINT-7：代码样式的纯文本块请使用 ```` ```text ````．直接使用 ```` ``` ```` 而不指定纯文本块里的语言，可能会导致内容被错误地缩进．

标点符号的使用

- PUNC-1：请在每句话的末尾添加 **句号**．
- PUNC-2：请正确使用 **全角** 标点符号与 **半角** 标点符号．汉语请使用全角符号，英语请使用半角符号．中文中夹用英文时，请参考 中文出版物夹用英文的编辑规范[49]．特别地，请用全角句点「．」替代中文句号「。」．
- PUNC-3：由于 `“……”` 和 `‘……’` 未区分全半角，请使用 `「……」` 作为全角双引号，`"..."` 作为半角双引号，`『……』` 作为全角单引号，`'...'` 作为半角单引号．
- PUNC-4：注意区分 **顿号** 与 **逗号** 的使用．
- PUNC-5：注意 **括号** 的位置．句内括号与句外括号的位置不同．
- PUNC-6：通常使用 **分号** 来表示列表环境中各复句之间的关系．
- PUNC-7：对于有序列表，推荐在每一项的后面添加 **分号**，在列表最后一项的后面添加 **句号**；对于无序列表，推荐在每一项的后面添加 **句号**．
- PUNC-8：注意区分各种不同的连接号，如 hyphen（一般使用 U+002D hyphen-minus（-），即键盘上的「减号」代替），U+2013 en dash（–）和 U+2014 em dash（—）．（英文中连接多个人名时，须用 en dash，但是极常误用为 hyphen．其他误用较为罕见，基本上只需记住这一点即可．）详见 连接号 - 维基百科[50]．

> ✓ **示例**
>
> - 中学生学科竞赛主要包括信息学奥林匹克竞赛、信息学奥林匹克竞赛、信息学奥林匹克竞赛、信息学奥林匹克竞赛和信息学奥林匹克竞赛（谁写的这个示例，建议抬走）．
> - 「你吃了吗?」李四问张三．
> - 我想对你说：「我真是太喜欢你了．」
> - 「苟利国家生死以，岂因祸福避趋之!」
> - 张华考上了大学；李萍进了技校；我当了工人：我们都有美好的前途．[12]
> - 以下是这个算法的基本流程：
>   1. 初始化到各点的距离为无穷大，将所有点设置为未被访问过，初始化一个队列；
>   2. 将起点放入队列，将起点设置为已被访问过，更新到起点的距离为 0；
>   3. 取出队首元素，将该元素设置为未被访问过；
>   4. 遍历所有与此元素相连的边，若到这个点存在更短的距离，则进行松弛操作；
>   5. 若这个点未被访问过，则将这个点放入队列，且设置这个点为已经访问过；
>   6. 回到第三步，直到队列为空．
> - KMP 算法（Knuth–Morris–Pratt algorithm, KMP algorithm）由 Knuth、Pratt 和 Morris 在 1977 年共同发布．[13]

Markdown 格式与主题扩展格式要求

- MDFM-1：表示强调时请使用 `**SOMETHING**` 和 `「」`，而非某级标题，因为使用标题会导致文章结构层次混乱和（或）目录出现问题．
- MDFM-2：当需要引用题目链接时，应尽可能使用原 OJ 题库中的链接而不是镜像链接．

---

[12] （冒号）表示总结上文．

[13] 科学技术名称的英文全称与其缩略形式间，应使用英文逗号．中文句子内夹用了用以注释、补充或说明的英文句子或语段，该英文句子或语段用中文圆括号标示．

- MDFM-3：请正确使用 Markdown 的区块功能．插入行内代码请使用一对反引号包围代码区块；行间代码请使用一对 ``` 包围代码区块，其中反引号就是键盘左上角波浪线下面那个符号，行间代码请在第一个 ``` 的后面加上语言名称（如：```cpp）．

  ✓ **示例**

  ````
  ```cpp
  // #include<stdio.h>     //不好的写法
  #include <cstdio>  //好的写法
  ```
  ````

  ```cpp
  // #include<stdio.h>     //不好的写法
  #include <cstdio>  //好的写法
  ```

- MDFM-4：「参考资料与注释」使用 Markdown 的脚注功能进行编写．格式为：

  ```
  文本内容．[^脚注名]
  [^脚注名]：参考资料内容．注意：冒号是英文冒号，冒号后面跟着一个空格．
  ```

  脚注名既可以使用数字也可以使用文本．脚注名摆放的位置与括号的用法一致．为美观起见，建议同一个页面内的脚注名遵循统一的命名规律，如：ref1、ref2、note1……

  脚注的内容统一放在 `## 参考资料与注释` 二级标题下．

  ✓ **示例**

  ````
  当 `#include <cxxxx>` 可以替代 `#include <xxxx.h>` 时，应使用前者．[^ref1]

  2020年1月21日，CCF宣布恢复NOIP．[^ref2]

  ## 参考资料与注释

  [^ref1]: [cstdio stdio.h namespace](https://stackoverflow.com/questions/10460250/cstdio-stdio-h-namespace)

  [^ref2]: [CCF关于恢复NOIP竞赛的公告-中国计算机学会](https://www.ccf.org.cn/c/2020-01-21/694716.shtml)
  ````

  当 `#include <cxxxx>` 可以替代 `#include <xxxx.h>` 时，应使用前者．[14]

  2020 年 1 月 21 日，CCF 宣布恢复 NOIP．[15]

- MDFM-5：建议使用主题扩展的 `???+note` 格式（即 Collapsible Blocks[53]）来描述题面和参考代码．也可以用这种格式来展示其他需要补充介绍的内容．

  示例代码（下面的代码中用 `␣` 表示空格）：

  ```
  ??? note "标题"
  ␣␣␣␣这个文本框会被默认折叠．
  ␣␣␣␣
  ␣␣␣␣推荐将 **解题代码** 放在折叠文本框内．

  ```

[14] cstdio stdio.h namespace[51]

[15] CCF 关于恢复 NOIP 竞赛的公告 - 中国计算机学会[52]

```
???+note "[HDOJ 的「A + B Problem」](https://acm.hdu.edu.cn/showproblem.php?pid=1000)"
␣␣␣␣标题也可以使用 Markdown 的超链接．这里的超链接是 HDOJ 的「A + B Problem」.
␣␣␣␣
␣␣␣␣而且推荐以这种方式**标注原题链接**.
␣␣␣␣
␣␣␣␣注意双引号的位置.
```

效果：

> **标题**
>
> 这个文本框会被默认折叠.
>
> 推荐将 **解题代码** 放在折叠文本框内.

> **HDOJ 的「A + B Problem」**[54]
>
> 标题也可以使用 Markdown 的超链接．这里的超链接是 HDOJ 的「A + B Problem」.
>
> 而且推荐以这种方式 **标注原题链接**.
>
> 注意双引号的位置.

两种格式的区别是，带 `+` 的会默认保持展开，而不带 `+` 的会默认保持折叠.

折叠框的标题，即 `???+note` 中 `note` 后的内容应以 `"` 包裹起来．其中的内容支持 Markdown 语法. 详见 Admonition - Changing the title[55].（不具备折叠功能的为一般的 Admonitions，参考 Admonitions - Material for MkDocs[56]）

- MDFM-6：当需要添加不同语言的代码时，推荐使用 Content tabs，可以实现不同语言代码的切换. Content tabs 还有其他的用法，详见 Content tabs[57]．其使用方法和效果如下.

> **示例**
>
> 注意需要在文本前面添加 4 个空格（下面用 `␣` 表示）．其他的语法还是与 Markdown 语法一致.

```
=== "C"
␣␣␣␣```c
␣␣␣␣#include <stdio.h>
␣␣␣␣
␣␣␣␣int main(void) {
␣␣␣␣    printf("Hello world!\n");
␣␣␣␣    return 0;
␣␣␣␣}
␣␣␣␣```

=== "C++"
␣␣␣␣```cpp
␣␣␣␣#include <iostream>
␣␣␣␣
␣␣␣␣int main(void) {
␣␣␣␣    std::cout << "Hello world!" << std::endl;
␣␣␣␣    return 0;
␣␣␣␣}
```

````
    ````
````

- C

```c
#include <stdio.h>

int main(void) {
  printf("Hello world!\n");
  return 0;
}
```

- C++

```cpp
#include <iostream>

int main(void) {
  std::cout << "Hello world!" << std::endl;
  return 0;
}
```

如果对 mkdocs-material（我们使用的这个主题）还有什么问题，还可以查阅 MkDocs 使用说明[58]，其介绍了 mkdocs-material 主题的插件使用方式.

### 文本内容的格式要求

- CONT-1：所有的 OI Wiki 文本都应使用粗体标记.
- CONT-2：在页面的开头应有一段简短的文字（如「本页面将介绍……」），用于概述页面内容.

> ✓ **示例**
>
> 本页面将列出在 OI Wiki 编写过程时推荐使用的格式规范与编辑方针.

- CONT-3：涉及到「前置知识」的页面，请在开头添加一行 **前置知识：……**，放在页面概述前. 格式如下：

`前置知识：[站内页面1](url1)、[站内页面2](url2)和[站内页面3](url3)`

> ✓ **示例**
>
> 前置知识：时间复杂度→5.2.1@p544
>
> 本页面将介绍基础的计算理论的知识.

- CONT-4：请注意文档结构. 文档结构应当十分条理，层次清晰. 请不要让诸如「五级标题」这种事情再次发生了，一篇正常的文章是用不到如此复杂的结构层次的.
- CONT-5：请注意内容的表述. 作为一个百科网站，OI Wiki 使用的语言应该是书面的，客观的. 诸如「抖机灵」性质的，对读者理解帮助不大的内容，不应该出现在 OI Wiki 当中.
- CONT-6：请尽量为链接提供完整的标题、或者可被识别的提示，避免使用裸地址和「这」、「此」之类的模糊不清的描述. 每一个超链接都应尽量对其加以清楚明确的描述，方便读者明白该超链接将指向何处.

  建议使用源文章或者标签页的标题.

> × **不推荐的写法**
```

```
请参考[这个页面](https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/syncing-a-fork)

请参考 <https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/syncing-a-fork>
```

请参考 这个页面[59]

请参考 https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/syncing-a-fork[60]

✓ **推荐的写法**

```
请参考 GitHub 官方的帮助页面 [Syncing a fork - GitHub Docs](https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/syncing-a-fork)
```

请参考 GitHub 官方的帮助页面 Syncing a fork - GitHub Docs[61]

- CONT-7：受 Markdown 格式限制，`## 参考资料与注释` 二级标题必须放在文末.
- CONT-8：所有用作序号的数字建议使用中文. 示例：
  - 数列的第一项.
  - 输入文件的第一行.
- CONT-9：请尽量避免在标题中使用 MathJax 公式，无论是几级标题. 在标题中使用公式有可能会导致目录显示错误. [16]
- CONT-10：请注意代码的可读性.
  - CONT-10.1.1：代码应拥有清晰的逻辑，尽可能简洁易懂. 不要过度压行，不要引入过多无关代码. 尽量避免与算法思想无关的内容.
  - CONT-10.1.2：建议在参考代码中添加适当注释以方便读者理解.

  对 C/C++ 类语言：
  - CONT-10.2.1：尽量避免出现影响阅读的预编译指令和宏定义.
  - CONT-10.2.2：不要用 `0` 代替 `false/NULL/nullptr` 等，不要用 `1` 代替 `true` 等.
  - CONT-10.2.3：在声明 类型别名[62] 时，不推荐使用 `typedef`，推荐使用 `using`.
  - CONT-10.2.4：不推荐用宏定义定义常量，推荐直接使用 `constexpr/const` 等关键字定义常量.
  - CONT-10.2.5：不推荐对函数使用 `inline` 关键字，详见 编译优化→4.4.10@p456.
  - CONT-10.2.6：尽量避免类型萃取、偏特化等复杂的模板元编程技巧. 如确需使用，则需添加注释解释含义.

    × **不推荐的写法**

```
template <
    typename U, typename V,
    enable_if_t<is_integral<U>::value && is_integral<V>::value &&
                (is_unsigned<U>::value && is_unsigned<V>::value)>*
= nullptr>
typename common_type<U, V>::type gcd(U x, V y) {
```

[16] 我的公式为什么在目录里没有正常显示？好像双倍了→1.7@p59

```
  return y == 0 ? x : gcd(y, x % y);
}

template <typename U, typename V,
          enable_if_t<is_integral<U>::value &&
is_integral<V>::value &&
                      (is_signed<U>::value ||
is_signed<V>::value)>* = nullptr>
auto gcd(U x, V y) {
  return gcd((typename make_unsigned<U>::type)(abs(x)),
             (typename make_unsigned<V>::type)(abs(y)));
}

template <typename U, typename V, typename... Args,
          enable_if_t<is_integral<U>::value &&
is_integral<V>::value &&
                      0 < sizeof...(Args)>* = nullptr>
auto gcd(U x, V y, Args... args) {
  return gcd(gcd(x, y), args...);
}

template <class T,
          decltype(declval<T>().begin(), declval<T>().end(), 0)*
= nullptr>
auto gcd(const T& container) {
  auto ans = (typename make_unsigned<typename
T::value_type>::type)(
      *container.begin());
  for (auto it = container.begin(); it != container.end(); ++it)
    ans = gcd(ans, *it);
  return ans;
}

```

该代码给出了一个求 最大公约数→9.11.4@p1188 的复杂实现，其中：

- 第一个 `gcd` 接受两个无符号整数 `x`，`y`，返回 `x`，`y` 的最大公约数，返回值类型的范围保证能同时包含 `x` 和 `y`.
- 第二个 `gcd` 接受两个整数 `x`，`y`，其中 `x`，`y` 至少有一个是有符号整数，返回 `x`，`y` 的最大公约数.
- 第三个 `gcd` 接受超过两个整数，返回这些整数的最大公约数.
- 第四个 `gcd` 接受一个容器，返回容器中所有数的最大公约数.

对 OI Wiki 来说，我们只关注最大公约数这个算法的思想，这份代码涵盖了过多无关且复杂的技术细节，是需要避免的.

✓ **推荐的写法**

```
int gcd(int x, int y) { return y == 0 ? x : gcd(y, x % y); }

```

诸如「添加类型检查」、「处理负数输入」、「让函数支持多参数」等更多是工程上关注的话题，我们的重点始终应该是算法的思想.

### LaTeX 公式的格式要求

LaTeX 作为公式排版的首选，我们应当正确地使用它．因此对于 LaTeX 的使用我们有严格的要求．如果您想要快速上手，可以阅读本章节末给出的表格．

- MATH-1.1：您使用的符号不应与 数学符号表→1.6@p48 规定的符号冲突．
- MATH-1.2：使用 Roman 体表示数字、常量、算子和函数．使用 Italic 体表示变量、下标．LaTeX 已经预先定义好了一些常见的常量、函数、运算符等，我们可以直接调用，包括但不限于：

```
\log, \ln, \lg, \sin, \cos, \tan, \sec, \csc, \cot, \gcd, \min, \max, \exp,
\inf, \mod, \bmod, \pmod
```

  所以在输入常量、函数名、运算符等时，请先检查一下是否应该使用 Roman 体或其它字体．LaTeX 符号的书写可参考 KaTeX 的 Supported Functions 页面[63]（不是全部），也可以搜索求解．

  由于 LaTeX 书写 Roman 体小写希腊字母较为困难，故小写希腊字母常量、算子和函数可以使用 Italic 体，如 $\pi$ 以及 $\delta x$ 中的 $\delta$．

  如果遇到没有预先定义好的需要使用 Roman 体的 **函数名**，我们可以使用 `$\operatorname{something}$` 来产生，如我们可以使用 `$\operatorname{lcm}$` 产生正体的最小公倍数（函数）符号．同理，产生 Roman 体的 **常量** 应用 `$\mathrm{}$`；产生 Roman 体粗体符号应用 `$\mathbf{}$`；产生 Italic 体粗体符号应用 `$\boldsymbol{}$`（如向量 $\boldsymbol{a}$）．对于多字母的变量，应当使用 `$\textit{}$`．其他非数学内容，包括英文、特殊符号等，一律使用 `$\text{}$`．中文我们则建议不放在 LaTeX 公式中．
- MATH-1.3：如果表达式须折行（常见于较长的行间公式中），则应遵循如下换行规则：
  - MATH-1.3.1：将换行符放在 $=$，$+$，$-$，$\pm$，$\mp$ 之前，如果有必要，也可放在 $\times$，$\cdot$，$/$ 之前，如：

$$\begin{aligned} \mathrm{e}^x &= \sum_{n=0}^{\infty} \frac{x^n}{n!} \\ &= 1 + x + \frac{x^2}{2} \\ &\quad + \frac{x^3}{6} + \frac{x^4}{24} + \cdots \end{aligned}$$

  - MATH-1.3.2：同一运算符不应在换行符前后同时出现，
  - MATH-1.3.3：换行符尽量不要出现在括号内的表达式中．
- MATH-1.4：在行内使用分数的时候，请使用 `$\dfrac{}{}$`．比如 `$\dfrac{1}{2}$`，效果 $\dfrac{1}{2}$，而不是 `$\frac{1}{2}$`，效果 $\frac{1}{2}$．
- MATH-1.5：组合数请使用 `\dbinom{n}{m}`，效果 $\dbinom{n}{m}$，而不是 `{n \choose m}`（在 LaTeX 中这种写法已不推荐）；与上一条关于分数的约定相似，请不要使用 `\binom{n}{m}`，效果 $\binom{n}{m}$．
- MATH-1.6：尽可能避免在行内使用巨运算符（如 $\sum$，$\prod$，$\int$ 等）．
- MATH-1.7：在不会引起歧义的情况下，请用 `$\times$` 代替星号，叉乘请使用 `$\times$`，点乘请使用 `$\cdot$`．如 $a \times b$，$a \cdot b$，而不是 $a * b$．
- MATH-1.8：请用 `$\cdots$`（居于排版基线与顶线中间），`$\ldots$`（居于排版基线的位置），`$\vdots$`（竖着的省略号）代替 `$...$`．如 $a_1, a_2, \cdots a_n$，而不是 $a_1, a_2, ...a_n$．
- MATH-1.9：请注意，不要在非代码区域使用任何程序设计语言的表示方式，而是使用 LaTeX 公式．例如，使用 `$=$` 而不是 `$==$`（如 $a = b$，而不是 $a == b$）、使用 `` `a<<1` `` 或者 `$a\times 2$` 而不是 `$a<<1$`、使用 `$a\bmod b$` 代替 `$a\%b$`（如 $a \bmod b$，而不是 $a\%b$）等．
- MATH-1.10：公式中不要使用中括号连缀（即 C++ 高维数组的表示方式）而多使用下标．即 $a_{i,j,k}$ 而不是 $a[i][j][k]$．在公式中下标较复杂的情况下建议改用多元函数（$f(i,j,k)$）或内联代码格式．对于一元简单函数使用 `$f_i$`、`$f(i)$` 或 `$f[i]$` 均可．

- MATH-1.11：为了统一且书写方便，复杂度分析时大 $O$ 记号请直接使用 `$O()$` 而不是 `$\mathcal O()$`.
- MATH-1.12：在表示等价关系时，请使用 `$\iff$`，效果 $\iff$，而不是 `$\Leftrightarrow$`，效果 $\Leftrightarrow$.
- MATH-1.13：分段函数环境 `cases` **只能有两列**（即一个 `&` 分隔符）.
- MATH-1.14：请不要滥用 LaTeX 公式. 这不仅会造成页面加载缓慢（因为 MathJax 的效率低是出了名的），同时也会导致页面的排版混乱. 我们通常使用 LaTeX 公式字体表示变量名称. 我们的建议是，如非必要，尽量减少公式与普通正文字体的 **大量** 混合使用，如非必要，尽量不要使用公式，如：

  ```
  我们将要学习 $Network-flow$ 中的 $SPFA$ 最小费用流，需要使用 $Edmonds-Karp$ 算法进行增广.
  ```

  就是一个典型的 **滥用公式字体** 的例子.（在页面中使用斜体请用 `*文本*` 表示.）
- MATH-1.15：请正确使用对应的 LaTeX 符号，尤其是公式中的希腊字母等特殊符号. 如欧拉函数请使用 `$\varphi$`，圆的直径请使用 `$\Phi$`，黄金分割请使用 `$\phi$`. 这些符号虽然同样表示希腊字母 Phi，但是在不同的环境下有不同的含义. 切记 **不要使用输入法的插入特殊符号** 来插入这种符号.

  另外，由于 LaTeX 历史原因，空集的符号应为 `$\varnothing$` 而不是 `$\emptyset$`；其他的符号应参照 数学符号表→1.6@p48 书写.

我们可以使用一个表格来总结一下上述内容. 注意本表格没有举出所有符号的用法，只给出常见的错误. 类似的情况类比即可.

| 不符合规定的用法 | 渲染效果 | 符合规定的用法 | 渲染效果 |
| --- | --- | --- | --- |
| `$log, ln, lg$` | $log, ln, lg$ | `$\log$, $\ln$, $\lg$` | $\log$, $\ln$, $\lg$ |
| `$sin, cos, tan$` | $sin, cos, tan$ | `$\sin$, $\cos$, $\tan$` | $\sin$, $\cos$, $\tan$ |
| `$gcd, lcm$` | $gcd, lcm$ | `$\gcd$, $\operatorname{lcm}$` | $\gcd$, $\operatorname{lcm}$ |
| `$e$, $\text{e}$, e`（自然对数的底） | $e$, $\text{e}$, e | `$\mathrm{e}$` | $\mathrm{e}$ |
| `$i$, $\text{i}$, i`（虚数单位） | $i$, $\text{i}$, i | `$\mathrm{i}$` | $\mathrm{i}$ |
| `$ 小于 a 的质数 $` | 小于$a$的质数 | `小于 $a$ 的质数` | 小于 $a$ 的质数 |
| `$...$` | $...$ | `$\cdots$, $\ldots$, $\vdots$, $\ddots$` | $\cdots$, $\ldots$, $\vdots$, $\ddots$ |
| `$a*b$`（两个数相乘） | $a*b$ | `$a\times b$, $a\cdot b$` | $a \times b$, $a \cdot b$ |
| `$SPFA$`（英文名称） | $SPFA$ | `SPFA` | SPFA |
| `$a==b$` | $a == b$ | `$a=b$` | $a = b$ |
| `$f[i][j][k]$` | $f[i][j][k]$ | `$f_{i,j,k}$, $f(i,j,k)$` | $f_{i,j,k}$, $f(i,j,k)$ |
| `$R,N^*$`（集合） | $R, N^*$ | `$\mathbf{R}$, $\mathbf{N}^*$` | $\mathbf{R}$, $\mathbf{N}^*$ |
| `$\emptyset$` | $\emptyset$ | `$\varnothing$` | $\varnothing$ |
| `$size$` | $size$ | `$\textit{size}$` | $\textit{size}$ |

### 对数学公式的附加格式要求

请注意，尽管上述输入公式的语法和真正的 LaTeX 排版系统非常相似，但 **MathJax 和 LaTeX 是两个完全没有关系的东西**，MathJax 仅仅使用了一部分与 LaTeX 非常相似的语法而已. 实际上，二者之间有不少细节差别，而这些差别经常导致写出来的公式在二者之间不通用.

由于 OI Wiki 使用 LaTeX 排版引擎开发了 PDF 导出工具，因此有必要强调公式在 MathJax 和 LaTeX 之间的兼容性. **请各位在 Wiki 中书写数学公式时注意以下几点.**

这些规则已经向 MathJax 做了尽可能多的妥协. 导出工具兼容了一部分原本仅能在 MathJax 中正常输出的写法.

- MATH-2.1：请使用 `\begin{aligned} ... \end{aligned}` 表示多行对齐的公式；

- MATH-2.2：如果这些多行对齐的公式需要 **编号**，请用 `align` 或 `equation` 环境；
- MATH-2.3：不要使用 `split`、`eqnarray` 环境；
- MATH-2.4：不要使用 `\lt`,`\gt` 来表示大于号和小于号，请直接使用 `<`，`>`；
- MATH-2.5：不要直接用 `\\` 换行（需要换行的公式，请套在 `aligned` 或其他多行环境下）；
- MATH-2.6：若要输出 LaTeX 符号 LATEX，请用 `$\rm{\LaTeX}$`，而不是 `mathrm`；（`\LaTeX` 在 TeX 排版系统中是一个不能用于数学模式下的命令，而 `\mathrm` 又不能在普通模式下使用；另外，`\text` 命令虽然在 TeX 上正常输出，但是在 MathJax 中 `\text` 命令的参数会被原样输出，而不是按命令转义）；
- MATH-2.7：数学公式中的中文文字 **必须置于 `\text{}` 命令之中**，而变量、数字、运算符、函数名称则必须置于 `\text{}` 命令之外. **请不要在 `\text{}` 命令中嵌套数学公式**；
- MATH-2.8：使用 `array` 环境时请注意 **实际列数与对齐符号的数量保持一致**. 例如下面的公式中，数据实际有 3 列（`&` 是列分隔符），因此需要 3 个对齐符号（`l`/`r`/`c` 分别表示左、右、居中对齐）.

```
$$
\begin{array}{lll}
F_1=\{\frac{0}{1},&&\frac{1}{1}\}\\
F_2=\{\frac{0}{1},&\frac{1}{2},&\frac{1}{1}\}\\
\end{array}
$$
```

### 伪代码格式

伪代码具体格式没有严格要求，请参考算法导论或学术论文. 注意不要写成 Python.

PCOD-1：Wiki 内使用 LaTeX 书写伪代码，整体处于 array 环境中，缩进使用 `$\qquad$`，文字描述使用 `$\text$`，关键字使用 `$\textbf$`，多字母变量使用 `$\textit$`，赋值使用 `$\gets$`.

参考示例：

**Input**. The edges of the graph $e$, where each element in $e$ is $(u, v, w)$ denoting that there is an edge between $u$ and $v$ weighted $w$.
**Output**. The edges of the MST of the input graph.
**Method**.
1 $result \leftarrow \emptyset$
2 sort $e$ into nondecreasing order by weight $w$
3 **for** each $(u, v, w)$ in the sorted $e$
4 **if** $u$ and $v$ are not connected in the union-find set
5 connect $u$ and $v$ in the union-find set
6 $result \leftarrow result \bigcup \{(u, v, w)\}$
7 **return** $result$

```
$$
\begin{array}{l}
\textbf{Input.} \text{The edges of the graph } e \text{, where each element in } e \text{ is } (u, v, w) \\
\text{denoting that there is an edge between } u \text{ and } v \text{ weighted } w . \\
\textbf{Output.} \text{The edges of the MST of the input graph}. \\
\textbf{Method.} \\
\begin{array}{ll}
1 & \textit{result} \gets \varnothing \\
2 & \text{sort } e \text{ into nondecreasing order by weight } w \\
3 & \textbf{for} \text{ each } (u, v, w) \text{ in the sorted } e \\
4 & \qquad \textbf{if} \ u \text{ and } v \text{ are not connected in the union-find set} \\
5 & \qquad\quad \text{connect } u \text{ and } v \text{ in the union-find set} \\
6 & \qquad\quad result \leftarrow result \bigcup \{(u, v, w)\} \\
7 & \textbf{return} \ result
\end{array}
\end{array}
$$
```

```
5 &  \qquad\qquad \text{connect } u \text{ and } v \text{ in the union-find set}
\\
6 &  \qquad\qquad \textit{result} \gets \textit{result}\;\bigcup\ \{(u, v, w)\} \\
7 &  \textbf{return } \textit{result}
\end{array}
\end{array}
$$
```

### 代码块的格式要求

代码块目前分为两种：片段和例题.

关于片段代码：

- CODE-1.1：若代码片段足够短且没有必要测试，可以直接在 Markdown 文档中修改.
- CODE-1.2：由于 Markdown 文档中内嵌的代码难以实现自动化测试，所以推荐使用例题代码的格式插入片段代码. 可以选择 多文件编译方案[64] 或 Snippet Sections[65] 语法：

  多文件编译方案示例：冒泡排序[66]. 正文引用 bubble-sort_1.cpp[67]，测试代码放在 bubble-sort_1.aux1.cpp[68] 中.

  Snippet Sections 示例：前缀和[69]. 正文中不需要引用 prefix-sum_1.cpp[70] 中的测试部分，所以选择插入主要的代码片段.

  **注意**：不要使用 Snippet Lines[71] 语法.

  为了提高代码复用率，您也可以将代码拆分成头文件，测试时在不同的测试代码里引用. 如果正文中需要出现完整的测试代码作为例题的参考实现，那么正文中应该另外用 Snippet Sections 语法拼接成单文件代码，以便读者阅读. 示例：红黑树[72].

关于例题代码：

- CODE-2.1：例题代码的表示形式为 `--8<-- "path"`，代码均存储在 `path` 中. 路径通常为 `docs/主题/code/内容/内容_编号.cpp`.
- CODE-2.2：修改例题代码时，请保证你的代码是正确的. 例题代码均拥有一组测试数据，存储在 `/docs/主题/examples/内容/内容_编号.in/ans` 中.

如果你需要添加例题：

- 请在 `docs/主题/code/内容` 中添加你的例题代码，并编号. 通常，该 `内容` 文件夹中已经有了一个或者多个代码. 例子：如果需要修改 `dag.md` 的代码，那么路径为 `docs/dp/code/dag`，其中 `dp` 为主题，而 `dag` 为内容.
- 如果需要在所有例题的最后添加一个例题代码，请顺延目前的编号. 比如已经存在了 `code/prefix-sum/prefix-sum_3.cpp`, 如果需要在最后一个例题后继续添加一个例题, 请将你的代码命名为 `prefix-sum_4.cpp` 并添加到 `docs/basic/code/prefix-sum` 中.
- 如果需要在文章中间添加一个例题代码，请插入并改变原先的编号. 比如已经存在了 `prefix-sum_2.cpp` 和 `prefix-sum_3.cpp`，如果你需要在第二个例题和第三个例题中间再添加一个例题，请将你的代码命名为 `prefix-sum_3.cpp` 并将原先的 `prefix-sum_3.cpp` 改名为 `prefix-sum_4.cpp` 同时 **在 Markdown 文档和测试数据存放的文件夹中同步修改编号**.
- **别忘记，你还要对你的代码添加一组测试数据，以保证这个代码是可以成功运行的.** 你需要在 `docs/主题/examples/内容` 文件夹中添加一组测试数据，将输入数据存储为 `内容_编号.in`，将标准答案存储为 `内容_编号.ans`.
- 最后，可以将代码添加到文档中了. 请直接在文档中用添加代码块的格式，并将代码块内部直接写成 `--8<-- "你的代码路径"` 的格式就可以了.

OI Wiki 会对例题代码进行全平台测试，为保证您的代码能够顺利通过测试，请遵守如下规则：

- CODE-3.1：您的代码需要同时支持在 C++14、C++17、C++20 标准下编译和运行.

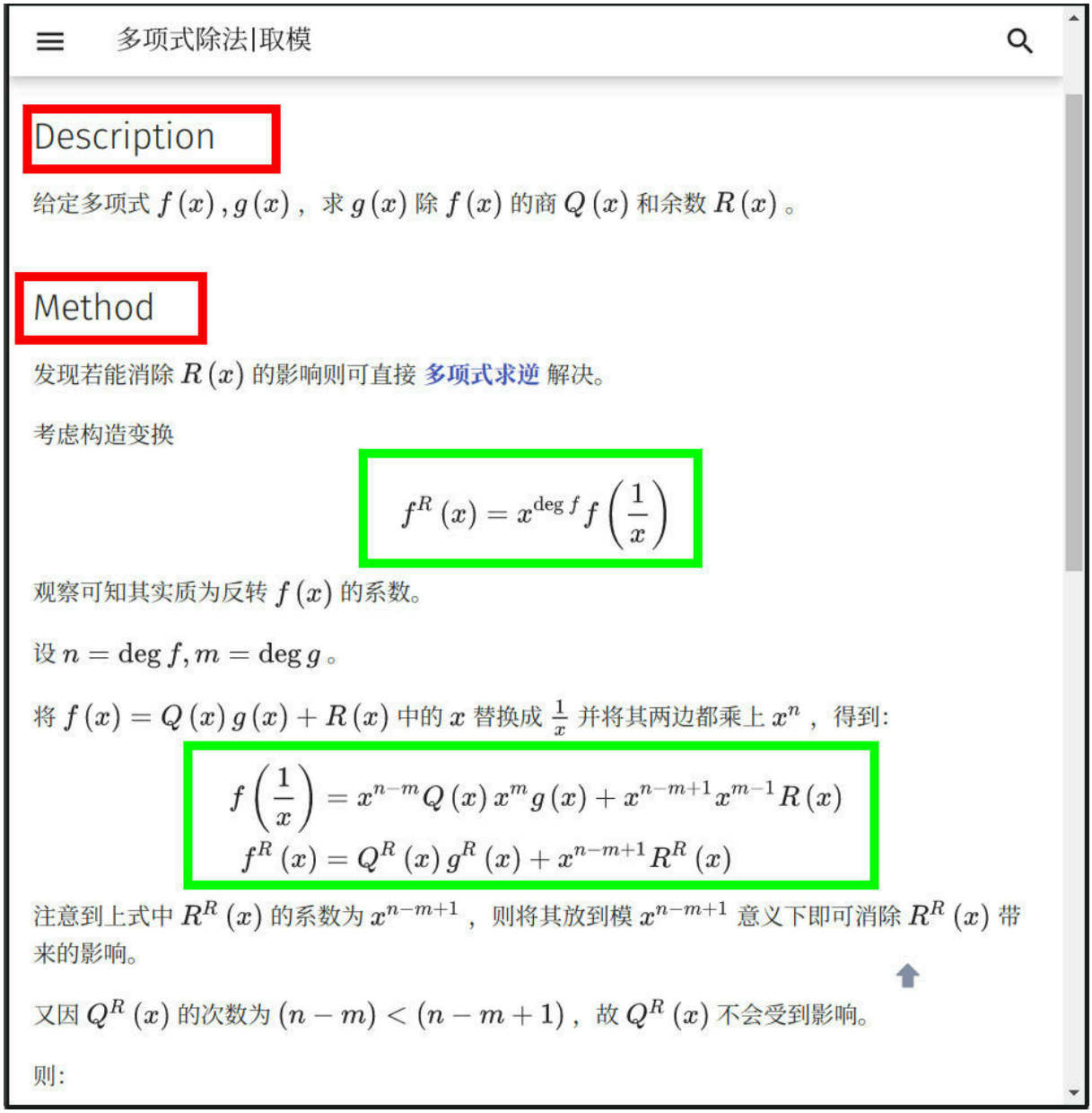
- CODE-3.2：不要使用 `<bits/stdc++.h>`、`<bits/extc++.h>` 等非标准头文件.
- CODE-3.3：标准答案文件不要有多余空格.
- CODE-3.4：不要使用 代用记号[73].
- CODE-3.5：使用 聚合初始化[74] 时，`object{args}` 不可写成 `(object){args}`.
- CODE-3.6：使用 运算符重载[75] 时注意格式，如重载比较运算符时，若使用成员函数写法，则不可省略 `const` 限定符.
- CODE-3.7：不要使用类似 `#define int long long` 的宏定义.
- CODE-3.8：若您需要使用 C 风格的 有格式输入/输出[76]，请特别留意格式指示符的写法：如 `size_t` 对应 `%zu`，`ptrdiff_t` 对应 `%td`. 例如输出某 STL 容器的大小时，代码应类似 `printf("%zu", container.size());`.
- CODE-3.9：由于当前测试环境 libstdc++ 的 `<chrono>` 库有 BUG[77]，所以请避免使用 `<chrono>` 库.
- CODE-3.10：由于 `long` 与 `unsigned long` 在某些测试环境下为 32 位，而在另一些测试环境下为 64 位，为确保各平台代码行为一致，故不推荐使用这两种类型. 推荐使用 定宽整数类型→4.2.3@p328.
- CODE-3.11：不建议使用 `__gcd`、`__int128`、`__builtin_` 系列函数等非标准内容. 如果您需要使用，则需确保您的代码能通过全平台测试，如 此代码[78] 提供了 libstdc++ 中 std::bitset→4.3.4@p389 特有成员函数 `_Find_first()` 的全平台实现.

此外，为了提高代码的可读性，建议遵守 CONT-10→1@p15.

## 图解

可能上述要求把握起来有些困难，接下来我们给出一些图片来具体分析哪种格式应该使用，哪种不该使用：

**例 1**



多项式除法|取模

## Description

给定多项式 $f(x), g(x)$，求 $g(x)$ 除 $f(x)$ 的商 $Q(x)$ 和余数 $R(x)$。

## Method

发现若能消除 $R(x)$ 的影响则可直接 **多项式求逆** 解决。

考虑构造变换

$$f^R(x) = x^{\deg f} f\left(\frac{1}{x}\right)$$

观察可知其实质为反转 $f(x)$ 的系数。

设 $n = \deg f, m = \deg g$。

将 $f(x) = Q(x)\,g(x) + R(x)$ 中的 $x$ 替换成 $\frac{1}{x}$ 并将其两边都乘上 $x^n$，得到：

$$f\left(\frac{1}{x}\right) = x^{n-m} Q(x)\, x^m g(x) + x^{n-m+1} x^{m-1} R(x)$$
$$f^R(x) = Q^R(x)\, g^R(x) + x^{n-m+1} R^R(x)$$

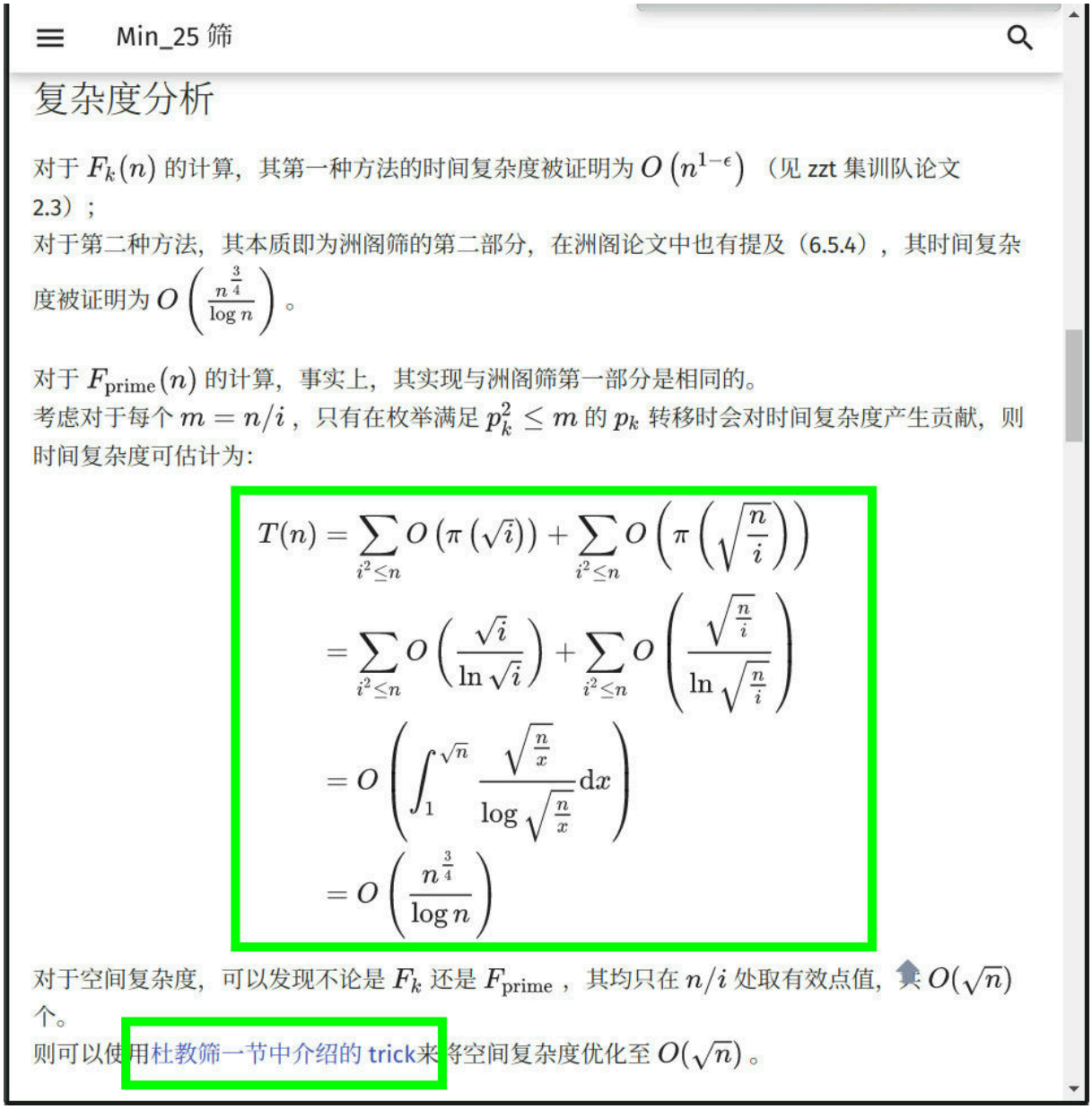
注意到上式中 $R^R(x)$ 的系数为 $x^{n-m+1}$，则将其放到模 $x^{n-m+1}$ 意义下即可消除 $R^R(x)$ 带来的影响。

又因 $Q^R(x)$ 的次数为 $(n-m) < (n-m+1)$，故 $Q^R(x)$ 不会受到影响。

则：

将复杂的 LaTeX 公式使用行间格式，可以使得页面错落有致．但 OI Wiki 作为一个以中文为主体的站点，我们希望大部分纲领性的信息（如标题）尽量使用中文（除英文专有名词）．

**例 2**



Min_25 筛

## 复杂度分析

对于 $F_k(n)$ 的计算，其第一种方法的时间复杂度被证明为 $O\left(n^{1-\epsilon}\right)$ （见 zzt 集训队论文 2.3）；
对于第二种方法，其本质即为洲阁筛的第二部分，在洲阁论文中也有提及（6.5.4），其时间复杂度被证明为 $O\left(\frac{n^{\frac{3}{4}}}{\log n}\right)$。

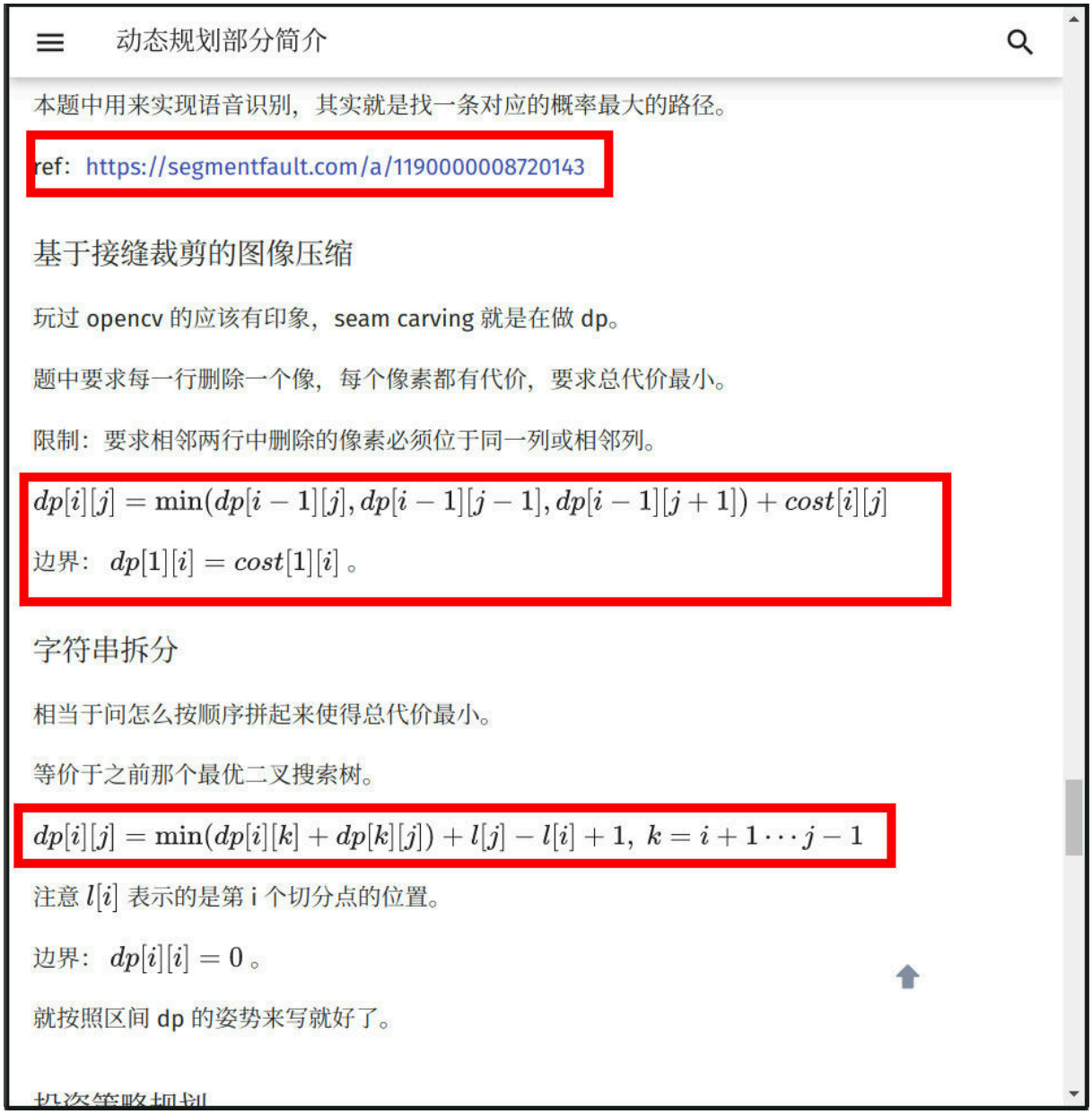
对于 $F_{\text{prime}}(n)$ 的计算，事实上，其实现与洲阁筛第一部分是相同的。
考虑对于每个 $m=n/i$，只有在枚举满足 $p_k^2 \leq m$ 的 $p_k$ 转移时会对时间复杂度产生贡献，则时间复杂度可估计为：

$$
\begin{aligned}
T(n) &= \sum_{i^2 \leq n} O\left(\pi\left(\sqrt{i}\right)\right) + \sum_{i^2 \leq n} O\left(\pi\left(\sqrt{\frac{n}{i}}\right)\right) \\
&= \sum_{i^2 \leq n} O\left(\frac{\sqrt{i}}{\ln \sqrt{i}}\right) + \sum_{i^2 \leq n} O\left(\frac{\sqrt{\frac{n}{i}}}{\ln \sqrt{\frac{n}{i}}}\right) \\
&= O\left(\int_1^{\sqrt{n}} \frac{\sqrt{\frac{n}{x}}}{\log \sqrt{\frac{n}{x}}} \mathrm{d}x\right) \\
&= O\left(\frac{n^{\frac{3}{4}}}{\log n}\right)
\end{aligned}
$$

对于空间复杂度，可以发现不论是 $F_k$ 还是 $F_{\text{prime}}$，其均只在 $n/i$ 处取有效点值，共 $O(\sqrt{n})$ 个。
则可以使用杜教筛一节中介绍的 trick 来将空间复杂度优化至 $O(\sqrt{n})$。

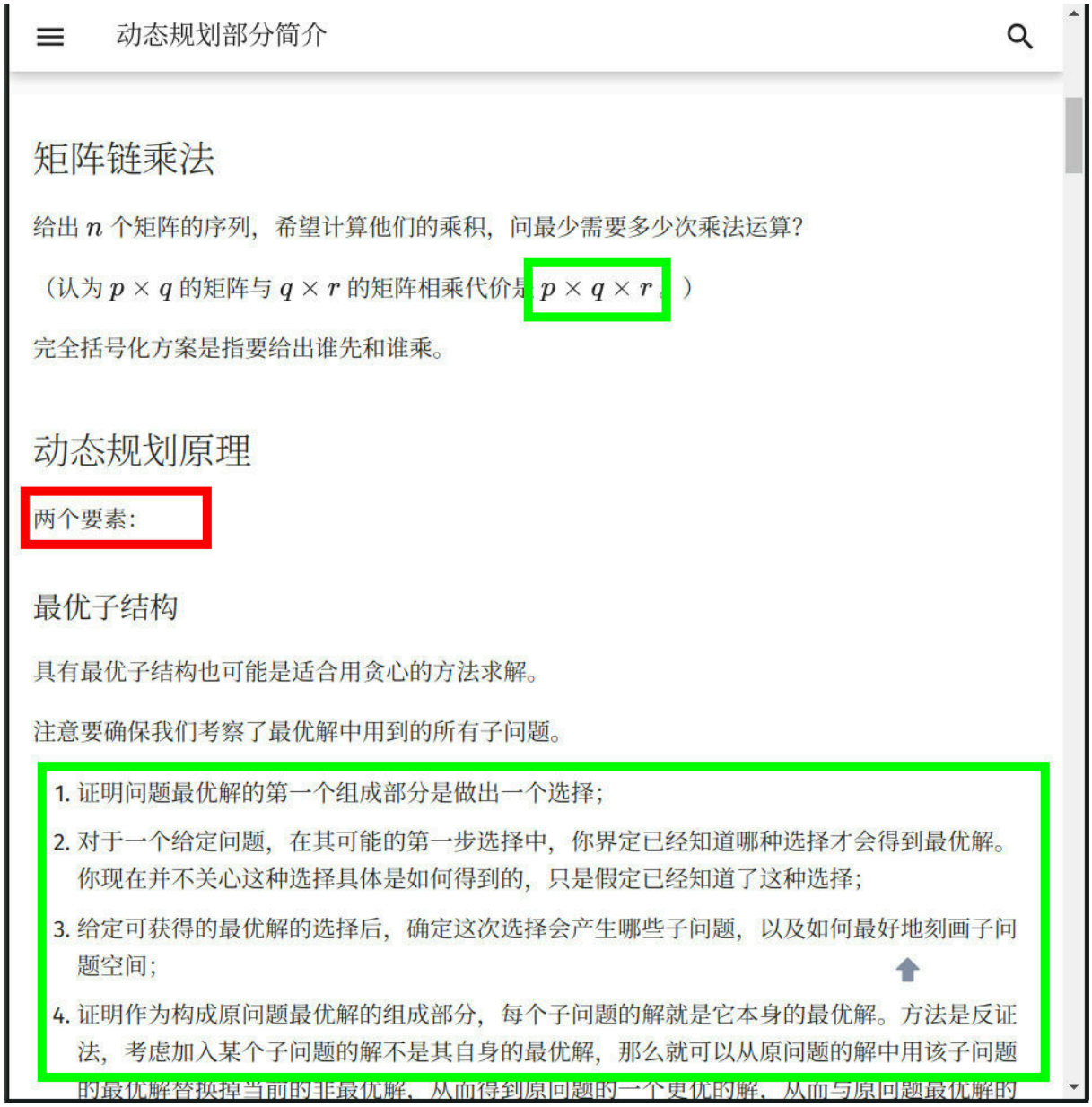
较复杂度的 LaTeX 公式请注意等号的对齐，同时可以适当引用 Wiki 的页面 **链接** 来完善内容.

**例 3**



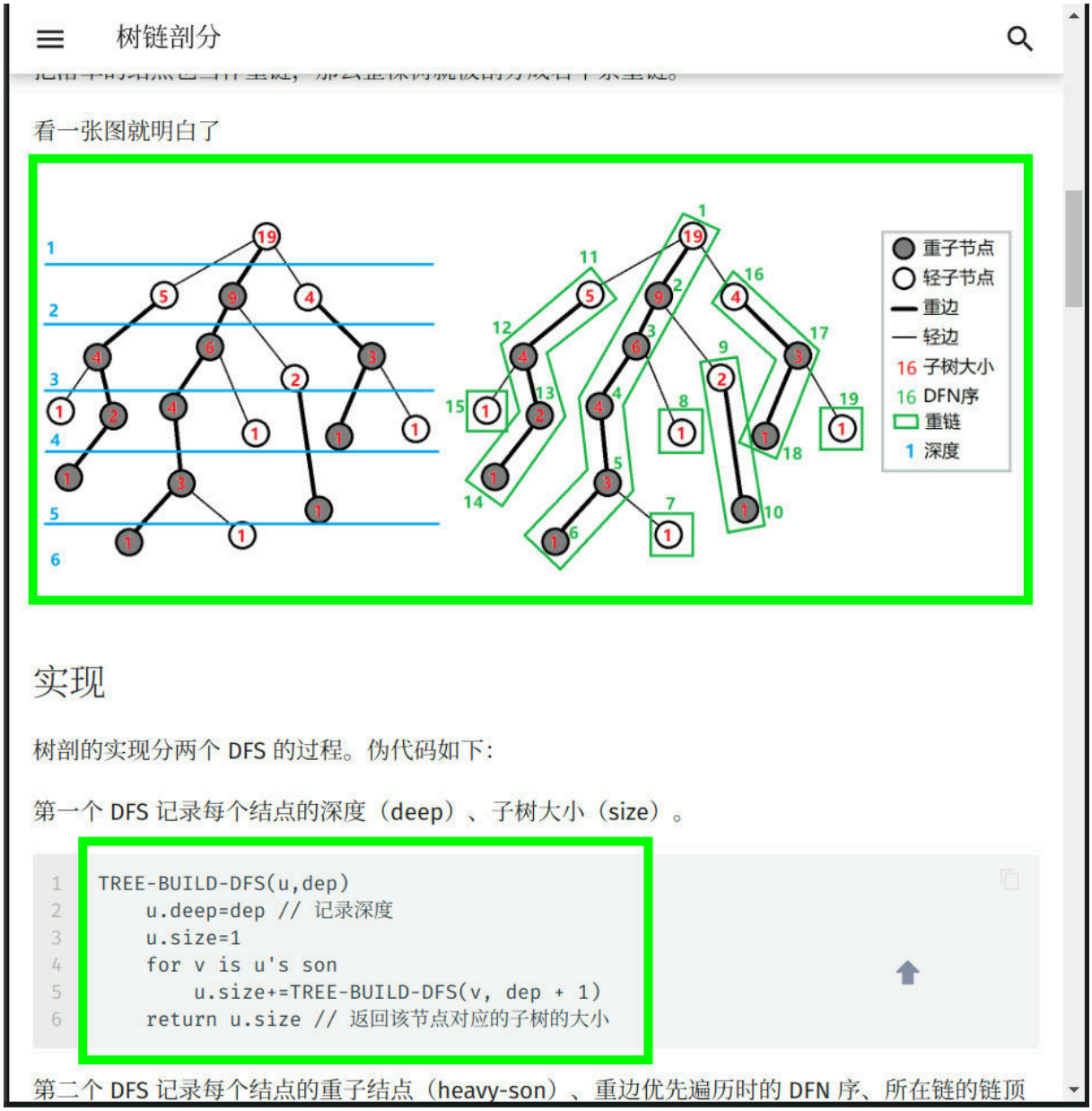

一般情况下，我们建议将引用的资料列在文末的 `## 参考资料与注释` 一节，并在原句后面加上脚注，而不是直接给出链接. 同时一定要避免使用 LaTeX 公式表达代码，上图中两个中括号就是不规范的写法. 我们建议使用 `dp(i,j)` 或者 `dp_{i,j}`.

**例 4**



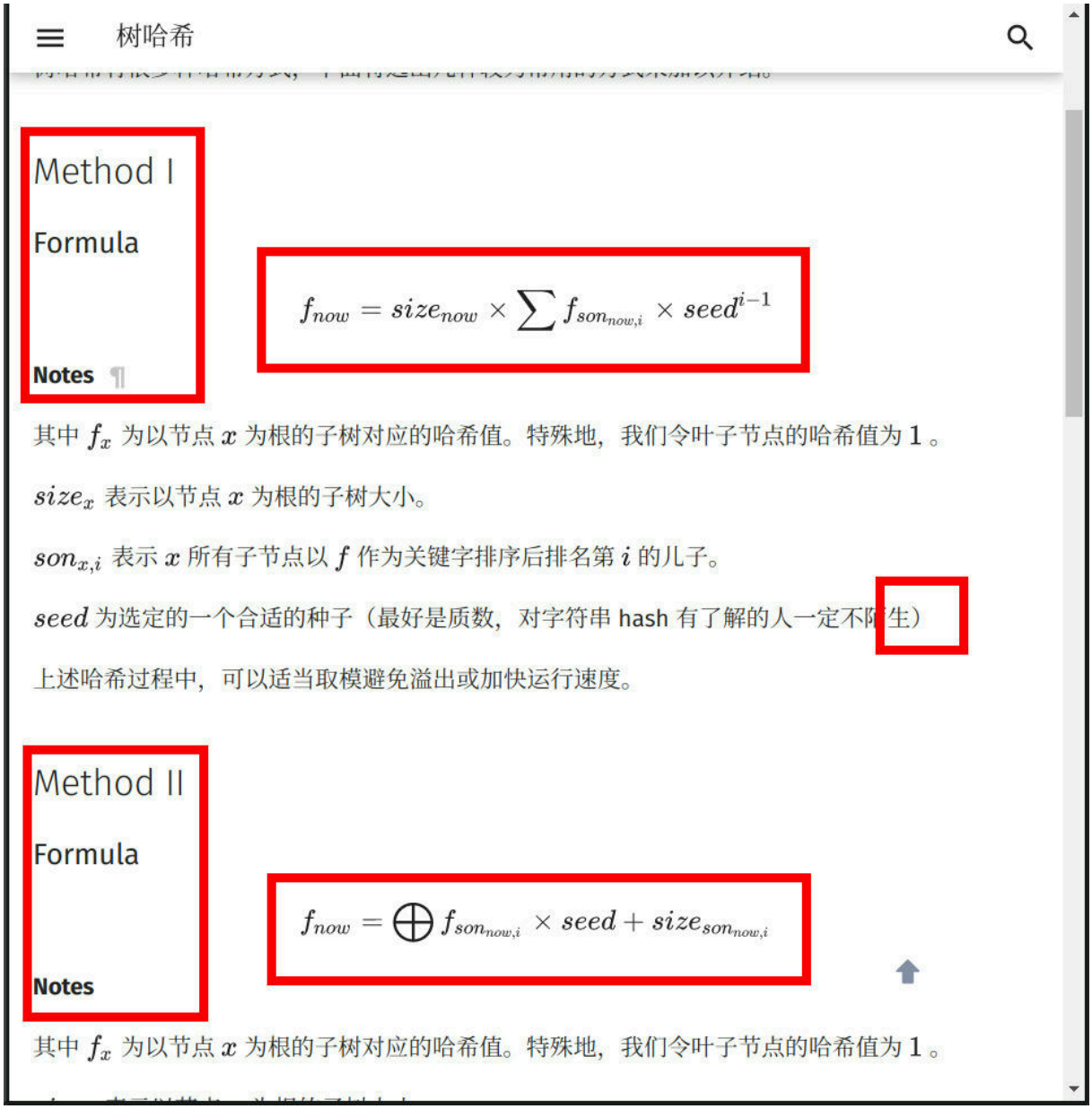

注意我们描述 **乘法** 的时候一般使用 `\times` 或者 `\cdot`，特殊情况（如卷积）下会使用 `*`（也可以写成 `\ast`）. 标题是简洁的词组，但我们不希望正文部分由词组拼凑而成. 上图中「两个要素」，建议更改为「动态规划的原理具有以下两个要素」，上下文保持连贯. 可取的地方是，适当使用 **有序** 列表可以更有条理地表述内容. 再次提醒，在使用列表的时候，每一项如果是一句话，需要在末位添加 **标点符号**. 有序列表通常添加分号，在最后一项末位添加句号；无序列表统一添加句号.

**例 5**




适当引用 **图片** 可以增强文章易读性. 使用 **伪代码** 的方式表达算法过程可以方便又简洁地描述算法过程，相比于直接贴模板代码更加好懂.

**例 6**




同样的问题，标题使用英文．并且在使用完括号后没有句号．另外，上图中的行间公式虽然没有使用括号，但是由于下标嵌套过多，使得最底层的下标字体很小，整个公式也并不美观．建议将 `son_{now,i}` 更换为 `son(now,i)`，或者把 `f_{now}` 替换为 `f(now)`．我们希望尽量控制上下标嵌套在两层以内（需要多次嵌套上标时建议使用 Knuth 箭头，如用 $2 \uparrow (2 \uparrow (2 \uparrow (2 \uparrow \cdots)))$ 代替 $2^{2^{2^{\cdots}}}$ ，《上帝造题的七分钟》）．

**例 7**




使用 MkDocs 扩展语法，让例题题面与算法描述区分开．将代码折叠，可以让文章更紧凑．（毕竟看 Wiki 的大多数是了解思路，除了模板代码需要阅读外，习题的代码大多可以折叠．）在描述函数操作时，使用行内代码和 LaTeX 公式都是不错的选择．

**例 8**




在文末罗列出参考文献，可以使页面的内容更严谨，真实可信.

## 外部链接

- 标点符号用法（GB/T 15834—2011）[79]
- 维基百科：格式手册/标点符号[80]
- 中文文案排版指北（简体中文版）[81]
- 中文文案风格指南 - PDFE GUIDELINE[82]
- 一份（不太）简短的 LATEX2ε 介绍或 106 分钟了解 LATEX2ε[83]
- 中文出版物夹用英文的编辑规范[84]

## 参考资料与注释

[1] #6793 [2] #6746 [3] #6412

[4] #6409
[5] #5912
[6] #5924
[7] #5482
[8] Content tabs
[9] #5152
[10] #5023
[11] #4792
[12] #4587
[13] #4726
[14] #4500
[15] #4057
[16] #4043
[17] #3517
[18] #3499
[19] #3447
[20] #3422
[21] #3255
[22] #3197
[23] #3061
[24] #2950
[25] 互联网档案馆
[26] #2918
[27] #2744
[28] #2215
[29] #2502
[30] #2466
[31] #2442
[32] #2422
[33] Collapsible Blocks
[34] `<details>` 元素
[35] #2412
[36] #2350
[37] Collapsible Blocks
[38] `<details>` 元素
[39] Content tabs
[40] SVG|MDN
[41] SVG 1.1
[42] APNG
[43] ScreenToGif
[44] Peek
[45] OI-wiki/OI-wiki#3422
[46] Save Page in Internet Archive
[47] remark-lint
[48] .remarkrc
[49] 中文出版物夹用英文的编辑规范
[50] 连接号 - 维基百科
[51] cstdio stdio.h namespace
[52] CCF 关于恢复 NOIP 竞赛的公告 - 中国计算机学会
[53] Collapsible Blocks
[54] HDOJ 的「A + B Problem」
[55] Admonition - Changing the title
[56] Admonitions - Material for MkDocs
[57] Content tabs
[58] MkDocs 使用说明
[59] 这个页面
[60] https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/syncing-a-fork
[61] Syncing a fork - GitHub Docs
[62] 类型别名
[63] KaTeX 的 Supported Functions 页面

[64] 多文件编译方案 

[65] Snippet Sections 

[66] 冒泡排序 

[67] bubble-sort_1.cpp 

[68] bubble-sort_1.aux1.cpp 

[69] 前缀和 

[70] prefix-sum_1.cpp 

[71] Snippet Lines 

[72] 红黑树 

[73] 代用记号 

[74] 聚合初始化 

[75] 运算符重载 

[76] 有格式输入/输出 

[77] BUG 

[78] 此代码 

[79] 标点符号用法（GB/T 15834—2011） 

[80] 维基百科：格式手册/标点符号 

[81] 中文文案排版指北（简体中文版） 

[82] 中文文案风格指南 - PDFE GUIDELINE 

[83] 一份（不太）简短的 LATEX2ε 介绍或 106 分钟了解 LATEX2ε 

[84] 中文出版物夹用英文的编辑规范 

# 1.6 数学符号表

本文规定了 OI Wiki 中数学符号的推荐写法，并给出了一些应用范例.

本文参考了 GB/T 3102.11-1993[1]、ISO 80000-2:2019[2] 和《具体数学》的符号表修订，故基本与国内通行教材的符号体系和 OI 场景的惯用符号体系兼容.

符号的 LaTeX 写法请参考 本文章的源代码[3]

## 数理逻辑

| 编号 | 符号，表达式 | 意义，等同表述 | 备注与示例 |
| --- | --- | --- | --- |
| n1.1 | $p \land q$ | $p$ 和 $q$ 的合取 | $p$ 与 $q$. |
| n1.2 | $p \lor q$ | $p$ 和 $q$ 的析取 | $p$ 或 $q$此处的 "或" 是包含的，即若 $p$，$q$ 中有一个为真陈述，则 $p \lor q$ 为真. |
| n1.3 | $\neg p$ | $p$ 的否定 | 非 $p$. |
| n1.4 | $p \Longrightarrow q$ | $p$ 蕴含 $q$若 $p$ 为真，则 $q$ 为真 | $q \Longleftarrow p$ 和 $p \Longrightarrow q$ 同义. |
| n1.5 | $p \Longleftrightarrow q$ | $p$ 等价于 $q$ | $(p \Longrightarrow q) \land (q \Longrightarrow p)$ 和 $p \Longleftrightarrow q$ 同义. |
| n1.6 | $(\forall\ x \in A)\ \ p(x)$ | 对 $A$ 中所有的 $x$, 命题 $p(x)$ 均为真 | 如果从上下文中可以得知考虑的是哪个集合 $A$, 可以使用记号 $(\forall\ x)\ \ p(x)$.$\forall$ 称为全称量词. $x \in A$ 的含义见 n2.1→1@p15. |
| n1.7 | $(\exists\ x \in A)\ \ p(x)$ | 存在一个属于 $A$ 的 $x$ 使得 $p(x)$ 为真 | 如果从上下文中可以得知考虑的是哪个集合 $A$, 可以使用记号 $(\exists\ x)\ \ p(x)$.$\exists$ 称为存在量词. $x \in A$ 的含义见 n2.1→1@p15.$(\exists!\ x)\ \ p(x)$（唯一量词）用来表示恰有一个 $x$ 使得 $p(x)$ 为真. $\exists!$ 也可以写作 $\exists^1$. |

## 集合论

| 编号 | 符号，表达式 | 意义，等同表述 | 备注与示例 |
| --- | --- | --- | --- |
| n2.1 | $x \in A$ | $x$ 属于 $A$，$x$ 是集合 $A$ 中的元素 | $A \ni x$ 和 $x \in A$ 同义. |
| n2.2 | $y \notin A$ | $y$ 不属于 $A$，$y$ 不是集合 $A$ 中的元素 | |

| | | | |
|---|---|---|---|
| n2.3 | $\{x_1, x_2, ..., x_n\}$ | 含元素 $x_1, x_2, ..., x_n$ 的集合 | 也可写作 $\{x_i \mid i \in I\}$, 其中 $I$ 表示指标集. |
| n2.4 | $\{x \in A \mid p(x)\}$ | $A$ 中使命题 $p(x)$ 为真的所有元素组成的集合 | 例如 $\{x \in \mathbf{R} \mid x \geq 5\}$如果从上下文中可以得知考虑的是哪个集合 $A$，可以使用符号 $\{x \mid p(x)\}$（如在只考虑实数集时可使用 $\{x \mid x \geq 5\}$）|也可以使用冒号替代, 如 $\{x \in A : p(x)\}$. |
| n2.5 | $\operatorname{card} A$\| $A$ \|$\#A$ | $A$ 中的元素个数，$A$ 的基数 | |
| n2.6 | $\emptyset$ | 空集 | 不应使用 $0$. |
| n2.7 | $B \subseteq A$ | $B$ 包含于 $A$ 中，$B$ 是 $A$ 的子集 | $B$ 的每个元素都属于 $A$.$\subset$ 也可用于该含义, 但请参阅 n2.8→1@p15 的说明. $A \supseteq B$ 和 $B \subseteq A$ 同义. |
| n2.8 | $B \subset A$ | $B$ 真包含于 $A$ 中，$B$ 是 $A$ 的真子集 | $B$ 的每个元素都属于 $A$, 且 $A$ 中至少有一个元素不属于 $B$.若 $\subset$ 的含义取 n2.7→1@p15, 则 n2.8→1@p15 对应的符号应使用 $\subsetneq$.$A \supset B$ 与 $B \subset A$ 同义. |
| n2.9 | $A \cup B$ | $A$ 和 $B$ 的并集 | $A \cup B := \{x \mid x \in A \vee x \in B\}$:= 的定义参见 n4.3→1@p15 |
| n2.10 | $A \cap B$ | $A$ 和 $B$ 的交集 | $A \cap B := \{x \mid x \in A \wedge x \in B\}$:= 的定义参见 n4.3→1@p15 |
| n2.11 | $\bigcup_{i=1}^{n} A_i$ | 集合 $A_1, A_2, ..., A_n$ 的并集 | $\bigcup_{i=1}^{n} A_i = A_1 \cup A_2 \cup ... \cup A_n$也可使用 $\bigcup_{i=1}^{n}$, $\bigcup_{i \in I}$, $\bigcup_{i \in I}$, 其中 $I$ 表示指标集；进一步，令 $P(i)$ 为某个与 $i$ 相关的命题，可使用 $\bigcup_{P(i)} A_i$ 表示所有使 $P(i)$ 为真的 $i$ 对应的 $A_i$ 之并集 |
| n2.12 | $\bigcap_{i=1}^{n} A_i$ | 集合 $A_1, A_2, ..., A_n$ 的交集 | $\bigcap_{i=1}^{n} A_i = A_1 \cap A_2 \cap ... \cap A_n$也可使用 $\bigcap_{i=1}^{n}$, $\bigcap_{i \in I}$, $\bigcap_{i \in I}$, 其中 $I$ 表示指标集；进一步，令 $P(i)$ 为某个与 $i$ 相关的命题，可使用 $\bigcap_{P(i)} A_i$ 表示所有使 $P(i)$ 为真的 $i$ 对应的 $A_i$ 之交集 |
| n2.13 | $A \setminus B$ | $A$ 和 $B$ 的差集 | $A \setminus B = \{x \mid x \in A \wedge x \notin B\}$不应使用 $A - B$当 $B$ 是 $A$ 的子集时也可使用 $\complement_A B$, 如果从上下文中可以得知考虑的是哪个集合 $A$，则 $A$ 可以省略．不引起歧义的情况下也可使用 $\overline{B}$ 表示集合 $B$ 的补集. |
| n2.14 | $(a, b)$ | 有序数对 $a$，$b$有序偶 $a$，$b$ | $(a, b) = (c, d)$ 当且仅当 $a = c$ 且 $b = d$. |
| n2.15 | $(a_1, a_2, ..., a_n)$ | 有序 $n$ 元组 | 参见 n2.14→1@p15. |
| n2.16 | $A \times B$ | 集合 $A$ 和 $B$ 的笛卡尔积 | $A \times B = \{(x, y) \mid x \in A \wedge y \in B\}$. |
| n2.17 | $\prod_{i=1}^{n} A_i$ | 集合 $A_1, A_2, ..., A_n$ 的笛卡尔积 | $\prod_{i=1}^{n} A_i = \{(x_1, x_2, ..., x_n) \mid x_1 \in A_1, x_2 \in A_2, ..., x_n \in A_n\}A \times A \times ... \times A$ 记为 $A^n$, 其中 $n$ 是乘积中的因子数；该符号的另一种用法参见 n6.8→1@p15 |

| | | | |
|---|---|---|---|
| n2.18 | $\mathrm{id}_A$ | $A \times A$ 的对角集 | $\mathrm{id}_A = \{(x,x) \mid x \in A\}$如果从上下文中可以得知考虑的是哪个集合 $A$, 则 $A$ 可以省略. |
| n2.19 | $\mathbf{1}_A$ | 指示函数 | $\mathbf{1}_A(a) = [a \in A]$, $[\cdot]$ 的定义参见 n6.24→1@p15. |
| n2.20 | $\mathcal{P}(A)2^A$ | 幂集 | $\mathcal{P}(A) = \{S : S \subseteq A\}$ |

## 标准数集和区间

| 编号 | 符号，表达式 | 意义，等同表述 | 备注与示例 |
|---|---|---|---|
| n3.1 | $\mathbf{N}$ | 自然数集 | $\mathbf{N} = \{0,1,2,3,...\}\mathbf{N}^* = \mathbf{N}_+ = \{1,2,3,...\}$可用如下方式添加其他限制：$\mathbf{N}_{>5} = \{n \in \mathbf{N} \mid n > 5\}$也可使用 $\mathbb{N}$. |
| n3.2 | $\mathbf{Z}$ | 整数集 | $\mathbf{Z}^* = \mathbf{Z}_+ = \{n \in \mathbf{Z} \mid n \neq 0\}$可用如下方式添加其他限制：$\mathbf{Z}_{>-3} = \{n \in \mathbf{Z} \mid n > -3\}$也可使用 $\mathbb{Z}$. |
| n3.3 | $\mathbf{Q}$ | 有理数集 | $\mathbf{Q}^* = \mathbf{Q}_+ = \{r \in \mathbf{Q} \mid r \neq 0\}$可用如下方式添加其他限制：$\mathbf{Q}_{<0} = \{r \in \mathbf{Q} \mid r < 0\}$也可使用 $\mathbb{Q}$. |
| n3.4 | $\mathbf{R}$ | 实数集 | $\mathbf{R}^* = \mathbf{R}_+ = \{x \in \mathbf{R} \mid x \neq 0\}$可用如下方式添加其他限制：$\mathbf{R}_{>0} = \{x \in \mathbf{R} \mid x > 0\}$也可使用 $\mathbb{R}$. |
| n3.5 | $\mathbf{C}$ | 复数集 | $\mathbf{C}^* = \mathbf{C}_+ = \{z \in \mathbf{C} \mid z \neq 0\}$也可使用 $\mathbb{C}$. |
| n3.6 | $\mathbf{P}$ | （正）素数集 | $\mathbf{P} = \{2,3,5,7,11,13,17,...\}$也可使用 $\mathbb{P}$. |
| n3.7 | $[a,b]$ | $a$ 到 $b$ 的闭区间 | $[a,b] = \{x \in \mathbf{R} \mid a \leq x \leq b\}$. |
| n3.8 | $(a,b]$ | $a$ 到 $b$ 的左开右闭区间 | $(a,b] = \{x \in \mathbf{R} \mid a < x \leq b\}(-\infty,b] = \{x \in \mathbf{R} \mid x \leq b\}$. |
| n3.9 | $[a,b)$ | $a$ 到 $b$ 的左闭右开区间 | $[a,b) = \{x \in \mathbf{R} \mid a \leq x < b\}[a,+\infty) = \{x \in \mathbf{R} \mid a \leq x\}$. |
| n3.10 | $(a,b)$ | $a$ 到 $b$ 的开区间 | $(a,b) = \{x \in \mathbf{R} \mid a < x < b\}(-\infty,b) = \{x \in \mathbf{R} \mid x < b\}$ $(a,+\infty) = \{x \in \mathbf{R} \mid a < x\}$. |

## 关系

| 编号 | 符号，表达式 | 意义，等同表述 | 备注与示例 |
|---|---|---|---|
| n4.1 | $a = b$ | $a$ 等于 $b$ | $\equiv$ 用于强调某等式是恒等式该符号的另一个含义参见 n4.18→1@p15. |
| n4.2 | $a \neq b$ | $a$ 不等于 $b$ | |
| n4.3 | $a := b$ | $a$ 定义为 $b$ | 参见 n2.9→1@p15,n2.10→1@p15 |
| n4.4 | $a \approx b$ | $a$ 约等于 $b$ | 不排除相等. |
| n4.5 | $a \simeq b$ | $a$ 渐近等于 $b$ | 例如：当 $x \to a$ 时，$\frac{1}{\sin(x-a)} \simeq \frac{1}{x-a} x \to a$ 的含义参见 n4.15→1@p15. |
| n4.6 | $a \propto b$ | $a$ 与 $b$ 成正比 | 也可使用 $a \sim b$.$\sim$ 也用于表示等价关系. |
| n4.7 | $M \cong N$ | $M$ 与 $N$ 全等 | 当 $M$ 和 $N$ 是点集（几何图形）时. 该符号也用于表示代数结构的同构. |
| n4.8 | $a < b$ | $a$ 小于 $b$ | |
| n4.9 | $b > a$ | $b$ 大于 $a$ | |
| n4.10 | $a \leq b$ | $a$ 小于等于 $b$ | |
| n4.11 | $b \geq a$ | $b$ 大于等于 $a$ | |
| n4.12 | $a \ll b$ | $a$ 远小于 $b$ | |
| n4.13 | $b \gg a$ | $b$ 远大于 $a$ | |
| n4.14 | $\infty$ | 无穷大 | 该符号 **不** 是数字. 也可以使用 $+\infty$, $-\infty$. |

| | | | |
|---|---|---|---|
| n4.15 | $x \to a$ | $x$ 趋近于 $a$ | 一般出现在极限表达式中．$a$ 也可以为 $\infty$，$+\infty$，$-\infty$. |
| n4.16 | $m \mid n$ | $m$ 整除 $n$ | 对整数 $m$，$n$:$(\exists\ k \in \mathbf{Z})\ \ m \cdot k = n$. |
| n4.17 | $m \perp n$ | $m$ 与 $n$ 互质 | 对整数 $m$，$n$:$(\nexists\ k \in \mathbf{Z}_{>1})\ \ (k \mid m) \wedge (k \mid n)$该符号的另一种用法参见 n5.2→1@p15 |
| n4.18 | $n \equiv k \pmod m$ | $n$ 模 $m$ 与 $k$ 同余 | 对整数 $n$，$k$，$m$:$m \mid (n-k)$不要与 n4.1→1@p15 中提到的相混淆. |

## 初等几何学

| 编号 | 符号，表达式 | 意义，等同表述 | 备注与示例 |
|---|---|---|---|
| n5.1 | $\parallel$ | 平行 | |
| n5.2 | $\perp$ | 垂直 | 该符号的另一种用法参见 n4.17→1@p15 |
| n5.3 | $\angle$ | （平面）角 | |
| n5.4 | $\overline{\mathrm{AB}}$ | 线段 AB | |
| n5.5 | $\overrightarrow{\mathrm{AB}}$ | 有向线段 AB | |
| n5.6 | $d(\mathrm{A},\mathrm{B})$ | 点 A 和 B 之间的距离 | 即 $\overline{\mathrm{AB}}$ 的长度. |

## 运算符

| 编号 | 符号，表达式 | 意义，等同表述 | 备注与示例 |
|---|---|---|---|
| n6.1 | $a + b$ | $a$ 加 $b$ | |
| n6.2 | $a - b$ | $a$ 减 $b$ | |
| n6.3 | $a \pm b$ | $a$ 加或减 $b$ | |
| n6.4 | $a \mp b$ | $a$ 减或加 $b$ | $-(a \pm b) = -a \mp b$. |
| n6.5 | $a \cdot b a \times b a b$ | $a$ 乘 $b$ | 若出现小数点，则应只使用 $\times$部分用例参见 n2.16→1@p15,n2.17→1@p15,n14.11→1@p15,n14.12→1@p15 |
| n6.6 | $\frac{a}{b} a/b a : b$ | $a$ 除以 $b$ | $\frac{a}{b} = a \cdot b^{-1}$可用：表示同一量纲的数值的比率．不应使用 $\div$. |
| n6.7 | $\sum_{i=1}^{n} a_i$ | $a_1 + a_2 + \ldots + a_n$ | 也可使用 $\sum_{i=1}^{n} a_i$，$\sum_{i} a_i$，$\sum_{i} a_i$，$\sum a_i$；令 $P(i)$ 为某个与 $i$ 相关的命题，可使用 $\sum_{P(i)} a_i$ 表示所有使 $P(i)$ 为真的 $i$ 对应的 $a_i$ 之和. |
| n6.8 | $\prod_{i=1}^{n} a_i$ | $a_1 \cdot a_2 \cdot \ldots \cdot a_n$ | 也可使用 $\prod_{i=1}^{n} a_i$，$\prod_{i} a_i$，$\prod_{i} a_i$，$\prod a_i$；令 $P(i)$ 为某个与 $i$ 相关的命题，可使用 $\prod_{P(i)} a_i$ 表示所有使 $P(i)$ 为真的 $i$ 对应的 $a_i$ 之积；该符号的另一种用法参见 n2.17→1@p15 |
| n6.9 | $a^p$ | $a$ 的 $p$ 次幂 | |
| n6.10 | $a^{1/2} \sqrt{a}$ | $a$ 的 $1/2$ 次方，$a$ 的平方根 | 应避免使用 $\sqrt{a}$. |
| n6.11 | $a^{1/n} \sqrt[n]{a}$ | $a$ 的 $1/n$ 次幂，$a$ 的 $n$ 次根 | 应避免使用 $\sqrt[n]{a}$. |
| n6.12 | $\bar{x} \bar{x}_a$ | $x$ 的算数均值 | 其他均值有：调和均值 $\bar{x}_h$几何均值 $\bar{x}_g$二次均值/均方根 $\bar{x}_q$ 或 $\bar{x}_{rms}$.$\bar{x}$ 也用于表示复数 $x$ 的共轭，参见 n11.6→1@p15. |
| n6.13 | $\operatorname{sgn} a$ | $a$ 的符号函数 | 对实数 $a$:$\operatorname{sgn} a = 1 \quad (a > 0) \operatorname{sgn} a = -1 \quad (a < 0) \operatorname{sgn} 0 = 0$参见 n11.7→1@p15. |

| | | | |
|---|---|---|---|
| n6.14 | $\inf M$ | $M$ 的下确界 | 非空集合 $M$ 的最大下界. |
| n6.15 | $\sup M$ | $M$ 的上确界 | 非空集合 $M$ 的最小上界. |
| n6.16 | $\lvert a\rvert$ | $a$ 的绝对值 | 也可使用 $\operatorname{abs} a$. |
| n6.17 | $\lfloor a\rfloor$ | 向下取整小于等于实数 $a$ 的最大整数 | 例如：$\lfloor 2.4\rfloor = 2 \lfloor -2.4\rfloor = -3$. |
| n6.18 | $\lceil a\rceil$ | 向上取整大于等于实数 $a$ 的最小整数 | 例如：$\lceil 2.4\rceil = 3 \lceil -2.4\rceil = -2$. |
| n6.19 | $\min(a,b) \min\{a,b\}$ | $a$ 和 $b$ 的最小值 | 可推广到有限集中. 要表示无限集中的最小值建议使用 inf, 参见 n6.14→1@p15 |
| n6.20 | $\max(a,b) \max\{a,b\}$ | $a$ 和 $b$ 的最大值 | 可推广到有限集中. 要表示无限集中的最大值建议使用 sup, 参见 n6.15→1@p15 |
| n6.21 | $n \bmod m$ | $n$ 模 $m$ 的余数 | 对正整数 $n$, $m$:$(\exists\, q \in \mathbf{N}, r \in [0,m))\ \ n = qm + r$其中 $r = n \bmod m$. |
| n6.22 | $\gcd(a,b) \gcd\{a,b\}$ | 整数 $a$ 和 $b$ 的最大公因数 | 可推广到有限集中. 不引起歧义的情况下可写为 $(a,b)$. |
| n6.23 | $\operatorname{lcm}(a,b) \operatorname{lcm}\{a,b\}$ | 整数 $a$ 和 $b$ 的最小公倍数 | 可推广到有限集中. 不引起歧义的情况下可写为 $[a,b] (a,b) [a,b] = \lvert ab\rvert$. |
| n6.24 | $[P]$ | Iverson 括号 | 若命题 $P$ 为真, 则 $[P] = 1$, 否则 $[P] = 0$. |
| n6.25 | $a \uparrow b$; $a \overset{n}{\uparrow} b$ | Knuth 箭头 | 对非负整数 $a, b, n$: $a \overset{n}{\uparrow} b = a \underbrace{\uparrow \ldots \uparrow}_{n \text{ times}} b$; $a \overset{0}{\uparrow} b = ab$; $a \overset{1}{\uparrow} b = a \uparrow b = a^b$; $a \uparrow 0 = 1 \quad (n > 0)$; $a \overset{n}{\uparrow} b = a \overset{n-1}{\uparrow} (a \overset{n}{\uparrow} (b-1))$. |
| n6.26 | $[x^n] f(x)$ | 多项式/形式幂级数/形式 Laurent 级数 $f(x)$ 中 $x^n$ 项的系数 | 若 $f(x) = \sum_i a_i x^i$, 则 $[x^n] f(x) = a_n$; 可推广到多元情况, 如若 $f(x,y) = \sum_{i,j} a_{i,j} x^i y^j$, 则 $[x^n y^m] f(x,y) = a_{n,m}$. |

## 组合数学

本节中的 $n$ 和 $k$ 是自然数，$a$ 是复数，且 $k \le n$.

| 编号 | 符号，表达式 | 意义，等同表述 | 备注与示例 |
|---|---|---|---|
| n7.1 | $n!$ | 阶乘 | $n! = \prod_{k=1}^{n} k = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n \quad (n > 0) 0! = 1$. |
| n7.2 | $a^{\underline{k}} (a)_{-k}$ | 下降阶乘幂 | $a^{\underline{k}} = a \cdot (a-1) \cdot \ldots \cdot (a-k+1) \quad (k > 0) a^{\underline{0}} = 1 n^{\underline{k}} = \dfrac{n!}{(n-k)!}$. |
| n7.3 | $a^{\overline{k}} (a)_{+k}$ | 上升阶乘幂 | $a^{\overline{k}} = a \cdot (a+1) \cdot \ldots \cdot (a+k-1) \quad (k > 0) a^{\overline{0}} = 1 n^{\overline{k}} = \dfrac{(n+k-1)!}{(n-1)!}$. |
| n7.4 | $\binom{n}{k}$ | 组合数 | $\binom{n}{k} = \dfrac{n!}{k!(n-k)!}$. |
| n7.5 | $\begin{bmatrix} n \\ k \end{bmatrix}$ | 第一类 Stirling 数 | $\begin{bmatrix} n+1 \\ k \end{bmatrix} = n \begin{bmatrix} n \\ k \end{bmatrix} + \begin{bmatrix} n \\ k-1 \end{bmatrix} x^{\overline{n}} = \sum_{k=0}^{n} \begin{bmatrix} n \\ k \end{bmatrix} x^k$. |
| n7.6 | $\begin{Bmatrix} n \\ k \end{Bmatrix}$ | 第二类 Stirling 数 | $\begin{Bmatrix} n \\ k \end{Bmatrix} = \dfrac{1}{k!} \sum_{i=0}^{k} (-1)^i \binom{k}{i} (k-i)^n \sum_{k=0}^{n} \begin{Bmatrix} n \\ k \end{Bmatrix} x^{\underline{k}} = x^n$. |

## 函数

| 编号 | 符号，表达式 | 意义，等同表述 | 备注与示例 |
|---|---|---|---|
| n8.1 | $f$ | 函数 | |
| n8.2 | $f(x)$, $f(x_1, \ldots, x_n)$ | 函数 $f$ 在 $x$ 处的值函数 $f$ 在 $(x_1, \ldots, x_n)$ 处的值 | |
| n8.3 | $\operatorname{dom} f$ | $f$ 的定义域 | 也可使用 $\mathrm{D}(f)$. |
| n8.4 | $\operatorname{ran} f$ | $f$ 的值域 | 也可使用 $\mathrm{R}(f)$. |

| | | | |
|---|---|---|---|
| n8.5 | $f:A\to B$ | $f$ 是 $A$ 到 $B$ 的映射 | $\operatorname{dom} f=A$ 且 $(\forall\ x\in\operatorname{dom} f)\quad f(x)\in B$. |
| n8.6 | $x\mapsto T(x),x\in A$ | 将所有 $x\in A$ 映射到 $T(x)$ 的函数 | $T(x)$ 仅用于定义，用来表示某个参数为 $x\in A$ 的某个函数值．若这个函数为 $f$, 则对所有 $x\in A$ 均有 $f(x)=T(x)$. 因此 $T(x)$ 通常用来定义函数 $f$.例如：$x\mapsto 3x^2y,x\in[0,2]$这是由 $3x^2y$ 定义的一个关于 $x$ 的二次函数．若未引入函数符号，则用 $3x^2y$ 表示该函数 |
| n8.7 | $f^{-1}$ | $f$ 的反函数 | 函数 $f$ 的反函数 $f^{-1}$ 有定义当且仅当 $f$ 是单射．若 $f$ 是单射，则 $\operatorname{dom}(f^{-1})=\operatorname{ran} f,\quad \operatorname{ran}(f^{-1})=\operatorname{dom} f$，且 $(\forall\ x\in\operatorname{dom} f)\quad f^{-1}(f(x))=x$.不要与函数的倒数 $f(x)^{-1}$ 混淆. |
| n8.8 | $g\circ f$ | $f$ 和 $g$ 的复合函数 | $(g\circ f)(x)=g(f(x))$. |
| n8.9 | $f:x\mapsto y$ | $f(x)=y$，$f$ 将 $x$ 映射到 $y$ | |
| n8.10 | $f\ \vert_a^b f(...,u,...)\vert_{u=a}^{u=b}$ | $f(b)-f(a)f(...,b,...)-f(...,a,...)$ | 主要用于定积分的计算中. |
| n8.11 | $\lim\limits_{x\to a}f(x)\lim_{x\to a}f(x)$ | 当 $x$ 趋近于 $a$ 时 $f(x)$ 的极限 | $\lim_{x\to a}f(x)=b$ 可以写成 $f(x)\to b\quad(x\to a)$.右极限和左极限的符号分别为 $\lim_{x\to a+}f(x)$ 和$\lim_{x\to a-}f(x)$. |
| n8.12 | $f(x)=O(g(x))$ | $\vert f(x)/g(x)\vert$ 在上下文隐含的限制中有上界，$f(x)$ 的阶不高于 $g(x)$ | 当 $f/g$ 与 $g/f$ 均有界时称 $f$ 与 $g$ 是同阶的．使用符号 "=" 是出于历史原因，其在此处不表示等价，因为不满足传递性．例如：$\sin x=O(x)\quad(x\to 0)$. |
| n8.13 | $f(x)=o(g(x))$ | 在上下文隐含的限制中有 $f(x)/g(x)\to 0$，$f(x)$ 的阶高于 $g(x)$ | 使用符号 "=" 是出于历史原因，其在此处不表示等价，因为不满足传递性．例如：$\cos x=1+o(x)\quad(x\to 0)$. |
| n8.14 | $\Delta f$ | $f$ 的有限增量 | 上下文隐含的两函数值的差分．例如：$\Delta x=x_2-x_1\Delta f(x)=f(x_2)-f(x_1)$. |
| n8.15 | $\dfrac{\mathrm{d}f}{\mathrm{d}x}f'$ | $f$ 对 $x$ 的导（函）数 | 仅用于一元函数．可以显式指明自变量，如 $\dfrac{\mathrm{d}f(x)}{\mathrm{d}x}$，$f'(x)$. |
| n8.16 | $\left(\dfrac{\mathrm{d}f}{\mathrm{d}x}\right)_{x=a}\quad f'(a)$ | $f$ 在 $a$ 处的导（函）数值 | 参见 n8.15→1@p15 |
| n8.17 | $\dfrac{\mathrm{d}^nf}{\mathrm{d}x^n}f^{(n)}$ | $f$ 对 $x$ 的 $n$ 阶导（函）数 | 仅用于一元函数．可以显式指明自变量，如 $\dfrac{\mathrm{d}^nf(x)}{\mathrm{d}x^n}$，$f^{(n)}(x)$.可用 $f''$ 和 $f'''$ 分别表示 $f^{(2)}$ 和 $f^{(3)}$. |
| n8.18 | $\dfrac{\partial f}{\partial x}f_x$ | $f$ 对 $x$ 的偏导数 | 仅用于多元函数．可以显式指明自变量，如 $\dfrac{\partial f(x,y,...)}{\partial x}$，$f_x(x,y,...)$.可以扩展到高阶，如 $f_{xx}=\dfrac{\partial^2f}{\partial x^2}=\dfrac{\partial}{\partial x}\left(\dfrac{\partial f}{\partial x}\right)f_{xy}=\dfrac{\partial^2f}{\partial y\partial x}=\dfrac{\partial}{\partial y}\left(\dfrac{\partial f}{\partial x}\right)$. |
| n8.19 | $\dfrac{\partial(f_1,...,f_m)}{\partial(x_1,...,x_n)}$ | Jacobi 矩阵 | 参见[1] |
| n8.20 | $\mathrm{d}f$ | $f$ 的全微分 | $\mathrm{d}f(x,y,...)=\dfrac{\partial f}{\partial x}\mathrm{d}x+\dfrac{\partial f}{\partial y}\mathrm{d}y+....$ |

[1] $\dfrac{\partial(f_1,...,f_m)}{\partial(x_1,...,x_n)}=\begin{pmatrix}\dfrac{\partial f_1}{\partial x_1}&\cdots&\dfrac{\partial f_1}{\partial x_n}\\\vdots&\ddots&\vdots\\\dfrac{\partial f_m}{\partial x_1}&\cdots&\dfrac{\partial f_m}{\partial x_n}\end{pmatrix}$ 矩阵的定义参见 n12.1→1@p15

| | | | |
|---|---|---|---|
| n8.21 | $\delta f$ | $f$ 的（无穷小）变分 | |
| n8.22 | $\int f(x)\mathrm{d}x$ | $f$ 的不定积分 | |
| n8.23 | $\int_a^b f(x)\mathrm{d}x$ | $f$ 从 $a$ 到 $b$ 的定积分 | 也可使用 $\int_a^b f(x)\mathrm{d}x$定积分还可以定义在更一般的域上. 如 $\int_C$, $\int_S$, $\int_V$, $\oint$, 分别表示在曲线 $C$, 曲面 $S$, 三维区域 $V$, 和闭曲线或曲面上的定积分. 多重积分可写成 $\iint$, $\iiint$ 等. |
| n8.24 | $f*g$ | 函数 $f$ 和 $g$ 的卷积 | $(f*g)(x)=\int_{-\infty}^{\infty} f(y)g(x-y)\mathrm{d}y.$ |

## 指数和对数函数

$x$ 可以是复数.

| 编号 | 符号，表达式 | 意义，等同表述 | 备注与示例 |
|---|---|---|---|
| n9.1 | e | 自然对数的底 | $\mathrm{e}=\lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n=2.718\ 281\ 8...$不要写成 $e$. |
| n9.2 | $a^x$ | $x$ 的指数函数（以 $a$ 为底） | 参见 n6.9→1@p15. |
| n9.3 | $\mathrm{e}^x\exp x$ | $x$ 的指数函数（以 e 为底） | |
| n9.4 | $\log_a x$ | $x$ 的以 $a$ 为底的对数 | 当底数不需要指定的时候可以使用 $\log x$.不应用 $\log x$ 替换 $\ln x$，$\lg x$，$\mathrm{lb}\,x$ 中的任意一个. |
| n9.5 | $\ln x$ | $x$ 的自然对数 | $\ln x=\log_{\mathrm{e}} x$参见 n9.4→1@p15. |
| n9.6 | $\lg x$ | $x$ 的常用对数 | $\lg x=\log_{10} x$参见 n9.4→1@p15. |
| n9.7 | $\mathrm{lb}\,x$ | $x$ 的以 2 为底的对数 | $\mathrm{lb}\,x=\log_2 x$参见 n9.4→1@p15. |

## 三角函数和双曲函数

| 编号 | 符号，表达式 | 意义，等同表述 | 备注与示例 |
|---|---|---|---|
| n10.1 | $\pi$ | 圆周率 | $\pi=3.141\ 592\ 6...$. |
| n10.2 | $\sin x$ | $x$ 的正弦 | $\sin x=\frac{\mathrm{e}^{\mathrm{i}x}-\mathrm{e}^{-\mathrm{i}x}}{2\mathrm{i}}$，$(\sin x)^n$，$(\cos x)^n (n\geq 2)$ 等通常写为 $\sin^n x$，$\cos^n x$ 等. |
| n10.3 | $\cos x$ | $x$ 的余弦 | $\cos x=\sin(x+\pi/2)$. |
| n10.4 | $\tan x$ | $x$ 的正切 | $\tan x=\sin x/\cos x$不可使用 $\mathrm{tg}\,x$. |
| n10.5 | $\cot x$ | $x$ 的余切 | $\cot x=1/\tan x$不可使用 $\mathrm{ctg}\,x$. |
| n10.6 | $\sec x$ | $x$ 的正割 | $\sec x=1/\cos x$. |
| n10.7 | $\csc x$ | $x$ 的余割 | $\csc x=1/\sin x$不可使用 $\mathrm{cosec}\,x$. |
| n10.8 | $\arcsin x$ | $x$ 的反正弦 | $y=\arcsin x\iff x=\sin y\quad(-\pi/2\leq y\leq\pi/2)$. |
| n10.9 | $\arccos x$ | $x$ 的反余弦 | $y=\arccos x\iff x=\cos y\quad(0\leq y\leq\pi)$. |
| n10.10 | $\arctan x$ | $x$ 反正切 | $y=\arctan x\iff x=\tan y\quad(-\pi/2\leq y\leq\pi/2)$不可使用 $\mathrm{arctg}\,x$. |
| n10.11 | $\mathrm{arccot}\,x$ | $x$ 反余切 | $y=\mathrm{arccot}\,x\iff x=\cot y\quad(0\leq y\leq\pi)$不可使用 $\mathrm{arcctg}\,x$. |
| n10.12 | $\mathrm{arcsec}\,x$ | $x$ 反正割 | $y=\mathrm{arcsec}\,x\iff x=\sec y\quad(0\leq y\leq\pi, y\neq\pi/2)$. |
| n10.13 | $\mathrm{arccsc}\,x$ | $x$ 的反余割 | $y=\mathrm{arccsc}\,x\iff x=\csc y\quad(-\pi/2\leq y\leq\pi/2, y\neq 0)$不可使用 $\mathrm{arccosec}\,x$. |
| n10.14 | $\sinh x$ | $x$ 的双曲正弦 | $\sinh x=\frac{\mathrm{e}^x-\mathrm{e}^{-x}}{2}$不可使用 $\mathrm{sh}\,x$. |
| n10.15 | $\cosh x$ | $x$ 的双曲余弦 | $\cosh^2 x=\sinh^2 x+1$不可使用 $\mathrm{ch}\,x$. |

| | | | |
|---|---|---|---|
| n10.16 | $\tanh x$ | $x$ 的双曲正切 | $\tanh x = \sinh x / \cosh x$不可使用 $\operatorname{th} x$. |
| n10.17 | $\coth x$ | $x$ 的双曲余切 | $\coth x = 1 / \tanh x$. |
| n10.18 | $\operatorname{sech} x$ | $x$ 的双曲正割 | $\operatorname{sech} x = 1 / \cosh x$. |
| n10.19 | $\operatorname{csch} x$ | $x$ 的双曲余割 | $\operatorname{csch} x = 1 / \sinh x$不可使用 $\operatorname{cosech} x$. |
| n10.20 | $\operatorname{arsinh} x$ | $x$ 的反双曲正弦 | $y = \operatorname{arsinh} x \iff x = \sinh y$不可使用 $\operatorname{arsh} x$. |
| n10.21 | $\operatorname{arcosh} x$ | $x$ 的反双曲余弦 | $y = \operatorname{arcosh} x \iff x = \cosh y \quad (y \geq 0)$不可使用 $\operatorname{arch} x$. |
| n10.22 | $\operatorname{artanh} x$ | $x$ 的反双曲正切 | $y = \operatorname{artanh} x \iff x = \tanh y$不可使用 $\operatorname{arth} x$. |
| n10.23 | $\operatorname{arcoth} x$ | $x$ 的反双曲余切 | $y = \operatorname{arcoth} x \iff x = \coth y \quad (y \neq 0)$. |
| n10.24 | $\operatorname{arsech} x$ | $x$ 的反双曲正割 | $y = \operatorname{arsech} x \iff x = \operatorname{sech} y \quad (y \geq 0)$. |
| n10.25 | $\operatorname{arcsch} x$ | $x$ 的反双曲余割 | $y = \operatorname{arcsch} x \iff x = \operatorname{csch} y \quad (y \geq 0)$不可使用 $\operatorname{arcosech} x$. |

## 复数

| 编号 | 符号，表达式 | 意义，等同表述 | 备注与示例 |
|---|---|---|---|
| n11.1 | $\mathrm{i}$ | 虚数单位 | $\mathrm{i}^2 = -1$不可使用 $i$ 或 $\mathtt{i}$ |
| n11.2 | $\operatorname{Re} z$ | $z$ 的实部 | 参见 n11.3→1@p15. |
| n11.3 | $\operatorname{Im} z$ | $z$ 的虚部 | 若 $z = x + \mathrm{i}y \quad (x, y \in \mathbf{R})$, 则 $x = \operatorname{Re} z$， $y = \operatorname{Im} z$. |
| n11.4 | $\lvert z \rvert$ | $z$ 的模 | $\lvert z \rvert = \sqrt{(\operatorname{Re} z)^2 + (\operatorname{Im} z)^2}$. |
| n11.5 | $\arg z$ | $z$ 的辐角 | 若 $z = r\mathrm{e}^{\mathrm{i}\varphi}$, 其中 $r = \lvert z \rvert$ 且 $-\pi < \varphi \leq \pi$, 则 $\varphi = \arg z$.$\operatorname{Re} z = r\cos\varphi$, $\operatorname{Im} z = r\sin\varphi$. |
| n11.6 | $\bar{z}z^*$ | $z$ 的复共轭 | $\bar{z} = \operatorname{Re} z - \mathrm{i}\operatorname{Im} z$. |
| n11.7 | $\operatorname{sgn} z$ | $z$ 的单位模函数 | $\operatorname{sgn} z = z/\lvert z \rvert = \exp(\mathrm{i}\arg z) \quad (z \neq 0)\operatorname{sgn} 0 = 0$参见 n6.13→1@p15. |

## 矩阵

| 编号 | 符号，表达式 | 意义，等同表述 | 备注与示例 |
|---|---|---|---|
| n12.1 | $A$参见[2] | $m \times n$ 型矩阵 $A$ | $a_{ij} = (A)_{ij}$也可使用 $A = (a_{ij})$. 其中 $m$ 为行数，$n$ 为列数$m = n$ 时称为方阵可用方括号替代圆括号. |
| n12.2 | $A + B$ | 矩阵 $A$ 和 $B$ 的和 | $(A + B)_{ij} = (A)_{ij} + (B)_{ij}$矩阵 $A$ 和 $B$ 的行数和列数必须分别相同. |
| n12.3 | $xA$ | 标量 $x$ 和矩阵 $A$ 的乘积 | $(xA)_{ij} = x(A)_{ij}$. |
| n12.4 | $AB$ | 矩阵 $A$ 和 $B$ 的乘积 | $(AB)_{ik} = \sum_j (A)_{ij}(B)_{jk}$矩阵 $A$ 的列数必须等于矩阵 $B$ 的行数. |
| n12.5 | $IE$ | 单位矩阵 | $(I)_{ik} = \delta_{ik}\delta_{ik}$ 的定义参见 n14.9→1@p15. |
| n12.6 | $A^{-1}$ | 方阵 $A$ 的逆 | $AA^{-1} = A^{-1}A = I \quad (\det A \neq 0)$.$\det A$ 的定义参见 n12.10→1@p15. |
| n12.7 | $A^{\mathrm{T}}A'$ | $A$ 的转置矩阵 | $(A^{\mathrm{T}})_{ik} = (A)_{ki}$. |
| n12.8 | $\overline{A}A^*$ | $A$ 的复共轭矩阵 | $\left(\overline{A}\right)_{ik} = \overline{(A)_{ik}}$. |
| n12.9 | $A^{\mathrm{H}}A^\dagger$ | $A$ 的 Hermite 共轭矩阵 | $A^{\mathrm{H}} = \left(\overline{A}\right)^{\mathrm{T}}$. |
| n12.10 | $\det A$参见[3] | 方阵 $A$ 的行列式 | 也可使用 $\lvert A \rvert$. |
| n12.11 | $\operatorname{rank} A$ | 矩阵 $A$ 的秩 | |

[2] $\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}$

[3] $\begin{vmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \cdots & a_{nn} \end{vmatrix}$

| | | | |
|---|---|---|---|
| n12.12 | $\mathrm{tr}\, A$ | 方阵 $A$ 的迹 | $\mathrm{tr}\, A = \sum_i (A)_{ii}$. |
| n12.13 | $\|\|A\|\|$ | 矩阵 $A$ 的范数 | 满足三角不等式：若 $A+B=C$, 则 $\|\|A\|\|+\|\|B\|\| \geq \|\|C\|\|$. |

## 坐标系

本节考虑三维空间中的一些坐标系．点 O 为坐标系的 **原点**．任意点 P 均由从原点 O 到点 P 的 **位置向量** 确定．

| 编号 | 坐标 | 位置向量和微分 | 坐标名 | 备注 |
|---|---|---|---|---|
| n13.1 | $x$，$y$，$z$ | $\boldsymbol{r} = x\boldsymbol{e}_x + y\boldsymbol{e}_y + z\boldsymbol{e}_z$ $\mathrm{d}\boldsymbol{r} = \mathrm{d}x\ \boldsymbol{e}_x + \mathrm{d}y\ \boldsymbol{e}_y + \mathrm{d}z\ \boldsymbol{e}_z$ | 笛卡尔坐标 | 基向量 $\boldsymbol{e}_x$，$\boldsymbol{e}_y$，$\boldsymbol{e}_z$ 构成右手正交系，见 图 1→1@p15 和 图 4→1@p15．基向量也可用 $\boldsymbol{e}_1$，$\boldsymbol{e}_2$，$\boldsymbol{e}_3$ 或 $\boldsymbol{i}$，$\boldsymbol{j}$，$\boldsymbol{k}$ 表示，坐标也可用 $x_1$，$x_2$，$x_3$ 或 $i$，$j$，$k$ 表示． |
| n13.2 | $\rho$，$\varphi$，$z$ | $\boldsymbol{r} = \rho\ \boldsymbol{e}_\rho + z\ \boldsymbol{e}_z$ $\mathrm{d}\boldsymbol{r} = \mathrm{d}\rho\ \boldsymbol{e}_\rho + \rho\ \mathrm{d}\varphi\ \boldsymbol{e}_\varphi + \mathrm{d}z\ \boldsymbol{e}_z$ | 柱坐标 | $\boldsymbol{e}_\rho(\varphi)$，$\boldsymbol{e}_\varphi(\varphi)$，$\boldsymbol{e}_z$ 组成右手正交系，见 图 2→1@p15．若 $z=0$, 则 $\rho$ 和 $\varphi$ 是平面上的极坐标． |
| n13.3 | $r$，$\vartheta$，$\varphi$ | $\boldsymbol{r} = r\boldsymbol{e}_r$ $\mathrm{d}\boldsymbol{r} = \mathrm{d}r\ \boldsymbol{e}_r + r\ \mathrm{d}\vartheta\ \boldsymbol{e}_\vartheta + r\ \sin\vartheta\ \mathrm{d}\varphi\ \boldsymbol{e}_\varphi$ | 球坐标 | $\boldsymbol{e}_r(\vartheta,\varphi)$, $\boldsymbol{e}_\vartheta(\vartheta,\varphi)$, $\boldsymbol{e}_\varphi(\varphi)$ 组成右手正交系，见 图 3→1@p15． |

如果不使用 右手坐标系→1@p15，而使用 左手坐标系→1@p15，则应在之前明确强调，以免符号误用．

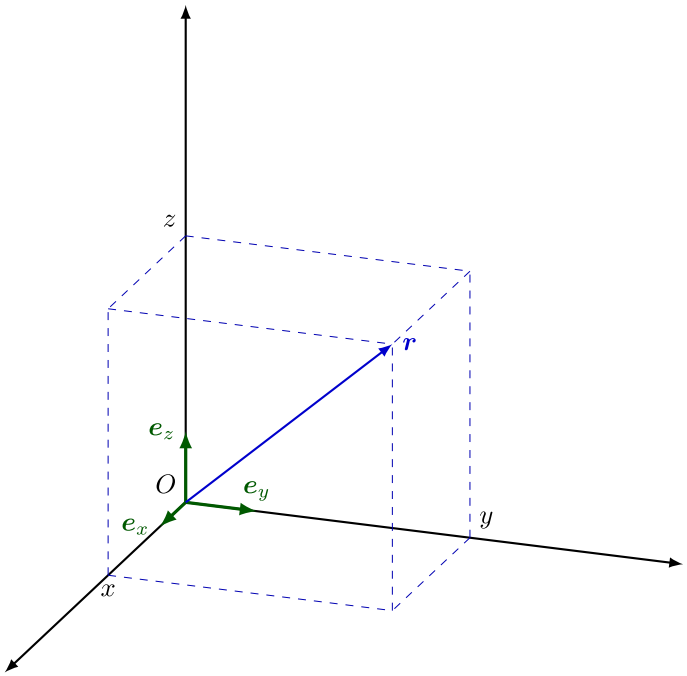



**图 1** 右手笛卡尔坐标系

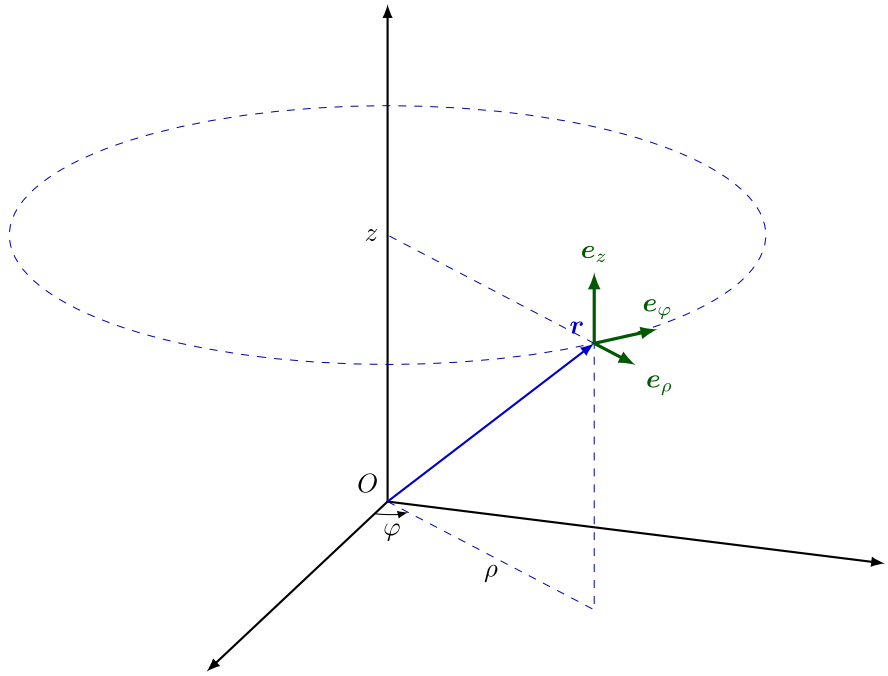



**图 2** 右手柱坐标系

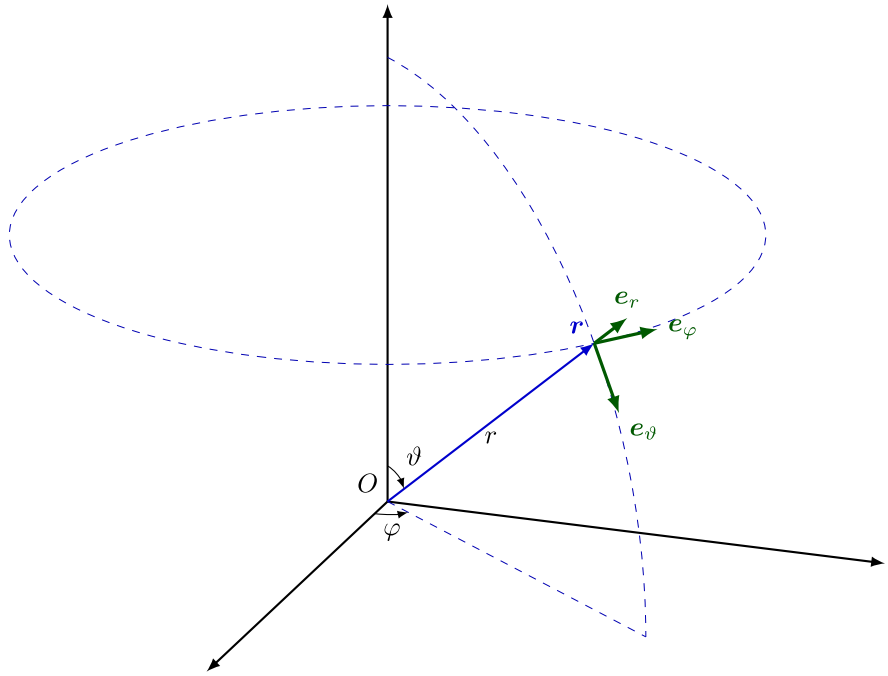



**图 3** 右手球坐标系

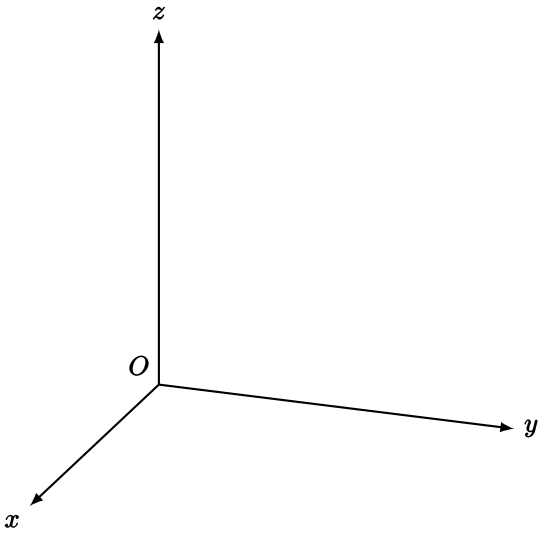



**图 4** 右手坐标系

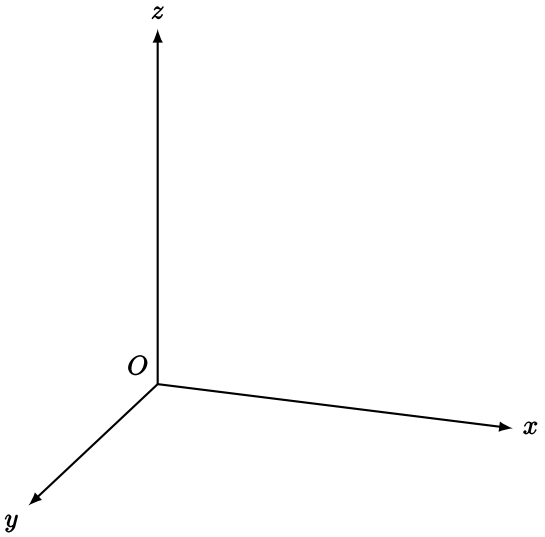



**图 5** 左手坐标系

## 标量和向量

本节中，基向量用 $\boldsymbol{e}_1$，$\boldsymbol{e}_2$，$\boldsymbol{e}_3$ 表示．本节中的许多概念都可以推广到 $n$ 维空间.

标量和向量本身与坐标系的选择无关，而向量的每个标量分量与坐标系的选择有关.

对于基向量 $\boldsymbol{e}_1$，$\boldsymbol{e}_2$，$\boldsymbol{e}_3$, 每个向量 $\boldsymbol{a}$ 都可以表示为 $\boldsymbol{a}=a_1\boldsymbol{e}_1+a_2\boldsymbol{e}_2+a_3\boldsymbol{e}_3$, 其中 $a_1$，$a_2$ 和 $a_3$ 是唯一确定的标量值，将其称为向量相对于该组基向量的 "坐标"，$a_1\boldsymbol{e}_1$，$a_2\boldsymbol{e}_2$ 和 $a_3\boldsymbol{e}_3$ 称为向量相对于该组基向量的分向量.

在本节中，只考虑普通空间的笛卡尔（正交）坐标．笛卡尔坐标用 $x$，$y$，$z$ 或 $a_1$，$a_2$，$a_3$ 或 $x_1$，$x_2$，$x_3$ 表示.

本节所有下标 $i$，$j$，$k$ 的范围均为 1 到 3.

| 编号 | 符号，表达式 | 意义，等同表述 | 备注与示例 |
| --- | --- | --- | --- |
| n14.1 | $\boldsymbol{a}\vec{a}$ | 向量 $\boldsymbol{a}$ | |
| n14.2 | $\boldsymbol{a}+\boldsymbol{b}$ | 向量 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 的和 | $(\boldsymbol{a}+\boldsymbol{b})_i=a_i+b_i$. |
| n14.3 | $x\boldsymbol{a}$ | 标量 $x$ 与向量 $\boldsymbol{a}$ 的乘积 | $(x\boldsymbol{a})_i=xa_i$. |

| | | | |
|---|---|---|---|
| n14.4 | $\lvert\boldsymbol{a}\rvert$ | 向量 $\boldsymbol{a}$ 的大小，向量 $\boldsymbol{a}$ 的范数 | $\lvert\boldsymbol{a}\rvert=\sqrt{a_x^2+a_y^2+a_z^2}$也可使用 $\lVert a\rVert$. |
| n14.5 | $\vec{\mathbf{0}}\mathbf{0}$ | 零向量 | 零向量的大小为 0. |
| n14.6 | $\boldsymbol{e}_{\boldsymbol{a}}$ | $\boldsymbol{a}$ 方向的单位向量 | $\boldsymbol{e}_{\boldsymbol{a}}=\boldsymbol{a}/\lvert\boldsymbol{a}\rvert\quad(\boldsymbol{a}\neq\mathbf{0})$. |
| n14.7 | $\boldsymbol{e}_x$，$\boldsymbol{e}_y$，$\boldsymbol{e}_z\boldsymbol{e}_1$，$\boldsymbol{e}_2$，$\boldsymbol{e}_3$ | 笛卡尔坐标轴方向的单位向量 | 也可使用 $\boldsymbol{i}$，$\boldsymbol{j}$，$\boldsymbol{k}$. |
| n14.8 | $a_x$，$a_y$，$a_za_i$ | 向量 $\boldsymbol{a}$ 的笛卡尔分量 | $\boldsymbol{a}=a_x\boldsymbol{e}_x+a_y\boldsymbol{e}_y+a_z\boldsymbol{e}_z$如果上下文确定了基向量，则向量可以写为 $\boldsymbol{a}=(a_x,a_y,a_z).a_x=\boldsymbol{a}\cdot\boldsymbol{e}_x,a_y=\boldsymbol{a}\cdot\boldsymbol{e}_y,a_z=\boldsymbol{a}\cdot\boldsymbol{e}_z\boldsymbol{r}=x\boldsymbol{e}_x+y\boldsymbol{e}_y+z\boldsymbol{e}_z$ 是坐标为 $x$，$y$，$z$ 的位置向量. |
| n14.9 | $\delta_{ik}$ | Kronecker delta 符号 | $\delta_{ik}=[i=k]$，其中 $[\cdot]$ 的定义参见 n6.24→1@p15，即：$\delta_{ik}=1\quad(i=k)\delta_{ik}=0\quad(i\neq k)$. |
| n14.10 | $\varepsilon_{ijk}$ | Levi-Civita 符号 | $\varepsilon_{123}=\varepsilon_{231}=\varepsilon_{312}=1\varepsilon_{132}=\varepsilon_{321}=\varepsilon_{213}=-1$ 其余的 $\varepsilon_{ijk}$ 均为 0. |
| n14.11 | $\boldsymbol{a}\cdot\boldsymbol{b}$ | 向量 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 的标量积/内积 | $\boldsymbol{a}\cdot\boldsymbol{b}=\sum_i a_ib_i$. |
| n14.12 | $\boldsymbol{a}\times\boldsymbol{b}$ | 向量 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 的向量积/外积 | 右手笛卡尔坐标系中，$(\boldsymbol{a}\times\boldsymbol{b})_i=\sum_j\sum_k\varepsilon_{ijk}a_jb_k\varepsilon_{ijk}$ 的定义参见 n14.10→1@p15. |
| n14.13 | $\boldsymbol{\nabla}$ | nabla 算子 | $\boldsymbol{\nabla}=\boldsymbol{e}_x\frac{\partial}{\partial x}+\boldsymbol{e}_y\frac{\partial}{\partial y}+\boldsymbol{e}_z\frac{\partial}{\partial z}=\sum_i\boldsymbol{e}_i\frac{\partial}{\partial x_i}$. |
| n14.14 | $\boldsymbol{\nabla}\varphi\mathbf{grad}\,\varphi$ | $\varphi$ 的梯度 | $\boldsymbol{\nabla}\varphi=\sum_i\boldsymbol{e}_i\frac{\partial\varphi}{\partial x_i}\mathbf{grad}$ 应使用 `\operatorname{\mathbf{grad}}`. |
| n14.15 | $\boldsymbol{\nabla}\cdot\boldsymbol{a}\mathbf{div}\,\boldsymbol{a}$ | $\boldsymbol{a}$ 的散度 | $\boldsymbol{\nabla}\cdot\boldsymbol{a}=\sum_i\frac{\partial a_i}{\partial x_i}\mathbf{div}$ 应使用 `\operatorname{\mathbf{div}}`. |
| n14.16 | $\boldsymbol{\nabla}\times\boldsymbol{a}\mathbf{rot}\,\boldsymbol{a}$ | $\boldsymbol{a}$ 的旋度 | $(\boldsymbol{\nabla}\times\boldsymbol{a})_i=\sum_j\sum_k\varepsilon_{ijk}\frac{\partial a_k}{\partial x_j}\mathbf{rot}$ 应使用 `\operatorname{\mathbf{rot}}`.不应使用 $\mathbf{curl}$.$\varepsilon_{ijk}$ 的定义参见 n14.10→1@p15. |
| n14.17 | $\boldsymbol{\nabla}^2\Delta$ | Laplace 算子 | $\boldsymbol{\nabla}^2=\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}+\frac{\partial^2}{\partial z^2}$. |

## 特殊函数

本节中的 $z$，$w$ 是复数，$k$，$n$ 是自然数，且 $k\leq n$.

| 编号 | 符号，表达式 | 意义，等同表述 | 备注与示例 |
|---|---|---|---|
| n15.1 | $\gamma$ | Euler–Mascheroni 常数 | $\gamma=\lim_{n\to\infty}\left(\sum_{k=1}^{n}\frac{1}{k}-\ln n\right)=0.577\ 215\ 6....$ |
| n15.2 | $\Gamma(z)$ | gamma 函数 | $\Gamma(z)=\int_0^\infty t^{z-1}\mathrm{e}^{-t}\mathrm{d}t\quad(\mathrm{Re}\,z>0)\Gamma(n+1)=n!$. |
| n15.3 | $\zeta(z)$ | Riemann zeta 函数 | $\zeta(z)=\sum_{n=1}^{\infty}\frac{1}{n^z}\quad(\mathrm{Re}\,z>1)$. |
| n15.4 | $\mathrm{B}(z,w)$ | beta 函数 | $\mathrm{B}(z,w)=\int_0^1 t^{z-1}(1-t)^{w-1}\mathrm{d}t\quad(\mathrm{Re}\,z>0,\quad\mathrm{Re}\,w>0)\mathrm{B}(z,w)=\frac{\Gamma(z)\Gamma(w)}{\Gamma(z+w)}\frac{1}{(n+1)\,\mathrm{B}(k+1,n-k+1)}=\binom{n}{k}$. |

[1] GB/T 3102.11-1993 [2] ISO 80000-2:2019 [3] 本文章的源代码

# 1.7 F.A.Q.

本页面主要解答一些常见的问题.

## 我想问点与这个 Wiki 相关的问题

Q：你们是为什么想要做这个 Wiki 的呢?

A：不知道你在学 OI 的时候，面对庞大的知识体系，有没有感到过迷茫无助的时候? OI Wiki 想要做的事情可能类似于「让更多竞赛资源不充裕的同学能方便地接触到训练资源」. 当然这么表述也不完全，做 Wiki 的动机可能也很纯粹，只是简单地想要对 OI 的发展做出一点点微小的贡献吧. XD

* * *

Q：我很感兴趣，怎么参与?

A：OI Wiki 现在托管在 GitHub 上，你可以直接访问这个 repo[1] 来查看最新进展. 参与的途径包括在 GitHub 上面开 Issue[2]、Pull Request[3]，或者在交流群中分享你的想法、直接向管理员投稿. 目前，我们使用的框架是用 Python 开发的 MkDocs[4]，支持 Markdown 格式（也支持插入数学公式）.

* * *

Q：可是我比较弱……不知道我能做点什么.

A：一切源于热爱. 你可以协助其他人审核修改稿件，帮助我们宣传 OI Wiki，为社区营造良好学习交流氛围!

* * *

Q：现在主要是谁在做这件事啊? 感觉这是个大坑，真的能做好吗?

A：最开始主要是一些退役老年选手在做这件事，后来遇到了很多志同道合的小伙伴：有现役选手，退役玩家，也有从未参加过 OI 的朋友. 目前，这个项目主要是由 OI Wiki 项目组来维护（下面是一张合影）.

当然，这个项目只靠我们的力量是很难做得十全十美的，我们诚挚地邀请你一起来完善 OI Wiki.

* * *

Q：你们怎么保证我们添加的内容不会突然消失?

A：我们把内容托管在 GitHub[5] 上面，即使我们的服务器翻车了，内容也不会丢失. 另外，我们也会定期备份大家的心血，即使有一天 GitHub 倒闭了（?），我们的内容也不会丢失.

* * *

Q：OI Wiki 好像有空的页面啊!

A：是的. 受限于项目组成员的水平和时间，我们暂时无法完成这些空页面. 所以我们在这里进行征稿和招募，希望可以遇到有同样想法的朋友，我们一起把 OI Wiki 完善起来.

* * *

Q：为什么不直接去写 中文维基百科[6] 呢?

A：因为我们希望可以真正帮到更多的选手或者对这些内容感兴趣的人. 而且由于众所周知的原因，中文维基上的内容并不是无门槛就可以获取到的.

## 我想参与进来!

Q：我要怎么与项目组交流?

A：可以通过 关于本项目里的交流方式→1.2@p16 联系我们.

* * *

Q：我要怎么贡献代码或者内容?

请参考 如何参与→1.3@p17 页面.

* * *

Q：目录在哪?

A：目录在项目根目录下的 mkdocs.yml[7] 文件中.

* * *

Q：如何修改一个 topic 的内容?

A：在对应页面右上方有一个编辑按钮 edit，点击并确认阅读了 如何贡献→1.3@p17 之后会跳转到 GitHub 上对应文件的位置.

或者也可以自行阅读目录 (mkdocs.yml)[8] 查找文件位置.

* * *

Q：如何添加一个 topic?

A：有两种选择：

- 可以开一个 Issue，注明希望能添加的内容.
- 可以开一个 Pull Request，在目录 (mkdocs.yml)[9] 中加上新的 topic，并在 docs[10] 文件夹下对应位置创建一个空的 `.md` 文件. 文档的格式细节请参考 格式手册→1.5@p23.

* * *

Q：我尝试访问 GitHub 的时候遇到了困难.

A：推荐在 hosts 文件中加入如下几行[1]：

```
# GitHub Start
140.82.114.25                 alive.github.com
140.82.113.5                  api.github.com
185.199.110.153               assets-cdn.github.com
185.199.111.133               avatars.githubusercontent.com
185.199.111.133               avatars0.githubusercontent.com
185.199.111.133               avatars1.githubusercontent.com
185.199.111.133               avatars2.githubusercontent.com
185.199.111.133               avatars3.githubusercontent.com
185.199.111.133               avatars4.githubusercontent.com
185.199.111.133               avatars5.githubusercontent.com
185.199.111.133               camo.githubusercontent.com
140.82.112.22                 central.github.com
185.199.111.133               cloud.githubusercontent.com
140.82.114.9                  codeload.github.com
140.82.113.22                 collector.github.com
185.199.111.133               desktop.githubusercontent.com
185.199.111.133               favicons.githubusercontent.com
140.82.112.3                  gist.github.com
52.216.163.147                github-cloud.s3.amazonaws.com
52.217.124.1                  github-com.s3.amazonaws.com
52.216.144.83                 github-production-release-
asset-2e65be.s3.amazonaws.com
52.217.121.249                github-production-repository-
file-5c1aeb.s3.amazonaws.com
52.217.206.57                 github-production-user-
asset-6210df.s3.amazonaws.com
192.0.66.2                    github.blog
140.82.114.4                  github.com
140.82.113.18                 github.community
185.199.110.154               github.githubassets.com
151.101.1.194                 github.global.ssl.fastly.net
185.199.110.153               github.io
185.199.111.133               github.map.fastly.net
185.199.110.153               githubstatus.com
```

[1] GitHub520[11]

```
140.82.112.25               live.github.com
185.199.111.133             media.githubusercontent.com
185.199.111.133             objects.githubusercontent.com
13.107.42.16                pipelines.actions.githubusercontent.com
185.199.111.133             raw.githubusercontent.com
185.199.111.133             user-images.githubusercontent.com
13.107.253.40               vscode.dev
140.82.112.21               education.github.com
# GitHub End
```

可以在 GitHub520[12] 上了解到最新内容和更多信息.

Linux 和 macOS 用户可以尝试使用 依云[13] 的 gh-check 脚本[14] 获取访问最快的 IP，使用 `--hosts` 参数可以直接更新 hosts 文件. 使用 `--help` 参数可以获取使用帮助. 使用先需要安装 Python3 和 aiohttp（`pip install aiohttp -i https://pypi.tuna.tsinghua.edu.cn/simple/`）. 依云博客的介绍：寻找最快的 GitHub IP[15].

同时，您可以使用 Gitclone[16] 服务加速 Clone，可以阅读其首页上的说明.

如果您仅仅是想 Clone **OI Wiki** 的仓库，那么：

```
git clone https://gitclone.com/github.com/OI-wiki/OI-wiki
```

如果您需要向 **OI Wiki** 贡献，那么首先 fork **OI Wiki** 的仓库，然后（将 `username` 替换为您的用户名），需要注意的是提供的示例将使您使用 SSH 连接到 GitHub[2]：

```
git clone https://gitclone.com/github.com/username/OI-wiki
git remote set-url origin git@github.com:username/OI-wiki.git
```

* * *

Q：我这里 pip 也太慢了！

A：可以选择更换国内源[3]，或者：

```
pip install -U -r requirements.txt -i https://pypi.tuna.tsinghua.edu.cn/simple/
```

* * *

Q：我在客户端 clone 了这个项目，速度太慢.

A：如果有安装 `git bash`，可以加几个限制来减少下载量. [4]

```
git clone https://github.com/OI-wiki/OI-wiki.git --depth=1 -b master
```

* * *

Q：我没装过 Python 3.

A：可以访问 Python 官网[22] 了解更多信息.

* * *

Q：好像提示我 pip 版本过低.

A：进入 cmd/shell 之后，执行以下命令：

```
python -m pip install --upgrade pip
```

* * *

[2] GitHub 弃用了基于密码身份验证的 HTTPS 协议，连接必须使用 SSH 或者 Personal Access Token，参见 我应使用哪个远程 URL? [17]，创建个人访问令牌[18] 和 使用 SSH 连接到 GitHub[19].

[3] 更改 pip 源至国内镜像 - L 瑜 - CSDN 博客[20]

[4] GIT--- 看我一步步入门（Windows Git Bash）[21]

Q：我安装依赖失败了.

A：检查一下：网络？权限？查看错误信息？

* * *

Q：我已经 clone 下来了，为什么部署不了？

A：检查一下是否安装好了依赖？

* * *

Q：我 clone 了很久之前的 repo，怎么更新到新版本呢？

A：请参考 GitHub 官方的帮助页面 Syncing a fork - GitHub Docs[23].

* * *

Q：如果是装了之前的依赖怎么更新？

A：请输入以下命令：

```
pip install -U -r requirements.txt
```

* * *

Q：为什么我的 markdown 格式乱了？

A：可以查阅 cyent 的笔记[24]，或者 MkDocs 使用说明[25].

我们目前使用 remark-lint[26] 来自动化修正格式，可能还有一些 配置[27] 不够好的地方，欢迎指出.

* * *

Q：GitHub 是不是不显示我的数学公式？

A：是的，GitHub 的预览不显示数学公式. 但是请放心，MkDocs 是支持数学公式的，可以正常使用，只要是 MathJax 支持的句式都可以使用.

* * *

Q：我的数学公式怎么乱码了？

A：如果是行间公式（用的 `$$`），目前已知的问题是需要在 `$$` 两侧留有空行，且 `$$` 要 **单独** 放在一行里（且不要在前加空格）. 格式如下：

```
// 空行
$$
a_i
$$
// 空行
```

* * *

Q：我的公式为什么在目录里没有正常显示？好像双倍了.

A：是的，这个是 python-markdown 的一个 bug，可能近期会修复.

如果想要避免目录中出现双倍公式，可以参考 string 分类下 SAM 的目录写法[28].

```
结束位置 <script type="math/tex">endpos</script>
```

在目录中会变成

```
结束位置 endpos
```

注：现在请尽量避免在目录中引入 MathJax 公式.

* * *

Q：如何给一个页面单独声明版权信息？

A：在页面开头加一行即可. [5]

[5] Metadata - Material for MkDocs[29]

比如：

```
copyright: SATA
```

注：默认的是 CC BY-SA 4.0 和 SATA.

* * *

Q：为什么作者信息统计处没有我的名字？

A：如果你发现自己写过一个页面中的部分内容，但是你没有被记录进作者列表，可以把自己的 GitHub ID 加入到文件头的 author 字段→1.3@p17.

* * *

感谢你看到了最后，我们现在亟需的，就是你的帮助.

OI Wiki 项目组

2018.8

## 参考资料与注释

[1] repo

[2] Issue

[3] Pull Request

[4] MkDocs

[5] GitHub

[6] 中文维基百科

[7] mkdocs.yml

[8] (mkdocs.yml)

[9] (mkdocs.yml)

[10] docs

[11] GitHub520

[12] GitHub520

[13] 依云

[14] gh-check 脚本

[15] 寻找最快的 GitHub IP

[16] Gitclone

[17] 我应使用哪个远程 URL?

[18] 创建个人访问令牌

[19] 使用 SSH 连接到 GitHub

[20] 更改 pip 源至国内镜像 - L 瑜 - CSDN 博客

[21] GIT--- 看我一步步入门（Windows Git Bash）

[22] Python 官网

[23] Syncing a fork - GitHub Docs

[24] cyent 的笔记

[25] MkDocs 使用说明

[26] remark-lint

[27] 配置

[28] string 分类下 SAM 的目录写法

[29] Metadata - Material for MkDocs

# 1.8 镜像站列表

OI Wiki 部署在国外服务器上，有时可能会因为各种原因，出现访问不通畅的情况.

我们搭建了一个状态页：https://status.oi-wiki.org[1]，用于监控 OI Wiki 站点的在线情况. 如果你遇到了无法访问的问题，可以打开状态页，寻找可以连接的镜像站.

以下是一个 OI Wiki 的镜像站列表，可供选用：

- OI Wiki 主站，线路：DMIT
  - https://oi-wiki.org[2]

- 维护者：OI Wiki，线路：阿里云，同步频率：与主站相同
  - http://oi-wiki.com[3]
- 维护者：OI Wiki，线路：Netlify，同步频率：与主站相同
  - https://demo.oi-wiki.org[4]
- 维护者：琴春（vx.st[5]），线路：AWS，同步频率：与主站相同
  - https://oi-wiki.net[6]
  - https://oi-wiki.wiki[7]
  - https://oi-wiki.win[8]
  - https://oi-wiki.xyz[9]
  - https://oiwiki.moe[10]
  - https://oiwiki.net[11]
  - https://oiwiki.org[12]
  - https://oiwiki.vx.st[13]
  - https://oiwiki.wiki[14]
  - https://oiwiki.win[15]
  - https://oiwiki.com[16]
- 维护者：Menci（men.ci[17]），线路：Azure + 阿里云 CDN，同步频率：与主站相同
  - https://oi.wiki[18]

## 外部参考资料

[1] https://status.oi-wiki.org

[2] https://oi-wiki.org

[3] http://oi-wiki.com

[4] https://demo.oi-wiki.org

[5] vx.st

[6] https://oi-wiki.net

[7] https://oi-wiki.wiki

[8] https://oi-wiki.win

[9] https://oi-wiki.xyz

[10] https://oiwiki.moe

[11] https://oiwiki.net

[12] https://oiwiki.org

[13] https://oiwiki.vx.st

[14] https://oiwiki.wiki

[15] https://oiwiki.win

[16] https://oiwiki.com

[17] men.ci

[18] https://oi.wiki

# 1.9 致谢

本项目目前暂不接受捐赠.

所有款项将被用于 OI Wiki 的域名、服务器、运维等必需支出.

大额捐赠将会记录在本页面下方或日后更合适的位置来表示感谢.

* * *

| id | amount | date |
| --- | --- | --- |
| 匿名捐赠者 | 10 元 | 2021.6.20 |
| 匿名捐赠者 | 100 元 | 2021.6.10 |
| wood3 | 256 元 | 2021.6.4 |
| 海外兔 | 102 元 | 2021.5.12 |

| | | |
|---|---|---|
| 逐梦之人 | 80.7 元 | 2021.5.3 |
| 十寸雨 zsg | 10.24 元 | 2021.2.27 |
| 匿名捐赠者 | 2 元 | 2020.12.10 |
| 吾有一數名之曰詼 | 10.24 元 | 2020.10.19 |
| Huah | 66 元 | 2020.9.7 |
| icedream61 | 66 元 | 2020.9.2 |
| 三鸽酸鸽可爱 | 2.33 元 | 2020.8.31 |
| Apoi2333 | 23.33 元 | 2020.8.31 |
| 草莓熊 | 50 元 | 2020.7.4 |
| 匿名捐赠者 | 200 元 | 2020.6.29 |
| Hiroid | 30 元 | 2020.5.13 |
| dcmfqw | 20.48 元 | 2020.5.4 |
| akira | 66 元 | 2020.3.30 |
| Hermione | 10 元 | 2020.3.26 |
| Siyuan | 10 元 | 2020.2.28 |
| hsh | 15 元 | 2020.2.21 |
| 陌陌 | 10 元 | 2020.2.6 |
| 匿名捐赠者 | 10 元 | 2020.2.5 |
| Yisin | 10 元 | 2020.2.4 |
| GinRyan | 50 元 | 2019.11.30 |
| JuicyMio | 10 元 | 2019.11.30 |
| 匿名捐赠者 | 10 元 | 2019.11.14 |
| QQ 联系 12 | 5 元 | 2019.10.28 |
| 匿名捐赠者 | 5 元 | 2019.10.27 |
| 增肥中的小肥 | 50 元 | 2019.10.24 |
| ianahao | 10 元 | 2019.10.12 |
| Sundy | 10 元 | 2019.10.11 |
| 三鸽最可爱 | 23.33 元 | 2019.8.17 |
| Fburan | 10.24 元 | 2019.8.17 |
| 匿名捐赠者 | 30 元 | 2019.8.8 |
| Billchenchina | 100 元 | 2019.8.7 |
| 贷款捐头的匿名入土 | 30 元 | 2019.8.4 |
| sshwy | 50 元 | 2019.8.4 |
| 匿名捐赠者 | 10 元 | 2018.9.9 |
| 匿名捐赠者 | 20 元 | 2018.8.31 |
| Xeonacid | 30 元 | 2018.8.30 |
| 匿名捐赠者 | 240 元 | 2018.8.29 |
| Anguei | 5 元 | 2018.8.29 |

# 第二章

# 比赛相关

# 2.1 比赛相关简介

本章主要介绍计算机编程比赛直接相关的知识，包括各种赛事、赛制、题型，以及赛场上常见的坑点与技巧.

学习路线，与常用的学习资源也可以在本章找到.

本章亦设出题板块，介绍出竞赛题的相关知识.

# 2.2 赛事

## 2.2.1 OI 赛事与赛制

> **Authors:** Ir1d, Planet6174, abc1763613206, StudyingFather, cjsoft, Marcythm, luoguyuntianming, ChungZH, Xeonacid, YZircon, i-Yirannn, H-J-Granger, NachtgeistW, YuzhenQin, Andycode3759, HHH2309, shigengxin123456, Re-Ori, hcx1204

### 赛事简介

**信息学奥林匹克竞赛**（英语：Olympiad in Informatics，简称：OI）是一门在中学生中广泛开展的学科竞赛，和物理、数学等竞赛性质相同. OI 考察的内容是参赛者运用算法、数据结构和数学知识，通过编写计算机程序解决实际问题的能力.

OI 竞赛种类繁多，仅中国就包括：

- 全国青少年信息学奥林匹克联赛（NOIP）
- 全国青少年信息学奥林匹克竞赛（NOI）
- 全国青少年信息学奥林匹克竞赛冬令营（WC）
- 国际信息学奥林匹克竞赛中国队选拔赛（CTSC）

国际性的 OI 竞赛包括：

- 国际信息学奥林匹克（IOI）
- 美国计算机奥林匹克竞赛（USACO）
- 日本信息学奥林匹克（JOI）
- 亚太地区信息学奥林匹克（APIO）

……

对于大部分选手而言，每年的新赛季从 9 月的 CSP-J/S 第一轮开始.

在中国，OI 竞赛允许使用的语言只有 C++（曾经也开放过 C 和 Pascal 语言，但都已停止支持）. 其中，不同的竞赛对 C++ 的版本有不同的规定. 考试题目一般为算法或者数据结构相关的内容，题目形式包括传统题（最常见的规定输入和输出到文件的题目）和非传统题（提交答案题、交互题、补全代码题……等等）.

### 赛制介绍

#### OI 赛制

选手仅有一次提交机会. 比赛时无法看到评测结果，评分会在赛后公布. 每道题都有多个测试点，根据每道题通过的测试点的数量获得相应的分数；每个测试点还可能会有部分分，即使只有部分数据通过也能拿到分数.

> **自评测工具 selfEval**
>
> 现如今，在一些 NOI 系列赛中，提供了 selfEval 自评测工具. selfEval 内置于全国赛定制版 NOI Linux 中. 自 NOI2023 正式公布并投入使用后，selfEval 陆续用于其后的 NOI 全国赛、APIO（中国区）和 NOI 冬令营等. 选手可以使用 selfEval 在一组测试数据（称为预测试数据）上测试自己的程序，并

> 得到反馈结果. 选手在每场比赛中的自测次数有指定上限（NOI2024 自测次数上限为 50 次，NOI2025 自测次数上限为 30 次），而且预测试数据也是选手不可见的. 由于预测试数据不同于正式测试数据，因此自测结果仅用于调试，不能被视作正式评测成绩. 选手在同一题目上多次进行预测试，所使用的预测试数据是相同的.

CSP-J/S 第二轮、NOIP、省选、NOI 都是 OI 赛制.

### IOI 赛制

选手在比赛时有多次提交机会. 比赛实时评测并返回结果，如果提交的结果是错误的，不会有任何惩罚. 每道题都有多个测试点，根据每道题通过的测试点的数量获得相应的分数.

APIO、IOI 都是 IOI 赛制. 目前国内比赛也在逐渐向 IOI 赛制靠拢.

### Codeforces (CF) 赛制

Codeforces[1] 是一个在线评测系统，会定期举办比赛.

它的比赛特点是在比赛过程中只测试一部分数据（Pretests），而在比赛结束后返回完整的所有测试点的测试结果（System Tests）. 比赛时可以多次提交，允许 Hack 别人的代码（此处 Hack 的意思是提交一个测试数据，使得别人的代码无法给出正确答案）. 如果想要 Hack，选手必须要锁定自己的代码（换言之，比赛时无法重新提交该题）. Hack 时不允许将选手程序拷贝到本地进行测试，源代码会被转换成图片.

Codeforces 同时提供另外一种赛制，称作扩展 ICPC（Extended ICPC 或 ICPC+）. 在这一赛制中，在比赛过程中会测试全部数据，但比赛结束以后会有 12 小时的全网 Hack 时间. Hack 时允许将选手程序拷贝到本地进行测试.

## 主要比赛

### CSP-J/S

CSP-J/S（英文：Certified Software Professional Junior/Senior）是 NOIP 在 2019 年被取消之后，CCF 开设的非专业级软件能力认证测试，在 2025 年以前面向全年龄段，后改为 12 周岁以上[2].

CSP-J/S 分为入门级（Junior，简写为 CSP-J）与提高级（Senior，简写为 CSP-S）两组，赛程分为第一轮（一般在每年 9 月）和第二轮（一般在每年 10 月）两场. 第一轮为笔试，考察计算机理论和操作常识和基本的算法与数学知识；第二轮为上机考试，入门组与提高组都为 4 题，其中入门组考试时间 3.5 个小时，提高组 4 个小时（CSP-S 2019 除外，该场比赛使用旧 NOIP 提高组赛制，赛程分为两天，一天 3 题 3.5 小时）. 第一轮面向社会全体 12 周岁以上学生报名，经过一定的排名筛选后成绩优秀者有机会参加第二轮.

报名参加第一/二轮、第二轮后进行题目申诉等都需要向 CCF 缴费.

两轮测试都会以省为单位按照排名对选手成绩进行评级认证，分为一、二、三等.

### NOIP

NOIP（英语：National Olympiad in Informatics in Provinces，中文：全国青少年信息学奥林匹克联赛）是中华人民共和国组织的、面向中国（含港澳）中学生的信息学竞赛.

2018 年及以前的旧赛制：NOIP 按参赛对象分为普及组和提高组，2018 年于上海试点入门组；按阶段分为初赛和复赛两个阶段. 初赛会考察一些计算机基础知识和算法基础，复赛为上机考试. 时间上一般是 11 月的第二个周末，周六上午提高组一试 8:30-12:00（3.5 小时，共 3 题），下午 14:30-18:00 普及组（3.5 小时，共 4 题），周日上午提高组二试 8:30-12:00（3.5 小时，共 3 题）. 全国使用同一套试卷，但是评奖规则按照省内情况由 CCF（中国计算机学会）统一指定，并于赛后在 NOI 官方网站[3] 上公布. 各省的一等奖分数线略有不同.

NOIP 于 2019 年 8 月 16 日 被 CCF 暂停[4]，于 2020 年 1 月 21 日 被宣布恢复[5]. 2020 年起的 NOIP 赛制与以往有所不同，具体如下：

- 取消初赛，由 CSP-J/S 第一轮替代；
- 取消普及组，由 CSP-J 替代，此后 NOIP 仅有一个组别，面向提高组水平选手；
- 赛程由以往的两天共 6 题、每天 3.5 个小时，缩减为一天 4 题、共 4.5 个小时.
- 选手需要在 CSP-S 第二轮中取得一定名次才能获得 NOIP 参赛资格，具体名额各省有所差异. NOIP 省级参赛资格由该省在去年赛季中的参赛人数和成绩等有关.

报名参加 NOIP 和进行题目申诉不需要额外缴费.

NOIP 以省为单位排名评奖. 截至 2019 年，大部分高校的选手获得提高组省一等奖可以得到自主招生资格.

> 2020 年 1 月，中华人民共和国教育部发布 关于在部分高校开展基础学科招生改革试点工作的意见[6]. 意见指出，2020 年起，不再组织开展高校自主招生工作，并在部分一流大学建设高校开展基础学科招生改革试点（强基计划）.

### 省队选拔赛

**省队选拔赛**（简称：省选）用于选拔各省参加全国赛的代表队，一般举行于每年的 1~4 月. 赛程上一般分为两天，每天 3 题 4.5 小时.

省选题目由各个省自行决定，目前的趋势是很多省份选择联合命题.

各个省队的名额有复杂的计算公式，一般和之前的成绩和参赛人数有关. 通常来讲，NOIP 分数需要在省选的指标中占一定比例. 根据规则，初中选手只能被选拔为 E 类选手，不能参加 A、B 类选拔. A 类选手有 5 人（至少 1 女[7]），其他选手根据给定名额和所得分数依次进入 B 队. 一个学校参加 NOI 的名额不超过本省 A、B 名额总数的三分之一（四舍五入），得分最高且入选 A 队的女选手不占该比例（简称 1/3 限制或 1/3 淘汰，详见 CCF 官方说明[8]）.

自 2020 年起，NOI 省队选拔由 CCF 统一命题和评测，有能力命题的省可自行命题，但选拔方式需得到 CCF 的批准. 自 2024 年起，NOI 省队选拔恢复各省自主命题，有需求的省份可组织联考或使用他省试题，但具体方案需要得到 CCF 的批准.

### NOI

NOI（英文：National Olympiad in Informatics，中文：全国信息学奥林匹克竞赛）是国内包括港澳在内的省级代表队最高水平的大赛.

NOI 一般在七月份举行，选手分为正式选手与夏令营选手两类. 正式选手又分为三类，其中 A、B 类为省队正式选手，C 类选手为邀请赛选手. A、B 类对应省队的 A、B 类选手（其中 A 类在计算成绩时会有 5 分加分）；C 类名义上是学校对 CCF 做出突出贡献后的奖励名额. 夏令营选手分为 D、E 类，分别对应以非正式选手身份参赛的高中组与初中组选手. 夏令营选手如果成绩超过分数线的话，只有成绩证明而没有奖牌（同等分数含金量要低一些）. 排名前 50 的正式选手组成国家集训队，获得保送资格.

在国际平台上，为了与其他同样称作 NOI 的比赛区分，有时会被称作 CNOI.

### CTT

CTT（英文：China Team Training，中文：国际信息学奥林匹克国家集训队培训）是每年冬天为 IOI 国家集训队选手举办的集训与选拔活动，由 3-4 场测试组成. 除国家集训队外，部分在当年 NOI 中取得优异成绩的选手也可以「精英集训」的名义参加 CTT.

CTT 与平时作业等其他流程共同组成了国家队选拔的第一阶段. 自 2021 年起，在第一阶段排名前 30 的选手将成为国家候选队，进入第二阶段的选拔（WC）.

### WC

WC（英文：Winter Camp，中文：全国青少年信息学奥林匹克竞赛冬令营）是每年冬天在当年 NOI 举办地进行的一项活动．虽然该活动主要用于集训队培训与国家队选拔，但是前一年 NOIP 与 CSP-S 第二轮取得较好成绩的选手也可作为非正式营员参加．

WC 的内容包括若干天的培训和测试，测试成绩将和先前阶段的成绩汇总，算出集训队选手的综合排名．在 2020 年前，测试仅有一场，且集训队与非正式营员的测试题目相同，集训队综合成绩前 15 的选手将成为国家候选队，参与最后阶段的选拔（CTS 等）；而自 2021 年起，随着 CTS 的国家队选拔功能并入到 WC 中，国家候选队的测试变为两场，而非正式营员的测试仍为一场，且非正式营员的测试题目和候选队测试题目有部分重合．候选队中综合排名前 6 的选手将进入到最终的面试，并选出 4 名正式选手和 2 名替补选手参与当年的 IOI 比赛．

### APIO

APIO（英文：Asia-Pacific Informatics Olympiad，中文：亚太地区信息学奥林匹克竞赛）是一个面向亚太地区在校中学生的信息学学科竞赛．CCF 每年会在五月初举办中国赛区镜像赛．在比赛日前后会有培训活动．

APIO 参赛选手可分为 A 类和 B 类，A 类选手中的前六名（含并列）可参与 APIO 的国际奖项评选；而 B 类选手只能参与中国赛区的奖项评选．

### CTS

CTS（旧称：CTSC, 英文：China Team Selection Competition，中文：国际信息学奥林匹克竞赛中国队选拔赛）用来从国家候选队（15 人）中选拔国家队（6 人）准备参加当年夏天的 IOI 比赛，其中正式选手 4 人，替补选手 2 人．与 WC 一样，前一年 NOIP 取得较好成绩的选手也可以参加（不参与选拔）．

APIO 和 CTS 都以省为单位报名，一般按照 NOIP 的成绩排序来确定参加 APIO 和 CTS 的人员（二者一般时间上非常接近）．

2020 年的 CTS 因为疫情而停办，当年的国家集训队通过 NOI 选出；2021 年起，CTS 的选拔流程被 WC 取代．

### IOI

IOI（英文：International Olympiad in Informatics，中文：国际信息学奥林匹克竞赛）是一年一度的面向全球中学生的信息学科竞赛．每个国家有四人参赛，比赛一般会有直播．IOI 赛制中每个题目会有 Subtask（子任务），每个子任务对应一定的分数．

## 学科营

### 北京大学（PKU）

- 北京大学信息学冬季体验营（PKUWC）：在冬令营前后举行．
- 北京大学信息学体验营（PKUSC）：一般在六月份在校内举行．由于在学校机房比赛，机房环境是 Windows，比赛系统是 OpenJudge．
- 北京大学中学生暑期课堂（信息学）：在暑假举行，面向高二年级理科学生．

### 清华大学（THU）

- 计算机系 "大中衔接" 冬季研讨与教学活动：相当于信息学冬令营，有时也会用英文简写为 THUWC．一般共两天，上午为竞赛（第一天是标准 OI 竞赛，第二天为清华独创的 "工程题" 竞赛），下午为课程培训．

## 其他国家和地区的 OI 竞赛

### 美国：USACO

官网地址：http://www.usaco.org/[9]

USACO 或许是国内选手最熟悉的外国 OI 竞赛（可能也是中文题解最多的外国 OI 竞赛）.

每年冬季到初春，USACO 会每月举办一场网络赛．一场比赛持续 3~5 个小时.

根据官网的介绍，USACO 的比赛分成这 4 档难度（2015~2016 学年之前为 3 档）:

- 铜牌组，适合编程初学者，尤其是只学了最最基础的算法（如：排序，二分查找）的学生;
- 银牌组，适合开始学习基本的算法技巧（如：递归，搜索，贪心算法）和基础数据结构的学生;
- 金牌组，学生会遇到更复杂的算法（如：最短路径，DP）和更高级的数据结构;
- 铂金组，适合有着扎实的算法设计能力的选手，铂金组可以帮助他们以复杂且更开放的问题来挑战自我.

在国内，目前 USACO 题目最齐全的 OJ 平台是洛谷.

### 波兰：POI

官网地址：https://oi.edu.pl/[10]

官方提交地址：https://szkopul.edu.pl/p/default/problemset/[11]

POI 是不少省选选手最常刷的外国 OI 比赛.

根据 POI 官网[12] 的描述，POI 的流程如下:

- 第一轮：六题（第 31 届及以前为五题），网络赛;
- 第二轮：包含一场练习赛，和两场正式赛，其中练习赛一题，正式赛每场两题;
- 第三轮：包含一场练习赛，和两场正式赛，其中练习赛一题，正式赛每场三题.

在部分年份，曾举办名为 ONTAK 的比赛，其正式名称为 POI 训练营，对标国内的国家集训队集训赛（CTT）.

另外, 波兰国内还举办名为 PA 的公开比赛, 大意为「算法大战」, 其官网地址是: https://potyczki.mimuw.edu.pl/[13].

目前在国内 OJ 中，POI 题目最全的是 BZOJ.

### 克罗地亚：COCI

官网地址（英文）：http://www.hsin.hr/coci/[14]

官网地址（克罗地亚语）：http://www.hsin.hr/honi/[15]

难度跨度很大的比赛，大约是从普及 - 到省选 -.

以往 COCI 所有的题目均提供题目、数据、题解和标程．2017 年底起，COCI 的题解和标程停止了更新．2019-2020 赛季重新开始更新题解和标程.

洛谷、BZOJ 和 LibreOJ 都有少量的 COCI 题目.

### 日本：JOI

官网地址：https://www.ioi-jp.org/[16]

JOI（日文：日本情報オリンピック，中文：日本信息学奥赛）所有的题目都提供题目、数据、题解和标程．近两年的 JOI 决赛和春训营提供了英语题面，但并没有英语题解．历年的 JOI Open 都提供了英语版题面和题解.

JOI 的流程:

- 预赛（予選）
- 决赛（本選/JOI Final）
- 春训营（春季トレーニング合宿/JOI Spring Camp/JOISC）
- 公开赛（通信教育/JOI Open Contest）

预赛难度较低，自 2019/2020 赛季起，预赛分为多轮．JOI Final 的难度从提高 - 到 提高 + 左右．JOISC 和 JOI Open 的题目的难度从提高到 NOI - 不等.

绝大部分 JOI 题可以前往 AtCoder[17] 提交．你可以在 JOI 官网或者 AtCoder 上找到更多的 JOI 题（日文题面）．

目前 LibreOJ 和 BZOJ 有近些年的 JOI Final、JOISC 和 JOI Open 的题目．

### 俄罗斯：ROI

官网地址：http://neerc.ifmo.ru/school/archive/index.html[18]

在线提交地址：https://contest.yandex.ru/roiarchive/[19] 和 Codeforces（部分）．

ROI（俄文：олимпиадная информатика，中文：俄罗斯信息学奥赛）是俄罗斯的信息学竞赛．

流程：

- 市级比赛（Municipal Stage/Муниципальный этап）
- 州级比赛（Regional Stage/Региональный этап）
- 决赛（Final Stage/Заключительный этап）

目前 LibreOJ 有近几年的 ROI 决赛题的译文．

除此之外，俄罗斯较大型的、面向中学生的比赛还有：

- 信息学网络奥赛（俄文：Интернет-олимпиады по информатике）
  - 官网地址：http://neerc.ifmo.ru/school/io/index.html[20]
  - 该比赛由 ROI 出题人举办．
- 全国中学生团队信息学竞赛（俄文：Всероссийской командной олимпиады школьников）
  - 官网地址：http://neerc.ifmo.ru/school/russia-team/index.html[21]
  - 该比赛的预选赛 Moscow Team Olympiad 可以在 Codeforces 上提交．
- Innopolis Open
  - 官网地址 https://olymp.innopolis.ru/en/ooui/information/[22]
- 中学生编程公开赛（Открытая олимпиада школьников по программированию）
  - 官网地址：https://olympiads.ru/zaoch/[23]
  - 官网称该比赛对标 ROI.

### 加拿大：CCC & CCO

CCC（英文：Canadian Computing Competition），CCO（英文：Canadian Computing Olympiad），可在其官网[24] 查询历届的信息和试题等．

在 DMOJ 上可以提交 CCC[25] 和 CCO[26]，该 OJ 上还有 CCC 题解．

CCC Junior/Senior 贴近 NOIP 普及组/提高组难度．CCO 想要拿到金牌可能得有 NOI 银牌的水平．

### 新加坡：NOI SG

官网地址：https://noisg.comp.nus.edu.sg/noi/[27]

全称 Singapore National Olympiad in Informatics，在新加坡国内语境且不引起歧义的情况下也作 NOI．赛制上分为 Online Qualification Contest（在线资格赛）和 Final Contest（全国决赛）．在线资格赛以学校为单位报名参加，选手在本校参赛，通过网络进行远程提交．资格赛成绩只在校内排名，前 5 名且非零分选手有资格作为校代表队参加全国决赛．

目前国内 OJ 对于 NOI SG 的题目收录比较匮乏，可以在 官方的 GitHub 账号[28] 上找到历年题面、测试数据和官方标准程序．

### 台湾地区：資訊奧林匹亞競賽

台湾地区把 OI 中的 informatics 翻译成「資訊」而非大陆通用的翻译「信息」．

台湾地区的选手如果想参加 IOI，需要经过这几轮比赛：

- 區域資訊學科能力競賽

- 全國資訊學科能力競賽
- 資訊研習營（TOI）

### 其他国家

- 澳大利亚：AIO：https://orac.amt.edu.au/hub/aio/[29]
  - 难度与 NOI 类似.
- 英国：British Informatics Olympiad：https://www.olympiad.org.uk/[30]
  - 难度太低.
- 捷克：Matematická olympiáda–kategorie P：http://mo.mff.cuni.cz/p/archiv.html[31]
- 罗马尼亚：Olimpiada Nationala de Informatica：http://olimpiada.info/[32]
  - 题面、测试数据、题解请在含有 Subiecte 字样的标签页中寻找.

## 其它国际 OI 竞赛

### BalticOI

**BalticOI** 面向的是波罗的海周边各国．BalticOI 2018 的参赛国有立陶宛、波兰、爱沙尼亚、芬兰等 9 国．题目难度大.

除了 2017 年，BalticOI 每年都公开题面、测试数据和题解．BalticOI 没有一个固定的官网，每年的主办方都会新建一个网站．历年的官网地址见 帖子[33].

目前 LibreOJ 有近十年的 BalticOI 题.

### BalkanOI

**BalkanOI** 面向巴尔干地区周边各国．BalkanOI 2018 的参赛国有罗马尼亚、希腊、保加利亚、塞尔维亚等 12 国．题目难度大.

BalkanOI 只有某几年公开题面、测试数据和题解，官网地址见 帖子[34].

### CEOI

CEOI 2018 的参赛国与上面两个比赛有部分重叠，包括波兰、罗马尼亚、格鲁吉亚、克罗地亚等国．题目难度大.

CEOI 每年都公开题面、测试数据和题解，官网地址见 帖子[35].

### eJOI

**eJOI** 全名 European Junior Olympiad in Informatics．参赛国包含俄罗斯、亚美尼亚、保加利亚、波兰等国．题目难度较大.

eJOI 每年都公开题面、测试数据和题解，官网地址见 帖子[36].

### NOI

> **⚠ Warning**
>
> 此处介绍的不是「全国信息学奥林匹克竞赛」.

**NOI** 全名 Nordic Olympiads in Informatics.

官网地址：http://nordic.progolymp.se[37]

近两年才开始举办的比赛，面向北欧各国.

## 参考资料

- ICPC/CCPC 赛事与赛制→2.2.2@p75
- 「翻译组」一些大洲级 OI 比赛的地址[38]

## 外部参考资料

[1] Codeforces 

[2] 后改为 12 周岁以上 

[3] NOI 官方网站 

[4] 被 CCF 暂停 

[5] 被宣布恢复 

[6] 关于在部分高校开展基础学科招生改革试点工作的意见 

[7] 至少 1 女 

[8] CCF 官方说明 

[9] http://www.usaco.org/ 

[10] https://oi.edu.pl/ 

[11] https://szkopul.edu.pl/p/default/problemset/ 

[12] POI 官网 

[13] https://potyczki.mimuw.edu.pl/ 

[14] http://www.hsin.hr/coci/ 

[15] http://www.hsin.hr/honi/ 

[16] https://www.ioi-jp.org/ 

[17] AtCoder 

[18] http://neerc.ifmo.ru/school/archive/index.html 

[19] https://contest.yandex.ru/roiarchive/ 

[20] http://neerc.ifmo.ru/school/io/index.html 

[21] http://neerc.ifmo.ru/school/russia-team/index.html 

[22] https://olymp.innopolis.ru/en/ooui/information/ 

[23] https://olympiads.ru/zaoch/ 

[24] 官网 

[25] CCC 

[26] CCO 

[27] https://noisg.comp.nus.edu.sg/noi/ 

[28] 官方的 GitHub 账号 

[29] https://orac.amt.edu.au/hub/aio/ 

[30] https://www.olympiad.org.uk/ 

[31] http://mo.mff.cuni.cz/p/archiv.html 

[32] http://olimpiada.info/ 

[33] 帖子 

[34] 帖子 

[35] 帖子 

[36] 帖子 

[37] http://nordic.progolymp.se 

[38]「翻译组」一些大洲级 OI 比赛的地址 

# 2.2.2 ICPC/CCPC 赛事与赛制

**Authors:** NachtgeistW, Ir1d, Xeonacid, H-J-Granger, abc1763613206, YuzhenQin

## 赛事介绍

### ICPC

ICPC（英文：International Collegiate Programming Contest，中文：国际大学生程序设计竞赛）由 ICPC 基金会（英文：ICPC Foundation）举办，是最具影响力的大学生计算机竞赛．由于以前 ACM 赞助这个竞赛，也有很多人习惯叫它 ACM 竞赛．

ICPC 主要分为区域赛（Regionals）和总决赛（World Finals）两部分．

官网地址：https://icpc.global[1]

### CCPC

官网地址：https://ccpc.io[2]

中国大学生程序设计竞赛．

和 ICPC 显著的区别是很多学校是不报销的.

### 赛制介绍

一般是三个人组成一队使用一台机器，在比赛时有多次提交机会. 比赛实时评测并返回结果，如果提交的结果错误会有 20 分钟的罚时，错误次数越多，加罚的时间也越长. 每个题目只有在所有数据点全部正确后才能得到分数. 比赛排名根据做题数来评判，做题数相同的，根据总用时来评判. 总用时是每题用时的和. 每题的用时是从比赛开始到做出该题的分钟数与该题的罚时之和.

一些 ICPC 相关赛事中，比赛结束前一小时进行封榜，封榜后的提交和排名将无法被其他选手看见.

在 ICPC 相关赛事中，选手允许带一定量的纸质资料.

除 ICPC 和 CCPC 外，众多比赛也采用该赛制，如 LeetCode 周赛及全国编程大赛、牛客小白赛练习赛挑战赛等.

### 赛季赛程

- ICPC/CCPC 网络赛（8 月底至 9 月初）
- ICPC/CCPC 区域赛（9 月底至 11 月底）
- ICPC EC Final/CCPC Final（12 月中旬）
- ICPC World Finals（次年 4 月至 6 月）

### 训练指南

#### 多校联合训练

暑期在 HDU OJ[3] 举行的训练赛. 有奖金，题目质量高，历经多年积累已有丰富资源.

OJ 里查询用的关键词：`Multi-University Training Contest`.

#### 国内区域赛

在 Virtual Judge[4] 里可以搜到精选题集.

#### 训练营

- 寒假的时候头条/清华/CCPC (Wannafly Camp) 举办的 Camp
- Wannafly Camp

### 训练资源

- QOJ：https://qoj.ac[5]
- Codeforces Gym：https://codeforces.com/gyms[6]

### 外部参考资料

[1] https://icpc.global 

[2] https://ccpc.io 

[3] HDU OJ 

[4] Virtual Judge 

[5] https://qoj.ac 

[6] https://codeforces.com/gyms 

## 2.3 题型

### 2.3.1 题型概述

> **Authors:** StudyingFather, NachtgeistW, countercurrent-time, Ir1d, H-J-Granger, Chrogeek, sshwy, Suyun514, hsfzLZH1, CBW2007, Xeonacid, kawa-yoiko, Konano

在算法竞赛中，有多种多样的问题类型.

## 传统题

**传统题** 是目前算法竞赛中较为常见的题型.

选手需要提交源代码, 评测系统会使用事先准备好一些输入数据和相应的输出数据作为测试点[1], 将选手提交的源代码编译后[2], 让选手程序读入输入数据, 通过将选手输出与事先准备好的输出比较, 来判断选手程序是否正确. 这种评测方式被称之为 **黑盒评测**[3].

对于一个测试点，往往还会设置时间限制和空间限制.

时间限制，指的是程序运行时间的限制[4]. 选手程序在一个测试点上的运行时间不能超过给定的时间限制.

空间限制, 指的是程序使用的内存量的限制. 选手程序在运行时占用的最大空间不能超过给定的空间限制.

在程序正常运行结束后, 选手的输出会和测试点输出进行比对. 这种比对一般采用过滤文末换行和行末空格之后，进行全文比对的方式. 对于某些特殊的题目，会使用 Special Judge→3.8@p261 来进行比对.

这一过程结束后，评测系统会根据程序的运行状态，给出不同的 **评测结果** [5]:

- Accepted（AC）：选手程序被接受.
- Compile Error（CE）：选手程序无法正常编译.
- Wrong Answer（WA）：选手程序正常结束，但是选手程序的输出与测试点输出不符.
- Presentation Error（PE）：选手程序正常结束，但是格式不符合要求 [6].
- Runtime Error（RE）：选手程序非正常结束（选手程序结束时的返回值不为零）.
- Time Limit Exceeded（TLE）：选手程序运行的时间超过了给定的时间限制.
- Memory Limit Exceeded（MLE）：选手程序占用的最大空间超过了给定的空间限制.
- Output Limit Exceeded（OLE）：选手程序输出的内容的量超过了最大限制.

在 ICPC 赛事中，你的程序需要在一道题目的所有测试点上都取得 AC 状态，才能视为通过相应的题目. 在 OI 赛事中，在一个测试点中取得 AC 状态，即可拿到该测试点的分数 [7].

## 提交答案题

**提交答案题** 是直接提交答案的题目. 该种题目一般会给出输入文件，要求提交包含有 `XXX1.out`、`XXX2.out`、`XXX3.out`..`XXXn.out` 的压缩包、文件夹或纯文件.

提交答案后, 评测系统会比较答案文件与标准答案, 根据选手答案的优劣情况和任务完成度, 给予一定的分数.

因为提交答案题不需要运行源程序，故提交答案题不存在时间和空间限制.

做这种题目一般有两种方法:

- 手玩. 这种方法简单粗暴，但是遇到较大的数据就没辙了.
- 编写一个程序来获得答案文件.

[1]因为技术上和资源上的限制，一道题目的测试点大多数情况下不能覆盖满足数据范围的全部数据.
[2]对于 Python 这样的解释性语言则直接由解释器解释运行程序.
[3]事实上评测系统的实现远比这个复杂，这里只是大概介绍了评测系统的评测过程.
[4]准确来说，一般是程序的用户态时间.
[5]这里的评测结果大多也适用于其他类型题目.
[6]大多数评测系统会将 PE 状态归到 WA 状态当中.
[7]一些测试点可能会有部分分, 选手在完成一个测试点的部分任务, 或者选手的输出正确但不够优的情况下, 可以获得一定比例的分数.

## 交互题

**交互题** 是需要选手程序与测评程序交互来完成任务的题目. 一类常见的情形是, 选手程序向测评程序发出询问, 并得到其反馈. 测评程序可能对选手的询问作出限制, 或调整应答策略来尽可能增加询问次数, 这也给题目带来了更多变化.

更详细的交互题讲解可以看 交互题→2.3.2@p80.

交互方式主要有如下两种. 虽然技术上有不小的差异, 但在考察算法的本质上它们并没有实际区别.

### STDIO 交互

STDIO 交互（标准 I/O 交互）是 Codeforces、AtCoder 等在线平台的交互手段, 也是 ICPC 系列赛事中的标准. Codeforces 提供了一个更加简要的 说明（英文）[1].

**例题 LOJ #559.「LibreOJ Round #9」ZQC 的迷宫**[2]

请注意最下方添加内容.

本题是一道交互题.

位于 $n \times m$ 个方格组成的黑暗迷宫的你, 需要走到这个迷宫的终点, 以完成迷宫挑战.

最开始, 你位于迷宫的起点即 $(1, 1)$ 处, 且面向右侧, 终点位于 $(n, m)$ 处. 迷宫中任意两个方格之间均连通, 且仅有唯一的一条路径, 两个相邻（即上、下、左、右四连通）方格间长度为一个单位长度. 两个相邻方格之间可能会有墙壁, 墙壁厚度相对于方格而言非常小, 粗略不计. 迷宫的边界均有墙壁, 且每一堵墙壁均与边界连通. 迷宫是完全黑暗的, 这意味着, 你无法得到除 $(n, m)$ 以外的任何信息.

为了在黑暗条件下尽量不迷路, 每次前进时你只能从当前格子出发, 沿着左侧或右侧墙壁, 左手或右手扶着墙壁前进, 并且使扶着墙壁的手移动距离恰好为一个单位长度. 需要注意的是, 若左侧或右侧墙壁不存在, 则沿该侧方向无法前进.

在黑暗中过久的你会感到恐惧, 因此你需要在你尽早走出迷宫. 如果你没有在限定步数内走出迷宫, 挑战将会失败.

对于这类题目, 选手只需像往常一样将询问写到标准输出, **刷新输出缓冲** 后从标准输入读取结果. 选手程序刷新输出缓冲后, 通过管道连接它的测评程序（称为交互器）才能立刻接收到这些数据. 在 C/C++ 中, `fflush(stdout)` 和 `std::cout << std::flush` 可以实现这个操作（使用 `std::cout << std::endl` 换行时也会自动刷新缓冲区, 但是 `std::cout << '\n'` 不会）; Pascal 则是 `flush(output)`.

### Grader 交互

Grader 交互方式常见于 IOI、APIO 等国际 OI 赛事（特别是 CMS 平台的竞赛）.

**例题 UOJ #206.【APIO2016】Gap**[3]

有 $N$ 个严格递增的非负整数 $a_1, a_2, \cdots, a_N (0 \le a_1 < a2 < \cdots < a_N \le 10^{18})$. 你需要找出 $a_{i+1} - a_i (0 \le i \le N-1)$ 里的最大的值.

你的程序不能直接读入这个整数序列, 但是你可以通过给定的函数来查询该序列的信息. 关于查询函数的细节, 请根据你所使用的语言, 参考下面的实现细节部分.

你需要实现一个函数, 该函数返回 $a_{i+1} - a_i (0 \le i \le N-1)$ 中的最大值.

对于这类题目, 选手只需编写一个特定的函数完成某项任务, 它通过调用给定的若干辅助函数来进行交互. 为了便于选手在本地测试, 题目会下发一个头文件与一个参考测评程序 `grader.cpp`（对于 Pascal 语言是一个库 `graderlib`）, 选手将自己的程序与 `grader.cpp` 一同编译方可得到可执行文件.

```
g++ grader.cpp my_solution.cpp -o my_solution -Wall -O2
./my_solution   # 执行程序
```

编译得到的程序表现与传统题程序类似. 它会打开固定的文件, 以固定的格式读取数据, 调用选手编写的函数, 并将结果和若干信息（例如询问的次数、答案正确性）显示在标准输出上.

实际测评时, 选手的程序会与一个不同的 `grader.cpp` 编译. 这个 `grader.cpp` 将以类似的方式调用选手编写的函数, 并记录其得分. 一般来说, 这个版本的 `grader.cpp` 所有全局符号都会设为 `static`, 也即不能通过冲突命名的方式破解它, 但是任何尝试突破 grader 限制的行为都会被判失格 (disqualification).

### 差别

STDIO 交互的一个明显优势在于它可以支持任何编程语言, 但是输入输出的耗时容易成为问题设计的瓶颈, 导致有时无法区分程序的时间效率差别; Grader 交互则恰好相反, 由于函数调用的开销不大, 常常可以允许 $10^6$ 数量级的询问次数, 但是语言的限制是其短板.

如果自己设计题目或举办比赛, 需要对二者认真权衡和比较.

### 通信题

**通信题** 是需要两个选手程序进行通信, 合作完成某项任务的题目. 第一个程序接收问题的输入, 并产生某些输出; 第二个程序的输入会与第一个的输出相关（有时是原封不动地作为一个参数, 有时会由评测端处理得到）, 它需要产生问题的解.

通信题的例子有: UOJ #178. 新年的贺电[4], #454.【UER #8】打雪仗[5] 等.

本地测试的方法由于题目设定的不同而多种多样, 常用的形式如:

- 手工输入
- 编写一个辅助程序, 转换第一个程序的输出到第二个程序的输入
- 用双向管道将两个程序的标准输入/输出连接起来

由于评测平台对于通信题的支持有限, 因而目前为止, 通信题只常见于 IOI 系列赛和 UOJ 等少数在线平台举办的比赛. 它仍是一个有待探索的领域.

### 函数补全题

**函数补全题** 是需要选手补全程序的题目. 可以理解为在一道交互题中, 题目给定了选手代码, 要求编写辅助函数.

通常有以下几种形式:

- 给定一个程序, 并告知要求补全的代码块将被嵌入在哪里.
- 不给出程序, 而将输入信息作为待提交函数的参数.

这种题在 LeetCode[6] 和 PTA - 拼题 A[7] 上比较多见.

### 其他类型

**例题 Quine[8]**

写一个程序, 使其能输出自己的源代码.

代码中必须至少包含十个可见字符.

题目很经典, 但是在绝大多数 OJ 上都很难实现.

**参考代码**

**注意**: 源代码不包含下方第一行（即 `// clang-format off`）.

```cpp
// clang-format off
#include<cstdio>

char *s={"#include<cstdio>%cchar *s={%c%s%c};%cint main()
{printf(s,10,34,s,34,10);return 0;}"};

int main(){printf(s,10,34,s,34,10);return 0;}
```

### 参考资料与注释

[1] 说明（英文）

[2] LOJ #559.「LibreOJ Round #9」ZQC 的迷宫

[3] UOJ #206.【APIO2016】Gap

[4] UOJ #178. 新年的贺电

[5] #454.【UER #8】打雪仗

[6] LeetCode

[7] PTA - 拼题 A

[8] Quine

## 2.3.2 交互题

**Authors:** countercurrent-time, StudyingFather

上个世纪的 IOI 就已涉及交互题．虽然交互题近年来没有在省选以下的比赛中出现，不过 2019 年里 NOI 系列比赛中连续出现《P5208[WC2019]I 君的商店》、《P5473[NOI2019]I 君的探险》两道交互题，这可能代表着交互题重新回到 NOI 系列比赛中．

交互题没有很高的前置算法要求，一般也没有严格的时间限制，程序的优秀程度往往仅取决于交互次数限制．所以学习交互题时，建议按照难度循序渐进．要是有意锻炼算法思维而不只是单纯地学习算法，那么完成交互题是很不错的方法．虽然交互题对选手已掌握算法的要求通常较低，但仍建议掌握一定提高和省选算法后再尝试做交互题，因为此时自己的算法思维水平和知识面已经达到了一定水平．基础的交互题介绍可以参考 OI Wiki 的 题型介绍 - 交互题→2.3.1@p76．

交互题的特殊错误：

- 选手每一次输出后都需要刷新缓冲区，否则会引起 Idleness limit exceeded 错误．另外，如果题目含多组数据并且程序可以在未读入所有数据前就知道答案，也仍然要读入所有数据，否则同样会因为读入混乱引起 ILE（可以一次提出多次询问，一次接收所有询问的回答）．同时尽量不要使用快读．
- 如果程序查询次数过多，则在 Codeforces 上会给出 Wrong Answer 的评测结果（不过评测系统会说明 Wrong Answer 的原因），而 UVa 会给出 Protocol Limit Exceeded (PLE) 的评测结果．
- 如果程序交互格式错误，UVa 会给出 Protocol Violation (PV) 的评测结果．

由于交互题输入输出较为繁琐，所以建议分别封装输入和输出函数．

比赛时如果出题人给出了 grader 头文件（用于 grader 交互题的调试）或者 checker 程序（用于 stdio 交互题的调试），则交互题的调试比较简单，因为交互题的对拍会比普通题目的对拍困难很多．没有 `testlib.h` 的情况下．交互细节较多的题目的 stdio 交互库会一般有 3k 代码量，再加上 3k 长度的对拍器，至少需要一小时实现．但是，无论是否有调试程序，调试交互题的代码都往往需要选手模拟与程序的交互过程，因此交互题需要选手能设计出高质量的程序，尽量保证一遍做对，同时拥有较强的静态查错能力．

例题：

- CF679A Bear and Prime 100[1]
- CF843B Interactive LowerBound[2]
- UOJ206[APIO2016]Gap[3]

- CF750F New Year and Finding Roots[4]
- UVa12731 太空站之谜 Mysterious Space Station[5]

### CF679A Bear and Prime 100

每个质数都有且只有两个因数，所以直接枚举要猜的数的因数．由于限制最多询问 20 次，并且对于较大的数（如 92）尝试分解质因数时发现需要最多枚举到 $\lfloor \frac{n}{2} \rfloor$ 的质数．所以我们先筛出 50 以内的质数，每次把所有这些数都询问一遍．

由于本题对拍比较容易，可以直接把值域内的数都尝试一遍．我们会发现程序无法有效处理质数的平方．所以我们要把 2,3,5,7 的平方 4,9,25,49 都放进去，总共 19 个数字，符合题意．

**参考代码**

```cpp
#include <cstdio>
constexpr int prime[] = {2,  3,  4,  5,  7,  9,  11, 13, 17, 19,
                         23, 25, 29, 31, 37, 41, 43, 47, 49};
int cnt = 0;
char res[5];

int main() {
  for (int i : prime) {
    printf("%d\n", i);
    fflush(stdout);
    scanf("%s", res);
    if (res[0] == 'y' && ++cnt == 2) return printf("composite"), 0;
  }
  printf("prime");
  return 0;
}
```

### CF843B Interactive LowerBound

链表最多有 $5 \times 10^4$ 个元素，但我们只能询问 1999 次，并且只能获取元素的后一个元素，所以普通的遍历整个链表的方法不可用．直接设法逼近目标元素的位置只有一种方法：随机撒点．

对于 $n < 2000$ 的情况直接枚举，$n \geq 2000$ 时，我们直接撒 1000 个点，这时这些点之间的期望距离很小，我们可以直接从小于 $x$ 的最大值开始向后遍历，可以证明在到达下一个点之前我们就已得到答案．遍历的过程中一旦找到大于等于 $x$ 的元素，就可以直接推出．

虽然整体思路简单，但实际情况下，如果没有学习过模拟退火等非完美随机算法，思考起来很可能会困难一些．

同时由于 Codeforces 具有 hack 机制，很多人会刻意卡掉没有初始化随机种子的代码，所以在 `random_shuffle()` 函数前需要 `srand((size_t)new char)`．

**参考代码**

```cpp
#include <algorithm>
#include <cstdio>
#include <cstdlib>
constexpr int N = 50005;
int n, start, x;
int a[N];
```

```cpp

int main() {
  scanf("%d%d%d", &n, &start, &x);
  if (n < 2000) {
    int ans = 2e9;
    for (int i = 1; i <= n; i++) {
      printf("? %d\n", i), fflush(stdout);
      int val, next;
      scanf("%d%d", &val, &next);
      if (val >= x) ans = std::min(ans, val);
    }
    if (ans == 2e9) ans = -1;
    printf("! %d", ans), fflush(stdout);
  } else {
    srand((size_t) new char);
    int p = start, ans = 0;
    for (int i = 1; i <= n; i++) a[i] = i;
    std::random_shuffle(a + 1, a + n + 1);
    for (int i = 1; i <= 1000; i++) {
      printf("? %d\n", a[i]), fflush(stdout);
      int val, next;
      scanf("%d%d", &val, &next);
      if (val < x && val > ans) p = a[i], ans = val;
    }
    while (p != -1 && ans < x) {
      printf("? %d\n", p), fflush(stdout);
      int val, next;
      scanf("%d%d", &val, &next);
      ans = val;
      p = next;
    }
    if (ans < x) ans = -1;
    printf("! %d", ans), fflush(stdout);
  }
  return 0;
}
```

### UOJ206[APIO2016]Gap

分两个子任务讨论：

1. 查询次数限制.

   我们考虑第一次查询. 因为我们一开始不知道任何数，所以我们需要询问范围 $[1, 10^{18}]$，获得最大最小值.

   由于查询次数限制刚好为 $\frac{N+1}{2}$, 所以考虑怎么每一次都能获取之前没有获取过的值, 这样能大概在次数范围内获取序列内的所有数. 方法也很简单：每次查询 $[s, t]$ 后，设获得的值为 $mn, mx$，则下一次查询 $[mn + 1, mx - 1]$.

2. 询问区间大小限制.

   由于题目要求询问区间内的数的数量之和不能超过 $3N$，所以考虑最小化询问区间. 上面的方法不再可用，因为其询问区间内的数数量之和规模为 $O(N^2)$. 我们可以考虑二分值域，但这种方法并

不可靠，最坏可能会被卡到 $O(N^2)$．所以我们需要更有效的划分值域的方法，避免查询区间内的点重复查询，浪费机会．

考虑到答案不会小于 $\lfloor \frac{a_n-a_1}{N-1} \rfloor$，所以我们可以考虑按这个值划分值域，设 $i$ 初始为 0，$ans$ 初始为上述值，每次询问 $[i, i+ans]$ 并且更新 $ans$，之后再以 $ans$ 为步长让 $i$ 自增．

不过这种方法也不能很好地适用于子任务 1，因为最坏可能很多询问的值域内一个数都没有．

**参考代码**

```cpp
#include <algorithm>
#include <cstdio>

#include "gap.h"

long long findGap(int T, int N) {
  static long long a[100005] = {}, ans = 0;
  long long s = 0, t = 1e18, s1, t1;
  if (T == 1) {
    int l = 1, r = N;
    while (l <= r) {
      MinMax(s, t, &s1, &t1);
      a[l++] = s1, a[r--] = t1;
      s = s1 + 1, t = t1 - 1;
    }
    for (int i = 2; i <= N; i++) ans = std::max(ans, a[i] - a[i - 1]);
  } else if (T == 2) {
    MinMax(s, t, &s1, &t1);
    ans = (t1 - s1) / (N - 1);
    long long l = s1 + 1, r = t1, last = s1;
    for (long long i = l; i <= r;) {
      MinMax(i, i + ans, &s1, &t1);
      i += ans + 1;
      if (s1 != -1) ans = std::max(ans, s1 - last), last = t1;
    }
  }
  return ans;
}
```

### CF750F New Year and Finding Roots

看到 $h \leq 7$，询问次数 $\leq 16$ 的严格要求，我们需要非常严格地最大化利用访问获得的信息．

$h \leq 4$ 时可以直接暴力枚举．然而 $h > 4$ 时需要很高效的遍历算法．

随机撒点不是好方法，因为随机撒点无法确定自己是否足够接近根节点了，并且单纯随机撒点，至少有一次碰到根节点的概率为 $1-(\frac{2^h-2}{2^h-1})$，即使排除重复撒点的情况后，碰到根节点的概率仍然非常小．

由于 $1 \leq k \leq 3$，并且我们并不知道哪一边更接近根节点，所以我们考虑最坏的情况，即如果 $k=3$ 时，前两次我们的遍历方向都是远离根节点的，第三次遍历方向是接近根节点的．所以我们必须往三个方向都遍历．

考虑 bfs 和 dfs 两种遍历方法．由于 bfs 搜索树可能很大，所以我们优先考虑 dfs．当然，如果我们知道当前的深度，并且当前深度小到深度范围内的搜索树规模小于等于剩余次数，我们就可以直接 bfs．

知道当前节点的深度，以及当前遍历的方向会获得很大优势. 然而知道当前在往根节点还是在往叶子节点遍历是非常困难的事情. 如果使用 dfs，只有当遍历到根节点（$k=2$）或者叶子节点（$k=1$）时才知道当前方向. 所以我们需要尽可能知道当前节点深度，并且不能采用类似迭代加深搜索的方法，遍历中途停下来.

考虑随机一个初始节点，从初始节点出发可能碰到上面的最坏情况.

如果 $k=1$，我们就可以直接知道当前节点的深度.

如果 $k=2$，那当前节点即根节点.

如果 $k=3$，我们直接考虑往三个方向 dfs. 考虑到其中两个方向是直接往叶子节点的方向，遍历路径长度相同；另一个方向是往根节点的方向，不过可能中途不小心往叶子节点的方向走了，遍历路径长度会较大. 此时我们就可以计算出当前节点的深度.

当 $k=1$ 或者 $k=3$ 时，我们需要考虑较长的遍历路径. 我们可以知道路径上深度最小的点（必定比初始节点深度小）. 如果我们为访问过的节点打标记，不再遍历，此时从该节点开始就只有一条遍历路径. 虽然这条路径可能还是会走向叶子节点，但是这条路径上同样必然存在深度比起点小的节点，我们就可以从这个节点开始继续重复上面的步骤.

当然，我们考虑 $h=7$ 的最坏情况时（每次只往根节点走一步，就直接往叶子节点走），会发现如果只 dfs，最坏需要 $\frac{(1+7)\times 7}{2}=28$ 次询问. 不过我们已经知道初始节点的深度，所以我们可以算出所有已遍历节点的深度，并且根据我们开始时对 bfs 的讨论，判断是否可以从深度最小的点直接 bfs.

这时，我们可以算出最坏需要 17 次. 所以我们考虑从搜索树上去掉一个节点（根据 dfs 只能盲目遍历的性质，我们考虑 bfs）：即当进行深度为 $k$ 的 bfs 时，搜索树节点最坏有 $2^k-1$ 个，可能需要 $2^k-1$ 次询问才能确定哪个节点的邻居恰有 2 个. 不过我们如果已经对其中 $2^k-2$ 个节点询问后，可以知道最后一个节点肯定是根节点.

此时最坏情况下的最优解为：$h=7$ 时，从叶子节点 dfs，每次都是只往根节点走一步，就直接往叶子节点走，询问 10 次后，当前已知最小深度的节点深度为 4，由于已知其父亲，直接从其父亲开始 bfs（搜索树深度为 3，节点数为 $2^3-1=7$）. 在 bfs 时询问了 $2^3-2=6$ 次后，确定 bfs 搜索树上最后一个节点为根节点.

此时我们的算法可以刚好卡到最坏 16 次.

**参考代码**

```cpp
#include <algorithm>
#include <cstdio>
#include <queue>
#include <vector>
using namespace std;
constexpr int N = 256 + 5;
int T, h, chance;
bool ok;
vector<int> to[N], path;

bool read(int x) {
  if (to[x].empty()) {
    printf("? %d\n", x), fflush(stdout);
    int k, t;
    scanf("%d", &k);
    if (k == 0) exit(0);
    for (int i = 0; i < k; i++) {
```

```
18        scanf("%d", &t);
19        to[x].push_back(t);
20      }
21      if (k == 2) {
22        printf("! %d\n", x), fflush(stdout);
23        return ok = true;
24      }
25      chance--;
26    }
27    return false;
28  }
29
30  bool dfs(int x) {
31    if (to[x].empty()) path.push_back(x);
32    if (read(x)) return true;
33    for (int i : to[x])
34      if (to[i].empty()) return dfs(i);
35    return false;
36  }
37
38  void bfs(int s, int k) {
39    queue<int> q;
40    for (int i : to[s])
41      if (to[i].empty()) q.push(i);
42    for (int i = 1; i < k; i++) {
43      int x = q.front();
44      q.pop();
45      if (read(x)) return;
46      for (int j : to[x])
47        if (to[j].empty()) q.push(j);
48    }
49    for (int i = 1; i < k; i++) {
50      int x = q.front();
51      q.pop();
52      if (read(x)) return;
53    }
54    printf("! %d\n", q.front()), fflush(stdout);
55  }
56
57  int main() {
58    for (scanf("%d", &T); T--;) {
59      ok = false;
60      for (int i = 0; i < N; i++) to[i].clear();
61      chance = 16;
62      scanf("%d", &h);
63      if (h == 0) exit(0);
64      vector<int> long_path;
65      if (read(1)) continue;
66      int root, dep;
67      if (to[1].size() == 1)
68        root = 1, dep = h;
69      else {
```

```
70        for (int i : to[1]) {
71          path.clear();
72          if (dfs(i)) break;
73          if (path.size() > long_path.size()) swap(path, long_path);
74        }
75        if (ok) continue;
76        dep = h - (path.size() + long_path.size()) / 2;
77        root = long_path.at((long_path.size() - (h - dep)) - 1);
78      }
79      while ((1 << (dep - 1)) - 2 > chance) {
80        path.clear();
81        if (dfs(root)) break;
82        dep = h - (h - dep + path.size()) / 2;
83        root = path.at((path.size() - (h - dep)) - 1);
84      }
85      if (!ok) bfs(root, 1 << (dep - 2));
86    }
87    return 0;
88  }
```

### UVa12731 太空站之谜 Mysterious Space Station

由于唯一的反馈是移动时是否撞墙，所以我们应该考虑在机器人不走丢的情况下，尽量接近墙边走路，这样有几个好处：

- 靠近墙边走路时，很容易知道自己会不会撞墙，获取到尽量多的信息.
- 墙边都是不会出现传送门的格子，可以避免机器人走丢.

所以，我们如果已知机器人可能在墙边的某个位置，要确定机器人是不是真的在这个位置，就可以通过「单手扶墙法」[6] 确定自己是不是真的在这个位置. 根据拓扑学原理，在两边都是墙的迷宫中，如果从入口进入，并且总是用一只手扶着同一边墙，就可以保证找到出口. 由于本题中的墙是闭合的，所以只需要沿着墙边的道路走，就可以保证可以回到原点而不会撞墙. 另外，由于墙边的道路是地图上的最大闭合回路，所以实际代码中并不需要特意撞墙以保证机器人在墙边，可以使用标记在地图中标明墙边道路. 而且一旦撞了墙，就需要赶快沿着原路返回，可以在避免机器人走丢的同时减少步数.

由上，可以推断出确定机器人是否在特定格子的试错法：将机器人在不走到未知格子或已知传送门的情况下走到墙边的道路上，然后绕着墙边道路走一圈. 这个过程中如果没有撞墙，就可以确定机器人确实是在特定格子.

我们可以采用上面的方法，一开始标出图中所有未知格子，然后从上到下，从左到右依次判断每个未知格子是否是传送门. 可以先走到未知格子上方，然后向下、向左走. 再用上面的方法判断机器人是不是在未知格子的左侧. 如果不是，说明机器人不在应该在的位置，即未知格子是传送门.

找出未知格子后就需要判断 2k 个未知格子的配对关系，实际方法也很简单：只需要暴力配对就可以了. 由于 $k \leq 5$，所以最多只需要 $9+7+5+3$ 次试错法. 作为对比，判断图中全部未知格子的情况最多需要 $121-40$ 次试错法.

由于目前下面这一份代码只能通过 UOJ 的镜像题：#247.【Rujia Liu's Present 7】Mysterious Space Station [7]，而无法通过 UVa 原题. 修改了 UOJ 上刘汝佳的标程后还是无法通过，并且暂时无法联系到刘汝佳. 所以下面的代码以 UOJ 为准.

不过刘汝佳的标程质量还是比下面这份代码质量高很多的，可以在 UOJ 上查看到 通过了 UOJ 镜像题的标程[8]. 同一份数据下，标程使用的移动次数非常少.

**参考代码**

```cpp
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <iostream>
#include <queue>
#include <stack>

#define Wall 0
#define Unknown 1
#define Space 2
#define Gate 3
#define Path 4

const int N = 20;
const int dir[8][2] = {{0, 1},  {1, 0}, {0, -1}, {-1, 0},
                       {-1, 1}, {1, 1}, {1, -1}, {-1, -1}};
const char dirs[5] = "ESWN";
int n, m, k;
int a[N][N], id[N][N];

struct point {
  int x, y;

  point(int x = 0, int y = 0) : x(x), y(y) {}

  bool operator==(const point& tmp) const { return x == tmp.x && y == tmp.y; }

  bool operator!=(const point& tmp) const { return !(*this == tmp); }

  point side(int d) const { return point(x + dir[d][0], y + dir[d][1]); }

  int check(int d) { return a[x + dir[d][0]][y + dir[d][1]]; }

  int id() { return ::id[x][y]; }
} start;

std::vector<std::pair<point, int>> path;
std::pair<point, point> ans[N];
std::pair<point, bool> vis[N];

bool walk(int d) {
  printf("MoveRobot %c\n", dirs[d]);
  fflush(stdout);
  int ret;
  scanf("%d", &ret);
  return ret;
}

bool walk(int d, std::stack<int>& st) {
  if (walk(d)) {
```

```
51      st.push(d);
52      return true;
53    }
54    return false;
55  }
56
57  bool read() {
58    if (scanf("%d%d%d", &n, &m, &k) != 3) return false;
59    if (n == 0) return false;
60    memset(a, 0, sizeof(a));
61    for (int i = 0; i < n; i++)
62      for (int j = 0; j < m; j++) {
63        char c;
64        std::cin >> c;
65        if (c == 'S') start = point(i, j);
66        if (c == '*')
67          a[i][j] = Wall;
68        else
69          a[i][j] = Unknown;
70      }
71    return true;
72  }
73
74  void answer() {
75    for (int i = 0; i < k; i++)
76      printf("Answer %d %d\n", ans[i].first.id(), ans[i].second.id());
77    fflush(stdout);
78  }
79
80  // 单手扶墙法，因为靠墙的 Path 是极大闭合环，所以只需要在沿着 Path
81  // 走的过程中没有碰到障碍就可以了
82  void wall_follower_init(point x, int last, int wallside, point s) {
83    if (x == s && !path.empty()) return;
84    if (x.check(wallside) == Path) {
85      path.push_back(std::make_pair(x, wallside));
86      wall_follower_init(x.side(wallside), wallside, last ^ 2, s);
87    } else if (x.check(last) == Wall) {
88      for (int i = 0; i < 4; i++)
89        if (i != (last ^ 2) && x.check(i) != Wall) {
90          path.push_back(std::make_pair(x, i));
91          wall_follower_init(x.side(i), i, last, s);
92          return;
93        }
94    } else {
95      path.push_back(std::make_pair(x, last));
96      wall_follower_init(x.side(last), last, wallside, s);
97    }
98  }
99
100 void init() {
101   int cnt = 1;
102   for (int i = 0; i < n; i++)
```

```
    for (int j = 0; j < m; j++) {
      if (a[i][j] == Unknown) {
        id[i][j] = cnt++;
        for (int k = 0; k < 8; k++)
          if (point(i, j).check(k) == Wall) {
            a[i][j] = Path;
            break;
          }
      } else
        id[i][j] = 0;
    }
  path.clear();
  int wallside = 0, last = 0;
  for (int i = 0; i < 4; i++)
    if (start.check(i) == Wall) {
      wallside = i;
      break;
    }
  for (int i = 0; i < 4; i++)
    if (start.check(i) == Path && i != (wallside ^ 2)) {
      last = i;
      break;
    }
  wall_follower_init(start, last, wallside, start);
}

void undo(std::stack<int>& st) {
  while (!st.empty()) walk(st.top() ^ 2), st.pop();
}

bool wall_follower(point x) {
  std::stack<int> st;
  bool ok = true;
  int i = 0;
  while (i < path.size() && path[i].first != x) i++;
  for (int j = i; ok && j < path.size(); j++) {
    if (walk(path[j].second))
      st.push(path[j].second);
    else
      ok = false;
  }
  for (int j = 0; ok && j < i; j++) {
    if (walk(path[j].second))
      st.push(path[j].second);
    else
      ok = false;
  }
  if (!ok) undo(st);
  return ok;
}

// 确定自己当前在
```

```cpp
// x，使用「摸着石头过河」的方法，只需要沿着可以避开障碍、未知格子和传送门的方向走到
// Path 就行.  在找传送门和配对传送门时使用
void bfs(point s, point t, std::vector<int>& v) {
  static int map[N][N] = {};
  memset(map, -1, sizeof(map));
  std::queue<point> q;
  map[s.x][s.y] = 4;
  q.push(s);
  while (!q.empty()) {
    point x = q.front();
    q.pop();
    if (x == t) break;
    for (int i = 0; i < 4; i++) {
      point y = x.side(i);
      if ((x.check(i) == Path || x.check(i) == Space) && map[y.x][y.y] == -1) {
        map[y.x][y.y] = i;
        q.push(y);
      }
    }
  }
  for (point x = t; x != s; x = x.side(map[x.x][x.y] ^ 2)) {
    v.push_back(map[x.x][x.y]);
  }
  std::reverse(v.begin(), v.end());
}

bool move(point s, point t, std::stack<int>& st) {  // 在靠近传送门时使用
  static std::vector<int> v;
  v.clear();
  bfs(s, t, v);
  for (int i : v)
    if (!walk(i, st)) return false;
  return true;
}

// 尽可能快地向墙边移动
bool make_sure(point x, int last) {
  if (a[x.x][x.y] == Path) return wall_follower(x);
  for (int i = 0; i < 4; i++)
    if ((x.check(i) == Path || x.check(i) == Space) && i != (last ^ 2)) {
      if (!walk(i)) return false;
      bool ret = make_sure(x.side(i), i);
      walk(i ^ 2);
      return ret;
    }
  return false;
}

void find_gate() {
  int cnt = 0;
  std::stack<int> st;
```

```
206   for (int i = 0; i < n; i++)
207     for (int j = 0; j < m; j++)
208       if (cnt == k * 2 && a[i][j] == Unknown)
209         a[i][j] = Space;
210       else if (a[i][j] == Unknown) {
211         bool ok = true;
212         if (!move(start, point(i - 1, j), st))
213           ok = false;
214         else if (!walk(1, st))
215           ok = false;
216         else if (!walk(2, st))
217           ok = false;
218         else if (!make_sure(point(i, j - 1), -1))
219           ok = false;
220         if (!ok) {
221           vis[cnt++] = std::make_pair(point(i, j), false);
222           a[i][j] = Gate;
223           for (int k = 0; k < 8; k++) {
224             point y = point(i, j).side(k);
225             if (point(i, j).check(k) == Unknown) a[y.x][y.y] = Space;
226           }
227         } else
228           a[i][j] = Space;
229         undo(st);
230       }
231 }
232
233 void make_gate_pair() {
234   int cnt = 0;
235   std::stack<int> st;
236   for (int i = 0; i < k * 2; i++)
237     if (!vis[i].second)
238       for (int j = 0; !vis[i].second && j < k * 2; j++)
239         if (j != i && !vis[j].second) {
240           bool ok = true;
241           if (!move(start, vis[i].first.side(2), st))
242             ok = false;
243           else if (!walk(0, st))
244             ok = false;
245           else if (!make_sure(vis[j].first.side(0), -1))
246             ok = false;
247           if (ok) {
248             ans[cnt++] = std::make_pair(vis[i].first, vis[j].first);
249             vis[i].second = vis[j].second = true;
250           }
251           undo(st);
252         }
253 }
254
255 int main() {
256   while (read()) {
257     init();
```

```
258     find_gate();
259     make_gate_pair();
260     answer();
261   }
262   return 0;
263 }
```

### 习题

- 刘汝佳的交互题专场比赛 Rujia Liu's Present 7 质量非常高，推荐一做. [9]
- P5473[NOI2019]I 君的探险[10]
- P5208[WC2019]I 君的商店[11]

### 参考资料与拓展阅读

- 用 Linux 管道实现 online judge 的交互题功能[12]

### 外部参考资料

[1] CF679A Bear and Prime 100

[2] CF843B Interactive LowerBound

[3] UOJ206[APIO2016]Gap

[4] CF750F New Year and Finding Roots

[5] UVa12731 太空站之谜 Mysterious Space Station

[6] 「单手扶墙法」

[7] #247.【Rujia Liu's Present 7】Mysterious Space Station

[8] 通过了 UOJ 镜像题的标程

[9] 刘汝佳的交互题专场比赛 Rujia Liu's Present 7 质量非常高，推荐一做.

[10] P5473[NOI2019]I 君的探险

[11] P5208[WC2019]I 君的商店

[12] 用 Linux 管道实现 online judge 的交互题功能

## 2.4 学习路线

**提示**

本文章正在编辑讨论中，欢迎补充更进一步的学习路线或在评论区提出你的想法！

本文将会介绍算法竞赛的学习路线.

该学习路线既是新手学习算法竞赛知识的指南，也是一份复习清单.

# 1 C++ 语言基础

先从 C++ 语法学起，一步一步来.

### 1.1 Hello, World!

以一句 `Hello, World!`，开始算法竞赛之旅吧！

同时了解一下 C++ 的源程序的大致框架是什么样子的.

- Hello, World!→4.2.1@p322
- C++ 语法基础→4.2.2@p323

### 1.2 变量与运算

计算机出现的最初目的就是计算. 因此我们先学习如何完成一些简单的运算任务吧.

- 变量→4.2.3@p328
- 运算→4.2.4@p339

### 1.3 流程控制

#### 1.3.1 分支结构

有的时候，我们需要在不同的条件下，选择执行不同的语句，这时候我们就需要借助分支语句.

- 分支→4.2.5.1@p345

分支语句包括下面几种:

- if 语句
- if-else 语句
- if-elif-else 语句
- switch 语句

#### 1.3.2 循环结构

将若干条语句重复执行多次，就需要用到循环语句.

- 循环→4.2.5.2@p348

循环语句包括下面几种:

- for 语句
- while 语句
- do-while 语句

### 1.4 数组与结构体

数组用于存储大量相同类型的数据. 而结构体则可以将若干变量捆绑起来.

- 数组→4.2.6.1@p352
- 结构体→4.2.6.2@p354

### 1.5 函数与递归

使用函数来让程序变得模块化，降低实现成本.

递归则是新手入门的一道坎，「自己调用自己」听起来并不是那么容易理解，不过仔细深究根本，就会发现「自己调用自己」和「自己调用别人」并没有本质差别.

- 函数→4.2.7@p363
- 递归 & 分治→5.5@p557

## 2 CSP-J 入门级

### 2.1 枚举与模拟

从现在开始，你已经会使用 C++ 语言完成一些简单的任务了，但是这远远不够.

为了做对一些简单的题目，你需要学会通过枚举或模拟脑海中的逻辑，来实现代码. 这看起来并不是很高效，但有的时候很管用.

- 枚举→5.3@p552
- 模拟→5.4@p555

### 2.2 递归与分治

递归是指函数定义中不断调用自己的方法; 而分治则是不断将这一个问题分解为若干子问题，求解后合并的操作.

- 递归 & 分治→5.5@p557

## 2.3 字符串

在做信息学题目时，经常会碰到的一个数据类型就是字符串，你需要学习一些用于操作字符串的 STL 函数．当然，模拟也是解决字符串问题的好方法．

- 字符串基础→8.2@p890
- STL 函数→8.3@p891

## 2.4 排序

当你获得了一组数据时，如何将他们从无序变成有序也是个很重要的问题．在你没有思路的时候，不妨考虑一下将数组排个序吧．这也是接下来的很多算法的基础．

排序的方法有点多，但理解后记住它们并不难．

- 排序简介→5.7.1@p567
- 选择排序→5.7.2@p568
- 冒泡排序→5.7.3@p571
- 插入排序→5.7.4@p572
- 计数排序→5.7.5@p574
- 基数排序→5.7.6@p578
- 快速排序→5.7.7@p585
- 归并排序→5.7.8@p592
- 堆排序→5.7.9@p596
- 桶排序→5.7.10@p598
- 排序相关 STL→5.7.14@p613

NOI 大纲中入门级只要求学习选择、冒泡、插入排序，共三个排序算法，但是其余的难度也并不大，且初赛中可能涉及，故一并列出．

## 2.5 二分与倍增

二分查找，本质上是运用分治的思想，不断减少查找范围的大小，直至找到答案．但是需要注意，这个查找方式必须应用在有序的数据结构中．

- 二分→5.9@p629

而倍增则不同，它是不断翻倍，以把线性范畴内的处理转化为对数级，大大优化时间复杂度．（这个知识点需要一点数学基础，暂时跳过也问题不大）

- 倍增→5.10@p645

## 2.6 搜索

在入门组，搜索的题目常常会在迷宫类题目中出现，一般会有地图类的数据；此外，搜索也十分常用于高效地枚举构造合法解的情况，亦可用于骗分．

### 2.6.1 深度优先搜索（DFS）

深度优先搜索指利用递归函数方便地实现暴力枚举的算法，与图论中的 DFS 算法有一定相似之处，但并不完全相同．

- DFS（搜索）→6.2@p654

### 2.6.2 广度优先搜索（BFS）

将每一个状态设计为图中的一个点，可以展开地毯式搜索．

- BFS（搜索）→6.3@p657

### 2.6.3 搜索优化

很多题目都可以用 DFS 来解决，而这个算法的复杂度显然是无法通过的．因此，需要一些优化使它跑得更快．这样的优化能够减少不可能成功的尝试，称为「剪枝」．BFS 相关的优化就要更加灵活了，但是基本思路和这里是一样的．

- DFS 剪枝优化→6.12@p708

## 2.7 数据结构入门

### 2.7.1 线性数据结构

数组，链表，队列，栈，都是线性结构．巧用这些结构可以做出不少方便的事情．

- 栈→10.2@p1978
- 队列→10.3@p1980
- 链表→10.4@p1988

### 2.7.2 复杂数据结构

- 树及二叉树→11.6.1@p2621
- 图的概念→11.2@p2598
- 图的存储→11.3@p2605

## 2.8 动态规划入门

动态规划（Dynamic Programming, DP）是一种通过把原问题分解为相对简单的子问题的方式求解复杂问题的方法．

由于动态规划并不是某种具体的算法，而是一种解决特定问题的方法，因此它会出现在各式各样的数据结构中，与之相关的题目种类也更为繁杂．

- 动态规划简介→7.1@p714

### 2.8.1 背包问题

即给出一个有限制容量的背包，选择放入若干有容量和价值的物品，求解如何放置能使得价值总和最大．这是阻挡很多 OIer 的第一道坎，从这里开始，算法就有些难以理解．

- 背包 DP→7.4.1@p724

### 2.8.2 线性动态规划

在动态规划中，最难的部分之一就是设计状态，需要用到构造相关技巧．当你写出了状态和状态转移方程之后，完成一道动态规划的题目就不难了．

- 构造→5.11@p648
- 动态规划基础→7.2@p714

记忆化搜索是一种通过记录已经遍历过的状态的信息，从而避免对同一状态重复遍历的搜索实现方式．有的题目也可以使用记忆化搜索来降低思维难度．

因为记忆化搜索确保了每个状态只访问一次，它也是一种常见的动态规划实现方式．

- 记忆化搜索→7.3@p720

### 2.8.3 复杂动态规划

区间类动态规划是线性动态规划的扩展，它在分阶段地划分问题时，与阶段中元素出现的顺序和由前一阶段的哪些元素合并而来有很大的关系．

- 区间 DP→7.5@p734

## 2.9 数学

### 2.9.1 高精度算法

就算是 long long（或 int64）还不够怎么办？用高精度算法．本质上就是模拟了四则运算．

- 高精度计算→9.6@p1110

### 2.9.2 进制转换

在计算机中，除了二进制，比较常用的还有八进制和十六进制．有的时候学会运用正确的进制对解题也有很大帮助．

- 进位制→9.3.2@p1087

### 2.9.3 位操作

位操作就是基于整数的二进制表示进行的运算．由于计算机内部就是以二进制来存储数据，位操作是相当快的．

基本的位操作共 6 种，分别为按位与、按位或、按位异或、按位取反、左移和右移．

- 位操作→9.4@p1099

### 2.9.4 数论

- 数论基础→9.11.1@p1160
- 素数→9.11.3@p1179
- 筛法→9.11.6@p1199
- 最大公因数→9.11.4@p1188
- 欧拉函数→9.11.5@p1196
- 分解质因数→9.11.7@p1209

### 2.9.5 组合计数

- 排列组合→9.13.1@p1612
- 抽屉原理→9.13.2@p1618
- 容斥原理→9.13.3@p1618

* * *

至此，你就学习完了入门组范畴内的所有算法，但是想要掌握它们，你需要继续进行足够数量的刷题，以巩固你所学到的知识点．

# 2.5 学习资源

> **Authors:** Suyun514, ChungZH, Enter-tainer, StudyingFather, Konano, JulieSigtuna, GldHkkowo, SukkaW, Rapiz1, Henry-ZHR, H-J-Granger, countercurrent-time, fouzhe, Ir1d, abc1763613206, EndlessCheng, Plaaant6, LUTLJS, ZsgsDesign, CB-X2-Jun, tallnutliu

本页面主要列举了一些与算法竞赛有关的在线评测网站、题目合集、书籍、工具等资源．

## 在线评测平台

在线评测平台（英语：Online Judging System，简称：OJ），一般用于刷题训练，参与和组织比赛，以及用户之间的交流分享．

### 国内

- 51Nod[1]：有许多值得尝试的数学题和思维题．
- Comet OJ[2]：始于 2018 年，旨在为广大算法爱好者提供一个竞技、练习、交流的平台，经常举办原创性的高质量比赛，有丰富的题库．

- HDU Online Judge[3] 始于 2005 年，杭州电子科技大学在线评测系统，有多校训练的题目.
- HydroOJ[4]：始于 2021 年，为开源项目 Hydro[5] 的官方站. 用户可以创建自己的 域[6]，域中可以使用题库、比赛、讨论等主站可以使用的功能.
- Judge Duck Online[7] 基于 松松松[8] 开发的开源项目 JudgeDuck[9]，可以将评测程序的运行时间精确到微秒.（题目较少）
- LibreOJ[10]：始于 2017 年. 基于开源项目 Lyrio[11]，Libre 取自由之意. 题目所有测试数据以及提交的代码均对所有用户开放. 目前由 Menci[12] 维护.
- CDOJ[13]：电子科技大学在线评测系统，始于 2012 年.
- 洛谷[14]：始于 2013 年，社区群体庞大，各类 OI 的真题和习题较全. 提供有偿教育服务.
- 牛客网[15]：始于 2014 年，提供技术类求职备考、社群交流、企业招聘等服务.
- OpenJudge[16]：始于 2005 年，由 POJ 团队开发的小组评测平台.
- POJ[17]：北京大学在线评测系统，始于 2003 年，国内历史最悠久的 OJ 之一. 内有很多英文题，既有基础题，也有值得一试的好题. 可以在 百练[18] 题库提交 POJ 的题目.
- PTA（拼题 A）[19]：始于 2016 年，浙江大学衍生的杭州百腾教育科技有限公司产品.
- QOJ[20]：收集了很多国内外 OI 和 ICPC 竞赛题目，具有训练价值.
- Universal Online Judge[21]：始于 2014 年，Universal 取通用之意，项目开源[22]；VFK[23] 的 OJ：多原创比赛题和 CCF/THU 题，难度较高.
- Vijos[24]：始于 2005 年. 服务端[25] 和 评测机[26] 等项目开源.
- WZOI[27]：始于 2017 年，由浙江省温州中学维护的 开源[28] 评测系统.
- ZOJ[29]：浙江大学在线评测系统，始于 2001 年.

## 国外

- AizuOJ[30]：日本会津大学在线评测系统，始于 2004 年. 包含日本若干高中和大学编程比赛的题目，自带编程/数据结构/算法的入门课程.
- AtCoder[31]：日本 OJ，日文版里会有日本高校的比赛，英文内不会显示. 题目有趣，质量较高.
- CodeChef[32]：印度 OJ，周期举办比赛. 系统基于 SPOJ 的 Sphere Engine.
- Codeforces[33]：俄罗斯 OJ，始于 2010 年，创始人是 Mike Mirzayanov[34]. 有多种系列的比赛，并支持个人出题、申请组织比赛. 题目质量较高.
- CSES[35](Code Submission Evaluation System)，按专题划分的题库，旨在[36] 成为综合的高质量题库，主要由 Competitive Programmer's Handbook[37] 作者 Antti Laaksonen 开发，始于 2013 年.
- CS Academy[38]
- DMOJ[39] 加拿大开源的 OJ，语言支持广；题库是各大比赛的存档，也有定期自行举办的比赛.
- HackerRank[40] 有很多比赛
- Kattis[41] 题库主要包含类似 ICPC 比赛的题目；根据用户解题情况评定用户等级，推荐适合该用户水平的 trivial/easy/medium/hard 四类难度的题目，其中题目难度采用类 ELO 等级分[42] 系统来评估.
- LeetCode[43] 码农面试刷题网站，有中文分站：LeetCode China[44].
- Light OJ[45]
- Open Trains[46] 俄罗斯 Open Cup 比赛的训练平台，基于 ejudge[47] 开源系统搭建，支持虚拟比赛；题库包含历年 Open Cup 赛题以及 Petrozavodsk 训练营的题目.
- SPOJ[48] 始于 2003 年，其后台系统 Sphere Engine[49] 于 2008 年商业化；支持题目点赞和标签功能.
- TopCoder[50] 始于 2001 年，其 竞技编程社区[51] 有很多比赛；目前主营业务是技术众包.
- TimusOJ[52] 始于 2000 年，由 Ural Federal University 开发，拥有俄罗斯最大的在线评测题库，题目主要来自乌拉尔联邦大学校赛、乌拉尔锦标赛、ICPC 乌拉尔区域赛、以及 Petrozavodsk 训练营.
- Online Judge（前 UVaOJ[53]）始于 1995 年，国际成名最早的 OJ，创始人是西班牙 University of Valladolid (UVa) 的 Miguel Ángel Revilla 教授；由于 Revilla 教授于 2018 年不幸离世[54]，且 Valladolid 大学终止

维护，UVaOJ 自 2019 年 7 月起更名为 Online Judge．现在该平台的维护者 正在 GitHub 上构建新的评测平台[55]．

- Yandex[56] 存档了近几年的全俄罗斯信息学奥赛．

## 教程资料

- OI Wiki[57]
- Codeforces 上网友整理的一份教程合集[58]
- 英文版 E-Maxx 算法教程[59]
- 演算法筆記[60]：台湾师范大学总结的教程
- 如何为 ACM-ICPC 做准备？- geeksforgeeks[61]
- Topcoder 整理的教程[62]
- 校招面试指南[63]
- 由 hzwer 收集整理自互联网的课件[64]
- Trinkle23897 的课件[65]
- huzecong 的课件[66]
- Open Data Structure[67]：内含众多数据结构讲稿
- IOI Syllabus (2020)[68]

## 书籍

本列表内注明了书籍作者，译者未列其中．因无重名书籍且易于寻找，故不标明 ISBN．

- 刘汝佳系列
  - 《算法竞赛入门经典》（紫）
    - 第一版 配套资源仓库（镜像）[69]
    - 第二版 配套资源仓库[70]
    - 第二版 习题选解[71]
  - 《算法竞赛入门经典 - 训练指南》（白/蓝）- 陈锋 合著
  - 《算法艺术与信息学竞赛》（蓝/黑）
- 《算法竞赛进阶指南》- 李煜东
  - 配套资源仓库[72]
- 《啊哈算法》- 纪磊
  - 面向初学者或有初步兴趣的人群，有幽默配图．
- CCF 中学生计算机程序设计系列
  - 《CCF 中学生计算机程序设计 - 入门篇》- 陈颖，邱桂香，朱全民
    - 建议配合勘误使用．[73]
  - 《CCF 中学生计算机程序设计 - 基础篇》- 江涛，宋新波，朱全民
  - 《CCF 中学生计算机程序设计 - 提高篇》- 徐先友，朱全民
  - 《CCF 中学生计算机程序设计 - 专业篇》（未出）
- 深入浅出系列
  - 《深入浅出程序设计竞赛 - 基础篇》- 洛谷网校教研组
- 一本通系列
  - 《信息学奥赛一本通》- 董永建
  - 《信息学奥赛一本通 - 提高篇》- 黄新军，董永建
    - 建议选择性阅读．[74]
  - 《信息学奥赛一本通 - 高手训练》- 黄新军，董永建
- 其他由国内著名 OI 教练写的教材
  - 《信息学奥赛课课通》- 林厚从

  - 《聪明人的游戏：信息学探秘 - 提高篇》- 江涛，陈茂贤
  - 《计算概论：C++ 编程与信息学竞赛入门》- 金靖
  - 《算法竞赛宝典》- 张新华
- ACM 国际大学生程序设计竞赛系列
  - 《ACM 国际大学生程序设计竞赛系列 知识与入门》- 俞勇
  - 《ACM 国际大学生程序设计竞赛系列 算法与实现》- 俞勇
  - 《ACM 国际大学生程序设计竞赛系列 题目与解读》- 俞勇
- 《算法竞赛入门到进阶》- 罗勇军，郭卫斌
- 《算法导论》第三版 - Thomas H.Cormen/Charles E.Leiserson/Ronald L.Rivest/Clifford Stein

  黑书，大学经典教材．英文版原名 *Introduction to Algorithms*
  - 答案解析 (English)[75]
- 《具体数学》第二版 - Ronald L. Graham/Donald E. Knuth/Oren Patashnik

  英文版原名 *Concrete Mathematics*
- 《组合数学》第五版 - Richard A.Brualdi

  英文版原名 *Introductory Combinatorics*
- 《挑战程序设计竞赛》全套 - 秋叶拓哉，岩田阳一，北川宜稔 通俗易懂．
- 《算法概论》- Sanjoy Dasgupta/Christos Papadimitriou/Umesh Vazirani
  - 提纲挈领，但内容较少．
- Legend-K 的数据结构与算法的笔记[76]
- acm-cheat-sheet[77]
- Competitive Programmer's Handbook[78]- Antti Laaksonen
  - 作者花了三年个人时间完成．面向算法竞赛，覆盖面广，详略得当．
- 《挑战编程：程序设计竞赛训练手册》[79]- Steven S. Skiena/Miguel A. Revilla
  - 由西班牙 University of Valladolid 的两位教授编写．
  - 阅读 经过翻译的在线电子版图书[80]
  - 购买 纸质版图书[81]
- 《C++，挑战编程——程序设计竞赛进阶训练指南》- 邱秋
  - 作者博客的介绍页[82]
- 《数据结构（C++ 语言版 第 3 版）》- 邓俊辉[83]
  - 建议随配套课程、配套课件和习题解析一起使用．
- 《计算几何：算法与应用》- 伯格（Berg,M.D.）著，邓俊辉 译

  英文版原名 *Computational Geometry: Algorithms and Applications*
- 《Handbook of Data Structures and Applications, 2nd Edition》[84]
  - 由许多著名教授如 Sartaj Sahni、Hanan Samet、Weiss 等合著，内容较多，建议有一定基础的数据结构爱好者阅读．
- 算法详解 系列[85]
  - 面向有语言基础的初学者的教材，建议同配套课程一起使用
  - 《Algorithms Illuminated, Part 1: The Basics》- Tim Roughgarden
  - 《算法详解，卷 1：算法基础》- 徐波 译
  - 《Algorithms Illuminated, Part 2: Graph Algorithms and Data Structures》- Tim Roughgarden
  - 《算法详解，卷 2：图算法和数据结构》- 徐波 译
  - 《Algorithms Illuminated, Part 3: Greedy Algorithms and Dynamic Programming》- Tim Roughgarden
  - 《Algorithms Illuminated, Part 4: Algorithms for NP-Hard Problems》- Tim Roughgarden

## 课程

- CMU 15-295 (2025)[86]
- LSU: CSC 2700 (2024)[87]
- NUS: CS 3233 (2021)[88]
- Reykjavik: T-414-ÁFLV (2016)[89]
- Stanford: CS 97SI (2015)[90]
- Stonybrook: CSE 392 (2012)[91]
- UBC: CPSC 490 (2021)[92]
- UCF: COP 4516 (2025)[93]
- THU: 数据结构[94]
- THU: 计算几何[95]
- StanfordOnline: Algorithms: Design and Analysis[96]

## 工具

- 《100 个 gdb 小技巧》[97]
- Algorithm Visualizer[98]
- cppreference[99]：一个全面的 C 和 C++ 语言及其标准库的在线参考资料
- Compiler Explorer[100]：在线查看编译后代码块对应的汇编语句，支持选择不同的编译器
- C++ Insights[101]：以编译器的视角去查看你的 C++ 源码
- Inverse Symbolic Calculator[102]：实数反查表达式，适用于反推常数
- LATEX 手写符号识别[103]
- LATEX 数学公式参考[104]
- Mathpix[105]：截图转 LATEX
- OEIS[106]：整数数列搜索引擎
- Python Tutor[107]: 代码执行过程可视化
- Quick C++ Benchmark[108]：在线比较两个及以上函数的运行速度
- Try It Online[109]：在线运行 600+ 种语言的代码，支持 IO 交互，超时 60s，可以分享代码
- 图论画板[110] 与 GraphViz[111]
- uDebug[112]：提供一些 OJ 题目的调试辅助
- USF[113] 与 VisuAlgo[114]：算法可视化
- Wandbox[115]: 在线代码运行，支持 30+ 种语言，可以分享代码，支持不同编译器版本
- Wolfram Alpha[116]：可以计算包括数学、科学技术、社会文化……等多个主题的问题

## 题集和资源

- POJ 训练计划[117]
- USACO[118]
- 洛谷题单[119]
- -Morass- 贴在 Codeforces 上的一份题单[120]
- Codeforces 社区高质量算法文章合集 之一[121] 之二[122]
- 北京大学 ICPC 暑期课课件例题[123]
- 北京大学 ICPC 暑期课课件[124]
- GitHub.com:OI-wiki/libs[125]
- 多校联合训练[126] 关键词：`Multi-University Training Contest`
- Vjudge[127]
- Project Euler[128]
- Junior Training Sheet[129]：对新手友好的训练计划

- USACO Guide[130]：针对 USACO 的各个级别分类的训练资源

## 外部参考资料

[1] 51Nod

[2] Comet OJ

[3] HDU Online Judge

[4] HydroOJ

[5] Hydro

[6] 域

[7] Judge Duck Online

[8] 松松松

[9] JudgeDuck

[10] LibreOJ

[11] Lyrio

[12] Menci

[13] CDOJ

[14] 洛谷

[15] 牛客网

[16] OpenJudge

[17] POJ

[18] 百练

[19] PTA（拼题 A）

[20] QOJ

[21] Universal Online Judge

[22] 项目开源

[23] VFK

[24] Vijos

[25] 服务端

[26] 评测机

[27] WZOI

[28] 开源

[29] ZOJ

[30] AizuOJ

[31] AtCoder

[32] CodeChef

[33] Codeforces

[34] Mike Mirzayanov

[35] CSES

[36] 旨在

[37] Competitive Programmer's Handbook

[38] CS Academy

[39] DMOJ

[40] HackerRank

[41] Kattis

[42] ELO 等级分

[43] LeetCode

[44] LeetCode China

[45] Light OJ

[46] Open Trains

[47] ejudge

[48] SPOJ

[49] Sphere Engine

[50] TopCoder

[51] 竞技编程社区

[52] TimusOJ

[53] UVaOJ

[54] Revilla 教授于 2018 年不幸离世

[55] 正在 GitHub 上构建新的评测平台

[56] Yandex

[57] **OI Wiki**

[58] Codeforces 上网友整理的一份教程合集

[59] 英文版 E-Maxx 算法教程

[60] 演算法筆記

[61] 如何为 ACM-ICPC 做准备？ - geeksforgeeks

[62] Topcoder 整理的教程

[63] 校招面试指南

[64] 由 hzwer 收集整理自互联网的课件

[65] Trinkle23897 的课件

[66] huzecong 的课件

[67] Open Data Structure

[68] IOI Syllabus (2020)

[69] 第一版 配套资源仓库（镜像）

[70] 第二版 配套资源仓库

[71] 第二版 习题选解

[72] 配套资源仓库

[73] 建议配合勘误使用.

[74] 建议选择性阅读.

[75] 答案解析 (English)

[76] Legend-K 的数据结构与算法的笔记

[77] acm-cheat-sheet

[78] Competitive Programmer's Handbook

[79]《挑战编程：程序设计竞赛训练手册》

[80] 经过翻译的在线电子版图书

[81] 纸质版图书

[82] 作者博客的介绍页

[83]《数据结构（C++ 语言版第 3 版）》- 邓俊辉

[84]《Handbook of Data Structures and Applications, 2nd Edition》

[85] 算法详解 系列

[86] CMU 15-295 (2025)

[87] LSU: CSC 2700 (2024)

[88] NUS: CS 3233 (2021)

[89] Reykjavik: T-414-ÁFLV (2016)

[90] Stanford: CS 97SI (2015)

[91] Stonybrook: CSE 392 (2012)

[92] UBC: CPSC 490 (2021)

[93] UCF: COP 4516 (2025)

[94] THU: 数据结构

[95] THU: 计算几何

[96] StanfordOnline: Algorithms: Design and Analysis

[97]《100 个 gdb 小技巧》

[98] Algorithm Visualizer

[99] cppreference

[100] Compiler Explorer

[101] C++ Insights

[102] Inverse Symbolic Calculator

[103] LATEX 手写符号识别

[104] LATEX 数学公式参考

[105] Mathpix

[106] OEIS

[107] Python Tutor

[108] Quick C++ Benchmark

[109] Try It Online

[110] 图论画板

[111] GraphViz

[112] uDebug

[113] USF

[114] VisuAlgo

[115] Wandbox

[116] Wolfram Alpha

[117] POJ 训练计划

[118] USACO 

[119] 洛谷题单 

[120] -Morass- 贴在 Codeforces 上的一份题单 

[121] 之一 

[122] 之二 

[123] 北京大学 ICPC 暑期课课件例题 

[124] 北京大学 ICPC 暑期课课件 

[125] GitHub.com:OI-wiki/libs 

[126] 多校联合训练 

[127] Vjudge 

[128] Project Euler 

[129] Junior Training Sheet 

[130] USACO Guide 

# 2.6 技巧

## 2.6.1 读入、输出优化

> **Authors:** Marcythm, YZircon, Chaigidel, Tiger3018, voidge, H-J-Granger, ouuan, Enter-tainer, lcfsih, Xeonacid, Ir1d

本文将介绍如何优化基于流的 I/O 与 C 风格的 I/O.

> **注意**
>
> 基于流的 I/O 与 C 风格的 I/O 的实际速度会随环境的不同（如编译器，操作系统与硬件规格）发生一定的改变. 如果想要进行更进一步的分析，请以实验结果为准. 但需要注意实验中的变量控制，避免因多变量影响导致结论错误.

### 基于流的 I/O

对于基于流的 I/O（如 `std::cin` 与 `std::cout`），最常用的优化方法为关闭与 C 流的同步与解除输入输出流的关联.

#### 关闭同步

使用 `std::ios::sync_with_stdio(false)`[1] 函数来关闭与 C 流的同步. C++ 为了兼容 C，也就是为了保证程序在同时使用了 `printf` 和 `std::cout` 时不发生混乱，因此对这两种流进行了同步. 同步的 C++ 流保证是线程安全的.

这其实是 C++ 为了兼容而采取的保守措施. 若开启同步，在每一次 I/O 操作时，C++ 流会立即将此操作应用于对应的 C 缓冲区中，而如果代码中并不涉及 C 风格的 I/O，这一操作便是多余的. 因此可以在进行 I/O 操作之前关闭与 C 流的同步，但是在这样做之后要注意后续代码中不能同时使用 `std::cin` 和 `scanf`，也不能同时使用 `std::cout` 和 `printf`，但是可以同时使用 `std::cin` 和 `printf`，也可以同时使用 `scanf` 和 `std::cout`.

#### 解除关联

使用 `tie()`[2] 函数解除输入流与输出流的关联.

在默认的情况下 `std::cin` 关联的是 `&std::cout`，因此每次进行格式化输入的时候都要调用 `std::cout.flush()` 清空输出缓冲区，这样会增加 I/O 负担. 可以通过 `std::cin.tie(nullptr)` 来解除关联，进一步加快执行效率.

> **注意**
>
> 使用时不可以省略参数写做 `std::cin.tie()`，这样不会解除关联，而是返回与 `std::cin` 关联的输出流．并且也无需进行 `std::cout.tie(nullptr)`，因为默认情况下没有另一条输出流与 `std::cout` 关联．

### 代码实现

```
std::ios::sync_with_stdio(false);
std::cin.tie(nullptr);
```

> **注意**
>
> 在同时进行了上述两个操作后，程序中必须手动 `flush` 才能确保每次 `std::cout` 展现的内容可以在 `std::cin` 前出现．这是因为这种情况下调用 `std::cin` 时 `std::cout` 不会自动刷新缓冲区．例如：
>
> ```
> std::ios::sync_with_stdio(false);
> std::cin.tie(nullptr);
> std::cout << "Please input your name: "
>           << std::flush;  // 或者: std::endl;
>                           // 因为每次调用 std::endl 都会 flush 输出缓冲区，而 \n
>                           // 则不会．
> // 若去掉 std::flush，则在输入姓名之前不会显示提示信息
> std::cin >> name;
> ```

## C 风格的 I/O

`scanf` 和 `printf` 依然有效率提升的空间，提升方法均基于整数与字符串之间的转化．

> **注意**
>
> 本页面中介绍的读入和输出优化均针对整型数据．浮点数的读入与输出优化十分复杂，读入相关优化可参考 Bellerophon 算法[3]，输出相关优化可参考 Ryū 算法[4]．

### 实现设计

> **注意**
>
> 当前的优化方法着重于进行更快的 I/O，而在数据转换过程中均采用朴素方法，并未充分利用硬件特性．现如今绝大多数 x86 架构 CPU 均支持 AVX2 指令集，可以利用 SIMD 加速整数与字符串之间的转换．标准库函数并未利用 SIMD 优化，如 libstdc++ 的 实现[5] 为一次转化连续两位，并通过查表的方式转化为字符，因此优化数据转换过程也可能会带来收益．但在竞赛范畴，本文中提到的优化方法已足够应对绝大多数场景．

#### 读入优化

每个整数由符号和数字两部分组成，并且符号一定在数字部分之前，因此首先会读入符号部分．对于符号部分，正整数的 `+` 通常是省略的，且不会对后面数字所代表的值产生影响，而 `-` 不可省略，因此要进行判定．如果输入不包含负整数，这部分的判定可以省略．对于数字部分，仅包含 0 至 9 的数字，因此在读入不应存在于整数中的字符（通常为空格）时，就可以判定此整数已经读入结束．

在读入时，由于是从左向右读入数字，恰好可以利用秦九韶算法进行整数转换．因此整个转换过程可以结合输入进行．

在读入数字部分的过程中，需要判断读入的字符是否为十进制数字字符．可简单采用 `ch >= '0' && ch <= '9'` 条件进行判断，也可利用 `isdigit()`[6] 函数．

#### 输出优化

输出时需要将整数转化为字符串，一般采取朴素算法，即直接从低到高计算出整数的每一位，转化为字符后逆序输出．

### 实现细节

#### 整型溢出问题

在实现中需要注意整型溢出的问题．如输出优化中不恰当地取相反数会导致整型最小值变为相反数后超出此整型能表示的最大值，这可能导致输出错误．读入整型最小值时也可能发生类似的溢出，但这种情况下可能并不会导致读入数据错误，这是由于溢出得到的值可能与实际输入的值相等．

有符号整型溢出是未定义行为，在实现时可以借助 C 语言中负整数除法计算向零取整的性质来避免上述问题．但如果无需输入输出负数，或不可能输入输出此整型的最小值，则这一问题将不会出现．

#### 提升实现的通用性

如果程序中使用了多个类型的整型变量，那么可能需要实现多个类型不同但逻辑相同的输入输出函数．此时可以使用 C++ 中的 `template`[7] 实现对于所有整数类型的输入输出优化．例如在 C++11 标准下使用

```
template <typename T>
typename std::enable_if<std::is_integral<T>::value &&
                        std::is_signed<T>::value>::type
read(T &x);
```

或在 C++20 标准下使用

```
template <std::signed_integral T>
void read(T &x);
```

定义函数．

为了方便阅读，后文中的实现假定仅需读入 `int` 类型的整数，这些实现已足够应对大部分题目的需要．

### 实现

主流实现仅在使用的读入输出函数上有区别，在整数转换部分逻辑均相同．下面按照各实现使用的读入输出函数进行介绍．

#### 使用 `getchar` 与 `putchar` 实现

核心代码如下．

```
void read(int &x) {
  bool neg = false;
  x = 0;
  char ch = 0;
  while (ch < '0' || ch > '9') {
    if (ch == '-') neg = true;
    ch = getchar();
  }
  if (neg) {
```

```
    while (ch >= '0' && ch <= '9') {
      x = x * 10 + ('0' - ch);
      ch = getchar();
    }
  } else {
    while (ch >= '0' && ch <= '9') {
      x = x * 10 + (ch - '0');
      ch = getchar();
    }
  }
}

void write(int x) {
  bool neg = false;
  if (x < 0) {
    neg = true;
    putchar('-');
  }
  static int sta[40];
  int top = 0;
  do {
    sta[top++] = x % 10;
    x /= 10;
  } while (x);
  if (neg)
    while (top) putchar('0' - sta[--top]);
  else
    while (top) putchar('0' + sta[--top]);
}

```

**使用 `fread` 与 `fwrite` 实现**

通过 `fread` 与 `fwrite` 可以实现更快的读入输出. 其函数签名如下.

```
std::size_t fread(void* buffer, std::size_t size, std::size_t count,
                  std::FILE* stream);
std::size_t fwrite(const void* buffer, std::size_t size, std::size_t count,
                   std::FILE* stream);
```

如 `fread(Buf, 1, SIZE, stdin)`, 表示从标准输入中读入 `SIZE` 个大小为 1 字节的数据块到 `Buf` 中. 返回值表示成功读入了多少字节的数据.

由于 `fread` 与 `fwrite` 是整段读取和写入, 因此速度相较 `getchar()` 与 `putchar()` 有优势. 如果缓冲区足够大, 可以一次性读入整个文件. 但如果缓冲区不够大, 则需要多次读取以确保读取输入的所有内容. 为了实现这个功能, 只需要重定义一下 `getchar`.

```
char buf[1 << 20], *p1, *p2;
#define gc()                                                          \
  (p1 == p2 && (p2 = (p1 = buf) + fread(buf, 1, 1 << 20, stdin), p1 == p2) \
       ? EOF                                                          \
       : *p1++)
```

输出类似于读入, 先将输出内容放入一个缓冲区中, 最后通过 `fwrite` 一次性将缓冲区的内容输出即可.

核心代码如下.

```cpp
// #define DEBUG 1  // 调试开关
struct IO {
#define MAXSIZE (1 << 20)
#define isdigit(x) (x >= '0' && x <= '9')
  char buf[MAXSIZE], *p1, *p2;
  char pbuf[MAXSIZE], *pp;
#if DEBUG
#else
  IO() : p1(buf), p2(buf), pp(pbuf) {}

  ~IO() { fwrite(pbuf, 1, pp - pbuf, stdout); }
#endif
  char gc() {
#if DEBUG  // 调试，可显示字符
    return getchar();
#endif
    if (p1 == p2) p2 = (p1 = buf) + fread(buf, 1, MAXSIZE, stdin);
    return p1 == p2 ? ' ' : *p1++;
  }

  void read(int &x) {
    bool neg = false;
    x = 0;
    char ch = gc();
    for (; !isdigit(ch); ch = gc())
      if (ch == '-') neg = true;
    if (neg)
      for (; isdigit(ch); ch = gc()) x = x * 10 + ('0' - ch);
    else
      for (; isdigit(ch); ch = gc()) x = x * 10 + (ch - '0');
  }

  void read(char *s) {
    char ch = gc();
    for (; isspace(ch); ch = gc());
    for (; !isspace(ch); ch = gc()) *s++ = ch;
    *s = 0;
  }

  void read(char &c) { for (c = gc(); isspace(c); c = gc()); }

  void push(const char &c) {
#if DEBUG  // 调试，可显示字符
    putchar(c);
#else
    if (pp - pbuf == MAXSIZE) fwrite(pbuf, 1, MAXSIZE, stdout), pp = pbuf;
    *pp++ = c;
#endif
  }

  void write(int x) {
    bool neg = false;
    if (x < 0) {
      neg = true;
```

```
55        push('-');
56      }
57      static int sta[40];
58      int top = 0;
59      do {
60        sta[top++] = x % 10;
61        x /= 10;
62      } while (x);
63      if (neg)
64        while (top) push('0' - sta[--top]);
65      else
66        while (top) push('0' + sta[--top]);
67    }
68
69    void write(int x, char lastChar) { write(x), push(lastChar); }
70  } io;
71
```

使用此方法时需要注意：

- 关闭调试开关时使用 `fread()`，`fwrite()`，退出时自动析构执行 `fwrite()`．开启调试开关时使用 `getchar()`，`putchar()`，便于调试．
- 若要进行文件读写，需在所有读写进行之前加入 `freopen()`．

**使用 `mmap` 实现**

`mmap` 是 Linux 系统调用，可以将文件一次性地映射到内存中，类似于可以指针引用的内存区域，在一些场景下有更优的速度．其函数签名如下：

```
1 void *mmap(void addr[.length], size_t length, int prot, int flags, int fd,
2            off_t offset);
```

> ⚠ **注意**
>
> `mmap` 不能在 Windows 环境下使用（例如 CodeForces 与 HDU 的评测机系统），同时也不建议在正式赛场上使用．实际上使用 `fread` 已经足够快了，且如果用 `mmap` 反复读取一小块文件，做一次内存映射并且内核处理缺页的开销会远比使用 `fread` 的开销大．

首先需获取文件描述符 `fd`，然后通过 `fstat` 获取文件大小，此后通过 `mmap` 获得文件映射到内存的指针 `*pc`．之后可以直接用 `*pc++` 替代 `getchar()` 进行文件读取．

如果需要从标准输入中读入时，可以将 `fd` 设为 `0`．**但是，对标准输入使用 mmap 是极其危险的行为，同时不能在终端输入，可以选择将文件重定向到标准输入．**

> **例题：洛谷 P10815【模板】快速读入**[8]
>
> 读入 $n$ 个范围在 $[-n, n]$ 的整数，求和并输出．其中 $n \leq 10^8$．数据保证对于序列的任何前缀，这个前缀的和在 32 位有符号整形的存储范围内．

参考代码如下．

```
1 #include <fcntl.h>
2 #include <sys/mman.h>
3 #include <sys/stat.h>
4 #include <unistd.h>
5
```

```cpp
#include <cstdio>

char *pc;

void rd(int &x) {
  bool neg = false;
  x = 0;
  char ch = 0;
  while (ch < '0' || ch > '9') {
    if (ch == '-') neg = true;
    ch = *pc++;
  }
  if (neg) {
    while (ch >= '0' && ch <= '9') {
      x = x * 10 + ('0' - ch);
      ch = *pc++;
    }
  } else {
    while (ch >= '0' && ch <= '9') {
      x = x * 10 + (ch - '0');
      ch = *pc++;
    }
  }
}

int main() {
  int fd = 0;  // 从 stdin 读入
  // int fd = open("xxx.in", O_RDONLY); // 从文件读入
  struct stat state;
  fstat(fd, &state);  // 获取文件大小
  pc = (char *)mmap(NULL, state.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
  int n, x, sum = 0;
  rd(n);
  while (n--) {
    rd(x);
    sum += x;
  }
  printf("%d\n", sum);
}

```

## 参考


cin.tie 与 sync_with_stdio 加速输入输出 - 码农场[9]

C++ 高速化 - Heavy Watal[10]

'Re: mmap/mlock performance versus read' - MARC[11]


## 外部参考资料

[1] `std::ios::sync_with_stdio(false)`

[2] `tie()`

[3] Bellerophon 算法

[4] Ryū 算法

[5] 实现

[6] `isdigit()`

[7] `template` [8] 洛谷 P10815【模板】快速读入 [9] cin.tie 与 sync_with_stdio 加速输入输出 - 码农场

[10] C++ 高速化 - Heavy Watal [11] 'Re: mmap/mlock performance versus read' - MARC

## 2.6.2 分段打表

前置知识：分块→10.8.1@p2047.

朴素的打表，指的是在比赛时把所有可能的输入对应的答案都计算出来并保存下来，然后在代码里开个数组把答案放里面，直接输出即可.

注意这个技巧只适用于输入的值域不大（如，输入只有一个数，而且范围很小）的问题，否则可能会导致代码过长、MLE、打表需要的时间过长等问题.

**例题**

规定 $f(x)$ 为整数 $x$ 的二进制表示中 1 的个数. 输入一个正整数 $n(n \le 10^9)$，输出 $\sum_{i=1}^{n} f^2(i)$.

如果对于每一个 $n$，都输出 $f(n)$ 的话，除了可能会 MLE 外，还有可能代码超过最大代码长度限制，导致编译不通过.

我们考虑优化这个答案表. 采用 分块→10.8.1@p2047 的思想，我们设置一个合理的步长 $m$（这个步长一般视代码长度而定），对于第 $i$ 块，计算出：

$$\sum_{k=\frac{n}{m}(i-1)+1}^{\frac{ni}{m}} f^2(k)$$

的值.

然后输出答案时采用分块思想处理即可. 即，整块的答案用预处理的值计算，非整块的答案暴力计算.

一般来说，这样的问题对于处理单个函数值 $f(x)$ 很快，但是需要大量函数值求和（求积或某些可以快速合并的操作），枚举会超出时间限制，在找不到标准做法的情况下，分段打表是一个不错的选择.

**注意事项**

当上题中指数不是定值，但是范围较小，也可以考虑打表.

### 例题

「BZOJ 3798」特殊的质数[1]：求 $[l, r]$ 区间内有多少个质数可以分解为两个正整数的平方和.

「Luogu P1822」魔法指纹[2]

### 外部参考资料

[1] 「BZOJ 3798」特殊的质数 [2] 「Luogu P1822」魔法指纹

## 2.6.3 常见错误

> **Authors:** Estrella-Explore, H-J-Granger, orzAtalod, ksyx, Ir1d, Chrogeek, Enter-tainer, yiyangit, shuzhouliu, broken-paint, CarvingAn

本页面主要列举一些竞赛中很多人经常会出现的错误.

### 因环境不同导致的错误

- `scanf` 或 `printf` 使用 `%I64d` 格式指示符在 Linux 下可能导致输入输出格式错误.

### 会引起 CE 的错误

这类错误多为词法、语法和语义错误，引发的原因较为简单，修复难度较低.

例:

- `int main()` 写为 `int mian()` 之类的拼写错误.
- 写完 `struct` 或 `class` 忘记写分号.
- 数组开太大，（在 OJ 上）使用了不合法的函数（例如多线程），或者函数声明但未定义，会引起链接错误.
- 函数参数类型不匹配.
  - 示例：如使用 `<algorithm>` 头文件中的 `max` 函数时，传入了一个 `int` 类型参数和一个 `long long` 类型参数.

```
// query 为返回 long long 类型的自定义函数
printf("%lld\n", max(0, query(1, 1, n, l, r));

//错误     没有与参数列表匹配的 重载函数 "std::max" 实例
```

- 使用 `goto` 和 `switch-case` 的时候跳过了一些局部变量的初始化.

### 不会引起 CE 但会引起 Warning 的错误

犯这类错误时写下的程序虽然能通过编译，但大概率会得到错误的程序运行结果. 这类错误会在使用 `-W{warningtype}` 参数编译时被编译器指出.

- 赋值运算符 `=` 和比较运算符 `==` 不分.
  - 示例:

```
std::srand(std::time(nullptr));
int n = std::rand();
if (n = 1)
  printf("Yes");
else
  printf("No");

// 无论 n 的随机所得值为多少，输出肯定是 Yes
// 警告     运算符不正确: 在 Boolean 上下文中执行了常量赋值. 应考虑改用「==」.
```

  - 如果确实想在原应使用 `==` 的语句里使用 `=`（比如 `while (foo = bar)`），又不想收到 Warning，可以使用 **双括号**：`while ((foo = bar))`.
- 由于运算符优先级产生的错误.
  - 示例:

```
// 错误
// std::cout << (1 << 1 + 1);
// 正确
std::cout << ((1 << 1) + 1);

// 警告     「<<」: 检查运算符优先级是否有可能的错误; 使用括号阐明优先级
```

- 不正确地使用 `static` 修饰符.
- 使用 `scanf` 读入的时候没加取地址符 `&`.
- 使用 `scanf` 或 `printf` 的时候参数类型与格式指定符不符.

- 同时使用位操作和逻辑运算符 `==` 并且未加括号.
  - 示例：`(x >> j) & 3 == 2`
- `int` 字面量溢出.
  - 示例：`long long x = 0x7f7f7f7f7f7f7f7f`，`1<<62`.
- 未初始化局部变量.

> **未初始化变量会发生什么**
>
> 原文：https://loj.ac/d/3679[1] by @hly1204
>
> 例如我们在 C++ 中声明一个 `int a;` 但不初始化，可能有时候会认为 `a` 是一个「随机」（其实可能不是真的随机）的值，但是可能将其认为是一个固定的值，但实际上并非如此.
>
> 我们在简单的测试代码中
>
> https://wandbox.org/permlink/T2uiVe4n9Hg4EyWT[2]
>
> 代码是：
>
> ```cpp
> #include <iostream>
>
> int main() {
>   int a;
>   std::cout << std::boolalpha << (a < 0 || a == 0 || a > 0);
>   return 0;
> }
> ```
>
> 在一些编译器和环境上开启优化后，其输出为 false.
>
> 有兴趣的话可以看 https://www.ralfj.de/blog/2019/07/14/uninit.html[3]，尽管其是用 Rust 做的实验，但是本质是一样的.

- 局部变量与全局变量重名，导致全局变量被意外覆盖.（开 `-Wshadow` 就可检查此类错误.）
- 运算符重载后引发的输出错误.
  - 示例：

```cpp
// 本意：前一个 << 为重载后的运算符，表示输出；后一个 << 为移位运算符，表示将 1 左移 1 位.  但由于忘记加括号，导致编译器将后一个 << 也判作输出运算符，而导致输出的结果与预期不同.  错误 std::cout << 1 << 1; 正确
std::cout << (1 << 1);
```

## 既不会引起 CE 也不会引发 Warning 的错误

这类错误无法被编译器发现，仅能自行查明.

### 会导致 WA 的错误

- 上一组数据处理完毕，读入下一组数据前，未清空数组.
- 读入优化未判断负数.
- 所用数据类型位宽不足，导致溢出.
  - 如习语「三年 OI 一场空，不开 `long long` 见祖宗」所描述的场景.选手因为没有在正确的地方开 `long long`（将整数定义为 `long long` 类型），导致得出错误的答案而失分.
- 存图时，节点编号 0 开始，而题目给的边中两个端点的编号从 1 开始，读入的时候忘记 -1.
- 大/小于号打错或打反.
- 在执行 `ios::sync_with_stdio(false);` 后混用 `scanf/printf` 和 `std::cin/std::cout` 两种 IO，导致输入/输出错乱.

- 示例：

```
// 这个例子将说明关闭与 stdio 的同步后，混用两种 IO 方式的后果
// 建议单步运行来观察效果
#include <cstdio>
#include <iostream>

int main() {
  // 关闭同步后，cin/cout 将使用独立缓冲区，而不是将输出同步至 scanf/printf
  // 的缓冲区，从而减少 IO 耗时
  std::ios::sync_with_stdio(false);
  // cout 下，使用'\n'换行时，内容会被缓冲而不会被立刻输出
  std::cout << "a\n";
  // printf 的 '\n' 会刷新 printf 的缓冲区，导致输出错位
  printf("b\n");
  std::cout << "c\n";
  // 程序结束时，cout 的缓冲区才会被输出
  return 0;
}
```

- 由于宏的展开，且未加括号导致的错误.
  - 示例：该宏返回的值并非 $4^2 = 16$ 而是 $2 + 2 \times 2 + 2 = 8$.

```
#define square(x) x* x
printf("%d", square(2 + 2));
```

- 哈希的时候没有使用 `unsigned` 导致的运算错误.
  - 对负数的右移运算会在最高位补 1. 参见：位操作符→4.2.4@p339.
- 没有删除或注释掉调试输出语句.
- 误加了 `;`.
  - 示例：

```
/* clang-format off */
while (1);
    printf("OI Wiki!\n");
```

- 哨兵值设置错误. 例如，平衡树的 `0` 节点.
- 在类或结构体的构造函数中使用 `:` 初始化变量时，变量声明顺序不符合初始化时候的依赖关系.
  - 成员变量的初始化顺序与它们在类中声明的顺序有关，而与初始化列表中的顺序无关. 参见：构造函数与成员初始化器列表[4] 的「初始化顺序」
  - 示例：

```
#include <iostream>

class Foo {
 public:
  int a, b;

  // a 将在 b 前初始化，其值不确定
  Foo(int x) : b(x), a(b + 1) {}
};

int main() {
  Foo bar(1, 2);
  std::cout << bar.a << ' ' << bar.b;
```

```
14 }
15
16 // 可能的输出结果：-858993459 1
```

- 并查集合并集合时没有把两个元素的祖先合并.
  - 示例：

```
1 f[a] = b;              // 错误
2 f[find(a)] = find(b);  // 正确
```

- `freopen` 使用 `a` 进行追加写
  - CCF 的检测环境不会清空输出文件，使用 `a` 会导致上一位选手的输出也被评测机读入引发 WA

### 换行符不同

> **Warning**
>
> 在正式比赛中会尽量保证选手答题的环境和最终测试的环境相同.
>
> 本节内容仅适用于模拟赛等情况，而我们也建议出题人尽量让数据符合 数据格式→2.7@p123.

不同的操作系统使用不同的符号来标记换行，以下为几种常用系统的换行符：

- LF（用 `\n` 表示）：`Unix` 或 `Unix` 兼容系统
- CR+LF（用 `\r\n` 表示）：`Windows`
- CR（用 `\r` 表示）：`Mac OS` 版本 9 及以前

而 C/C++ 利用转义序列 `\n` 来换行，这可能会导致我们认为输入中的换行符也一定是由 `\n` 来表示，而只读入了一个字符来代表换行符，这就会导致我们没有完全读入输入文件.

以下为解决方案：

- 多次 `getchar()`，直到读到想要的字符为止.
- 使用 `cin` 读入，**这可能会增大代码常数**.
- 使用 `scanf("%s",str)` 读入一个字符串，然后取 `str[0]` 作为读入的字符.
- 使用 `scanf(" %c",&c)` 过滤掉所有空白字符.

### 会导致未知的结果

未定义行为会导致未知的结果，可能是 WA，RE 等. 编译器通常会假定你的程序不会出现未定义行为，因此出现开 O2 与不开 O2 代码行为不一致的情况.

- 除以 0（求 0 的逆元）

> **示例**
>
> ```
> 1 cout << x / 0 << endl;
> ```

- 数组（下标）越界

  例如：

  - 未正确设置循环的初值导致访问了下标为 -1 的值.
  - 无向图边表未开 2 倍.
  - 线段树未开 4 倍空间.
  - 看错数据范围，少打一个零.
  - 错误预估了算法的空间复杂度.
  - 写线段树的时候，`pushup` 或 `pushdown` 叶节点.

正确的做法：不要越界，记得检查自己的代码，使得下标访问数 `x` 在定义的下标中.

- 除 main 外有返回值函数执行至结尾未执行任何 return 语句

  即使有一个分支有返回值，但是其他分支却没有，结果也是未定义的.

  可以向编译选项中追加 `-Wall`，检查编译器是否给出有关于函数未 return 的警告.

- 尝试修改字符串字面量

  **⚠ 示例**

  ```
  1 char *p = "OI-wiki";
  2 p[0] = 'o';
  3 p[1] = 'i';
  ```

  这样试图修改字符串字面量会导致 **未定义行为**, 应当使用其他 **合适** 的数据类型, 例如 `std::string` 和 `char[]`.

- 多次释放/非法解引用一片内存

  例如:

  - 未初始化就解引用指针.
  - 指针指向的内存区域已经释放.

    使用 `erase` 或 `delete` 或 `free` 操作应注意不要对同一地址/对象多次使用.

- 尝试释放由 `new []` 分配的整块内存的一部分

  例如:

  ```
  1 object *pool = new object[POOL_SIZE];
  2
  3 object *pointer = pool + 10;
  4
  5 // 报错!
  6 delete pointer;
  ```

  常见于使用内存池提前分配整块内存后，试图使用 `delete` 或 `free()` 释放从内存池中获取的单个对象.

- 解引用空指针/野指针

  对于空指针：先应该判断空指针，可以用 `p == nullptr` 或 `!p`.

  对于野指针：可以释放指针的时候将其置为 `nullptr` 以规避.

- 有符号数溢出

  例如我们有一个表达式 `x+1 > x`.

  正常输出应当是 `true`, 但是在 `INT_MAX` 作为 `x` 时输出 `false`, 这时称为 `signed integer overflow`.

  可以使用更大的数据类型（例如 `long long` 或 `__int128`），或判断溢出. 若保证无负数，亦可使用无符号整型.

  有符号整数溢出可能影响编译优化，例如代码:

  ```
  1 int foo(int x) {
  2   if (x > x + 1) return 1;
  3   return 0;
  4 }
  ```

  可能被编译器直接优化为:

```
int foo(int x) { return 0; }
```

因为编译器可以假定有符号整数永远不会溢出，因此 `x > x + 1` 永远不会成立.

- 使用未初始化的变量

> ⚠ **示例**
>
> ```
> int foo(int a) {
>   int t; /* 没有初始化 */
>   if (/* 使用 */ t > 3) return a;
>   return 0;
> }
> ```

### 会导致 RE

- 没删文件操作（某些 OJ）.
- 排序时比较函数的错误 `std::sort` 要求比较函数是严格弱序：`a<a` 为 `false`；若 `a<b` 为 `true`，则 `b<a` 为 `false`；若 `a<b` 为 `true` 且 `b<c` 为 `true`，则 `a<c` 为 `true`. 其中要特别注意第二点. 如果不满足上述要求，排序时很可能会 RE. 例如，编写莫队的奇偶性排序时，这样写是错误的:

```
bool operator<(const int a, const int b) {
  if (block[a.l] == block[b.l])
    return (block[a.l] & 1) ^ (a.r < b.r);
  else
    return block[a.l] < block[b.l];
}
```

上述代码中 `(block[a.l]&1)^(a.r<b.r)` 不满足上述要求的第二点. 改成这样就正确了:

```
bool operator<(const int a, const int b) {
  if (block[a.l] == block[b.l])
    // 错误: 不满足严格弱序的要求
    // return (block[a.l] & 1) ^ (a.r < b.r);
    // 正确
    return (block[a.l] & 1) ? (a.r < b.r) : (a.r > b.r);
  else
    return block[a.l] < block[b.l];
}
```

- Windows 下栈空间不足，导致栈空间溢出，Windows 向程序发出 SIGSEGV 信号，程序终止并返回 3221225725（即 0xC00000FD, NTSTATUS 定义为 `STATUS_STACK_OVERFLOW`）.

  若使用 gcc 编译器，可在编译时加入命令 `-Wl,--stack=SIZE` 以指定栈空间大小限制，其中 `SIZE` 为栈空间大小字节数.

  Linux 下栈空间不足，导致栈空间溢出，Linux 会在栈堆中乱写 `head_info`，此操作绝大多数情况下会导致程序立即退出，显示 `段错误（核心已转储）/segmentation fault (core dumped)` 等字样.

  可以在终端下使用 `ulimit -s SIZE` 修改当前终端的栈空间限制，其中 `SIZE` 为栈空间大小千字节数（KB）.

  **请注意，如果你将栈空间限制设置过大，无穷递归可能导致递归栈过大进而导致系统崩溃.**

### 会导致 TLE

- 分治未判边界导致死递归.
- 死循环.

    - 循环变量重名.
    - 循环方向反了.
- BFS 时不标记某个状态是否已访问过.
- 使用宏展开编写 min/max

  这种错误会大大增加程序的运行时间，甚至直接影响代码的时间复杂度. 在初学者写线段树时尤为多见.

  常见的错误写法是这样的:

```
#define Min(x, y) ((x) < (y) ? (x) : (y))
#define Max(x, y) ((x) > (y) ? (x) : (y))
```

  这样写虽然在正确性上没有问题，但是如果直接对函数的返回值取 max，如 `a = Max(func1(), func2())`，而这个函数的运行时间较长，则会大大影响程序的性能，因为宏展开后是 `a = func1() > func2() ? func1() : func2()` 的形式，调用了三次函数，比正常的 max 函数多调用了一次. 注意，如果 `func1()` 每次返回的答案不一样，还会导致这种 `max` 的写法出现错误. 例如 `func1()` 为 `return ++a;` 而 `a` 为全局变量的情况.

  示例：如下代码会被卡到单次查询 $\Theta(n)$ 导致 TLE.

```
#define max(x, y) ((x) > (y) ? (x) : (y))

int query(int t, int l, int r, int ql, int qr) {
  if (ql <= l && qr >= r) {
    ++ti[t];  // 记录结点访问次数方便调试
    return vi[t];
  }

  int mid = (l + r) >> 1;
  if (mid >= qr) return query(lt(t), l, mid, ql, qr);
  if (mid < ql) return query(rt(t), mid + 1, r, ql, qr);
  return max(query(lt(t), l, mid, ql, qr), query(rt(t), mid + 1, r, ql,
qr));
}
```

- 使用 + 运算符向 `std::string` 类字符串追加字符

  这种错误会创建一个临时 `string` 变量，修改完成后再赋值给原变量. 这种错误无法被编译器优化，在数据量大的情况下可能会导致时间复杂度退化.

  常见 错误写法:

```
std::string a;
char b = 'c';
a = a + b;
```

  当执行这段代码时，程序首先会创建一个临时 `string` 变量，随后将 `a` 的值存入临时变量，然后在末尾添加 `b` 的值，最后再存入 `a`.

  从 汇编结果[5] 可以看出，`a = a + b` 调用了三次 `std::__cxx11::basic_string` 中的功能，分别为 `operator+`、`operator=` 和创建变量.

  正确写法应该是:

```
std::string a;
char b = 'c';
a += b;
```

这种写法[6] 会直接将字符 `b` 附加到字符串 `a` 中，仅调用了一次 `operator+=`. 更详细的性能比较可参考 Benchmark[7].

- 没删文件操作（某些 OJ）.
- 在 `for/while` 循环中重复执行复杂度非 $O(1)$ 的函数. 严格来说，这可能会引起时间复杂度的改变.
- 进行二分搜索时中点公式或终止条件错误.

### 会导致 MLE

- 数组过大.

> **Linux 下的内存占用指标详解**
>
> 太长不看版：如果在 CCF 系列的考试时声明了一个特别大的全局静态数组，此时需要特别慎重. 因为程序声明的数组会全部计入内存占用中（而不像大部分在线评测平台仅计算实际使用的部分），在某些情况下这甚至有可能导致整道题全部 MLE.
>
> - 关于 RSS 和 VSZ[1][2]
>
>  1. VSZ (Virtual Memory Size，虚拟内存大小）[3]
>
>  VSZ 表示进程的 **虚拟内存大小**，是进程可以访问的虚拟地址空间的总大小，通常以 KB 显示.
>
>  虚拟内存是一个逻辑概念，通常会比实际内存使用大很多.
>
>  在 Linux 下你可以使用 `top` 命令来查看某个进程的内存占用组成，其中 `VIRT` 这一列就代表它占用的虚拟内存.
>
>  虚拟内存一般包括进程分配但未实际使用的地址空间，简而言之，申请了多少，虚拟内存就大约是多少.
>
>  需要特别注意的是，常用的在线评测平台通常只统计物理内存占用. 但 CCF **的评测环境统计的是虚拟内存**, 这意味着如果你声明了一个全局静态大数组, 即使只使用了其中的一小部分，也会占用大量空间.
>
>  2. RSS（Resident Set Size，常驻集大小）[4]
>
>  RSS 表示进程实际占用的 **物理内存大小**，即驻留在 RAM 中的页帧大小，通常以 KB 显示.
>
>  同样的，你也可以用 `top` 在 `RES` 这一列查看某个进程的物理内存.
>
>  RSS 一般仅包含实际加载到物理内存的部分，也就是说，实际使用了多少就是多少.
>
> - 内存占用行为分析
>
>  假设声明了以下数组:
>
>  ```
>  1 const int SIZE = 1e8;
>  2 int arr[SIZE];  // 占用空间: 4 字节 * 1 亿 = 400 MB
>  ```
>
>  这是一个静态数组, 分配在全局数据段. 这个数组未经显式初始化, 通常会被分配在 BSS 段（如果显式初始化（如全为 0 或其他值），则分配在 DATA 段）.
>
>  - 当完全未使用该数组时（假定编译器不会优化掉该数组）

[1] What is RSS and VSZ in Linux memory management - Stack Overflow[8]
[2] Need explanation on Resident Set Size/Virtual Size - Stack Overflow[9]
[3] 虚拟内存[10]
[4] 常驻集大小[11]

- 物理内存：如果数组未被访问，按需分页机制（Demand Paging）会使内存页尚未加载到物理内存中. 物理内存不会增加或仅略微增加一点（可能加载了一些元数据页）.
- 虚拟内存：数组的大小会计入虚拟内存（增加 `400MB`），因为整个数组的虚拟地址空间已经被分配.

– 部分使用该数组时

假设仅使用数组的少部分元素，例如：

```
arr[0] = 1;
arr[999999] = 2;
```

- 虚拟内存：虚拟内存不会改变，仍为 `400MB`.
- 物理内存：每次访问数组的一个元素，对应的虚拟页会被加载到物理内存中. 假定系统分页大小为 `4KB`，那么每页包含 $4\ \text{KB} \div 4\text{B} = 1024$ 个 `int` 元素. 访问数组两次可能加载 2 个页，即增加约 $2 \times 4\ \text{KB} = 8\ \text{KB}$ 的物理内存.

– 数组被大半使用

假设对数组的前 $50,000,000$ 个元素赋值：

```
for (int i = 0; i < 50000000; ++i) {
  arr[i] = i;
}
```

- 虚拟内存 (VSZ)：VSZ 仍为 `400MB`，不会发生变化.
- 物理内存 (RSS)：此时是连续访问（访问的 $50,000,000$ 个元素在内存地址上相邻），需要加载的分页数为 $\left\lceil \frac{50,000,000}{1024} \right\rceil = 48,828$ 页.

  假设每页大小为 `4KB`，因此总计 $48,828 \times 4\ \text{KB} \approx 190\ \text{MB}$，物理内存增加到约 `190MB`.

  注：如果对该数组赋值时下标随机，则会导致物理内存占用与预测差距较大（因为内存页加载是以地址为准，导致随机赋值时会加载大量内存页）.

简要总结：随着被访问的部分占的比例增加，物理内存趋近于虚拟内存（假定不存在页面回收）.

- STL 容器中插入了过多的元素.
  - 经常是在一个会向 STL 插入元素的循环中死循环了.
  - 也有可能被卡了.

### 会导致常数过大

- 定义模数的时候，未定义为常量.
  - 示例：

```
// int mod = 998244353;       // 错误
const int mod = 998244353;  // 正确，方便编译器按常量处理
```

- 使用了不必要的递归（尾递归不在此列）.
- 将递归转化成迭代的时候，引入了大量额外运算.

### 只在程序在本地运行的时候造成影响的错误

- 文件操作有可能会发生的错误：

  - 对拍时未关闭文件指针 `fclose(fp)` 就又令 `fp = fopen()`. 这会使得进程出现大量的文件野指针.
  - `freopen()` 中的文件名未加 `.in/.out`.
- 使用堆空间后忘记 `delete` 或 `free`.

### 参考资料与注释


[1] https://loj.ac/d/3679

[2] https://wandbox.org/permlink/T2uiVe4n9Hg4EyWT

[3] https://www.ralfj.de/blog/2019/07/14/uninit.html

[4] 构造函数与成员初始化器列表

[5] 汇编结果

[6] 这种写法

[7] Benchmark

[8] What is RSS and VSZ in Linux memory management - Stack Overflow

[9] Need explanation on Resident Set Size/Virtual Size - Stack Overflow

[10] 虚拟内存

[11] 常驻集大小


## 2.6.4 常见技巧

> **Authors:** H-J-Granger, Ir1d, ChungZH, Marcythm, StudyingFather, billchenchina, Suyun514, Psycho7, greyqz, Xeonacid, partychicken

本页面主要列举一些竞赛中的小技巧.

### 利用局部性

局部性是指程序倾向于引用邻近于其他最近引用过的数据项的数据项，或者最近引用过的数据项本身. 局部性分为时间局部性和空间局部性.

具体可参见 循环展开 (Loop Unroll)→4.4.10@p456、代码布局优化 (Code Layout Optimizations)→4.4.10@p456 等内容

### 循环宏定义

如下代码可使用宏定义简化:

```
for (int i = 0; i < N; i++) {
  // 循环内容略
}

// 使用宏简化
#define f(x, y, z) for (int x = (y), __ = (z); x < __; ++x)

// 这样写循环代码时，就可以简化成 `f(i, 0, N)` . 例如:
// a is a STL container
f(i, 0, a.size()) { ... }
```

另外推荐一个比较有用的宏定义:

```
#define _rep(i, a, b) for (int i = (a); i <= (b); ++i)
```

### 善用 namespace

使用 namespace 能使程序可读性更好，便于调试.

**例题：NOI 2018 屠龙勇士**

```cpp
// NOI 2018 屠龙勇士 40分部分分代码
#include <algorithm>
#include <cmath>
#include <cstring>
#include <iostream>
using namespace std;
long long n, m, a[100005], p[100005], aw[100005], atk[100005];

namespace one_game {
// 其实namespace里也可以声明变量
void solve() {
  for (int y = 0;; y++)
    if ((a[1] + p[1] * y) % atk[1] == 0) {
      cout << (a[1] + p[1] * y) / atk[1] << endl;
      return;
    }
}
}  // namespace one_game

namespace p_1 {
void solve() {
  if (atk[1] == 1) {  // solve 1-2
    sort(a + 1, a + n + 1);
    cout << a[n] << endl;
    return;
  } else if (m == 1) {  // solve 3-4
    long long k = atk[1], kt = ceil(a[1] * 1.0 / k);
    for (int i = 2; i <= n; i++)
      k = aw[i - 1], kt = max(kt, (long long)ceil(a[i] * 1.0 / k));
    cout << k << endl;
  }
}
}  // namespace p_1

int main() {
  int T;
  cin >> T;
  while (T--) {
    memset(a, 0, sizeof(a));
    memset(p, 0, sizeof(p));
    memset(aw, 0, sizeof(aw));
    memset(atk, 0, sizeof(atk));
    cin >> n >> m;
    for (int i = 1; i <= n; i++) cin >> a[i];
    for (int i = 1; i <= n; i++) cin >> p[i];
    for (int i = 1; i <= n; i++) cin >> aw[i];
    for (int i = 1; i <= m; i++) cin >> atk[i];
    if (n == 1 && m == 1)
      one_game::solve();  // solve 8-13
    else if (p[1] == 1)
      p_1::solve();  // solve 1-4 or 14-15
```

```
52      else
53        cout << -1 << endl;
54    }
55    return 0;
56  }
```

## 使用宏进行调试

编程者在本地测试的时候，往往要加入一些调试语句．而在需要提交到 OJ 时，为了不使调试语句的输出影响到系统对程序输出结果的判断，就要把它们全部删除，耗时较多．这种情况下，可以通过定义宏的方式来节省时间．大致的程序框架是这样的：

```
#define DEBUG
#ifdef DEBUG
// do something when DEBUG is defined
#endif
// or
#ifndef DEBUG
// do something when DEBUG isn't defined
#endif
```

`#ifdef` 会检查程序中是否有 `#define` 定义的对应标识符，如果有定义，就会执行后面的语句．而 `#ifndef` 会在没有定义相应标识符的情况下执行后面的语句．

这样，只需在 `#ifdef DEBUG` 里写好调试用代码，`#ifndef DEBUG` 里写好真正提交的代码，就能方便地进行本地测试．提交程序的时候，只需要将 `#define DEBUG` 一行注释掉即可．也可以不在程序中定义标识符，而是通过 `-DDEBUG` 的编译选项在编译的时候定义 `DEBUG` 标识符．这样就可以在提交的时候不用修改程序了．

不少 OJ 都开启了 `-DONLINE_JUDGE` 这一编译选项，善用这一特性可以节约不少时间．

## 对拍

对拍是一种进行检验或调试的方法，通过对比两个程序的输出来检验程序的正确性．可以将自己程序的输出与其他程序的输出进行对比，从而判断自己的程序是否正确．

对拍过程要多次进行，因此需要通过批处理的方法来实现对拍的自动化．

具体而言，对拍需要一个 数据生成器→3.9.3@p273 和两个要进行输出结果比对的程序．

每运行一次数据生成器都将生成的数据写入输入文件，通过重定向的方法使两个程序读入数据，并将输出写入指定文件，最后利用 Windows 下的 `fc` 命令比对文件（Linux 下为 `diff` 命令）来检验程序的正确性．如果发现程序出错，可以直接利用刚刚生成的数据进行调试．

对拍程序的大致框架如下：

```
#include <cstdio>
#include <cstdlib>

int main() {
  // For Windows
  // 对拍时不开文件输入输出
  // 当然，这段程序也可以改写成批处理的形式
  while (true) {
    system("gen > test.in");  // 数据生成器将生成数据写入输入文件
```

```
    system("test1.exe < test.in > a.out");  // 获取程序1输出
    system("test2.exe < test.in > b.out");  // 获取程序2输出
    if (system("fc a.out b.out")) {
      // 该行语句比对输入输出
      // fc返回0时表示输出一致，否则表示有不同处
      system("pause");  // 方便查看不同处
      return 0;
      // 该输入数据已经存放在test.in文件中，可以直接利用进行调试
    }
  }
}
```

## 内存池

当动态分配内存时，频繁使用 `new/malloc` 会占用大量的时间和空间，甚至生成大量的内存碎片从而降低程序的性能，可能会使原本正确的程序 TLE/MLE.

这时候需要使用到「内存池」这种技巧：在真正使用内存之前，先申请分配一定大小的内存作为备用. 当需要动态分配时直接从备用内存中分配一块即可.

在大多数 OI 题当中，可以预先算出需要使用到的最大内存并一次性申请分配.

示例：

```
// 申请动态分配 32 位有符号整数数组：
int* newarr(int sz) {
  static int pool[MAXN], *allocp = pool;
  return allocp += sz, allocp - sz;
}

// 线段树动态开点的代码：
Node* newnode() {
  static Node pool[MAXN << 1], *allocp = pool - 1;
  return ++allocp;
}
```

## 参考资料

洛谷日报 #86[1]

《算法竞赛入门经典 习题与解答》

## 外部参考资料

[1] 洛谷日报 #86 

# 2.7 出题

**Authors:** ouuan, Henry-ZHR, StudyingFather, ChungZH, xyf007, Cryflmind, oierlinch, xk2013awa

# 出题前的准备

## 具备一定的水平

一方面，一个人自己出题，很难出出难度大于自身水平的题目，一定的 OI 水平有助于想到更加优质的 idea 并想出优秀的做法；另一方面，OI 水平在一定程度上代表着 OI 资历，见识过更多的题目的选手也会对「好题」拥有自己的见解.

## 抱有认真负责的态度

出题是给别人做的，比起展示自己，更多是为了是服务他人．算法竞赛是选手之间的竞赛，而不是出题人与做题人之间的较量．因此，出题不应以考倒选手为目标（当然，适当的防 AK 与良好的区分度也是非常重要的），而应当让选手能在比赛中有所收获．花费足够的时间精力去学习如何出题并认真负责地出题非常重要.

## 做好耗费大量时间的准备

如果想要认真地出题，就必然要花费大量的时间．如果不做好心理准备，可能导致比赛准备匆忙，质量不过关，也可能在事后由于没有将时间花费在学习上而懊悔．但出题也可以带来很多美好的回忆，如果真的对出题抱有兴趣，并做好了充分的心理准备，出题带来的收获也能够弥补那些花费的时间.

## 认真阅读本文的内容

本文从如何出题、如何把题出好两个方面对整个出题流程进行了介绍．对于想要出题的人来说，认真阅读本文一定能够受益匪浅.

# 题目内容

出一道题，idea，即题目本质的内容，是题目的灵魂，也是出题的第一步.

## idea 的来源

1. 受到已有题目的启发（但不能照搬或无意义地加强，如：序列题目搬到仙人掌上）.
2. 受到学过的知识点的启发（但不能毫无联系地拼凑知识点）.
3. 从生活/游戏中受到启发（但注意不要把游戏出成大模拟）.
4. 不知道为什么，就是想到了一道题.

## 什么样的 idea 是不好的

### 关于原题

原题大致可分为完全一致、几乎一致和做法一致三种.

- 完全一致：使用一题的 AC 代码可以 AC 另一题.
- 几乎一致：由一题的 AC 代码改动至另一题的 AC 代码可以由一个不会该题的人完成.
- 做法一致：核心思路、做法一致，但代码实现上、不那么关键的细节上有差异.

这三种原题自下而上为包含关系.

以下情况不应出现：

1. 在明知有「几乎一致」的原题的情况下出原题.
2. 由于未使用搜索引擎查找导致自己不清楚有原题，从而出了「几乎一致」的原题.
3. 在「做法一致」的原题广为人知（如：NOIP、NOI 原题）时出原题.
4. 在带有选拔性的考试的非送分题中出现「做法一致」的原题.

以下情况最好不要出现：

1. 在明知有至少为「做法一致」的原题的情况下出原题.
2. 由于未使用搜索引擎查找导致自己不清楚有原题，从而出了「做法一致」的原题.

3. 在任何情况下出「几乎一致」的原题.

可以放宽要求的例外情况：

1. 校内模拟赛.
2. 以专题训练为目的的模拟赛.
3. 难度较低的比赛，或是定位为送分题的题目.

### 关于毒瘤题

「毒瘤题」是一个非常模糊而主观的观念，在这只是引用一些前人关于此的探讨，加以自己的一些理解. 这个话题是非常开放的，欢迎大家来发表自己的观点.

> 一道好题不应该是两道题拼在一起，一道好题会有自己的 idea——而它应该不加过多包装地突出这个 idea.
>
> 一道好题应该新颖. 真正的好题，应该是能让人脑洞出新的好题的好题.
>
> ——vfk《UOJ 精神之源流》[1]

例子：「XR-1」柯南家族[2]，做法的前后两部分完全割裂，前半部分为「模板」树上后缀排序[3]，后半部分是经典树上问题. 就算是随意输入树的点权，依然可以做第二部分，前后部分没有联系.

> 一类 OI 题以数学为主，无论是题目描述还是做法都是数学题的特征，并且解法中不含算法相关的知识点，这类 OI 题目统称为纯数学题.
>
> ——王天懿《论偏题的危害》[4]

经典例子：NOIP2017 小凯的疑惑[5]

OI 中的数学题与其它数学题的区别，也是体现 OI 本质的一个特点，是 OI 中的数学题往往重点不在答案 **是什么**，而在如何 **加快** 答案的计算. 如果一道题考察的重点是「怎么算」而非「怎么快速计算」，这样的数学题一般都是不适合出在 OI 中的.

> 一部分偏题中牵涉到了大学物理的内容，导致选手在面对这些从未接触过物理知识点时变得不知所措，造成了知识上的隔膜.
>
> ——王天懿《论偏题的危害》[6]

经典例子：「清华集训 2015」多边形下海[7]

不止是物理，OI 题目中不应过多涉及到其它学科的知识，如果涉及应当给予详细的解释，不应使其它学科的知识作为解题的重大障碍.

> 一道好题无论难度如何，都应该具有自己的思维难度，需要选手去思考并发现一些性质.
>
> 一道好题的代码可以长，但一定不是通过强行嵌套或者增加条件而让代码变长，而是长得自然，让人感觉这个题的代码就应该是这么长.
>
> ——王天懿《论偏题的危害》[8]

经典例子：「SDOI2010」猪国杀[9]，「集训队互测 2015」未来程序·改[10]

在一般的 OI 比赛中，思维难度应占主要部分. 当然，如 THUWC/THUSC 的 Day 2+ 那样的工程题也有其存在的道理——毕竟体验营的目的除了考察选手的算法设计能力，还有和大学学习对接的工程代码以及文档学习能力. 但在一般的 OI 比赛中，考察更多的应当还是算法设计与思维能力.

## 题面

### 使用 LaTeX 书写公式

网上有很多 LaTeX 的教程，如：

- LaTeX 入门→3.12@p291
- LaTeX 数学公式大全[11]
- LaTeX 各种命令，符号[12]

使用时请注意 LaTeX 公式的格式要求→1.5@p23.

## 题目背景

题目背景最好尽量简短．在题目背景较长时，应当与题目描述分开．

需要绝对避免题目背景严重影响题意的理解．

必要时，可以提供与背景结合的题目描述与简洁的题目描述两个版本．

## 题目描述

简而言之，题目描述需要 **清晰易懂**.

题面中的每个可能不被理解的定义都应得到解释，不应凭空冒出未加定义的概念．例如：在 CF1172D Nauuo and Portals[13] 中，你必须在题面中解释什么是「传送门」.

题面中涉及到的每个概念应当使用单一的词汇来描述. 例如: 不应一会儿说「费用」, 一会儿说「代价」.

不应不加说明地使用与原义、常见义不同的词汇．例如：不应不加说明地用「路径」代指一条边.

你需要保证你的题面不会自相矛盾．例如：在 CF1173A Nauuo and Votes[14] 中，没有把 "?" 作为一种 "result"，是因为 "?" 的含义是 "there are more than one possible results".

你需要保证你的题面不能被错误理解而自圆其说，即使这种理解是反常识、没有人会这么去想的．例如：在 CF1172D Nauuo and Portals[15] 中，之所以要繁琐地定义 "walk into" 并与 "teleport" 区分，是为了防止这种理解：通过传送门可以到另一个传送门，而到了传送门会传送，因此会反复横跳.

顺着读题目描述应当能看懂每一句话，并理解题目的任务与要求．至少在紧接着的下一段话中疑惑能够得到解释，而不是需要在若干段后才能得到解释，或者要看了输入输出格式才能明白题意，甚至需要根据样例来猜题意．例如：在「GuOJ Round #1」琪露诺的冰雪宴会[16] 中，在输出格式才第一次出现了题目的目标「雾之湖最终能接收到的最大水量」，再加上「灵梦当然能很快算出来清理完全部小溪的总费用是多少」这句带有误解性质的话，更容易使人读错题意，这是不可取的，应当在题目描述中就对题目的目标进行说明.（在这个例子中还存在题目背景严重影响题意理解的问题.）相同的错误还出现在 CF1423(4)N Bubblesquare Tokens[17] 中，在输出格式才第一次出现了题目的目标 "friend pairs and number of tokens each of them gets on behalf of their friendship".

## 输入输出格式

输入输出格式清晰 **完整** 即可，没有死板的要求，个人建议参照 CF 的题目来写输入输出格式，具体可以参考 CF 出题人须知[18].

为了方便选手做题，输入输出格式中最好说明每个变量的具体含义，除非变量的意义非常长，没法一句话说清楚（这时可以说「意义见题目描述」）.

需要特别注意的是，如果输出中含有小数，请尽量使用 SPJ→2@p67 来对误差的大小进行限制，而非要求「保留 x 位小数」.

「保留 x 位小数」对精度的要求可能是无限的．例如：要求保留三位小数，实际答案为 0.0015，此时只要有任意大小的误差导致计算出的答案小于 0.0015, 即使计算出的答案是 0.00149999… 也会输出错误的答案.

如果无法使用 SPJ，请保证对精度的要求是有限的，例如：请输出答案四舍五入后保留小数点后三位的结果．令标准答案为 $ans$，数据保证对于任意满足 $\frac{|x-ans|}{\max(1,ans)} < 10^{-9}$ 的 $x$，四舍五入后结果与 $ans$ 四舍五入后相同.

可以参考的一些句子：

```
输入的第一行包含三个正整数 $n$, $m$, $k$ ($1\le n,m\le 2\cdot 10^5$, $1\le k\le 100$) — $n$ 表示数列的长度，$m$ 表示操作个数，$k$ 的意义见题目描述.
```

```
输入的第二行包含 $n$ 个非负整数 $a_1,a_2,\ldots,a_n$ ($1\le a_i\le 10^9$) — 题目给出的数列.
```

```
接下来的 $m$ 行中的第 $i$ 行包含两个正整数 $l_i$ 和 $r_i$ ($1\le l_i\le r_i\le n$)，表示第 $i$ 次操作在区间 $[l_i,r_i]$ 上进行.
```

```
接下来的 $n-1$ 行，每行包含两个正整数 $u$ 和 $v$ ($1\le u,v\le n$)，表示 $u$ 和 $v$ 之间由一条边相连.

数据保证给出的边能构成一棵树.
```

```
输入的唯一一行包含一个由小写英文字母构成的非空字符串，其长度不超过 $10^6$.
```

```
输入的第二行包含一个小数点后不超过三位的实数 $x$ ($-10^6\le x\le 10^6$)，意义见题目描述.
```

```
输出包含一个实数，当你的输出与标准答案之间的绝对误差或相对误差小于 $10^{-6}$ 时视作正确.
```

```
输出的第二行包含 $n$ 个正整数，表示你构造的一组方案 — 其中第 $i$ 个数表示你打出的第 $i$ 张牌的编号.

如果有多组合法的答案，可以任意输出其中一组.
```

**在选手代码内由随机数生成器生成输入数据**

有的题目会因为输入数据过大，为了防止读入用时过长，而要求选手在代码内通过给定的数据生成器生成数据，代替通过标准输入或文件输入来读入数据.

采用这种做法需要谨慎考虑，因为它有很多缺点：

- 可能引入了正解所不需要的数据随机性，或者使得构造数据变得困难
- 可能增大了理解输入格式的难度
- 如果随机数生成器封装的不好，可能理解数据生成器本身的使用方法就有难度
- 如果选手没有使用出题者推荐的语言，可能需要自己写一个数据生成器

采用这种做法一般是为了防止读入数据用时过长，所以一个可能的替代方案是下发一个性能足够好的 读入、输出优化→2.6.1@p103 模板，以尽量保证所有人的读入用时一致，这样的话即使读入用时很久也不会影响不同选手用时的差异. 另一个解决方案是将题目包装成函数调用式（而非 IO 式）交互题，即使算法过程中没有交互，交互题也可以起到统一读入用时的作用，IOI 就采用了所有题目都是交互题的方案. 但是，这两种方案都对选手使用的语言有限制，需要出题者手动支持每种允许选手使用的语言.

回到问题的本源，还可以考虑一下过大的输入数据是否是必要的，有没有可能使用较小的输入数据达到目的，以及比正解复杂度稍劣的做法是否有卡掉的必要.

## 数据范围

按照 CF 的要求，数据范围要写在输入格式里，但在国内，数据范围往往是写在题目的最后的.

数据范围中最容易犯的错误就是不完整．输入中的每一个数、每一个字符串都应该有清晰的界定．在上文所给出的输入输出格式示例中就有一些数据范围的正确写法．

数据范围的常见遗漏：

1.「整数」中的「整」．
2. 题面中只说了是「整数」没说是「正整数」，并且数据范围中只有上限没有下限．
3. 字符串没说字符集．
4. 实数没说小数点后位数．
5. 某些变量没有给范围．

你需要保证标程可以通过满足题面所述数据范围的 **任何一组数据**．

> **关于「保证数据随机生成」**
>
> 有的题目中会「保证数据随机生成」，很多时候这样的限制并不是最优的解决方案，因为「随机生成」对数据的限制并不明确，会给判断具体数据范围、提供 hack 数据带来困难．
>
> 一般来说，「保证数据随机生成」可以换成解法所需要的数据性质．例如，随机生成一棵树往往可以换成限制树的高度．
>
> 如果一定要保证数据随机生成，应当指定随机生成的具体操作．例如，生成一棵树是随机选择父亲节点还是随机生成 Prüfer 序列．
>
> 需要注意的是，非确定性算法和依赖于数据随机性的算法是不同的．前者可以对于任意数据都有很高的概率得到正解，而后者是对于大部分的数据能得到正解，对于某些特定的数据则不可能得到正解．

### 样例

样例应当有一定的强度，能够查出一些简单的错误．读错题意的人应当能够通过样例发现自己读错了题意．

有多种操作的题，每种操作都应在样例中出现．

有多种输出的题（如 CF1173A Nauuo and Votes[19]），每种输出都应在样例中出现．例外：实际上不可能无解，但要求判断是否有解的题目．

### 样例解释

题目描述越复杂、越不易理解就越应当有详细的样例解释．

题目难度越简单就越应当有详细的样例解释．

详细的样例解释可以选择配上图片．

较大的样例可以没有样例解释．

为了照顾色觉障碍者，最好不要使颜色成为理解样例解释所必备的．可以用彩色图片来美化样例解释，但如果一定要用颜色传递一些必要的信息，最好不要同时出现红黄或者红绿．

## 时限、空间限制与部分分

时限与空间限制的目的是卡掉复杂度错误的做法．（当然，也是为了防止评测用时过长，如：只对交互次数有限制而对时间复杂度没有限制的交互题也有时间限制．）

因此，原则上时间限制应当选取不使错误做法通过的尽量大的值．

一般地，时限应满足以下要求：

1. 至少为 std 在最坏情况下用时的两倍．
2. 如果比赛允许使用 Java，应使 Java 能够通过．

3. 不应使错误做法通过（实在卡不掉、想放某种错解过除外）.

为了更好地在放大常数做法过的同时卡掉错解，一般可以采用同时增大数据范围和时限的方法. 但要注意，有时正解（由于缓存等玄学问题）会在数据范围增大时有极大的常数增加，此时增大数据范围不一定能够增大正解与错解之间用时的差距.

在有部分分的赛制中，还可以通过设置有梯度的数据、数据范围稍小的数据来使较为优秀的错解和大常数正解不能通过，同时使其获得较高的部分分.

需要注意的是，在数据范围小于 $5 \cdot 10^5$ 时，应当考虑是否能使用 指令集[20] 通过.

一般情况下空间限制应当设置的足够大，除非空间复杂度更优的做法的确十分巧妙，值得卡掉空间复杂度大的做法. 这种情况下可以考虑设置空间限制较松的部分分. 值得注意的是，如果不想卡掉空间消耗较大的做法，数据结构题一般需要设置较大的空间限制.

> 一道好题应该具有它的选拔性质，具有足够的区分度. 应该至少 4 档部分分，让新手可以拿到分，让高手能够展示自己的实力.
>
> ——vfk《UOJ 精神之源流》

部分分一般分为较小数据范围与特殊性质两种.

较小数据范围一般要设置多档，即使你想不到某种复杂度的做法，也可以考虑给这种复杂度一档分. 一般来说，为了避免卡常，可以设置一档极限数据除以二的部分分.

「数据有梯度」最好用多档部分分替代.

特殊性质部分分的设置要依具体题目而定. 理想的特殊性质部分分应当是能够引导选手思考正解的. 与较小数据范围部分分不同，在你不会针对某种特殊性质的做法时，最好不要给这种特殊性质一档分. 例如：「CTS2019」随机立方体[21] 的 $k = 1$ 这档部分分在讲题时就被很多人吐槽，称这档部分分妨碍了思考正解.

如果题目给分方式与默认方式不同（如：在一般的 OI 赛制比赛中绑 subtask 测试），一定要在题面中说明.

不推荐使用「百分之 XX 的数据满足 XX」的说法，尤其是数据范围有多个变量时. 例如，「30% 的数据满足 $n \leq 1000$」和「40% 的数据满足 $m \leq 100$」可能描述了 70% 的数据的性质，也可能只描述了 40% 数据的性质. 一般来说，subtask 或数据范围表格是更好的选择.

## 造数据

数据生成是出题过程中必要的一步，也是对拍时所必需的，掌握一些生成数据的技巧，就能使造数据的过程更加轻松，造出来的数据强度更高.

### 生成随机数据

#### 生成随机数

请参考 随机函数→13.6.1@p3252 页面.

需要特别提醒的是，在生成值域比随机函数返回值更大的数时，请 **不要** 使用 `rand() * rand()` 之类的写法，这样的写法生成的随机数非常不均匀.

另外，出题时推荐使用 testlib→3.9.3@p273 来造数据，可以保证在不同平台上同一个种子生成的随机数相同，并且种子会依据命令行参数自动生成.

### 生成随机排列

可以使用 STL 中的 `std::shuffle` 函数，形如 `std::shuffle(a, a + n, rng)`，这里 `rng` 是一个随机数生成器，比如 `std::mt19937 rng(std::chrono::steady_clock::now().time_since_epoch().count())`.

请 **不要** 使用 `std::random_shuffle`，它在 C++14 中弃用，C++17 中被移除.

### 生成随机区间

常见错误方法：在 $[1,n]$ 中随机生成左端点 $l$，再在 $[l,n]$ 中随机生成右端点 $r$. 这样的话生成的区间会比较靠右.

较为正确的方法（推荐做法）：在 $[1,n]$ 中随机生成两个数，取较小的作为左端点，较大的作为右端点.

真正均匀随机的方法：在 $[0,n]$ 中生成一个随机数 $x$，若 $x=0$，再在 $[1,n]$ 中生成一个随机数 $y$，区间为 $[y,y]$；否则按「较为正确的方法」生成.

### 生成随机树

常用方法是为 $2\sim n$ 的每个节点 $i$ 从 $[1,i-1]$ 中随机选择一个父亲. 这样做的话生成的树不是均匀随机的，期望高度为 $O(\log n)$.

还有一种随机方法：从 $[i\cdot low, i\cdot high]$ 中随机选择 $i$ 的父亲. 若 $low$ 和 $high$ 设置得当，可以造出强度较高的树.

真正均匀随机的方法是利用 Prüfer 序列→11.25@p3033，先生成一个随机 Prüfer 序列，再通过序列生成树. 这样做的话，树的期望高度为 $O(\sqrt{n})$.

除此之外，可以随机一个排列来给节点重编号/打乱边的顺序.

## 构造数据

### 区间相关的题目

常用构造：长度特别小（特殊地，全部为单点）、长度特别大（特殊地，全部为整个序列）.

### 需要分解因数的题目

可重质因数个数尽量多：2 的幂.

去重后质因数个数尽量多：最小的若干个质数相乘.

约数尽量多：可以参考 OEIS 上的 A002182[22] 数列.

### 需要求最大公因数的题目

让需要求最大公因数的两个数为 斐波那契数列→9.13.4@p1629 的相邻两项，可以让欧几里得算法达到最坏时间复杂度.

### 树上问题

常用构造：

- 链
- 菊花
- 完全二叉树
- 将完全二叉树的每个节点替换为一条长为 $\sqrt{n}$ 的链
- 菊花上挂一条链
- 链上挂一些单点
- 一棵高度为 $d$ 且 $d>1$ 的树的根节点有两个儿子，左子树是一条长为 $d-1$ 的链，右子树是一棵高度为 $d-1$ 的这样的树.

如果不是在考场上，还可以使用 Tree-Generator[23] 来生成各种各样的树.

## 批量生成数据

笔者推荐使用命令行参数 + bat/sh 的方法.

例如:

`gen.cpp`:

```cpp
#include "testlib.h"

using namespace std;

int n, m, k;
vector<int> p;

int main(int argc, char* argv[]) {
  registerGen(argc, argv, 1);

  int i;

  n = atoi(argv[1]);
  m = atoi(argv[2]);
  k = rnd.next(1, n);

  for (i = 1; i <= n; ++i) p.push_back(i);

  shuffle(p.begin(), p.end());
  // 使用 rnd.next() 进行 shuffle

  printf("%d %d %d\n", n, m, k);
  for (i = 0; i < n; ++i) {
    printf("%d%c", p[i], " \n"[i == n - 1]);
    // 把字符串当作数组用，中间空格，末尾换行，是一个造数据时常用的技巧
  }

  return 0;
}
```

`gen_scripts.bat`:

```
gen 10 10 > 1.in
gen 1 1 > 2.in
gen 100 200 > 3.in
gen 2000 1000 > 4.in
gen 100000 100000 > 5.in
```

这样做的好处是，对于不同的数据只需要写一个 generator，并且可以方便地修改某个测试点的参数.

## 造数据的要求

数据应当包含各个参数的最小值和最大值.

数据应当包含各种边角情况.

在使用 subtask 时，数据（包括输入、输出）最好覆盖到值域中的各个范围，而不是只有数据范围的最大值.

为了防止针对特殊构造的特判过掉，可以将不同的构造结合在一个测试点中，或者数据的大部分是构造，掺杂小部分的随机.

数据中应当包含各种各样的构造，即使你不知道什么错解会挂在这组构造上.（在按测试点给分的赛制中需要酌情处理.）

当然，如果你已知一个（正常人能想的到、写的出的）正确性有问题的错解，要尽量卡掉它.

需要特别提醒的是，如果有整型溢出的可能，一定要卡掉会溢出的做法. 在有部分分的赛制中，不应使不开 long long 的人得到和暴力一样甚至更低的分数.

如果有 pretests，pretests 应尽量强，（同时尽量少）. 换言之，你需要在 pretests 中（用尽量少的数据组数）包含该题的所有已知叉点.

如果你希望出现少量而非没有 FST，仍然应当保证 pretests 的强度，因为实际比赛中很可能出现你意想不到的错误，导致远远高出预期的 FST 数量.

### 数据的格式

这里提供一些通常情况下输入数据的格式要求，可作为一般情况下的参考：

> 1. 使用测试环境下的换行格式.
> 2. 文件最后一行的末尾有换行符，即整个文件的最后一个字符需要是 `\n`.
> 3. 任何一行的开头和末尾都没有空白字符.
> 4. 连续的空格不超过 1 个.

在 Windows 环境下生成的数据，其换行格式通常为 `\r\n`，而主流测评系统均在 Linux 环境下运行，其换行格式为 `\n`. 若在 Linux 环境下读入 Windows 格式的换行数据，可能会导致读入字符串时换行处理异常，进而导致不同环境下程序运行结果不同；若在 Linux 环境下比较 Linux 环境下生成的输出和 Windows 环境下生成的标准输出，可能由于换行格式不同而导致比较存在差异. 为了保持程序行为一致，所有数据的换行格式必须转换为程序运行环境下的换行格式.

一般可以通过如下方式生成 Linux 格式换行的数据：

1. 直接使用 Linux 环境生成数据.
2. 通过 `dos2unix`[24] 工具对输入输出文件进行转换，此工具包含于 Cygwin, MinGW 等工具链中.
3. 使用二进制方式打开输出文件，并且使用 `\n` 换行格式.
4. 参考 此页面[25] 中 `dos2unix.cpp` 代码自行编写工具.

## Special Judge

SPJ 编写教程→3.8@p261

输出方案题和输出浮点数题是两种较为常见的需要使用 SPJ 的题型，其它题目视情况也需要使用 SPJ. 在 CF 上，所有题目都必须使用基于 testlib 的 checker，例如：题目要求输出若干个整数时，使用 testlib 自带的 ncmp checker，选手可以任意输出空白字符（既可以空格也可以换行）.

checker 一般使用 testlib 编写. 由于 checker 要应对各种各样的不合法输出，需要极强的鲁棒性，不使用 testlib 是很难写好 checker 的.

编写 checker 需要注意以下两点：

1. 你需要应对各种不合法的输出，因此，请检查读入的每个变量是否在合法范围中（`readInt(minvalue, maxvalue)`）. 例如：读入一个在 check 过程中会作为数组下标的变量时必须检查其范围，否则可能引发数组越界，有时这会导致 RE，有时则可能判为 AC.
2. 原则上 checker 中不应检查空白字符（即，不应使用 `readSpace()`、`readEoln()`、`readEof()`，值得一提的是，testlib 会自动检查是否有多余的输出）.

## 题解

题解的目标是让预计会来参加比赛的人都能看懂．所以官方题解详细程度的要求会比一般的题解高．

### 关于部分分

在有部分分的题目中，题解里可以考虑写一写部分分的做法．

### 关于知识点

解题中用到的知识点应当明确指出．对于一些难度和题目难度相当的知识点，最好给出学习该知识点的资料（比如一篇博客的地址）．

### 关于定义

题解中不要凭空冒出来一些概念．

例如：dp 的题解要解释清楚状态的定义．

### 关于细节

具体的实现细节如果比较巧妙最好写出来，否则的话「详见代码」也是可以的．如果「详见代码」的话，最好在代码中加上一定的注释．

### 标程

标程中最好去掉冗余部分．比如，有的题解中保留了完整的 define 模板（为了提高做题速度，包含大量 define 与常用函数，常用于 CF 等在线比赛），并且其中很大一部分都没有用到，这是不好的．

如果涉及到一些题解中没有详细说明的实现细节，最好加上适量的注释．

## 比赛

### 比赛通知中的题目难度需真实

> Remember that authors tend to underestimate the difficulty of their problems.
>
> ——Codeforces PROPOSE A PROBLEM 页面的提醒

出题人很可能错误估计题目的难度，因此，如果要在比赛通知中写上比赛难度，需要谨慎考虑，最好提前请人来验题并进行评估．

### 题目难度的分配

在类国内 OI 的模拟赛中，往往是三道题的整体难度与比赛难度相当即可．

在类 CF/ATC 这种线上赛的比赛中，需要尽量保证难度的递增（虽然由于对难度的误估很多时候都并不能真正做到），并且尽量避免出现大的 difficulty gap．可以通过把一题分为难易两题（两个 subtask）来减少 difficulty gap，但是分 subtask 需要谨慎考虑，也有很多人不喜欢 CF 赛制中的 subtask（Are subtasks evil?[26]），原因包括但不限于：

- 由于赛制原因，可能先做 easy version 再做 hard version 罚时更少而总分更高
- subtask 的赋分往往与题目难度不成正比
- 很多时候 easy version 的题目并不是一道合格的题目（不有趣）
- 很多时候 easy version 的解法对于思考 hard version 的正解没有帮助

### 题目知识点的分配

一场比赛应尽量涵盖较广的知识点（专题训练赛当然除外）．

经典反例：涵盖了动态规划、期望、组合计数、容斥原理、多项式等多种知识点的 CTS2019．

> 我要从五道题里选六道，我也很无奈啊.
>
> ——CTS2019 组题人给出的理由，没有收到足够多的题目投稿

# 出题平台

## Polygon

Polygon 是一个功能非常强大的多人合作出题平台，可以作为在任何网站（使用 package 功能导出到不支持 Polygon 的网站）多人合作出题的首选方案，单人出题（尤其是在不同设备上出题）时也是很不错的选择，使用方法参见 Polygon 简介→3.10@p283.

## Codeforces

Codeforces 是全球最著名的算法竞赛网站之一，题目质量较高，非常适合有一定出题经验并且想进一步提升出题水平、想要出一套高质量题目的出题人. 不足之处是审核速度较慢（一般要几个月），但你也可以在审核期间就开始题目的准备（虽然有题目被否掉导致准备白费了的风险）.

### 出题资格

- 蓝名且参加过至少 25 场 rated 比赛；
- 紫名且参加过至少 15 场 rated 比赛；
- 橙名且参加过至少 5 场 rated 比赛；
- 红名或黑红名.

### 提交比赛申请

有了出题资格后，在侧边栏可以看到 Propose a contest/problems[27] 按钮.

点进去之后，先写一份 contest proposal（在 PROPOSE A CONTEST 里写），然后再写 problem proposal 并添加进比赛里.

题目决定好之后，就可以将 contest proposal open to review（提交审核）了.

### 在 Polygon 上准备题目

参考 Polygon 简介→3.10@p283.

### 与管理之间的联系

与管理联系有两个作用：

1. 加快审核速度.
2. 进入准备阶段后管理会提供建议和帮助.

正规的联系方式是在 proposal system 中以 proposal 的形式提交申请，管理开始审核之后以 comment 的形式在 proposal 的下方进行讨论.

实际上，如果 proposal 长时间没有过审，可以考虑私信联系管理（其实 CF 上写了 "Don't send private messages or emails to coordinators"，但 300iq 在 评论[28] 中表示可以私信他）.

## Comet OJ

Comet OJ 链接[29]

已经不再活跃（截至 2021 年 11 月，最后一场比赛是 2020 年 1 月的）.

出题申请：https://info.cometoj.com/contests/Questionnaire_IssuerInfo/[30]

## CodeChef

印度的算法竞赛平台，有三种赛制：10 天且带 challenge 的 Long Challenge，2.5h 类 ICPC 的 Cook-Off，3h 类 IOI 的 LunchTime.

出题 FAQ：https://www.codechef.com/wiki/faq-problem-setters[31]

出题指南：https://www.codechef.com/problemsetting[32]

### AtCoder

日本的算法竞赛平台，出题联系方式：contest@atcoder.jp[33].

### UOJ & LOJ

比赛不多的国内 OJ.

### 洛谷

参与出题工作人员需要有一定的奖项认证等级，创建比赛后由负责人在 工单系统[34] 中提交申请.

公开赛规范：https://help.luogu.com.cn/rules/academic/opencontest-standard[35]

## 参考资料

1. vfk《UOJ 精神之源流》[36]
2. 王天懿《论偏题的危害》[37]
3. CF 出题人须知[38]（国内可访问的图片版[39]）
4. CF 出题人的自我修养[40]

本文由作者本人自 ouuan 的出题规范[41] 搬运而来并有所修改、补充.

## 外部参考资料


[1] vfk《UOJ 精神之源流》

[2] 「XR-1」柯南家族

[3] 「模板」树上后缀排序

[4] 王天懿《论偏题的危害》

[5] NOIP2017 小凯的疑惑

[6] 王天懿《论偏题的危害》

[7] 「清华集训 2015」多边形下海

[8] 王天懿《论偏题的危害》

[9] 「SDOI2010」猪国杀

[10] 「集训队互测 2015」未来程序·改

[11] LaTeX 数学公式大全

[12] LaTeX 各种命令，符号

[13] CF1172D Nauuo and Portals

[14] CF1173A Nauuo and Votes

[15] CF1172D Nauuo and Portals

[16] 「GuOJ Round #1」琪露诺的冰雪宴会

[17] CF1423(4)N Bubblesquare Tokens

[18] CF 出题人须知

[19] CF1173A Nauuo and Votes

[20] 指令集

[21] 「CTS2019」随机立方体

[22] A002182

[23] Tree-Generator

[24] `dos2unix`

[25] 此页面

[26] Are subtasks evil?

[27] Propose a contest/problems

[28] 评论

[29] Comet OJ 链接

[30] https://info.cometoj.com/contests/Questionnaire_IssuerInfo/

[31] https://www.codechef.com/wiki/faq-problem-setters

[32] https://www.codechef.com/problemsetting

[33] contest@atcoder.jp

[34] 工单系统 

[35] https://help.luogu.com.cn/rules/academic/opencontest-standard 

[36] vfk《UOJ 精神之源流》 

[37] 王天懿《论偏题的危害》 

[38] CF 出题人须知 

[39] 国内可访问的图片版 

[40] CF 出题人的自我修养 

[41] ouuan 的出题规范 

# 第三章

# 工具软件

# 3.1 工具软件简介

本章主要介绍与竞赛有关的工具软件，包括一些代码编辑器的介绍和 OJ 相关工具.

程序是解决 OI 问题的工具，熟练运用代码编辑器是学习 OI 的前提.

了解 OJ 相关的工具能让 OI 之旅更加舒适、便捷.

# 3.2 代码编辑工具

## 3.2.1 Vim

> **Authors:** Enter-tainer, ouuan, Xeonacid, Ir1d, partychicken, ChungZH, LuoshuiTianyi, Kewth, s0cks5, Doveqise, StudyingFather, SukkaW, SodaCris, SkyeYoung, 383494, danielqfmai

Vim - 无处不在的文本编辑器.

### 简介

Vim 是从 vi 发展出来的一个文本编辑器. 其代码补完、编译及错误跳转等方便编程的功能特别丰富，在程序员群体中被广泛使用.

### 安装

Linux 系统通常自带 Vim，打开终端输入 `vim` 即可启用.

若需手动安装，Vim 的 官方网站[1] 提供了下载的 说明文档[2]，按照需求编译安装即可.

### Vim 的模式与常用键位

Vim 的基础操作在 Vim 自带的教程里将会讲述. 打开终端输入 `vimtutor` 即可进入教程.

这些操作通常需要二三十分钟来大致熟悉.

#### 命令模式 (Command Mode)

进入 Vim 后的默认模式.

此状态下敲击键盘动作会被 Vim 识别为命令，而非输入字符，比如我们此时按下 `i`，并不会输入一个字符，`i` 被当作了一个命令.

Vim 的方向键是 `↑`、`↓`、`←`、`→`，或者 `h`、`j`、`k`、`l`.

```
        ↑(k)
        ^
(h)← <    > →(l)
        v
        ↓(j)
```

以下是命令模式常用的命令:

- `i` 切换到输入模式，在光标当前位置开始输入文本. 按 `Esc` 键可回到普通模式.
- `x` 用于删除光标后的一个字符.
- `:` 切换到底线命令模式，以在最底一行输入命令.
- `a` 切换到输入模式，在光标后开始输入文本.
- `o` 切换到输入模式，在光标下插入新的一行；`O` 切换到输入模式，在光标上插入新的一行.
- `p` 粘贴剪贴板内容到光标下方；`P` 粘贴剪贴板内容到光标上方.
- `dd` 删除光标所在的一整行.
- `d` 命令也是删除，通常配合其他键使用.
- `u` 撤销上一次对文本的更改.

- `y` 命令可以复制被选中的区域．需要按 `v` 进入可视模式操作．
- `yy` 复制当前行．
- `Ctrl + r` 重做上次撤销的操作．
- `:w` 保存文件，常配合 q 保存退出．
- `:q` 退出 Vim．
- `:q!` 强制退出 Vim，不保存修改．

部分其他命令：

- `c` 命令用于修改，相当于 `di`．
- `=` 命令可以以默认格式对选中行应用自动缩进．
- `==` 自动缩进当前行．
- `.` 命令可以重复上次执行的命令．
- `gg` 命令可跳至代码的开头；`G` 命令可跳至代码最后一行的开头；`G` 命令前加数字可跳至指定行．
- `w` 可以跳到下个单词的开头；`e` 可以跳到当前单词或下一单词的结尾；`b` 可以跳到当前单词或上一单词的开头；`0` 可以跳至行首；`$` 可以跳至行尾．`w`、`e`、`-0`、`$` 还可以与其他命令组合，比如 `de`、`dw`、`d0` 和 `d$` 分别对应删至单词尾、删至下个单词头、删至行首和删至行尾．

命令模式下按 `/`，下方即会出现查找框，输入需要查找的字符，按回车后就能查看搜索结果．如果有多个查找结果，按 `n` 即可跳至下一个查找结果；按 `N` 可跳至上一个．

命令模式下按 `*` 可以查找当前光标下的单词．

在输入某个命令前，输入一个数字 n 的话，命令就会重复 n 次．

### 输入模式 (Insert Mode)

在命令模式下按下 `i` 就进入了输入模式，按 `Esc` 键可以返回到命令模式．

在输入模式中，可以使用以下按键：

- 字符按键以及 `Shift` 组合，输入字符
- `ENTER`，回车键，换行
- `BACK SPACE`，退格键，删除光标前一个字符
- `DEL`，删除键，删除光标后一个字符
- 方向键，在文本中移动光标
- `HOME/END`，移动光标到行首/行尾
- `Page Up/Page Down`，上/下翻页
- `Insert`，切换光标为输入/替换模式，光标将变成竖线/下划线
- `ESC`，退出输入模式，切换到命令模式

在输入模式下按 `Ctrl` + `o` 即可进入「输入 - 命令模式」，执行完一次操作后又会自动回到输入模式．

### 底线命令行模式

命令模式下按 `:`，进入底线命令模式．

底线命令模式可以输入单个或多个字符的命令，可用的命令非常多．

在底线命令模式中，基本的命令有：

- `:help`/`:h` 查看英文版 Vim 在线帮助文档．
- `:w` 保存文件．
- `:q` 退出 Vim．
- `:wq` 保存文件，退出 Vim．
- `:q!`/`:!q` 强制退出 Vim，不保存修改．
- `:e filename` 可以打开当前目录下的指定文件．
- `:s` 命令是替换．

```
" 把当前行第一个匹配的 str1 替换成 str2
:s/str1/str2/
" 把当前行所有的 str1 替换成 str2
:s/str1/str2/g
" 把当前行所有的 str1 替换成 str2，在替换前询问
:s/str1/str2/gc
" 把第 x1 行至 x2 行中，每一行第一个匹配的 str1 替换成 str2
:x1,x2 s/str1/str2/
" 把第 x1 行至 x2 行中所有的 str1 替换成 str2
:x1,x2 s/str1/str2/g
" 第 x1 行至 x2 行中所有的 str1 替换成 str2，在替换前询问
:x1,x2 s/str1/str2/gc
" 把所有行第一个匹配的 str1 替换成 str2
:%s/str1/str2/
" 把全文件所有的 str1 替换成 str2
:%s/str1/str2/g
" 把全文件所有的 str1 替换成 str2，在替换前询问
:%s/str1/str2/gc
```

如果命令形式是 `:! command`，则命令将在 bash 终端执行.

按 `Esc` 键可以退出底线命令模式.

### 可视模式 (Visual mode)

按 `v` 进入可视模式，多用于选中区域. 按 `V`（`Shift+v`）进入行可视模式，用于选中行.

按 `Ctrl` + `v` 或 `Ctrl` + `q` 进入块可视模式 (visual block).

进入块可视模式后，按 `I` 或 `A` 进入插入模式（相当于 `i` 和 `a`），退出插入模式后对本行所做的改动将被应用到选中的每一行同一位置. 常用于批量添加注释.

选中后输入 `y` 或 `d` 亦可执行相应命令.

三种可视模式可以通过按键相互转化.

## Vim 的快捷键

可参考 史上最全 Vim 快捷键键位图—入门到进阶[3]

## 进阶知识

### `.` 命令

Vim 的使用者不可避免地会抗拒重复的文本修改，因为 Vim 注定比其他编辑器会多出两次按键——`Esc` 与 `i`. 但是，Vim 其实提供了重复命令 `.`，它适用于重复的添加、修改、删除文本操作.

`.` 命令可以重复上次执行的命令. 但是这个「命令」并不只限于单一的命令，它也可以是 `数字 + 命令` 的组合；`进入插入模式 + 输入文本 + Esc` 也是命令的一种. 所以，适当使用 `.` 命令才能达到最高的效率.

例如，如下代码的每一行末尾都少了分号:

```
int a, b
cin >> a >> b
cout << a + b
return 0
```

将 `.` 与搭配移动到行尾插入命令 `A` 使用，就能高效地补上末尾的分号.

```
A;<Esc>
" 重复下面的命令
```

```
3 j.
```

再例如，如下代码中，后面五个赋值语句的数组名全部写错了：

```
1 int check() {
2   book[1] = 1, book[2] = 1, book[3] = 1, bok[1] = 1, bok[2] = 1, bok[3] = 1,
3   bok[4] = 1, bok[5] = 1;
4   return 0;
5 }
```

一个个改过于麻烦，而命令行模式的 `s` 命令又会全部改掉.

第一种改法是搭配普通模式下的 `s` 命令（删除光标处字符并进入插入模式）使用. 来到第一个错误的数组名首字母处，按下 `3s/cw`，输入正确的数组名并退出. 之后把光标一个个移过去，再使用 `.` 命令.

第二种比较节省时间的改法是利用查找模式修改. 键入 `/bok`，接着按下回车，并使用 `n` 键来到第一个错误的数组名首字母处，键入 `3s 新数组名 <Esc>`，最后重复 `n.`.

第三种改法是简易查找命令 `f`. 在一行中普通模式下，`f + 单个字符` 即可查找此行中出现的这个字符并将光标移至字符处；按 `;` 查找下一个，`,` 查找上一个. 所以对于上面的代码，只需键入 `fb;;;` 之后进入插入模式修改，然后 `;.` 即可. 这种改法适用于只需行内移动的情况.

### 宏

Vim 的宏功能可以重复任意长的命令.

使用宏之前要先「录制」，即把一串按键操作录下来再回放，这样就达到了重复的效果. 录制的方法很简单，普通模式下键入 `q` 开始录制. 下一步，为录制的宏指定一个执行的命令键，可以按下 26 个字母中的任意一个来指定. 这时左下方会显示 `记录中 @刚刚选择的字母`. 然后就可以开始录制命令了. 同理，普通模式下按 `q` 暂停录制.

使用方法为按下 `:` 进入命令行模式，键入 `@选择的记录字母`，然后之前录制的命令就被调用了.

将 `.` 和宏组合，即录制宏 → 调用宏 →`.` 重复命令 → 数字 +`.`，可以达到非常高的效率.

### normal 命令

该命令与普通模式有关，效果是在指定行重复命令.

按 `:` 进入命令行模式，输入如下命令：

```
1 :a,b normal command
```

或者：

```
1 :a,b norm command
```

以上命令的意思是在普通模式下，对 a~b 行执行 `command` 命令.

由于 `normal` 命令可以被 `.` 命令重复调用，且其易于理解，它的使用频率甚至更高于宏.

### 数字 +`.`+ 宏 + normal

以上三种命令可以组合使用. 例如：

> 我下载了一本书，我需要它的每一个章节都变成「标题」，以方便转换成 mobi 之类的格式，或者方便生成 TOC 目录跳转，怎么办呢?

以下是用 Vim 处理的过程：

1. 按下 `/` 调出查找框，输入正则表达式进行查找；
2. 用 `q` 命令开始录制宏；
3. 键入 `I#` 命令，然后按下 `ESC`；

4. 用 `q` 命令结束宏录制；
5. 键入 `normal n@字母` 转到下一处并重复上一步操作；
6. 键入 `数字 + .` 多次重复.

### 外部链接

- Vim 官网[4]
- 原作者提供的配置[5]
- Vim 调试：termdebug 入门[6]
- Vim scripting cheatsheet[7]
- Learn Vimscript the Hard Way[8]
- Linux vi/vim | 菜鸟教程[9]

### 外部参考资料

[1] 官方网站 

[2] 说明文档 

[3] 史上最全 Vim 快捷键键位图—入门到进阶 

[4] Vim 官网 

[5] 原作者提供的配置 

[6] Vim 调试：termdebug 入门 

[7] Vim scripting cheatsheet 

[8] Learn Vimscript the Hard Way 

[9] Linux vi/vim | 菜鸟教程 

## 3.2.2 Emacs

**Authors:** ouuan, akakw1, Ir1d, partychicken, Xeonacid

本页面为 Emacs 的入门教程.

15 分钟入门 Emacs.

### 简介

Emacs 是一款非常容易上手的编辑器，只需要简短的几行配置就能使用，但是想要非常熟练地使用 Emacs 进行各项工作还是需要一定的时间.

作为入门教程，这里仅介绍 Emacs 的基本功能，以及较方便地用 Emacs 编写、调试代码的方法.

### 入门

#### 命令

命令在 Emacs 中有很大的作用.

使用 Application 键[1]（Windows 系统下 Emacs 未指定这个键，需要手动设置）或者快捷键 M-x（`Alt` + `x`）可以打开命令输入，输入完按下回车可以执行命令.

通常使用 `es` 或者 `eshell` 命令来打开 Eshell（类似一个终端）.

输入命令通常可以用快捷键代替.

#### 缓冲 (buffer)

缓冲即打开的文件和进程，在不保存的情况下，在缓冲中修改并不会修改到文件.

在缓冲区的底部点击缓冲的名字或者使用快捷键可以切换缓冲.

[1] 该键的作用是调出鼠标右键菜单，一般为右 `Ctrl` 左边的第一个键.

### 编译、调试和运行

编译和调试功能的入口在顶部菜单栏的 Tools 下拉栏. 使用者也可以通过命令或者自定义快捷键使用编译和调试功能.

可以使用终端或 Eshell 运行程序.

按下 Tools 中的调试 (gud-gdb) 后，输入程序名（一般会自动输好，但如果中途将程序另存为或者打开了两个需要调试的程序，**自动输好的文件名可能会有误**）即可开始调试.

### 分屏

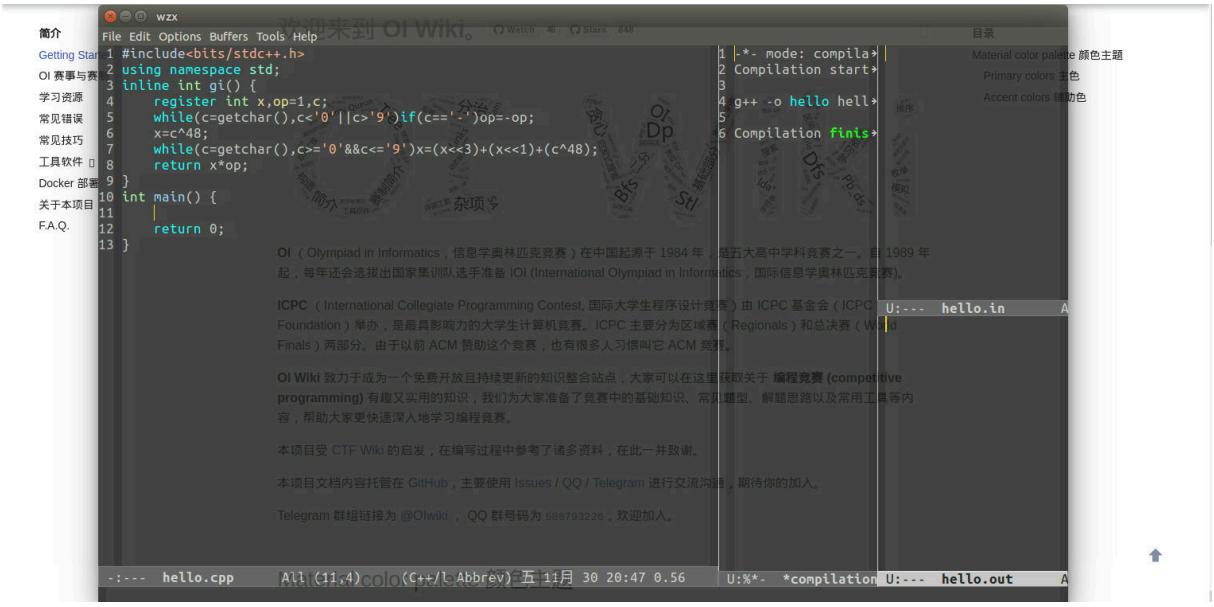
这个功能能让使用者同时查看各个缓冲的内容，而不需要来回切换缓冲，方便测试、调试代码.

分屏功能可以同时显示多个窗口，用鼠标拖动窗口的边缘可以缩放窗口.

几个快捷键:

- 删除分屏 "C-x 0"：将这个分屏删去
- 横向分屏 "C-x 3"：将这个分屏横向分成两半
- 纵向分屏 "C-x 2"：将这个分屏纵向分成两半

推荐的窗口布局为将窗口分为四块: 先横向分, 调整一块的宽度约为 3/4 屏, 作为编辑窗口. 将另一块横向分, 一块作为调试和编译信息显示的窗口, 另一块再纵向分, 一块打开输入文件, 一块打开输出文件.




### 快捷键

Emacs 拥有极为丰富的快捷键，可以大幅提高工作的效率. 使用者可以在配置中自定义快捷键或者设置快捷键的映射.

由于快捷键过多，所以 Emacs 快捷键的使用与操作系统不同.

为了方便描述，做如下约定:

| 字符 | 键位 |
| --- | --- |
| C | Ctrl |
| M | Alt |
| ? | 任意键位 |

一般有以下三种:

- F?、ESC：直接按下对应的功能键.

- `M-?`、`C-?`、`C-M-?`：按下 `Alt` 或者 `Ctrl` 的同时按下 `?`.
- `? ?`：先按下第一个 `?` 代表的键，松开再按下第二个 `?` 代表的键.

下面是一些常用的快捷键：

- `C-x h`：全选
- `C-x left`、`C-x right`：切换到上/下一个缓冲
- `C-x d`：打开一个目录
- `C-x C-f`：打开一个文件（如果不存在文件则新建文件）

## 个性化

刚安装好的 Emacs 外观难看且不好使用，因此需要对其进行个性化设置.

由于配置不好记，所以部分可以直接设置的部分建议不要记配置.

### 直接设置

- Options：`Highlight Matching Parentheses` 高亮匹配括号
- Options：`Blink Cursor` 设置光标闪烁
- Options Show/Hide：`Tool Bar` 显示/不显示工具栏（默认显示，建议不显示）
- Options：`Use CUA Keys` 勾选后可以使用 Ctrl + C,Ctrl + V 等快捷键进行复制粘贴
- Options Customize-Emacs：`Custom Theme` 选择配色方案，选择完后需要点击保存
- Options：`Save Options` **保存配置**

### 配置

在 home 目录下显示隐藏文件（Windows 系统在 **用户目录** 的 `AppData\Roaming` 目录下），".emacs" 就是配置文件（如果没有说明之前没保存），打开修改即可. 如果 Emacs 已打开，则需要重启 Emacs，配置才能生效.

考场推荐的配置如下.

```
;;设置一键编译 可以自行添加参数 难背考场不建议使用 不建议依赖一键编译
(defun compile-file ()(interactive)(compile (format "g++ -o %s %s -g -lm -Wall" (file-name-sans-extension (buffer-name))(buffer-name))))
(global-set-key [f9] 'compile-file)
;;;;设置编译快捷键（如果设置了一键编译不要与一键编译冲突）
;;(global-set-key [f9] 'compile)

(global-set-key (kbd "C-a") 'mark-whole-buffer) ;;全选快捷键
(global-set-key (kbd "C-z") 'undo) ;;撤销快捷键
(global-set-key [f10] 'gud-gdb) ;;GDB调试快捷键
(global-set-key (kbd "RET") 'newline-and-indent) ;;换行自动缩进
(global-set-key (kbd "C-s") 'save-buffer) ;;设置保存快捷键
(setq-default kill-ring-max 65535) ;;扩大可撤销记录

;;C++ 代码风格 一般控制缩进规则
;;;"bsd" 所有大括号换行
;;;"java" 所有大括号不换行. else 接在右大括号后面
;;;"awk" 只有命名空间旁、定义类、定义函数时的大括号换行. else 接在右大括号后面
;;;"linux" 只有命名空间旁、定义类、定义函数时的大括号换行. else 接在右大括号后面. 一般来说，这个风格应该有 8 格的空格缩进
(setq-default c-default-style "awk")
```

> **完整配置**

```
;;设置一键编译
(defun compile-file ()(interactive)(compile (format "g++ -o %s %s -g -lm -Wall" (file-name-sans-extension (buffer-name))(buffer-name))))
(global-set-key [f9] 'compile-file)
;;;;设置编译快捷键（如果设置了一键编译不要与一键编译冲突）
;;(global-set-key [f9] 'compile)

;;考场必备
(global-set-key (kbd "C-a") 'mark-whole-buffer) ;;全选快捷键
(global-set-key (kbd "C-z") 'undo) ;;撤销快捷键
(global-set-key [f10] 'gud-gdb) ;;GDB调试快捷键
(global-set-key (kbd "RET") 'newline-and-indent) ;;换行自动缩进
(global-set-key (kbd "C-s") 'save-buffer) ;;设置保存快捷键
(setq-default kill-ring-max 65535) ;;扩大可撤销记录
;;(define-key key-translation-map [apps] (kbd "M-x")) ;; windows 系统下设置命令快捷键

;;设置缩进
;;;C++ 代码缩进长度.
(setq-default c-basic-offset 4)
;;;使用 tab 缩进
(setq-default indent-tabs-mode t)
;;;tab 的长度. 务必和缩进长度一致
(setq-default default-tab-width 4)
(setq-default tab-width 4)

;;设置默认编码环境
(set-language-environment "UTF-8")
(set-default-coding-systems 'utf-8)

;;不显示欢迎页面
(setq-default inhibit-startup-screen t)

;;设置标题
(setq-default frame-title-format "")

;;显示行号
(global-display-line-numbers-mode t)

;;高亮
(global-hl-line-mode 1);;高亮当前行
(show-paren-mode t);;高亮匹配括号
(global-font-lock-mode t);;语法高亮

;;允许emacs和外部其他程序的粘贴 好像默认允许
(setq-default x-select-enable-clipboard t)

;;设置字体是 Ubuntu Mono 的 16 号，如果字体不存在会报错
(set-face-attribute 'default nil :font "Ubuntu Mono-16")
;(set-face-attribute 'default nil :font "Consolas-16") ;; windows 系统请用这条

;;鼠标滚轮支持
```

```
(mouse-wheel-mode t)

;;设置光标形状为竖线（默认为方块）
(setq-default cursor-type 'bar)

;;回答 yes/no 改成回答 y/n
(fset 'yes-or-no-p 'y-or-n-p)

;;透明度
(set-frame-parameter (selected-frame) 'alpha (list 85 60))
(add-to-list 'default-frame-alist (cons 'alpha (list 85 60)))

;;减少页面滚动的行数，防止整页地滚动
(setq-default scroll-margin 3 scroll-conservatively 10000)

;;优化文件树结构
(ido-mode t)

;;配色方案
(setq default-frame-alist
         '((vertical-scroll-bars)
           (top . 25)
           (left . 45)
           (width . 120)
           (height . 40)
           (background-color . "grey15")
           (foreground-color . "grey")
           (cursor-color . "gold1")
           (mouse-color . "gold1")
           (tool-bar-lines . 0)
           (menu-bar-lines . 1)
           (scroll-bar-lines . 0)
           (right-fringe)
           (left-fringe)))

(set-face-background 'highlight "gray5")
(set-face-foreground 'region "cyan")
(set-face-background 'region "blue")
(set-face-foreground 'secondary-selection "skyblue")
(set-face-background 'secondary-selection "darkblue")
(set-cursor-color "wheat")
(set-mouse-color "wheat")

(custom-set-variables
 '(ansi-color-faces-vector
   [default default default italic underline success warning error])
;;启动 Ctrl-x Ctrl-c Ctrl-v = 剪切 复制 粘贴
 '(cua-mode t nil (cua-base))
 '(show-paren-mode t)
;;隐藏工具栏
 '(tool-bar-mode nil))
;;关闭光标闪烁
```

```
103  '(blink-cursor-mode nil)
104 (custom-set-faces)
```

### 拓展阅读

要以终端模式启动 Emacs，在启动时添加参数 `-nw`. Emacs 有多种变体，如采用 native-comp 来减少延迟的 GCC Emacs[1] 及其纯 GTK 版本变体、针对 macOS 优化的 Emacs Macport.

Emacs 有中心化的软件仓库，配置后可通过 `M-x package-install` 来安装插件. 使用 镜像站[2] 可以加快下载速度.

Emacs 可以使用语言服务器（Language Server Protocol）来提高编辑体验，目前推荐的 C++ 后端是 Clangd[3]. 前端可以采用 Eglot[4] 或 Emacs LSP[5]，参阅 此条目[6] 可能对选择前端有所帮助.

拓展名为 `.org` 的 Org Mode 文档可以通过 Pandoc[7] 转换为 Markdown 文档.

### 参考资料与注释

[1] GCC Emacs

[2] 镜像站

[3] Clangd

[4] Eglot

[5] Emacs LSP

[6] 此条目

[7] Pandoc

## 3.2.3 VS Code

> **Authors:** NachtgeistW, Ir1d, ouuan, Enter-tainer, Xeonacid, ChungZH, keepthethink, abc1763613206, partychicken, Chrogeek, xkww3n, HeliumOI, Pinghigh, xiaofu-15191, Sekakou, fuxianhu

### 简介

Visual Studio Code（以下简称 VS Code）是一个由微软开发，同时支持 Windows、Linux 和 macOS 等操作系统且开放源代码的代码编辑器. 它是用 TypeScript 编写的，并且采用 Electron 架构. 它带有对 JavaScript、TypeScript 和 Node.js 的内置支持，并为其他语言（如 C、C++、Java、Python、PHP、Go）提供了丰富的扩展生态系统.

官网：Visual Studio Code - The open source AI code editor[1]

在阅读以下内容前，建议你先简单了解 VS Code 的基本使用方法：开始使用 Get started[2].

### 使用 C/C++ Compile Run 扩展运行代码

C/C++ Compile Run 是一个专注于 C/C++ 单文件编译与运行的插件. 它免去了传统 C/C++ 插件中繁琐的配置步骤，非常适合编程竞赛的需要，上手迅速，操作简单.

打开 VS Code，点击左侧边栏的「扩展」图标（或按下 `Ctrl` + `Shift` + `X`），在搜索框中输入 `C++`，找到 C/C++ Compile Run，点击「安装/Install」即可.




安装完成后，无需额外配置，插件会根据环境变量自动适配已配置好的 MinGW 编译器.

打开需要运行的文件，点击右上角的三角图标即可运行代码.

快捷键:

- F6 - 编译并在 VS Code 内置集成终端中运行
- F7 - 以自定义参数编译并以自定义参数在 VS Code 内置集成终端运行
- F8 - 编译并在外部终端中运行

## 使用 Code Runner 扩展运行代码

VS Code 安装并配置扩展后可实现对 C/C++ 的支持，但配置过程比较复杂. 一个简单的编译与运行 C++ 程序的方案是安装 Code Runner 扩展.

Code Runner 是一个可以一键运行代码的扩展，在工程上一般用来验证代码片段，支持 Node.js、Python、C、C++、Java、PHP、Perl、Ruby、Go 等 40 多种语言.

安装的方式是在扩展商店搜索 Code Runner 并点击 Install；或者前往 Marketplace[3] 并点击 Install，浏览器会自动打开 VS Code 并进行安装.



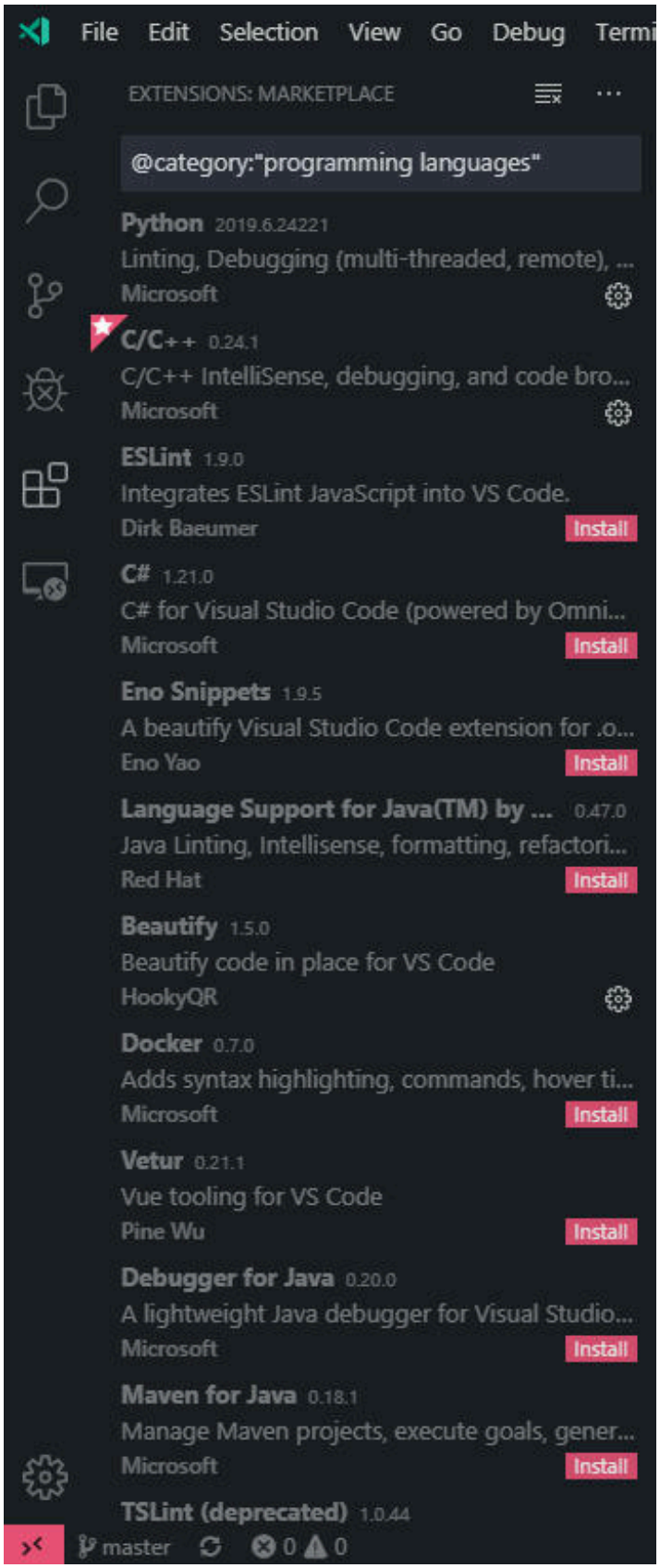
安装完成后，打开需要运行的文件，点击右上角的小三角图标即可运行代码；按下快捷键 `Ctrl` + `Alt` + `N`（在 macOS 下是 `Control` + `Option` + `N`）也可以得到同样的效果.

> ⚠ **Warning**
>
> 如果安装了 VS Code 与 Code Runner 后，代码仍然无法运行，很有可能是因为系统尚未安装 C/C++ 的运行环境，参考 Hello, World! 页面→4.2.1@p322 以安装.
>
> 记得勾选设置中的 Run In Terminal 选项，如图：
>
> 
> 

## 使用 C/C++ 扩展编译并调试/智能补全代码

### 安装扩展

在 VS Code 中打开扩展商店，在搜索栏中输入 `C++` 或者 `@category:"programming languages"`，然后找到 C/C++，点击 Install 安装扩展.


File Edit Selection View Go Debug Termi
EXTENSIONS: MARKETPLACE
@category:"programming languages"
Python 2019.6.24221
Linting, Debugging (multi-threaded, remote), ...
Microsoft
C/C++ 0.24.1
C/C++ IntelliSense, debugging, and code bro...
Microsoft
ESLint 1.9.0
Integrates ESLint JavaScript into VS Code.
Dirk Baeumer
Install
C# 1.21.0
C# for Visual Studio Code (powered by Omni...
Microsoft
Install
Eno Snippets 1.9.5
A beautify Visual Studio Code extension for .o...
Eno Yao
Install
Language Support for Java(TM) by ... 0.47.0
Java Linting, Intellisense, formatting, refactori...
Red Hat
Install
Beautify 1.5.0
Beautify code in place for VS Code
HookyQR
Docker 0.7.0
Adds syntax highlighting, commands, hover ti...
Microsoft
Install
Vetur 0.21.1
Vue tooling for VS Code
Pine Wu
Install
Debugger for Java 0.20.0
A lightweight Java debugger for Visual Studio...
Microsoft
Install
Maven for Java 0.18.1
Manage Maven projects, execute goals, gener...
Microsoft
Install
TSLint (deprecated) 1.0.44
master 0 0

> **⚠ Warning**
>
> 在配置前，请确保系统已经安装了 G++ 或 Clang，并已添加到了环境变量 `PATH` 中．请使用 CMD 或者 PowerShell，而不是 Git Bash 作为集成终端．

### 配置 GDB/LLDB 调试器

#### GDB

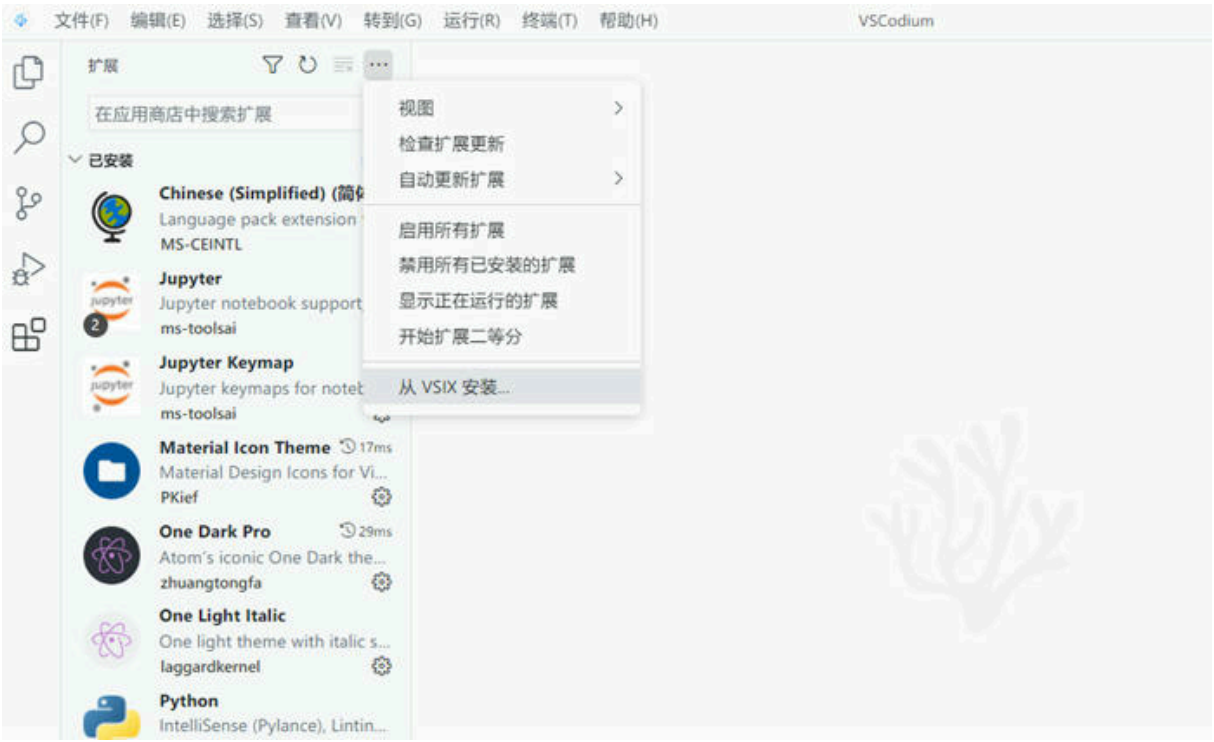
在 VS Code 中新建一份 C++ 代码文件，按照 C++ 语法写入一些内容（如 `int main(){}`），保存并按下 `F5`，进入调试模式．如果出现了「选择调试器」的提示，选择 `C++ (GDB/LLDB)`．在「选择配置」中，G++ 用户选择 `g++.exe - 生成和调试活动文件`；Clang 用户选择 `clang++ - 生成和调试活动文件`．

> **⚠ Warning**
>
> 配置名称并非固定，而是可以自定义的．不同的操作系统可能具有不同的配置名称．

完成后，VS Code 将自动完成初始化操作在下方的集成终端中启动调试．至此，GDB 所有的配置流程已经完毕．

#### LLDB

如果需要采用 LLDB，需要安装另外一款扩展[2]——CodeLLDB[4]．从该项目的 Release 页面下载 .vsix 文件后[3]，从 VS Code 的扩展页面安装．




先按照上文 GDB 的配置过程操作一遍，然后删除 `.vscode/launch.json`，按下 `F5`，选择 `LLDB`，再把 `launch.json` 中的 `${workspaceFolder}/<executable file>` 更改为 `${fileDirname}/${fileBasenameNoExtension}` 即可．

---

[2] VS Code 的 C/C++ 扩展如果选择 lldb 作调试器，则会默认采用 lldb-mi 程序，而它已经被 LLVM 开发团队从项目中分离出来，需要自己编译该程序．而它本身就有一些 bug，使用体验和方便程度都不如 CodeLLDB 扩展．

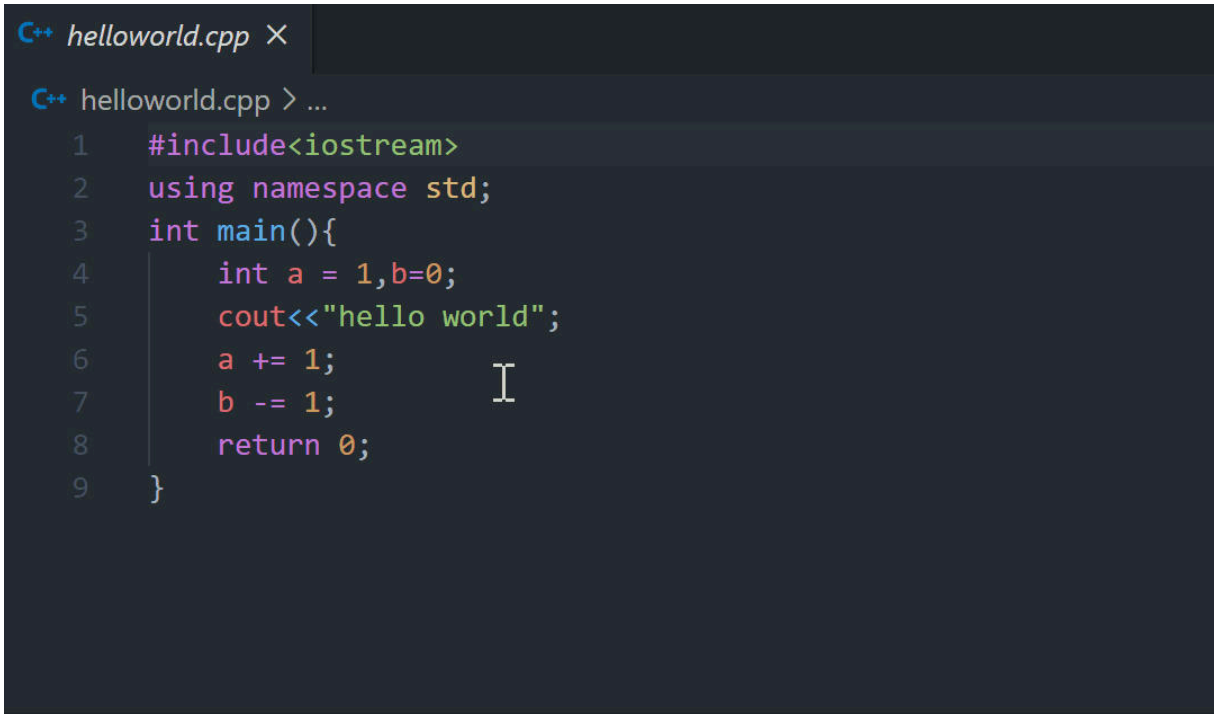
[3] 从扩展商店安装 CodeLLDB 后它会再从 GitHub 下载本体，下载速度奇慢，有时下载出错，所以最好直接下载本体然后安装．更新也可直接按照以上步骤下载安装．

至此，LLDB 配置完成．再次按下 F5 即可看到软件下方的调试信息．

若要在以后使用 VS Code 编译并调试代码，所有的源代码都需要保存至这个文件夹内．若要编译并调试其他文件夹中存放的代码，需要重新执行上述步骤（或将旧文件夹内的 .vscode 子文件夹复制到新文件夹内）．

### 开始调试代码

使用 VS Code 打开一份代码，将鼠标悬停在行数左侧的空白区域，并单击出现的红点即可为该行代码设置断点．再次单击可取消设置断点．




按下 F5 进入调试模式，编辑器上方会出现一个调试工具栏，四个蓝色按钮从左至右分别代表 GDB 中的 continue,next,step 和 until：



如果编辑器未自动跳转，点击左侧工具栏中的「调试」图标进入调试窗口，即可在左侧看到变量的值．

在「监视」中，你可以输入表达式，在每一次进行 next 或 step 等操作时都会重新求值并显示．

在「调用堆栈」中，你可以看见当前函数执行的栈帧．

> **Tip**
>
> 你可以参照 GDB 官方文档[5] 来查看某个数组一段区间内的内容．

在调试模式中，编辑器将以黄色底色显示下一步将要执行的代码．

### 配置 IntelliSense

用于调整 VS Code 的智能补全．

如果你使用 Clang 编译器，在「IntelliSense 模式」中选择 clang-x64 而非默认的 msvc-x64；如果你使用 G++ 编译器，选择 gcc-x64 以使用自动补全等功能．否则会得到「IntelliSense 模式 msvc-x64 与编译器路径不兼容．」的错误．




## 配置 clangd

> **Warning**
>
> 由于功能冲突，安装 clangd 扩展后 C/C++ 扩展的 IntelliSense 功能将被禁用（调试等功能仍然使用 C/C++ 扩展）．如果 clangd 扩展的功能出现问题，可以查看是否禁用了 C/C++ 扩展的 IntelliSense 功能．

### clangd 简介

LLVM 官网上对 clangd 的介绍是这样的：

> Clangd is an implementation of the Language Server Protocol leveraging Clang. Clangd's goal is to provide language "smartness" features like code completion, find references, etc. for clients such as C/C++ Editors.

简单来说，clangd 是 Clang 对语言服务器协定（Language Server Protocol）的实现，提供了一些智能的特性，例如全项目索引、代码跳转、变量重命名、更快的代码补全、提示信息、格式化代码等，并且能利用 LSP 与 Vim、Emacs、VSCode 等编辑器协作．虽然官方给出的定义是 LSP 的实现，但 clangd 的功能更接近语言服务器（Language Server）而不仅仅只是协议本身．

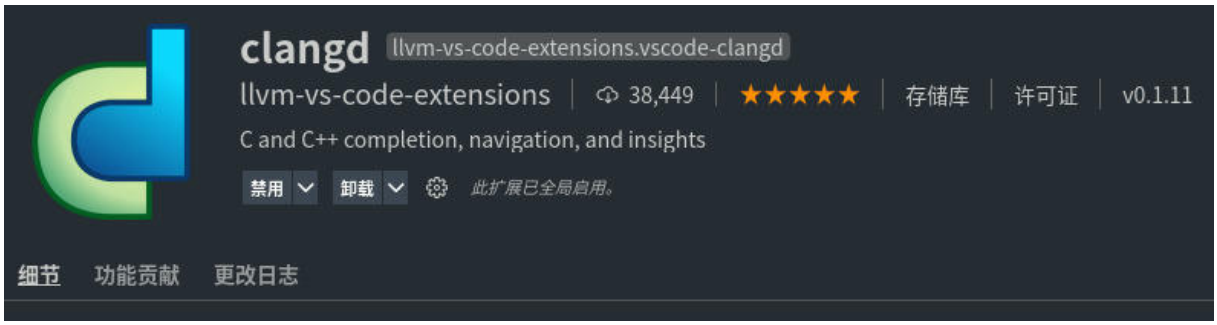
VS Code 的 C/C++ 扩展也有自动补全等功能，但在提示信息的易读程度的准确度等方面与 clangd 相比稍逊一筹，所以我们有时会使用 clangd 代替 C/C++ 扩展来实现代码自动补全等功能．

### 安装

参见 Getting started[6]．

### VS Code 扩展

打开 VS Code 扩展商店，在搜索栏中输入 `clangd` 找到 clangd 扩展并安装




如果下方弹出 clangd 要求关闭 Intellisense 的对话框，点击 "Disable Intellisense"，重新加载工作区，就可以享受 clangd 的自动补全等功能了.

## 编辑

### 语法设置

在新打开的编辑器中点击「选择语言」，即可打开对应的语法高亮，如图：




### 快捷键

官方快捷键 PDF 如下，也可以在 VS Code 中按下 `Ctrl` + `Shift` + `P` 打开命令面板并输入命令 `> Help: Keyboard Shortcuts Reference` 打开. [4]

- Windows 系统快捷键[8]
- Linux 系统快捷键[9]

[4] VS Code 官方文档[7].

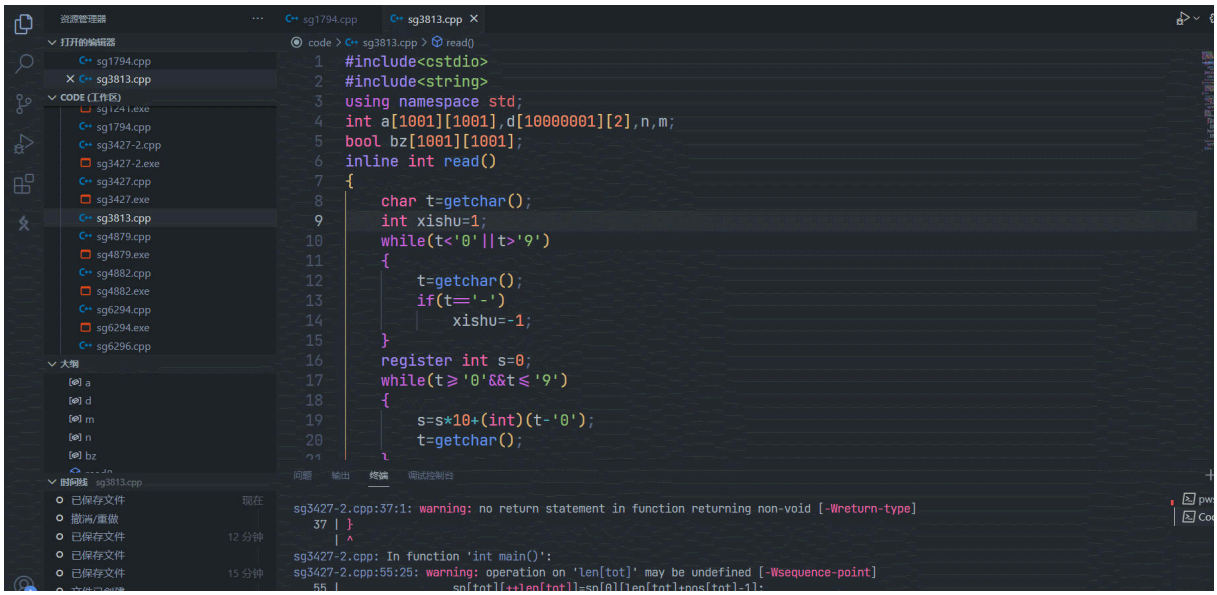
- Mac OS 系统快捷键[10]

部分快捷键：

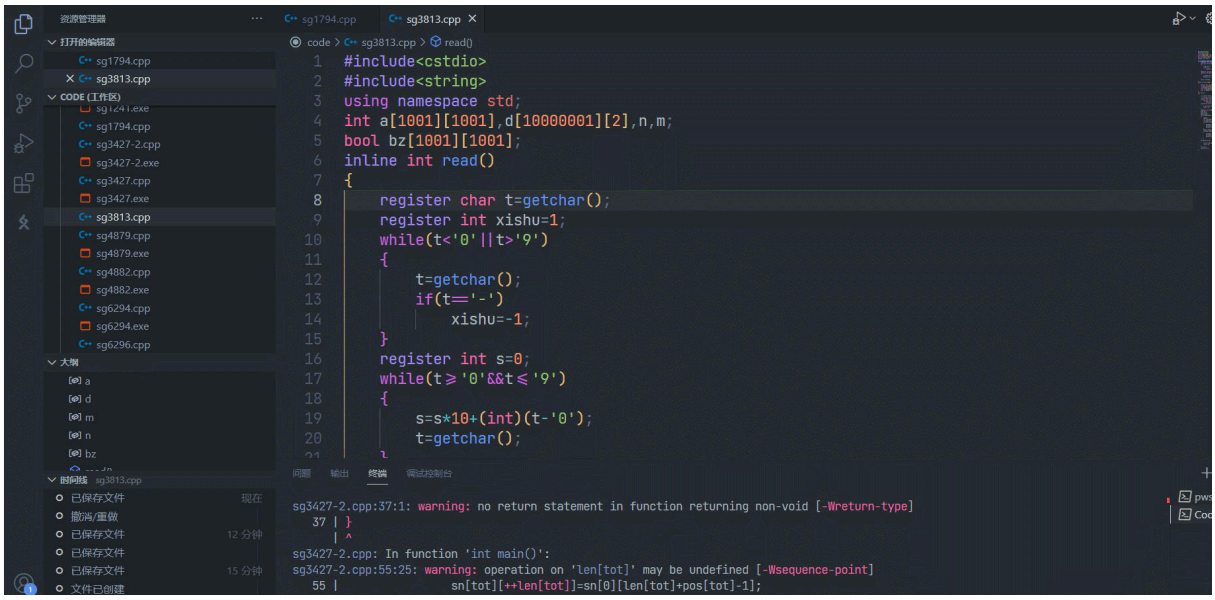
| 按键 | 操作 |
| --- | --- |
| Ctrl + C / X | 复制/剪切当前行（当没有选择内容时） |
| Ctrl + Shift + K | 删除当前行 |
| Alt + Up / Down | 行上移/下移 |
| Alt + Shift + Up / Down | 行向上/向下复制 |
| Ctrl + / | 切换行注释 |
| Ctrl + [ / ] | 行向左/右缩进 |
| Ctrl + Shift + [ / ] | 行折叠/展开 |
| Ctrl + P | 打开最近打开的文件 |
| Alt + Z | 切换自动折行 |
| Alt + F12 | 速览定义（如函数的定义） |
| Ctrl + Shift + \ | 跳转到匹配括号 |
| Ctrl + T | 在工作区中查找符号（在文件夹中查找指定名称函数等） |

## 多光标

按住 Alt 并单击即可在编辑器中添加光标，多数编辑操作都可同时进行；按住鼠标中键并在编辑器中拖动也可添加多行光标，如图：




按 Ctrl + F2 可在编辑器中同时更改所有匹配项，也可以在右键菜单中找到 Change All Occurrences，如图：




注意此时在右上角会有一个工具栏，可在其中开启查找匹配项时是否开启大小写匹配、全字匹配等.

## 参考资料与注释

[1] Visual Studio Code - The open source AI code editor

[2] 开始使用 Get started

[3] Marketplace

[4] CodeLLDB

[5] GDB 官方文档

[6] Getting started

[7] VS Code 官方文档

[8] Windows 系统快捷键

[9] Linux 系统快捷键

[10] Mac OS 系统快捷键

## 3.2.4 Atom

**Authors:** ouuan, ChungZH, partychicken, Xeonacid, Find-NICK

Atom，GitHub 家的编辑器.

## 简介

Atom 是一个免费、开源、跨平台的文本编辑器，由 GitHub 开发. 它是用 JavaScript 编写的，并且采用 Electron 架构. 它的一个较大缺点就是性能差.

> **⚠ Warning**
>
> 在 2022 年 6 月 8 日，Github 宣布将放弃 Atom 编辑器[1]，并于 2022 年 12 月 15 日对 Atom 编辑器存档. 如果你仍然喜欢 Atom 的界面，可以使用 Pulsar[2] 作为替代品，它的用户界面与 Atom 基本相似，目标是让他们最喜欢的编辑器保持活力 "keep their favorite editor alive"[5].

## 外部链接

- Atom 官网[4]
- Pulsar 官网[5]

[5] 来源：https://pulsar-edit.dev/about.html#the-team[3]

## 参考资料

[1] Github 宣布将放弃 Atom 编辑器

[2] Pulsar

[3] https://pulsar-edit.dev/about.html#the-team

[4] Atom 官网

[5] Pulsar 官网

## 3.2.5 Eclipse

**Authors:** ouuan, Doveqise, partychicken, Xeonacid, StudyingFather

### 介绍

Eclipse 是著名的跨平台开源集成开发环境（IDE）. 最初主要用来 Java 语言开发，当前亦有人通过插件使其作为 C++、Python、PHP 等其他语言的开发工具.

Eclipse 的本身只是一个框架平台，但是众多插件的支持，使得 Eclipse 拥有较佳的灵活性，所以许多软件开发商以 Eclipse 为框架开发自己的 IDE.

Eclipse 最初是由 IBM 公司开发的替代商业软件 Visual Age for Java 的下一代 IDE 开发环境，2001 年 11 月贡献给开源社区，现在它由非营利软件供应商联盟 Eclipse 基金会（Eclipse Foundation）管理. [6]

缺点:

- 实测这个 IDE 打开速度比 Visual Studio 慢
- 更新速度玄学，插件更新速度跟不上 IDE 的更新速度，对于经常更新的同学很不友好.

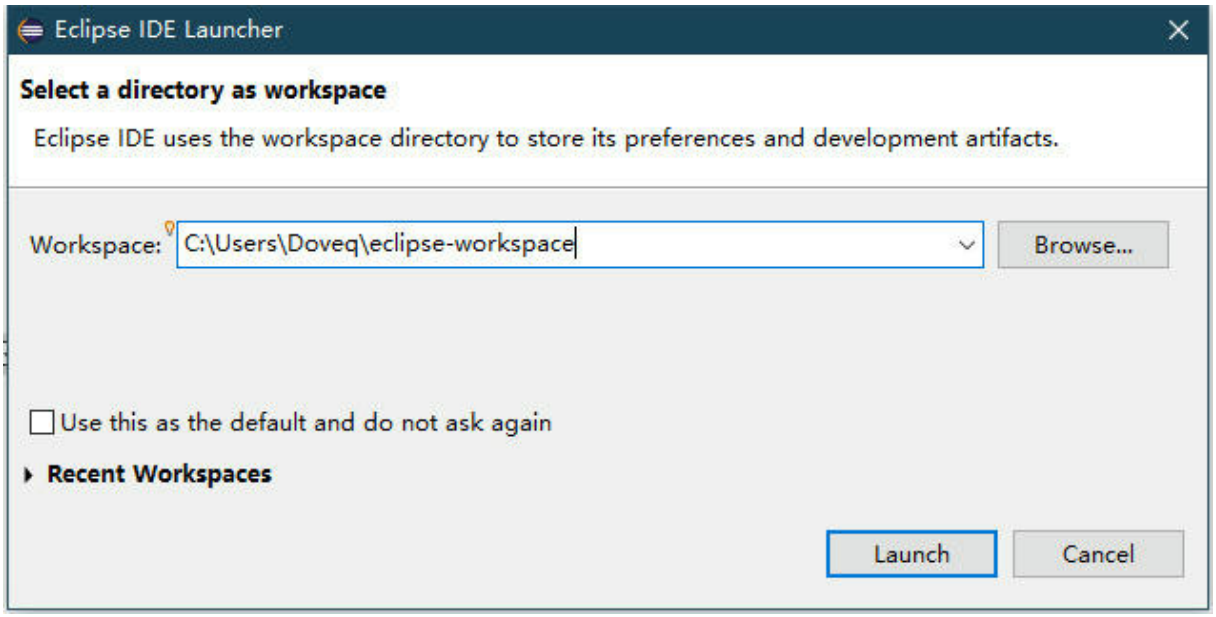
优点:

- 使用体验较好
- 能够快速上手，所以比较推荐 OIer 用这个 IDE.

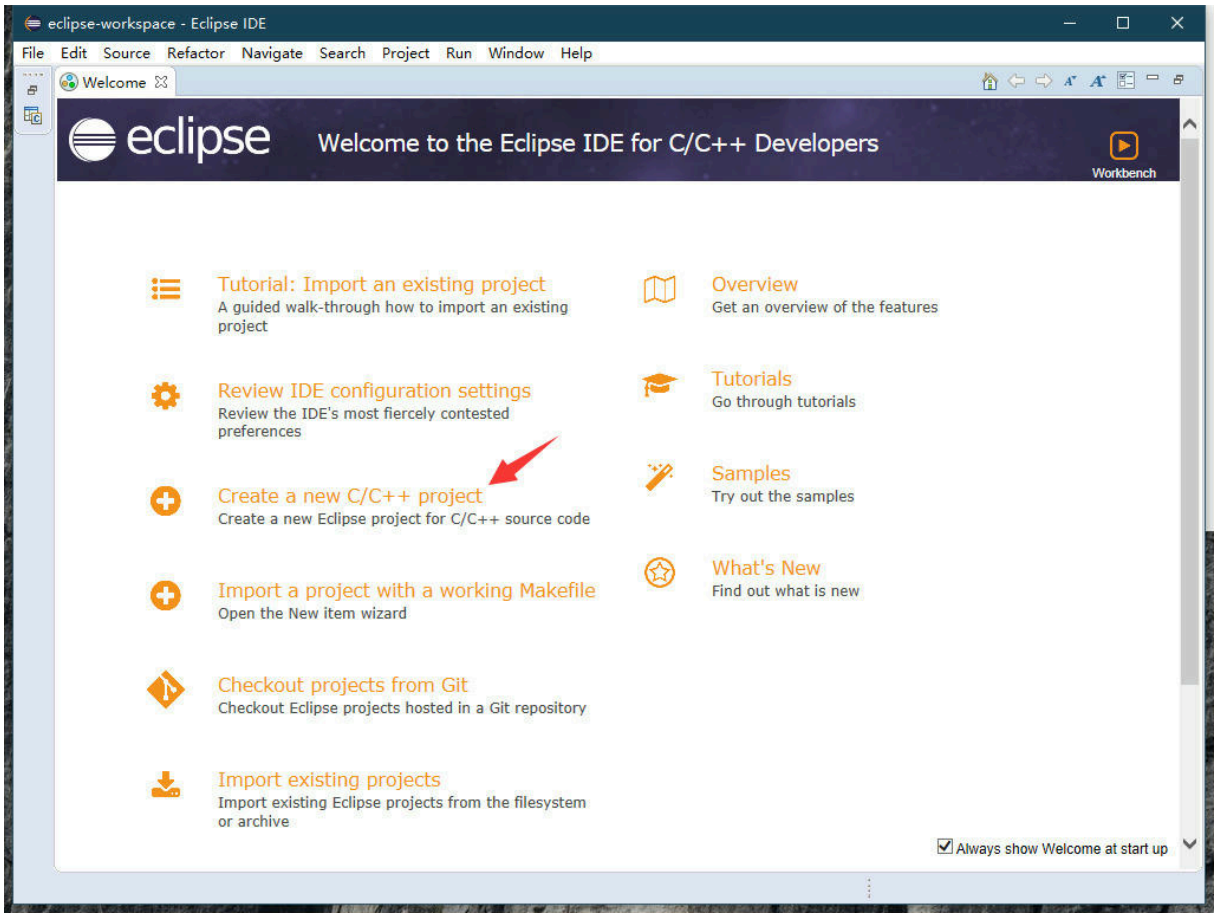
### 安装 & 配置指南

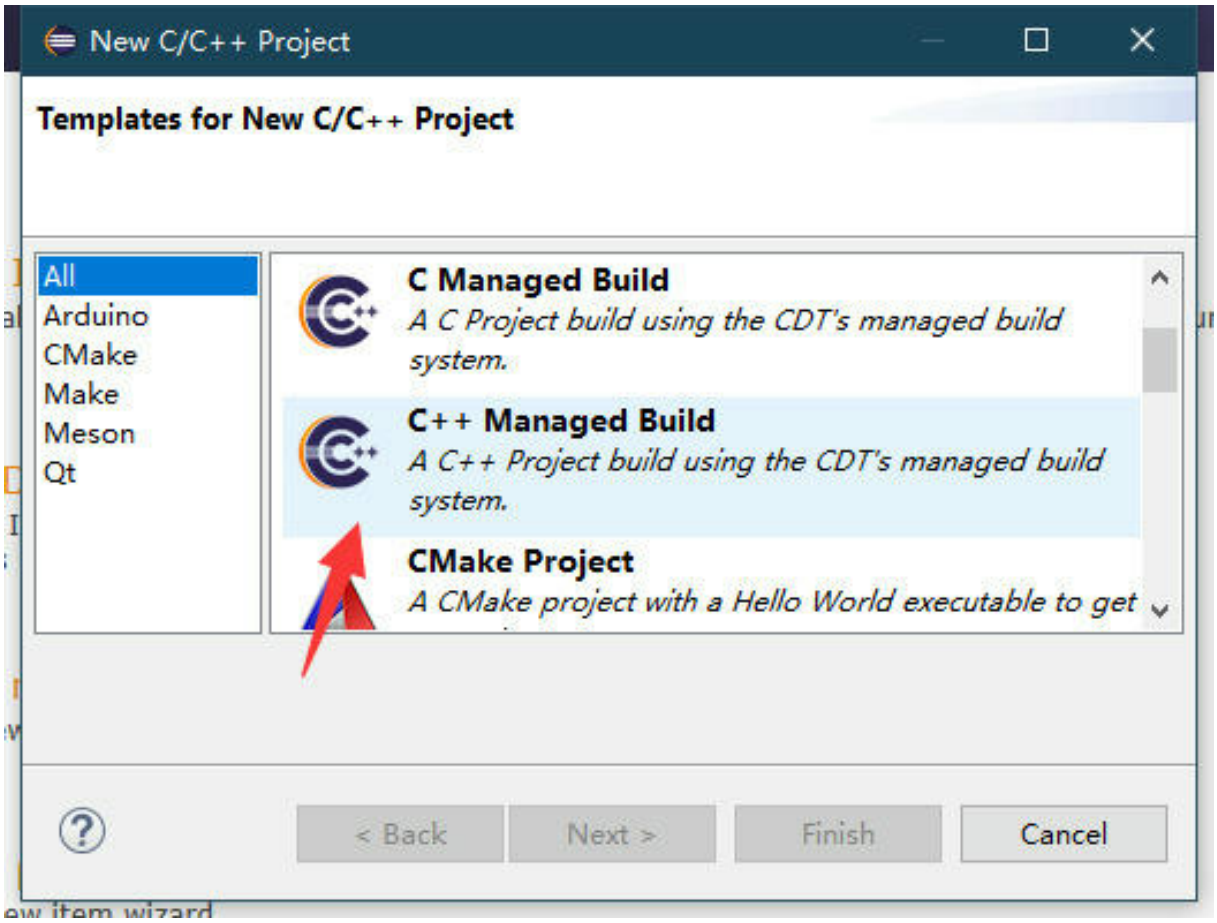
安装可参照 Eclipse/Installation - Eclipsepedia[2].

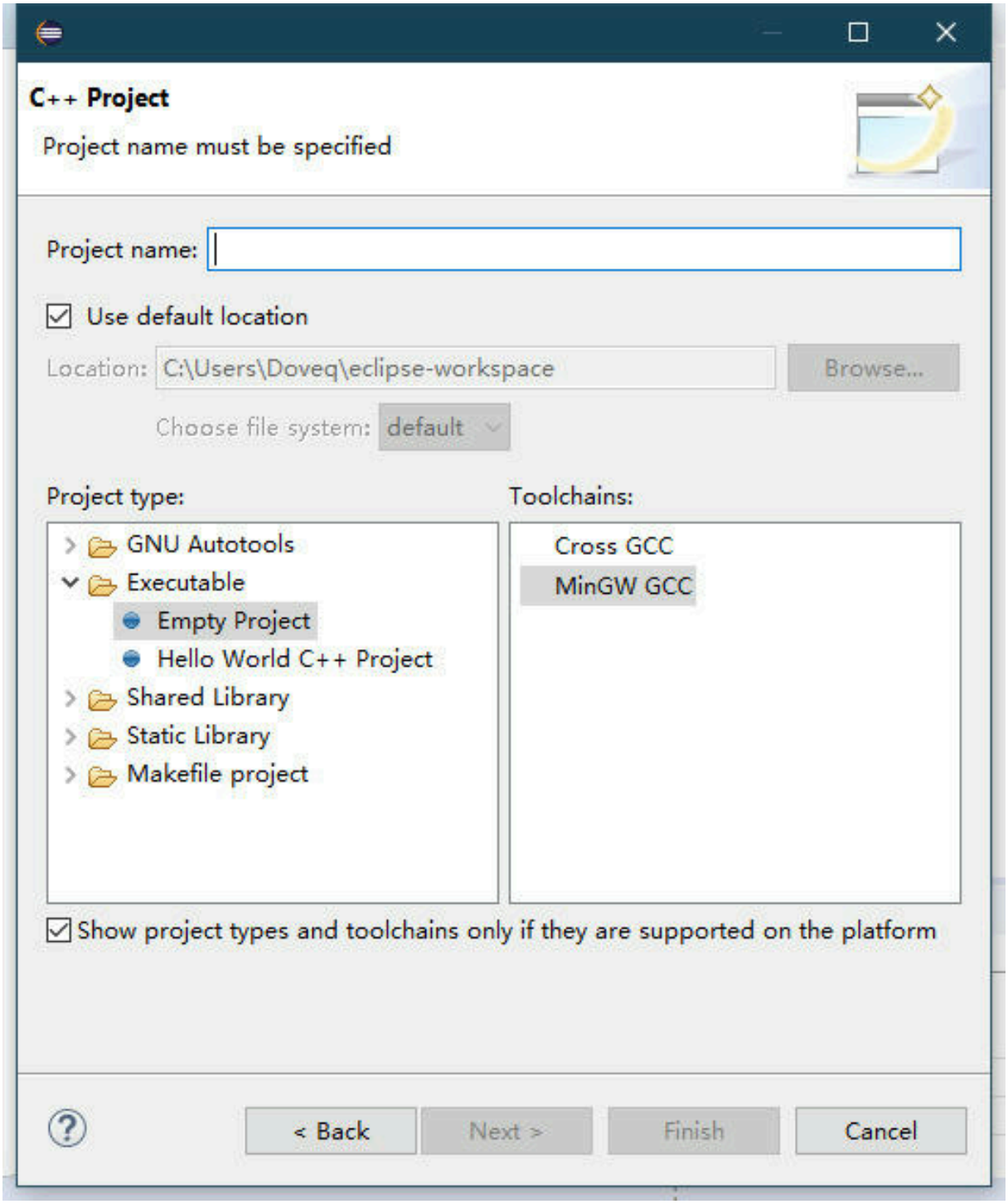
安装后如图填写目录信息以建造项目:




[6] Eclipse - 维基百科[1]


eclipse-workspace - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Welcome
eclipse
Welcome to the Eclipse IDE for C/C++ Developers
Workbench
Tutorial: Import an existing project
A guided walk-through how to import an existing project
Review IDE configuration settings
Review the IDE's most fiercely contested preferences
Create a new C/C++ project
Create a new Eclipse project for C/C++ source code
Import a project with a working Makefile
Open the New item wizard
Checkout projects from Git
Checkout Eclipse projects hosted in a Git repository
Import existing projects
Import existing Eclipse projects from the filesystem or archive
Overview
Get an overview of the features
Tutorials
Go through tutorials
Samples
Try out the samples
What's New
Find out what is new
Always show Welcome at start up


New C/C++ Project
Templates for New C/C++ Project
All
Arduino
CMake
Make
Meson
Qt
C Managed Build
A C Project build using the CDT's managed build system.
C++ Managed Build
A C++ Project build using the CDT's managed build system.
CMake Project
A CMake project with a Hello World executable to get
< Back
Next >
Finish
Cancel




## 拓展

这个软件的帮助手册很详细，建议刚接触的同学多看帮助手册，多百度，并且这个 IDE 的使用手感与 Visual Studio 相近.

和 VS Code→3.2.3@p147 类似，Eclipse 中也提供了很多插件，这些插件可以让 Eclipse 变得更加易用．[7]

[7] 曾经的 Java IDE 王者 Eclipse 真的没落了？21 款插件让它强大起来！[3]

## 参考资料与注释

[1] Eclipse - 维基百科

[2] Eclipse/Installation - Eclipsepedia

[3] 曾经的 Java IDE 王者 Eclipse 真的没落了？21 款插件让它强大起来！

## 3.2.6 Notepad++

**Authors:** ouuan, CBW2007, partychicken, StudyingFather, Xeonacid, Henry-ZHR

### 软件简介

Notepad++ 是 Windows 操作系统下的文本编辑器，支持多国语言、多种编码、多种编程语言的高亮和补全．它的 logo 也十分可爱，是一只变色龙（



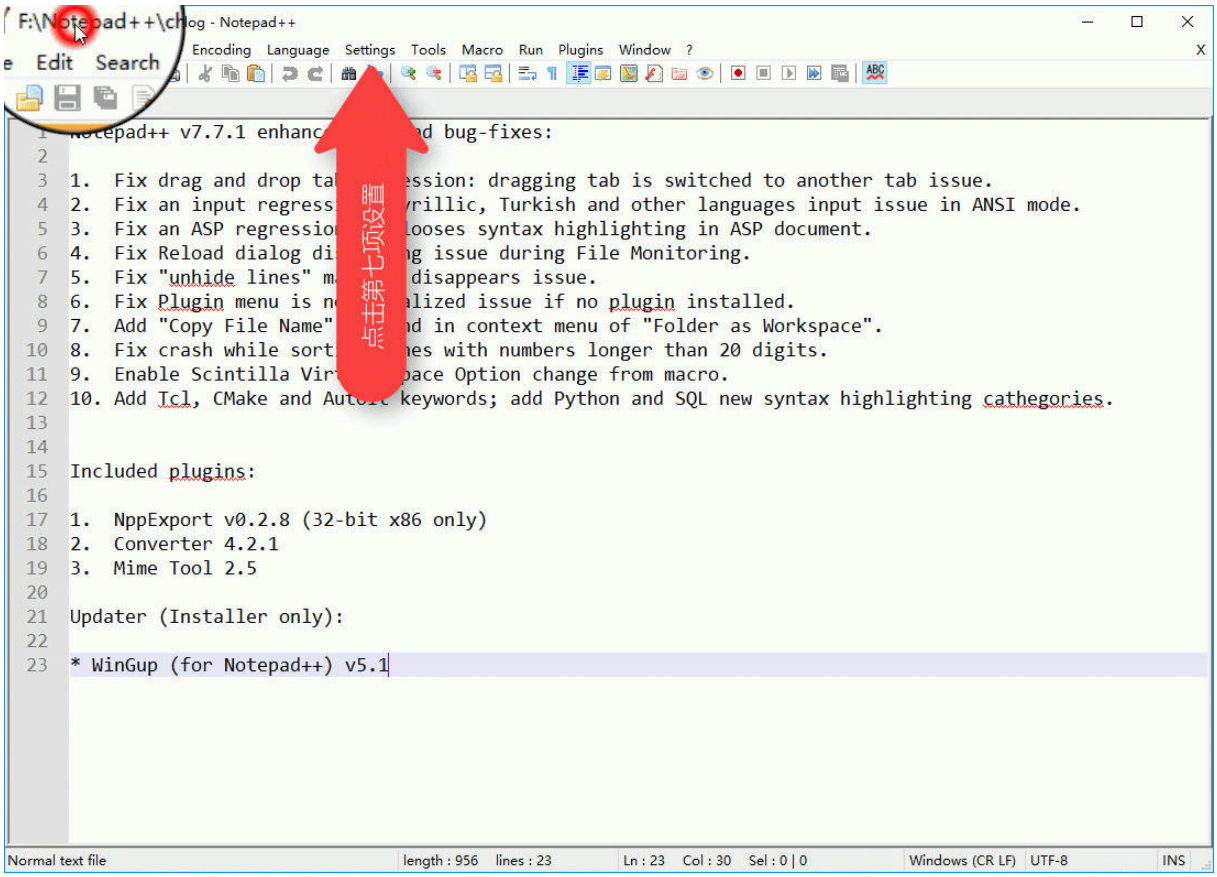
）

其功能比其他许多编辑软件强大许多，打开大文件时更加稳定，不断撤销不会出问题．关闭时也不需要保存，它会自动为你保存在缓冲区中．（可能需要配置）而且，它十分小巧，只有 10MB+，甚至可以放在 U 盘中随身携带．

### 下载与安装

参见 Getting started | Notepad++ User Manual[1].

### 更改界面语言



语言改完了，就可以随心所欲地魔改编辑器啦！

## 初级玩法

这里主要讲一些基础和特色功能.

### 查找与替换

依次单击「(菜单栏) 搜索」->「查找」(快捷键 CTRL+F) 即可打开「查找」页面 (如下图).




依次单击「(菜单栏) 搜索」->「替换」(快捷键 CTRL+H) 即可打开「替换」页面 (如下图).

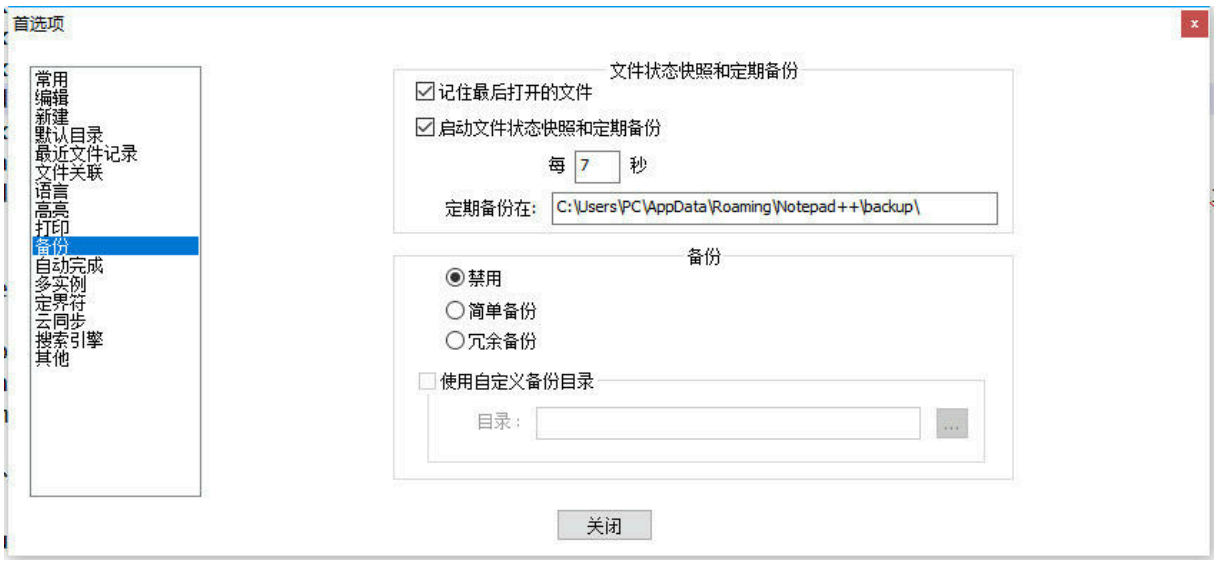



查找、替换之间其实是一个窗口，单击上面的标签页就可以完成切换.

其功能与普通编辑器大同小异，但是支持更多，如:

1. 严格匹配或大小写匹配等
2. 跨文档匹配
3. 转义字符，如'\r'，'\n'.
4. 正则表达式
5. 计数

## 定期备份




有了这个功能，就可以不用费心地担心意外情况代码丢失啦!

但是, 这个功能只是为你的文件拍了一个快照, 并没有真正保存, 所以还是建议要有良好的保存习惯. 或者说可以去自带插件商店安装 "Auto Save" 插件（详见 高级玩法 -> 插件→3.2@p138，下同）

### 书签功能

在你需要的行按 `Ctrl+F2` 即可设放置/取消书签，放置过书签的行前段有一个蓝色圆点.

按 `F2` 可以定位到下一个书签.

如果你抱怨不方便，可以去自带插件商店安装 "Bookmarks" 插件

### 代码高亮

右击左下角的 "XXX file"，可以选择许多种语言高亮，C、C++、PASCAL、Markdown 等任你挑选．你甚至可以自己定义高亮!

如果你认为每一次打开文件都要更改高亮很麻烦，可以在「设置 -> 首选项 -> 新建 -> 默认语言」中修改默认高亮.

需要渲染 Markdown 的，可以去插件商店安装 "Markdown Viewer"，还有更多类似插件等着你!

### 显示所有字符




点击红框所圈的按钮，就可以非（za）常（luan）清（wu）晰（zhang）地显示出「空格」、「TAB」、「换行」等原来不可见字符.

### 自动识别文件编码与换行符

Notepad++ 可以自动识别当前文件编码是 `UTF-8` 还是 `GB2312` 甚至其他．再也不用担心被 `锘斤拷` 抡死或被 `烫烫烫` 烫死了.

如果要使用不同的编码浏览文章，请依次单击「(菜单顶栏）编码」->「使用 XXX 编码」．如果想给文件换一个字符编码，请依次单击「(菜单顶栏）编码」->「转为 XXX 编码」.

它还可以自动识别换行符是 `CR`、`LF` 或 `CRLF`．不用担心下载下来的数据被吞换行.

在底部信息栏，你可以看到 "Windows(CR LF)" 等字样，这就是当前文件的换行符．右击它，可以改变当前文件换行符．此操作配合「显示所有字符」更直观哟!

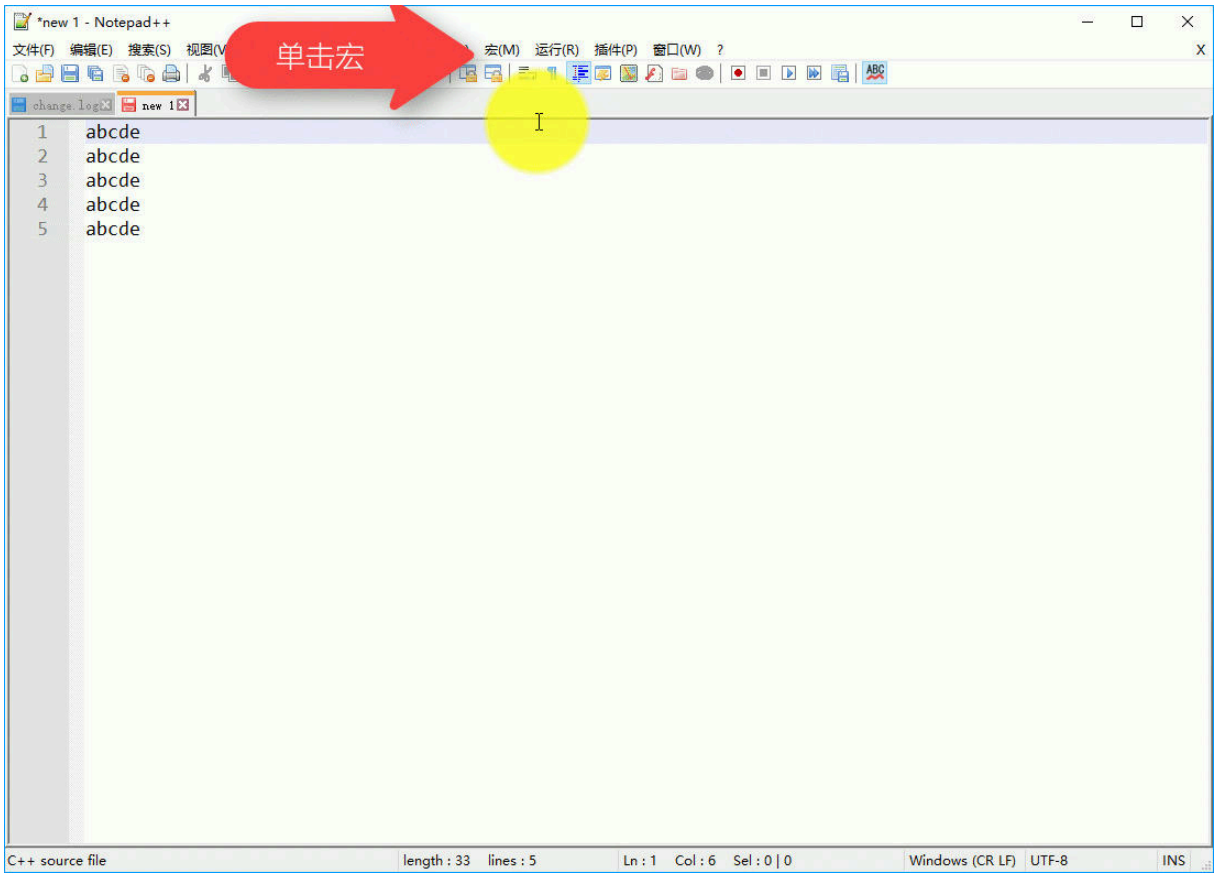
### 高级玩法

这个就适用于需求较高的用户.

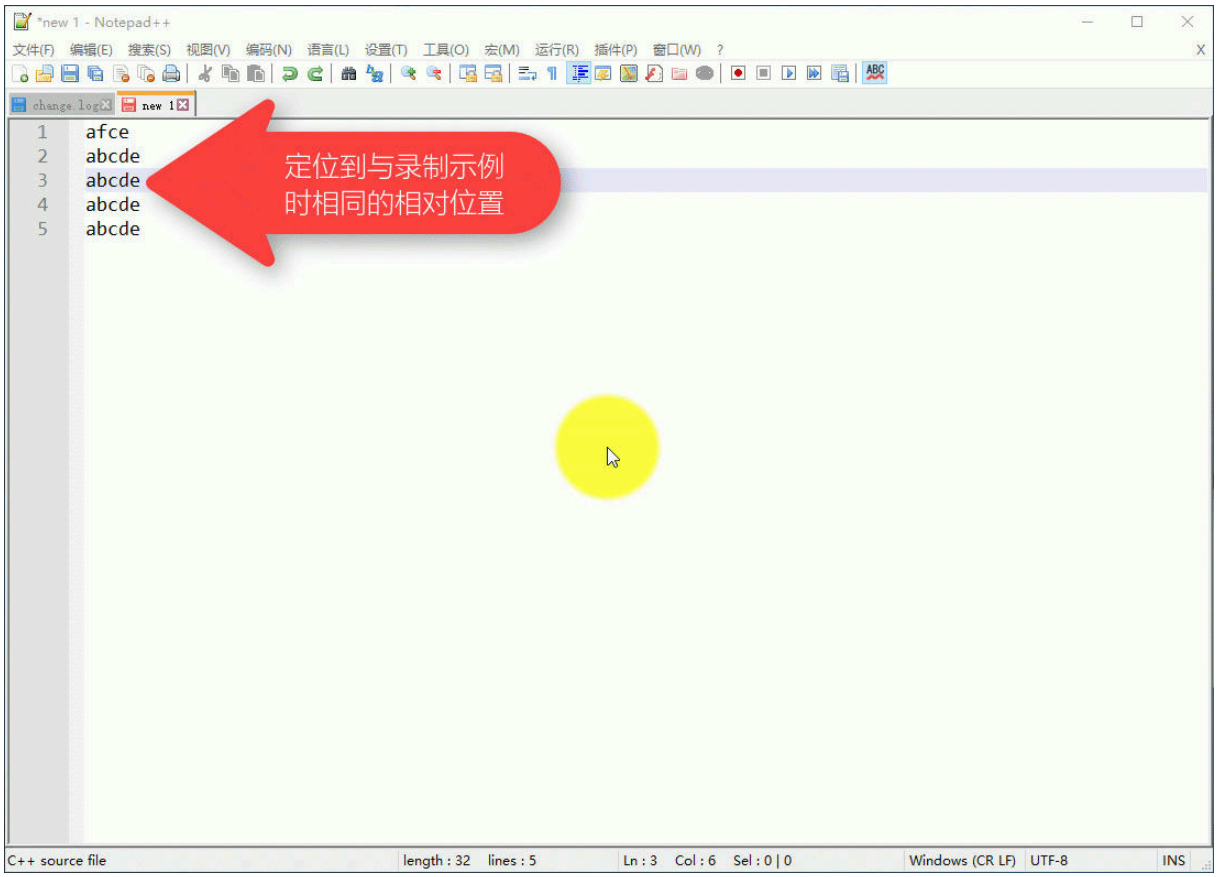
### 宏

宏可以帮助你完成许多重复的工作，例如，将奇数行的「abcde」改为「afce」，需要两步.

### 录制宏


*new 1 - Notepad++
单击宏
文件(F) 编辑(E) 搜索(S) 视图(V
宏(M) 运行(R) 插件(P) 窗口(W) ?
change.log
new 1
1 abcde
2 abcde
3 abcde
4 abcde
5 abcde
C++ source file
length : 33 lines : 5
Ln : 1 Col : 6 Sel : 0 | 0
Windows (CR LF) UTF-8
INS

### 使用宏




#### 大量处理，重复使用

如果是更多行呢？操作就需要一点改变.

首先是录制，一定要先按键盘上的 HOME 或 END 键将光标移动到行首或行尾，然后用方向键调整横向位置，再进行更改. 最后一定要用方向键将光标移动到下一个要处理的行.

比如刚刚的例子，可以先按 END 键，然后依次按 ←，Backspace,←,Backspace,F，最后按两下 ↓，最后停止录制.

然后是重播，先定位到第一个要处理的行（第 3 行），然后点击「宏」->「重复运行宏」. 在弹出窗口设置要运行的宏（刚录制的一般是第一个），设置运行次数（或者直接运行到文件尾），点确定即可.

#### 保存宏

点击「宏」->「保存录制宏」，并设置名称和快捷键，即可保存，方便后续使用.

### 插件

#### 插件管理

打开功能栏的「插件」按钮，列表中会显示所有你安装过的插件.

再选择「插件管理」选项，即可管理你的插件.

#### 安装插件（商店）

1. 打开「可用」选项卡，在列表中勾选你所要的插件
2. 点击右上角的「安装」按钮，按照提示重启软件即可.

### 安装插件（手动）

1. 下载插件（由第三方托管的官方地址：https://sourceforge.net/projects/npp-plugins/[2]）注意一定要选择 **与安装 Notepad++ 时处理器架构相同** 的插件.
2. 找到一个名为 "XXX.dll" 的文件（通常以插件名命名）.
3. 在 Notepad++ 中的功能栏点插件，并在列表中点「打开插件文件夹」.
4. 将刚才找到的 DLL 文件放入文件夹中，重启 Notepad++.
5. 【可选】删除刚才拷贝的文件，**但不要删除生成的文件夹!**

Tips: 如果多次不成功，可以尝试新建一个与插件名相同的文件夹在将 ".dll" 文件放入创建的文件夹中

### 更新插件

在插件管理中，选择「更新」选项卡，并勾选要更新的插件，然后点右上角的「更新」按钮.

### 移除插件

同样在插件管理中，选择「已安装」选项卡，并勾选要移除的插件，然后点右上角的「移除」按钮.

## 搭建开发环境

不只是编辑器！"Notepad++" 可谓神一样的存在，它可以通过傻瓜式地编译代码，甚至代替 IDE！这里以 C++ 为例

1. 安装编译器并将其必要的文件目录添加到 PATH 环境变量中.（C++ 需要添加 %APPPATH%\bin）当你在 cmd 中输入 g++ 时不再提示'g++'不是内部或外部命令.....即可（中间可能需要重启电脑）. 推荐 下载 ConsolePauser[3] 然后随便放并将其目录添加到环境变量（此为 Dev-C++ 的插件，在 Dev-C++ 软件根目录也有）.
2. 在菜单栏中选择「运行」->「运行.....」，打开「运行」窗口.
3. 分别输入以下命令

```
#编译命令:
cmd /c g++.exe -o $(CURRENT_DIRECTORY)\$(NAME_PART).exe $(FULL_CURRENT_PATH)
#运行命令:
cmd /c $(CURRENT_DIRECTORY)\$(NAME_PART).exe $(FULL_CURRENT_PATH) & pause
#调试命令:
cmd /c gdb $(CURRENT_DIRECTORY)\$(NAME_PART).exe

#如果下载了ConsolePauser可以使用下列代码获得更好的程序运行体验!（注意添加环境变量!）

#编译命令:
cmd /c (start ConsolePauser "g++.exe -o $(CURRENT_DIRECTORY)\$(NAME_PART).exe $(FULL_CURRENT_PATH)")
#运行命令:
cmd /c (start ConsolePauser "$(CURRENT_DIRECTORY)\$(NAME_PART).exe")
#调试命令:
cmd /c (start ConsolePauser "gdb $(CURRENT_DIRECTORY)\$(NAME_PART).exe")
```

4. 单击「保存」，名字可以自己取，如 "Compile","Run" 等，然后设定好你想要的快捷键（捡好记的来，如 Dev-C++ 就分别是 `F9` 和 `F10`）.
5. Enjoy it!

## 小彩蛋

1. 在运行安装程序时你会在下方看到这样一句话:

> "The best things in life are free. Notepad++ is free. So Notepad++ is the best(.)"
>
> （生活中最好的事情都是免费的. Notepad++ 是免费的. 所以 Notepad++ 是最好的.）

这牛吹的，不得不说，很有底气.

2. 在一个新开的页面中输入 "random" 并选中，再按 `F1` 就会得到一句很有意思的话.

### 外部参考资料

[1] Getting started | Notepad++ User Manual 

[2] https://sourceforge.net/projects/npp-plugins/ 

[3] 下载 ConsolePauser 

## 3.2.7 Kate

**Authors:** CoelacanthusHex

### 软件简介

Kate 是一个具有众多功能的跨平台文本编辑器. Kate 还附带了多种插件，包括一个嵌入式终端，可以让你直接从 Kate 中启动控制台命令，强大的搜索和替换插件，以及一个预览插件，可以渲染 MD、HTML 甚至 SVG 文件. 支持通过交换文件在系统崩溃时恢复数据，带参数提示的自动补全，同时支持 LSP (Language Server Protocol)[1] 以获得更为强大的补全.

### 下载与安装

可打开 Kate 官网[2]，然后进入 获取页面[3]. 随后，根据你使用的系统和喜欢的安装方式进行安装.

### 用法与功能

#### 交换文件防止数据丢失

与 Vim 类似，Kate 会将未保存的更改写入一个交换文件（一般是原文件名前面加点后面加 `.kate-swp`），如果遭遇断电或程序崩溃等意外，下次启动时不会丢失未保存的更改.

#### 代码高亮

Kate 支持三百余种语言的语法高亮. 一般来说，Kate 可以自动地选择对应的语言进行语法高亮，不过偶尔也有错误的时候，这时候可以点击最右下角的按钮，选择正确的语言.

#### 自己编写语法高亮文件

尽管 Kate 支持超过三百种语言的语法高亮，但是仍不免有语言未被覆盖到，此时可以自己动手编写语法高亮文件. Kate 自身自带的文件位于 Syntax Highlighting Powered By KSyntaxHighlighting Framework[4]，语法可参照 Working with Syntax Highlighting[5]，编写好的文件根据 Syntax definition files[6] 放置. CoelacanthusHex/dotfiles@80a913c/pam_env.xml[7] 有笔者编写的一个配置文件可供参考.

#### 切换语言

点击上方工具栏里的 `设置/Setting`，然后点击 `配置语言/Configure Language`，随后选择语言即可，注意可以选择备选语言.

#### 编码与行尾符

Kate 可以自动识别当前文件使用的是什么编码，如果识别错误，可以点击右下角倒数第二个按钮，选择正确的编码.

同时，Kate 也可以自动识别当前文件使用的行尾符，如果识别错误，可以点击 `工具→行尾/Tool→End of line` 选择正确的行尾符.

#### 查找与替换

依次单击 `编辑→查找`（快捷键 `Ctrl` + `F`）即可打开「查找」页面. 依次单击 `编辑→替换`（快捷键 `Ctrl` + `R`）即可打开「查找与替换」页面. 同时，点击左下角 `搜索与替换` 也可打开「查找与替换」页面.

具体操作和其他编辑器并无太大差别，但是支持一些额外的特性，例如：

1. 是否区分大小写
2. 支持正则表达式（包括捕获组）
3. 从当前文件到多文件再到当前工程不等的范围
4. 对查找的结果进行选择替换

### Language Server Protocol

Kate 自 19.12 起支持 LSP Client，最初仅支持 C/C++、D、Fortran、Go、Latex/BibTeX、OCaml、Python、Rust，现如今支持如下表中的语言：

| 语言 | LSP Server |
|---|---|
| Bash | bash-language-server[8] |
| LaTeX | texlab[9] |
| BibTeX | texlab[10] |
| C | clangd[11] |
| C++ | clangd[12] |
| D | serve-d[13] |
| Fortran | fortls[14] |
| Go | gopls[15] |
| Haskell | haskell-language-server-wrapper[16] |
| JavaScript | typescript-language-server[17] |
| OCaml | ocamllsp[18] |
| Perl | Perl-LanguageServer[19] |
| Python | pyls[20] |
| Rust | rls[21] |
| TypeScript | typescript-language-server[22] |
| R | RLanguageServer[23] |
| zig | zls[24] |

要启用 LSP 相关特性，需要前往菜单栏中 `设置→配置 Kate` 然后选择 `插件` 中 `LSP 客户端` 以启用相关特性．当打开对应语言的文件时，Kate 会自动拉起对应的 LSP Server.

### 增加配置

此外，用户还可以手动编写配置，具体格式为：

```
{
    "servers": {
        "bibtex": {
            "use": "latex",
            "highlightingModeRegex": "^BibTeX$"
        },
        "c": {
            "command": ["clangd", "-log=error", "--background-index"],
            "commandDebug": ["clangd", "-log=verbose", "--background-index"],
            "url": "https://clang.llvm.org/extra/clangd/",
            "highlightingModeRegex": "^(C|ANSI C89|Objective-C)$"
        },
        "cpp": {
            "use": "c",
            "highlightingModeRegex": "^(C\\+\\+|ISO C\\+\\+|Objective-C\\+\\+)$"
        },
        "haskell": {
```

```
            "command": ["haskell-language-server-wrapper", "--lsp"],
            "rootIndicationFileNames": ["*.cabal", "stack.yaml",
"cabal.project", "package.yaml"],
            "url": "https://github.com/haskell/haskell-language-server",
            "highlightingModeRegex": "^Haskell$"
        },
        "latex": {
            "command": ["texlab"],
            "url": "https://texlab.netlify.com/",
            "highlightingModeRegex": "^LaTeX$"
        },
        "rust": {
            "command": ["rls"],
            "rootIndicationFileNames": ["Cargo.lock", "Cargo.toml"],
            "url": "https://github.com/rust-lang/rls",
            "highlightingModeRegex": "^Rust$"
        }
    }
}
```

其中 `server` 里的每一项代表一种语言，在这个语言里，`command` 代表启动 LSP Server 所使用的命令，`command` 是一个数组，是所需要执行的命令以空格分词的结果；`url` 是 LSP 的网址；`rootIndicationFileNames` 是用于确定项目根目录的文件；`highlightingModeRegex` 则匹配某种语法高亮的名字，以确定使用哪个 LSP；如果存在 `use` 项，则代表使用 `use` 项对应的语言的配置.

该配置项位于 `设置→配置 Kate→LSP 客户端→用户服务器设置`，其中 `LSP 客户端` 部分要在 `插件` 中启用 `LSP 客户端` 插件后才可见.

### 内置终端

> **注意**
>
> 内置终端依赖了 KDE 的 Konsole[8]，而 Konsole 为 * nix 独有包. 也就是说，Windows 下该特性不可用.

按 `F4` 可打开或关闭内置终端，也可点击左下角 `终端` 按钮打开，内置终端的当前目录会自动与当前文件保持一致，并随着你选择的文件而改变. 其余与一般终端并无太大不同.

### 外部工具

点击 `工具→外部工具` 可执行.

点击 `工具→外部工具→配置` 可以配置外部工具.

#### 添加外部工具

##### 从预置配置中添加

进入配置页面后，点击左下角 `添加→从默认工具添加`，然后点击对应工具即可.

##### 手写配置添加

进入配置页面后，点击左下角 `添加→添加工具`，然后按提示填写即可. 可以参照 此文档（英文）[26] 来编写自己的外部工具配置. 注意可点击如下标志查看可使用的变量.

[8] Arch Linux 中对该包的描述[25] 中，其可选依赖了 `konsole`，描述为 `open a terminal in Kate`（在 Kate 中打开一个终端）.



### 常用的外部工具

#### 编译并执行单个 C++ 文件

在 * nix 系统下，打开任意 C++ 源文件，在外部工具里找到 `编译执行 cpp`，点击即可.

> **对于 Windows 用户**
>
> 在默认情况下，由于该工具的可执行文件为 `sh`，使得该工具在 Windows 下不可用. 然而，用户可以对该工具进行修改，使其可用于 Windows 系统.
>
> 要进行修改，请先确保你的系统内有一个可用的 C++ 编译器. 然后从默认工具添加 `编译运行 cpp`，将其中 `可执行文件` 从 `sh` 改为 `powershell`，参数改为 `-ExecutionPolicy Bypass -Command "g++ %{Document:FilePath} -o %{Document:FileBaseName}.exe;./%{Document:FileBaseName}.exe"`[9][10]即可.

#### Git Blame

打开任意文件，在外部工具里找到 `git blame`，点击之后，会打开一个窗口，展示 git blame 的结果.

#### 格式化

格式化功能要求对应包或应用程序可用，例如，C/C++ 的格式化要求 `clang-format` 可用. 对于其他语言，用户可以前往外部工具配置中查看其默认可执行文件作为参考.

打开任意源文件，在外部工具里找到 `用 xxx 格式化`，点击即可. 另外，对于 C/C++ 语言的源文件，`clang-format` 可格式化选中的文本.

#### Git Blame

要启用该特性，需要前往菜单栏中 `设置→配置 Kate` 然后选择 `插件` 中 `Git Blame`.

启用该特性后，Kate 会在每一行后面以较浅字体显示在 Git 中最后于什么时间被谁修改，将鼠标移动到文字上会出现一个悬浮窗显示 commit 的具体信息.

## 相关外部链接

- The Kate Handbook[27]
- 关于如何手写自己的 LSP 客户端配置（英文）[28]
- 关于如何手写自己的外部工具配置（英文）[29]

## 参考资料与脚注

[1] LSP (Language Server Protocol)

[2] Kate 官网

[3] 获取页面

[4] Syntax Highlighting Powered By KSyntaxHighlighting Framework

[5] Working with Syntax Highlighting

[6] Syntax definition files

[7] CoelacanthusHex/dotfiles@80a913c/pam_env.xml

[8] bash-language-server

[9] texlab

[10] texlab

[11] clangd

[12] clangd

[9] 若 `g++` 不在 `PATH` 环境变量中，则将其改为编译器的绝对路径

[10] 或者，如果使用 Clang，则将 `g++` 改为 `clang++`.

[13] serve-d 

[14] fortls 

[15] gopls 

[16] haskell-language-server-wrapper 

[17] typescript-language-server 

[18] ocamllsp 

[19] Perl-LanguageServer 

[20] pyls 

[21] rls 

[22] typescript-language-server 

[23] RLanguageServer 

[24] zls 

[25] Arch Linux 中对该包的描述 

[26] 此文档（英文） 

[27] The Kate Handbook 

[28] 关于如何手写自己的 LSP 客户端配置（英文） 

[29] 关于如何手写自己的外部工具配置（英文） 

### 3.2.8 Dev-C++

> **Authors:** topdevst, ksyx, ouuan, Doveqise, hsfzLZH1, wangqingshiyu, sshwy, NanoApe, DawnMagnet, CamberLoid, royqh1979

## 介绍

Dev-C++ 是一套用于开发 C/C++ 程序的自由的集成开发环境（IDE），并以 GPL 作为分发许可，使用 MinGW 及 GDB 作为编译系统与调试系统．Dev-C++ 运行在 Microsoft Windows 下．

Dev-C++ 的优点在于界面简洁友好，安装便捷，支持单文件编译，因此成为了许多入门 OI 选手以及 C++ 语言初学者的首选．在 NOIP 中，提供 Windows 作为比赛系统的省份一般预置 Dev-C++．

Dev-C++ 起源于 Colin Laplace 编写的 Bloodshed Dev-C++．该版本自 2005 年 2 月 22 日停止更新．2006 年，Dev-C++ 主要开发者 Colin Laplace 曾经对此作出了解释：「因忙于现实生活的事务，没有时间继续 Dev-C++ 的开发．」

Orwell Dev-C++ 是 Dev-C++ 的一个衍生版本，由独立程序员 Orwell (Johan Mes) 开发并维护．其对原版 Dev-C++ 进行了错误修正，并更新了编译器版本．一般而言，Dev-C++ 5.x 均为 Orwell Dev-C++．其最后一次更新于 2015 年，版本为 5.11．

Embarcadero Dev-C++[11]是 Bloodshed Dev-C++ 和 Orwell Dev-C++ 的继任者．2020 年，Embarcadero 赞助并接手了原有的 Dev-C++ 项目，继续开发．Embarcadero Dev-C++ 加入了对高 DPI 的支持，更新了编译器以加入更新版本的 C++ 标准支持，以及暗色模式．

以上的 Dev-C++ 分发都被认为是「官方的」．此外，在 2015 年 Orwell Dev-C++ 停止更新后，因为教学需要，一位来自中国的个人开发者 royqh1979[3] 决定继续开发他的 Dev-C++ 个人分支，命名为小熊猫 Dev-C++[12]，集成了智能提示和高版本的 MinGW64，非常便于国内的个人使用和学习．

小熊猫 Dev-C++ 6.7.5 版本发布后，作者使用 qt5 开发了全新的小熊猫 C++[13]，可在 windows、linux 和 macos 等系统下原生运行．小熊猫 C++ 的界面与 Dev-C++ 相似，除了提供和 Dev-C++ 相似但更加完善的单文件编译、调试、语法高亮、搜索/替换等功能外，还提供了诸如 **暗色主题、代码智能提示、变量/函数重命名、切换/自动识别文件编码** 等现代 IDE 常见的基本功能．此外小熊猫 C++ 还具备与 CP Editor 类似的试题集功能，可以自行编写或 **从常见的 OJ 竞赛网站上下载试题样例，自动运行和测试程序**．

[11]项目源代码托管于 GitHub[1] 和 SourceForge[2]．
[12]源代码托管于 Github[4]
[13]项目官网位于 小熊猫 C++[5]，源代码托管于 Github[6]

## 使用教程

### 常用快捷键

#### 文件部分

- `Ctrl + N`: 创建源代码
- `Ctrl + O`: 打开文件
- `Ctrl + W`: 关闭文件
- `Ctrl + Shift + W`: 关闭所有文件
- `Ctrl + P`: 打印文件

#### 格式部分

- `Ctrl + /`: 注释和取消注释
- `Tab`: 缩进
- `Shift + Tab`: 取消缩进

#### 行操作

- `Ctrl + E`: 复制行
- `Ctrl + D`: 删除行
- `Ctrl + Shift + Up`: 向上移动
- `Ctrl + Shift + Down`: 向下移动

#### 跳转部分

- `Ctrl + F`: 搜索
- `Ctrl + R`: 替换
- `F3`: 搜索下一个
- `Shift + F3`: 搜索上一个
- `Ctrl + G`: 到指定行号
- `Shift + Ctrl + G`: 到指定函数
- `Ctrl + [1 ~ 9]`: 设置书签
- `Alt + [1 ~ 9]`: 跳转书签

#### 显示部分

- `Ctrl + 滚轮`: 字号放大或缩小
- `Ctrl + F11`: 全屏或恢复

#### 运行部分

- `F9`: 只编译
- `F10`: 只运行
- `F11`: 编译并运行
- `F12`: 全部重新编译

#### 调试部分

- `F2`: 转到断点
- `F4`: 设置断点或取消
- `F5`: 调试运行
- `F6`: 停止
- `F7`: 逐步调试

### 调试流程

1. 将编译器配置设定为 `TDM-GCC 4.9.2 64-bit Debug`

2. 按 `F4` 设置或取消调试断点
3. 将光标放置在变量上，按 `Alt + A` 向调试窗口添加监控变量
4. 按 `F5` 启动调试
5. 按 `F7` 或 `Alt + N` 逐步调试
6. 按 `Alt + S` 跳至下一个调试断点
7. 按 `F6` 停止调试

## 扩展

### 增加编译选项

点击工具 -> 编译选项，然后选择 "代码生成/优化" 选项卡，下面介绍笔者常用的几个编译选项.

### 开启优化

优化代码运行时间或占用空间.

选择 "代码生成" 子选项卡中的 "优化级别（-Ox）" 选项标签.




**更换语言标准**

使用新语言特性或试图让代码在旧标准下编译.

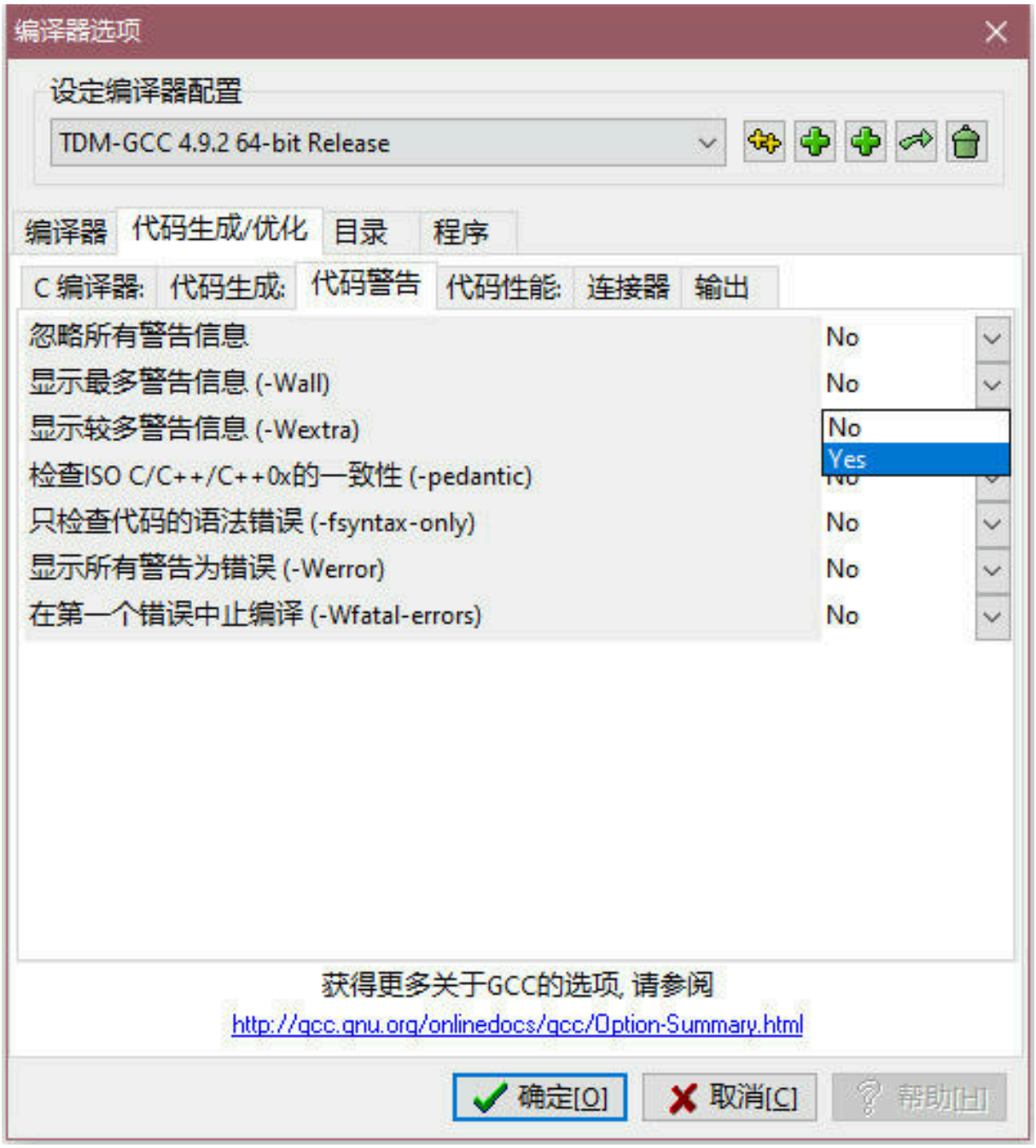
选择 "代码生成" 子选项卡中的 "语言标准（-std）" 选项标签.




**显示最多警告信息**

查错小助手.

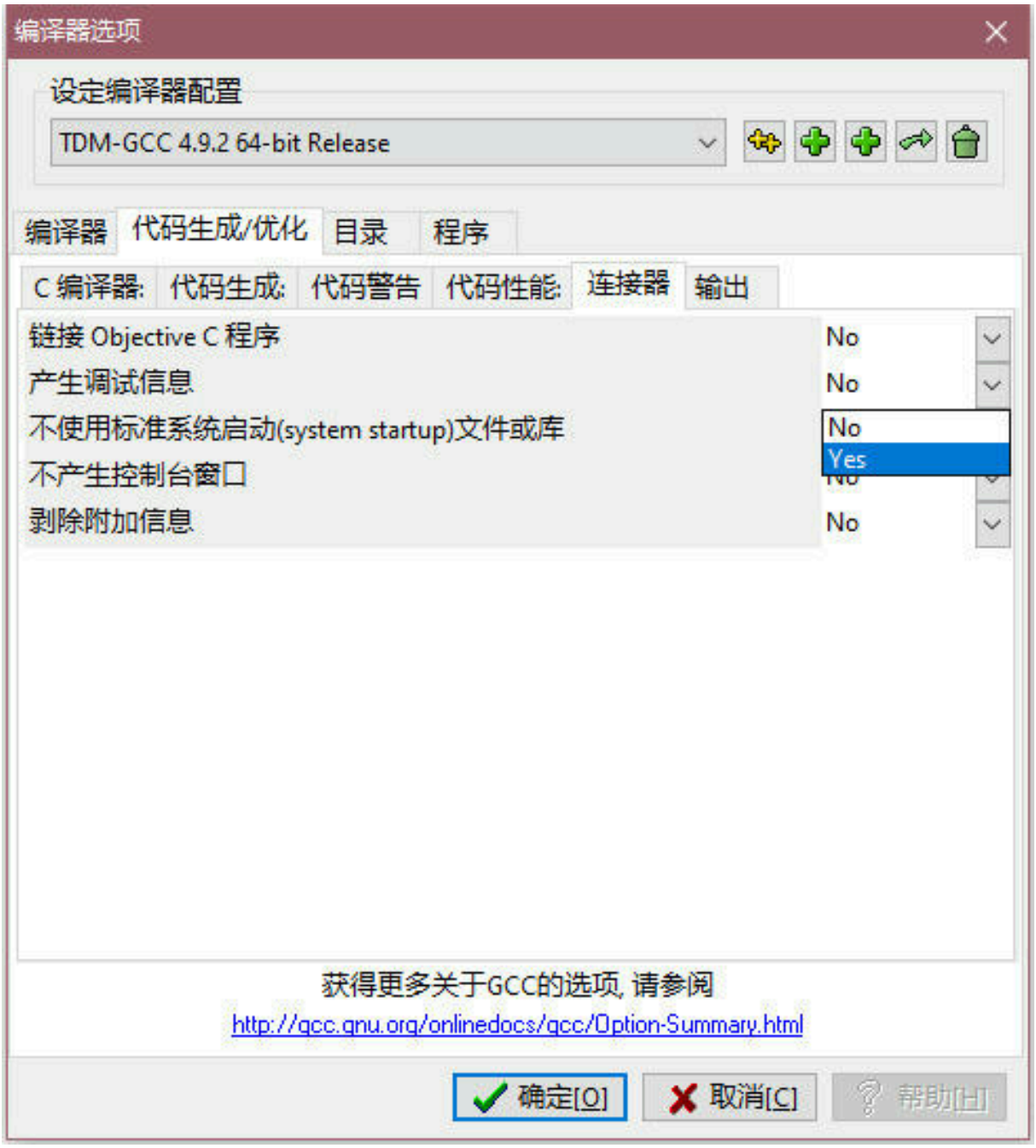
选择 "代码警告" 子选项卡中的 "显示最多警告信息（-Wall）" 选项标签.




**生成调试信息**

当显示"项目没有调试信息, 您想打开项目调试选项并重新生成吗?"点击后闪退或想使用调试功能时需开启此功能.

选择"连接器"子选项卡中的"产生调试信息"选项标签.




**编译小 trick**

点击工具 -> 编译选项，然后选择 "编译器" 选项卡，接下来介绍几个常用 trick.

**开大栈**

防止 DFS 爆系统栈之类的情况出现.

在 "连接器命令行加入以下命令" 中加入 `-Wl,--stack=128000000` 命令.

此命令将栈开到了约 128MB 的大小，有需要可以自行增加.



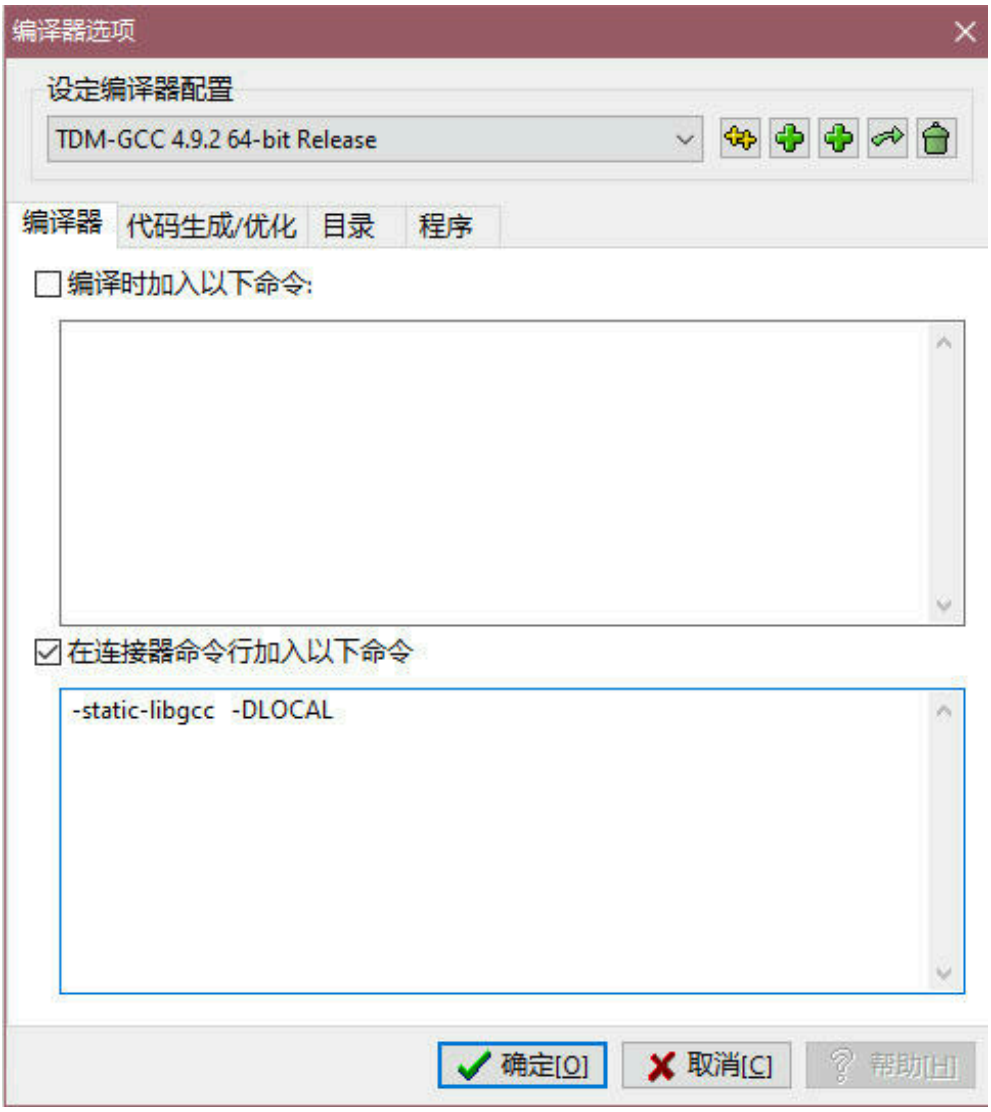

**定义宏**

方便本地评测使用文件输入输出或作其他用途.

在 "连接器命令行加入以下命令" 中加入 `-D[String]` 命令.

其中 `[String]` 改为你需要的宏名.

如图，当开启编译选项后便可将以下代码从 `test.in` 文件读入数据并在 `test.out` 文件中输出.




```
#ifdef LOCAL
freopen("test.in", "r", stdin);
freopen("test.out", "w", stdout);
#endif
```

### 代码格式化

点击 Astyle-> 格式化当前文件 或 按 Ctrl+Shift+A 进行代码格式化.




## 美化

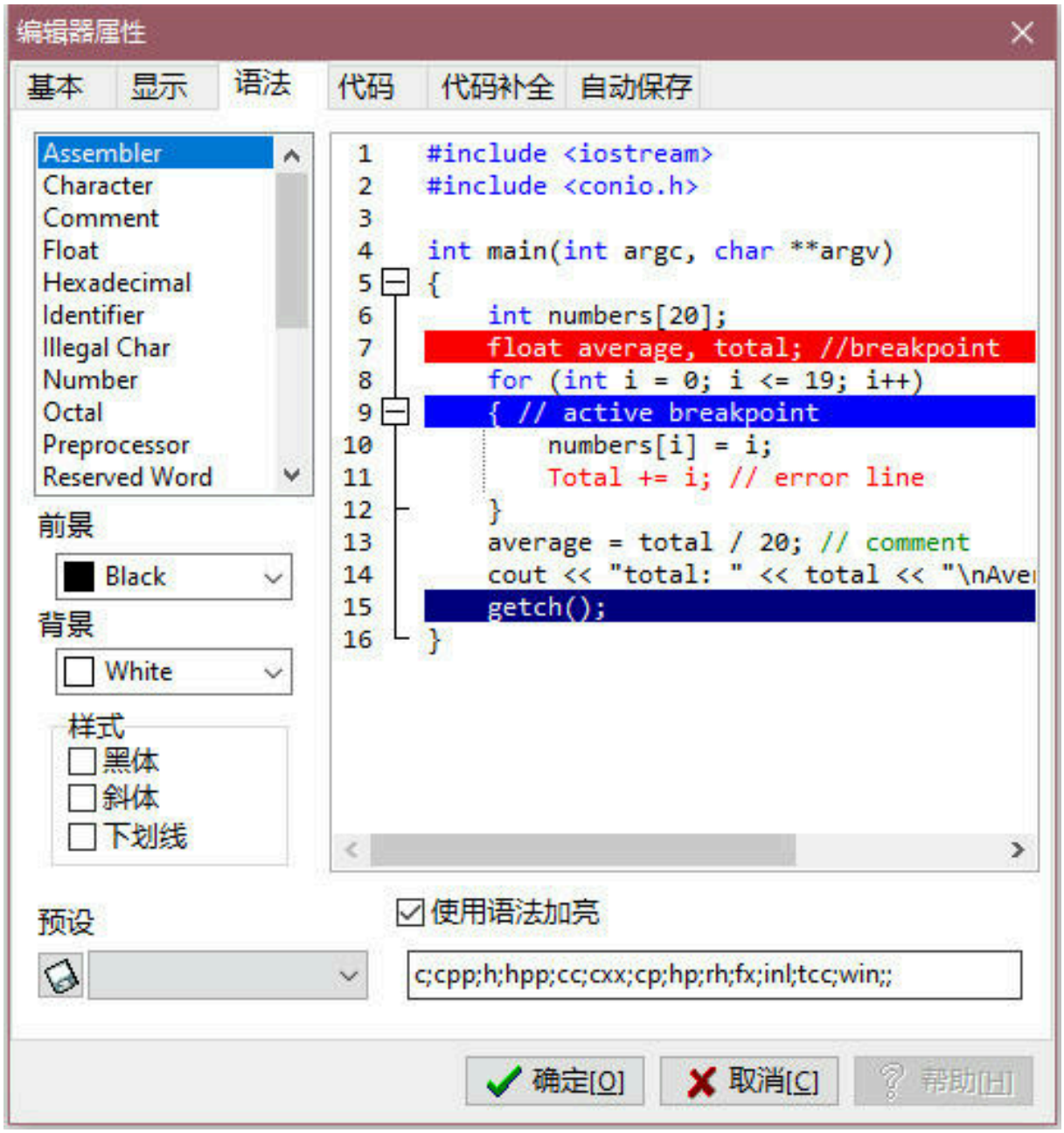
### 字体

点击工具 -> 编辑器选项，然后选择 "显示" 选项卡.




**主题**

点击工具 -> 编辑器选项，然后选择 "语法" 选项卡，可以使用预设主题，也可以自行调整.




## 参考资料

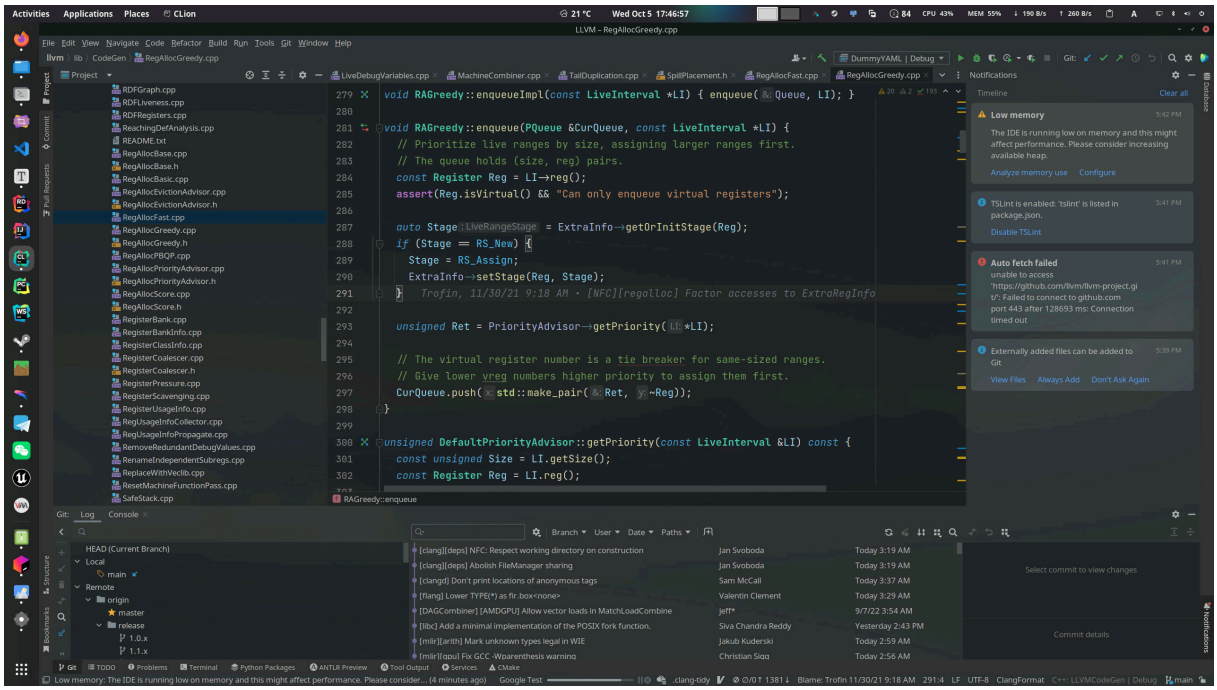
[1] GitHub [2] SourceForge [3] royqh1979

[4] Github [5] 小熊猫 C++ [6] Github

### 3.2.9 CLion

#### 简介

CLion 是一款由 JetBrains 公司开发的功能丰富且强大的跨平台 C/C++ 集成开发环境（IDE）.



## 官方教程

在官方网站中给出了 学习 CLion[1] 的教程.

## 安装

参见 Download CLion[2].

## 配置

### 工具链安装

CLion 默认不带编译器，构建工具和调试工具，需要手动进行安装.

#### Windows

参见 Tutorial: Configure CLion on Windows | CLion Documentation[3].

值得一提的是 CLion 的 Windows 版本中自带了 MinGW，所以可以不用额外安装 MinGW 工具链.

#### Linux

##### Debian/Ubuntu 及其衍生发行版

```
sudo apt install make cmake # build tools
sudo apt install gcc g++ gdb # compiler and debugger
sudo apt install clang clang++ llvm lldb # you can also choose to use clang toolchain
```

##### Arch Linux 及其衍生发行版

```
sudo pacman -S make cmake # build tools
sudo pacman -S gcc g++ gdb # compiler and debugger
sudo pacman -S clang clang++ llvm lldb # you can also choose to use clang toolchain
```

##### Fedora/RHEL/CentOS/Rocky Linux

```
sudo dnf install make cmake # build tools
sudo dnf install gcc g++ gdb # compiler and debugger
sudo dnf install clang clang++ llvm lldb # you can also choose to use clang toolchain
```

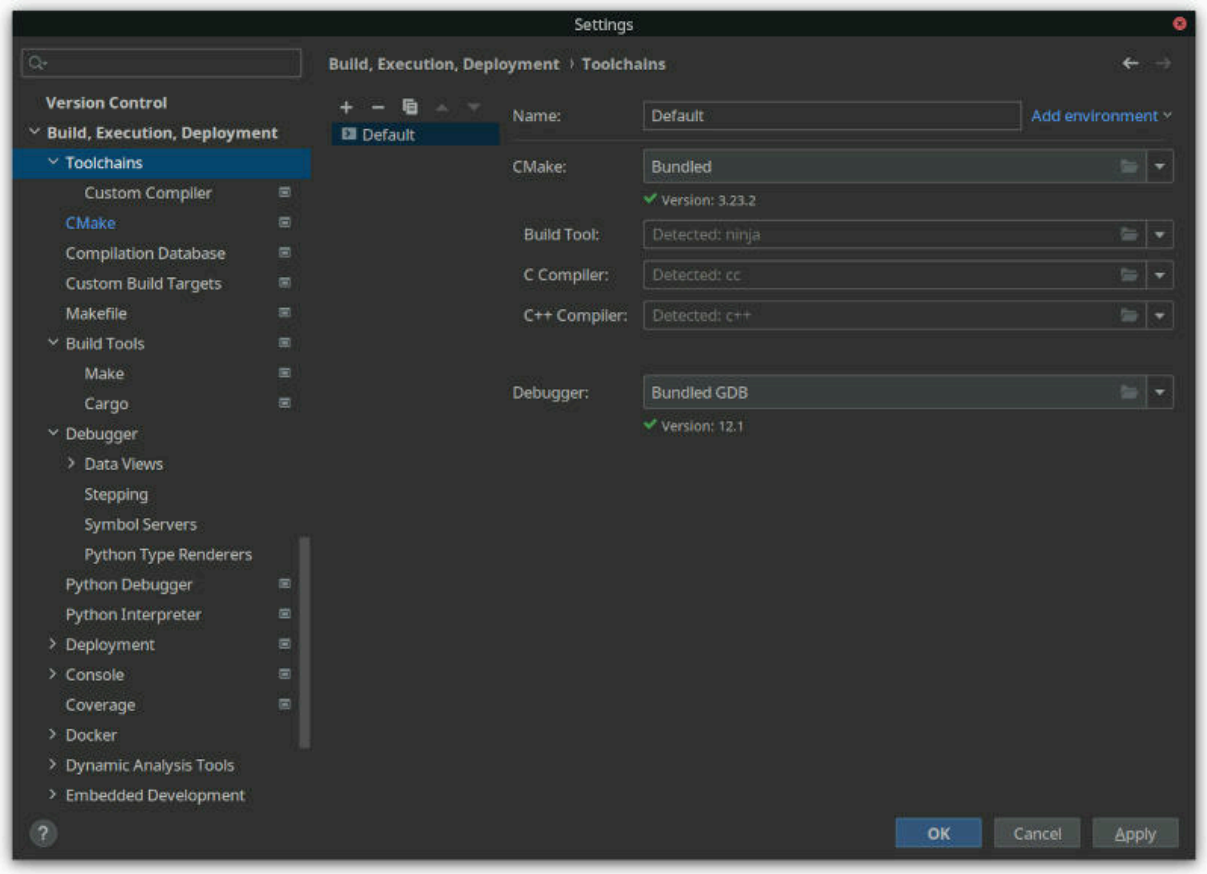
**macOS**

参见 Tutorial: Configure CLion on macOS | CLion Documentation[4].

## 工具链设置

### 手动设置工具链

新安装的 CLion 会自动检测系统中的 C/C++ 开发工具链，如果已安装的工具链无法自动检测到，可在 `Settings` 中找到 `Build, Execution, Deployment>Toolchains` 进行手动配置.

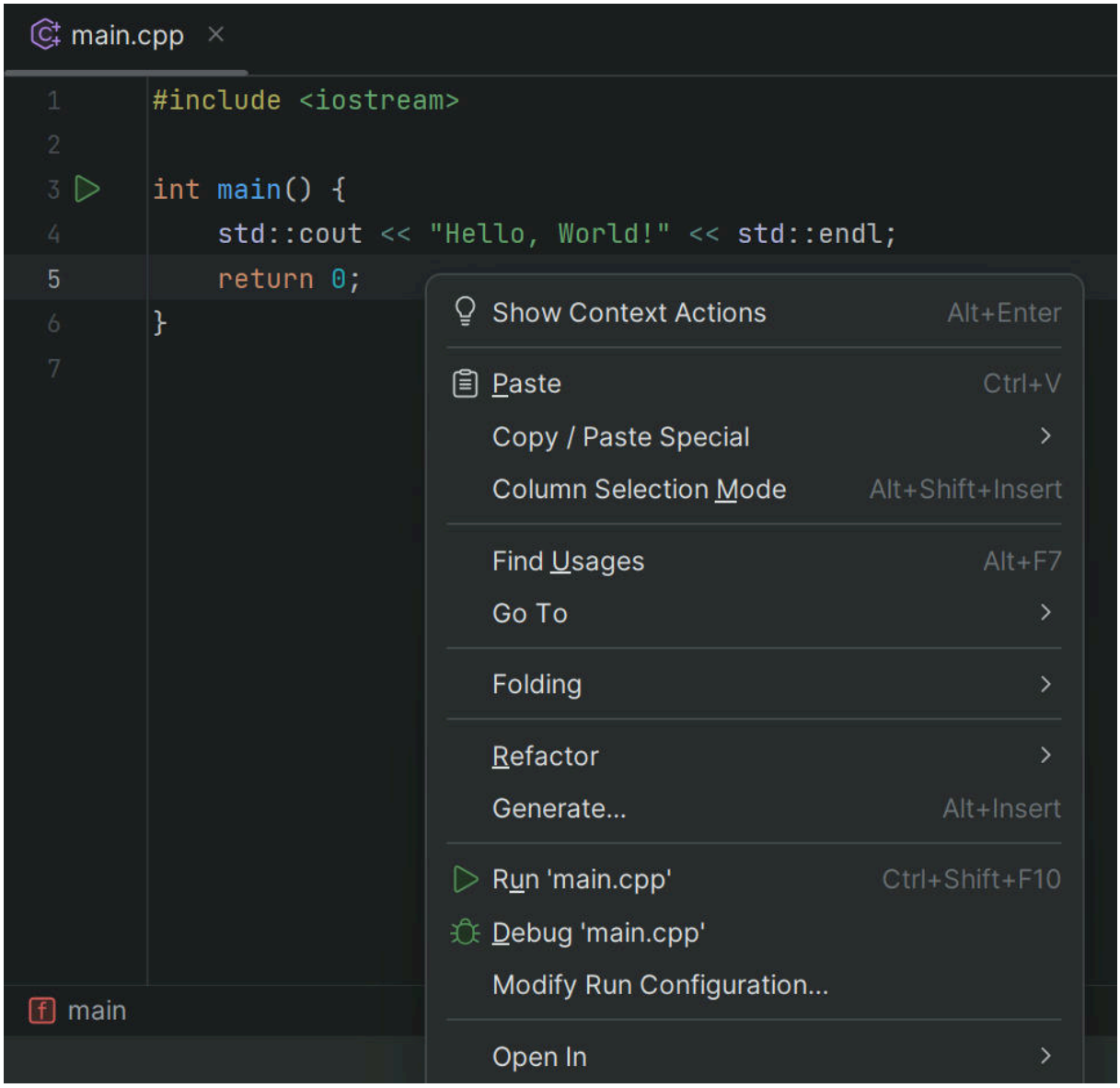



## 编译、运行和调试

虽然 CLion 诞生之初是面向多文件的复杂 C/C++ 项目诞生的，早些时候的 CLion 默认使用 CMake[5] 作为构建工具，但是自 CLion 2022.3 版本起，CLion 已经支持 C, C++ 单文件运行[6].

有多种方式来运行一个 C++ 程序，一个简单的流程如下：

1. 创建一个 C/C++ 项目：`New -> Project -> C++ Executable`，选择合适的地址和语言标准版本，点击 `Create`.
2. 打开项目，此时的项目目录下应当存在一个 `cmake-build-debug` 目录、一个 `CMakeLists.txt` 文件和一个 `main.cpp` 文件. 因为我们不需要使用 CMake 来管理项目，因此我们可以删去 `CMakeLists.txt` 文件和 `cmake-build-debug` 目录及其内所有文件.
3. 点击打开 `main.cpp` 文件，并在编辑区右键单击，可以看到 `Run 'main.cpp'` 选项. 选择此选项后，CLion 可以自动创建一个运行配置并运行程序.

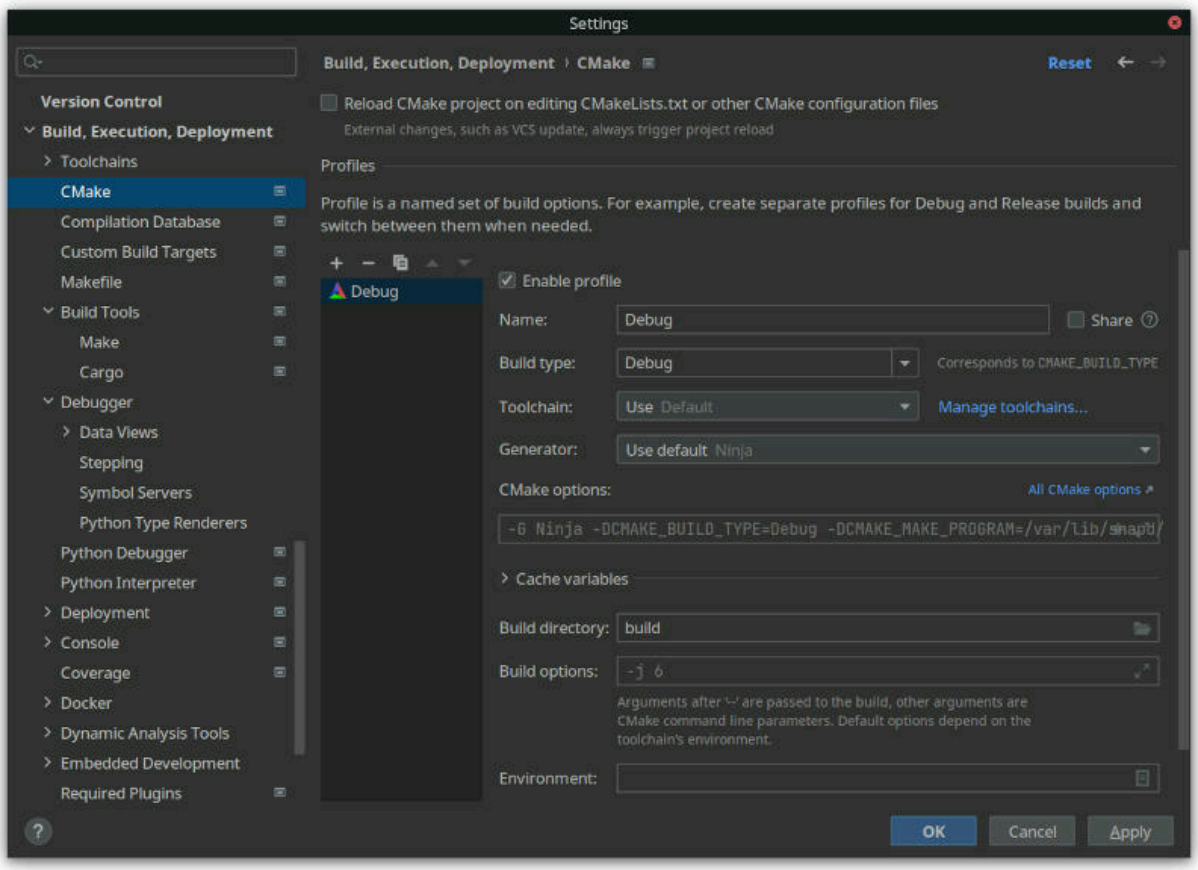



如需调试程序，可以编辑区打好断点，在编辑区右键单击，选择 `Debug 'main.cpp'` 选项.

### 通过 CMake 编译、运行和调试

#### 设置

CLion 也可使用 CMake[7] 作为构建工具，关于 CMake 的设置可以在 `Build, Execution, Deployment -> Toolchains -> CMake` 中修改.




### 编译选项

CMake 默认使用项目根目录下的 `CMakeList.txt` 作为构建项目的配置文件，可以使用 `add_compile_options` 命令来增加编译选项，例如：

```
1 add_compile_options(-std=c++17 -DDEBUG)
```

其他 CMake 的功能请参考 CMake 官方文档[8].

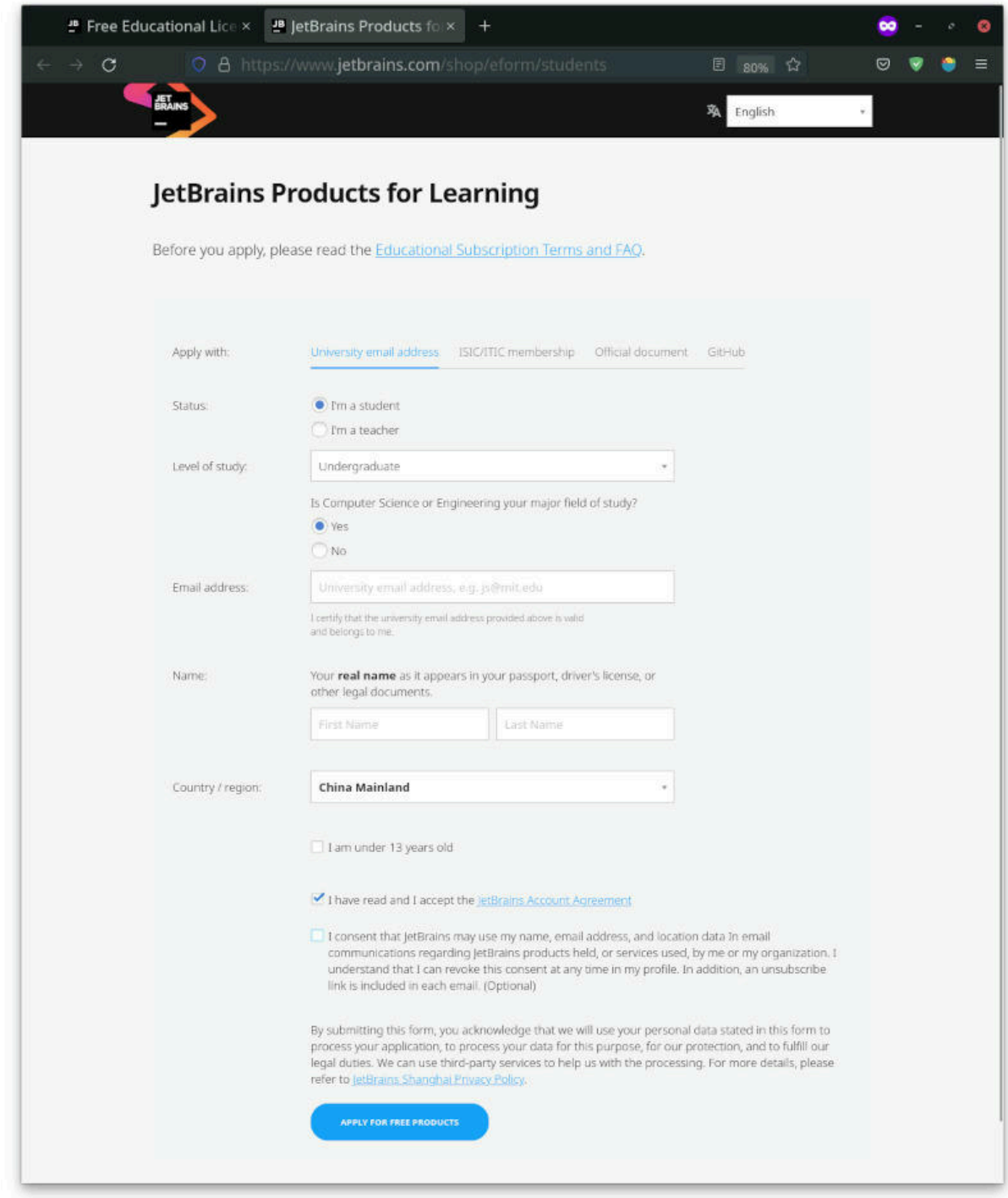
## 免费获取 CLion IDE 许可证

CLion 为付费产品，但是可以通过教育邮箱或开源项目申请特殊许可证．申请之后不仅可以免费使用正版 CLion IDE，还可以免费使用 JetBrains 公司开发的其他付费产品.

> **Note**
>
> 自 2025 年 5 月起，CLion 对非商业用途免费．[9]
>
> 根据 Toolbox 非商业用途订阅协议中的定义，商业产品是指有偿分发或提供或者作为您的商业活动的一部分使用的产品．但某些类别被明确排除在这一定义之外．常见的非商业用例包括学习和自我教育、任何形式的内容创作、开源代码和业余爱好开发.

### 使用教育邮箱获取

进入官网的 Free Educational Licenses 页面[10]，点击 `Apply` 按钮，填写相关信息即可申请.

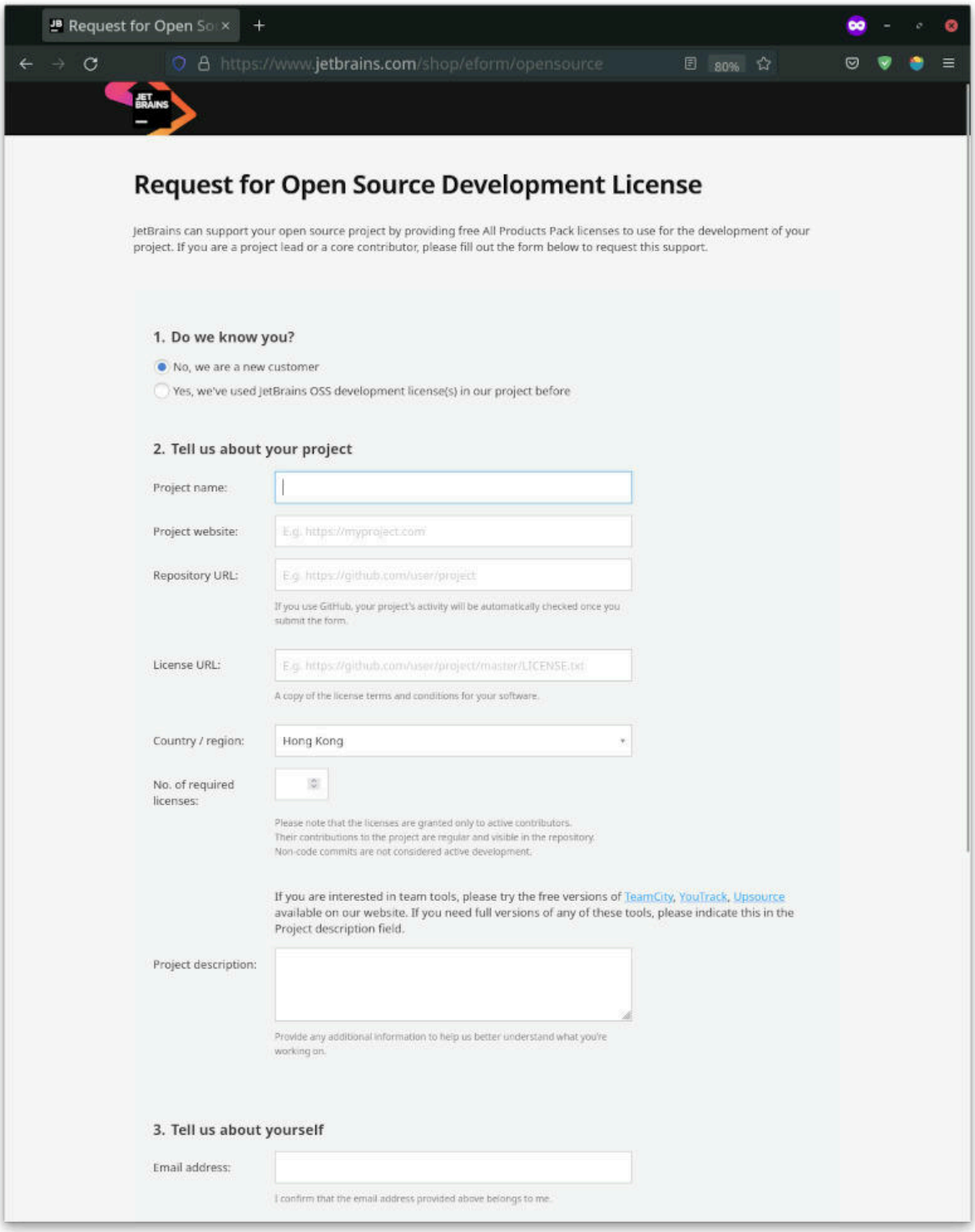



注意：在注册时于邮箱选项请填如 @edu.cn 后缀的教育邮箱，特殊许可证需要邮箱验证后方可拿到.

你可以到所在高校的教务中心官网去申请教育邮箱，如果申请不到需要使用 学信网[11] 进行认证（仅中国大陆）.

### 使用开源项目获取

如果您是某个开源项目的核心开发者或维护者之一，您可以尝试申请开源开发许可证 (Open Source Development License). 申请流程与教育许可证类似，但需要填写开源项目的仓库地址.




## 外部参考资料

[1] 学习 CLion

[2] Download CLion

[3] Tutorial: Configure CLion on Windows | CLion Documentation

[4] Tutorial: Configure CLion on macOS | CLion Documentation

[5] CMake

[6] C, C++ 单文件运行

[7] CMake

[8] CMake 官方文档

[9] 自 2025 年 5 月起，CLion 对非商业用途免费.

[10] Free Educational Licenses 页面  [11] 学信网 

## 3.2.10 Geany

**Authors:** xingjiapeng, MingqiHuang

Geany 是一个轻量、便捷的编辑器，对于 Linux 环境下的初学者较为友好.

与 Dev-C++ 一样，它可以编译运行单个文件.

不过，它可以在 Linux/Windows/macOS 下运行.

其官网为：https://geany.org/[1]

### 优缺点

#### 优点

1. 轻量；
2. 可以编译运行单个文件；
3. 不需要太多配置；
4. 跨平台.

#### 缺点

1. 没有太多人使用；
2. 在 macOS Catalina 下有一些权限问题[14]；
3. 新建文件时，默认不会有语法高亮，需要手动切换文件类型.

### 安装

参见 Download | Geany[3]

### 使用技巧

#### 切换文件类型

在文档 -> 设置文件类型中进行切换.

如 C++ 语言，点击文档 -> 设置文件类型 -> 编程语言 -> *C++* 源文件，即可看到文件已被转换为 C++ 语言的语法高亮了.

#### 设置文件模板

在配置文件目录下建立 templates/files 文件夹，建立在其中的文件即为模板文件，再次打开 Geany，就可以在文件 -> 从模板新建中找到它了.

配置文件目录可以通过帮助 -> 调试信息的第二、三行找出.

这里给出 macOS 和 Linux 下的默认模板配置文件目录：

- 系统目录：`/usr/share/geany/templates/files/`
- 用户目录：`~/.config/geany/templates/files/`[15]

### 常见问题

#### 兼容深度终端

在首选项 -> 工具 -> 虚拟终端，修改终端的命令为：

[14] 详见：https://github.com/geany/geany/issues/2344[2]

[15] 来源：https://wiki.geany.org/config/templates[4]

```
deepin-terminal -x "/bin/sh" %c
```

点击「应用」按钮即可。[16]

## 参考资料与注释

[1] https://geany.org/

[2] https://github.com/geany/geany/issues/2344

[3] Download | Geany

[4] https://wiki.geany.org/config/templates

[5] https://wiki.deepin.org/

## 3.2.11 Xcode

**Authors:** shenyouran, Xeonacid, StudyingFather, CoelacanthusHex

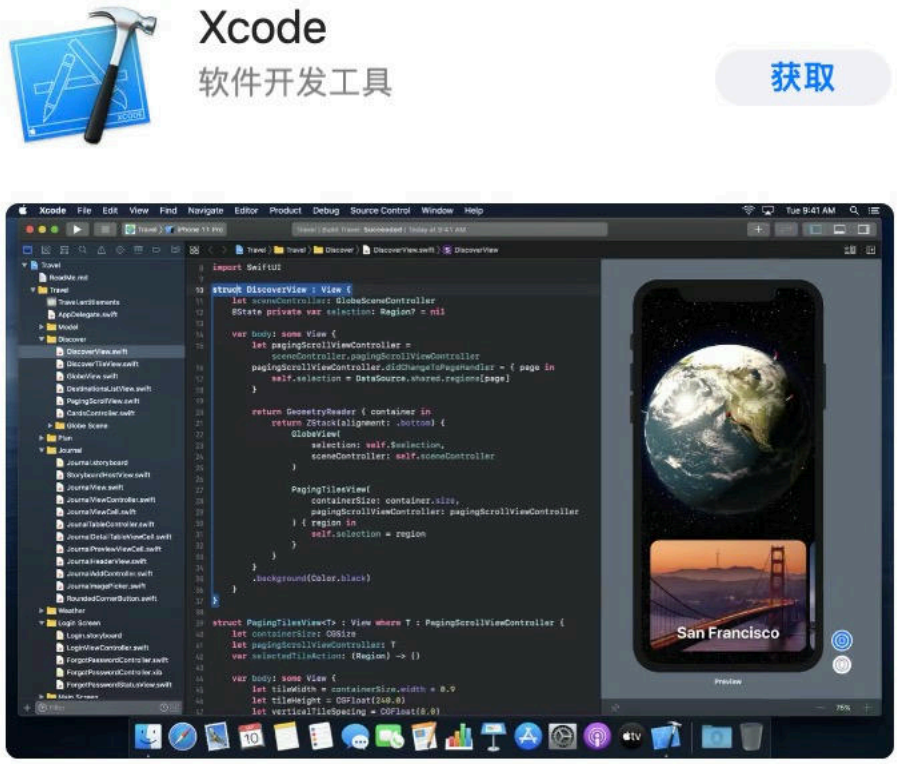
## 简介

Xcode 是一个运行在 macOS 上的集成开发工具（IDE），由 Apple Inc. 开发。

## 安装

### 方法一

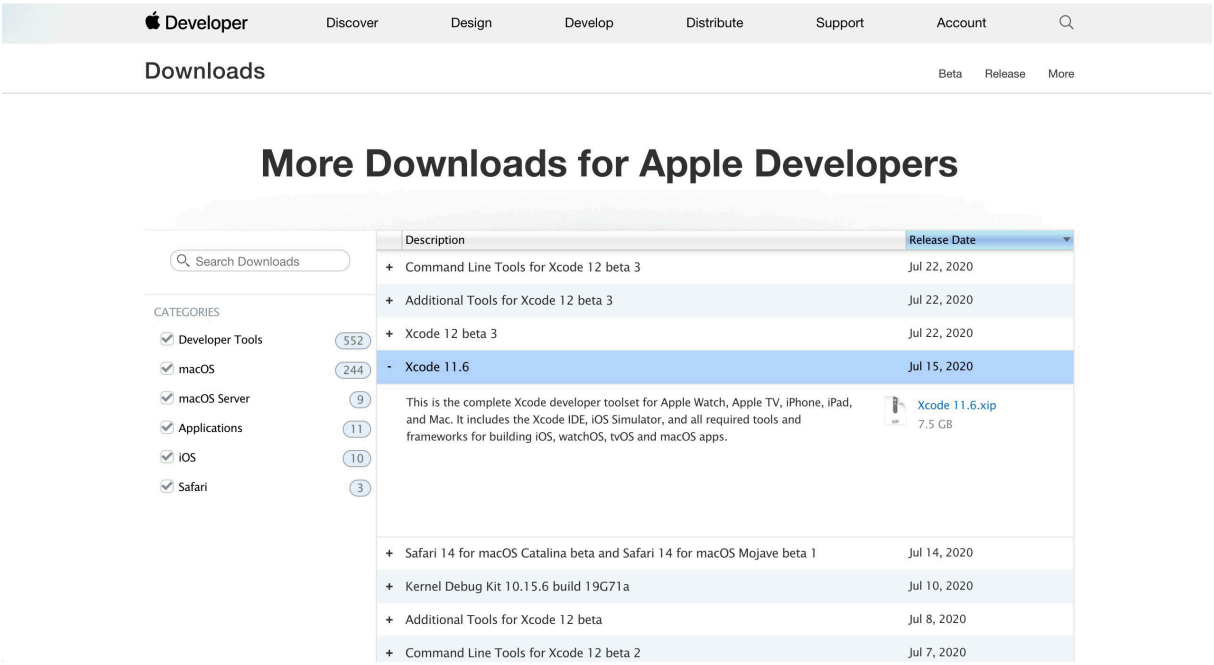
打开苹果电脑自带的 App Store（或者尝试 快捷链接[1]）下载 Xcode。点击获取，然后输入苹果账号密码开始下载安装。




[16] 来源：Deepin Wiki https://wiki.deepin.org/[5]

### 方法二

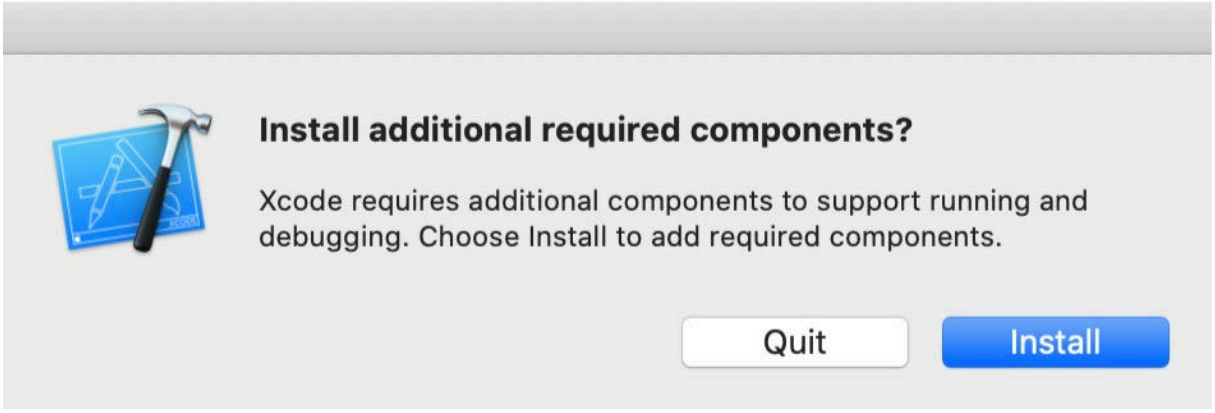
访问 苹果开发者下载页面[2]，用苹果账号登录，然后找到 Xcode 最新的稳定版本安装包（即不含 Beta 的最新版本，此处为 11.6）：




点击弹出框内蓝色的文件名即可下载．得到压缩包之后，用系统自带的工具进行解压，然后得到文件 Xcode.app．把这个文件移动到【应用程序】文件夹后即可使用．

### 基础配置

首次打开 Xcode 时，可能会遇到下列弹出窗口：

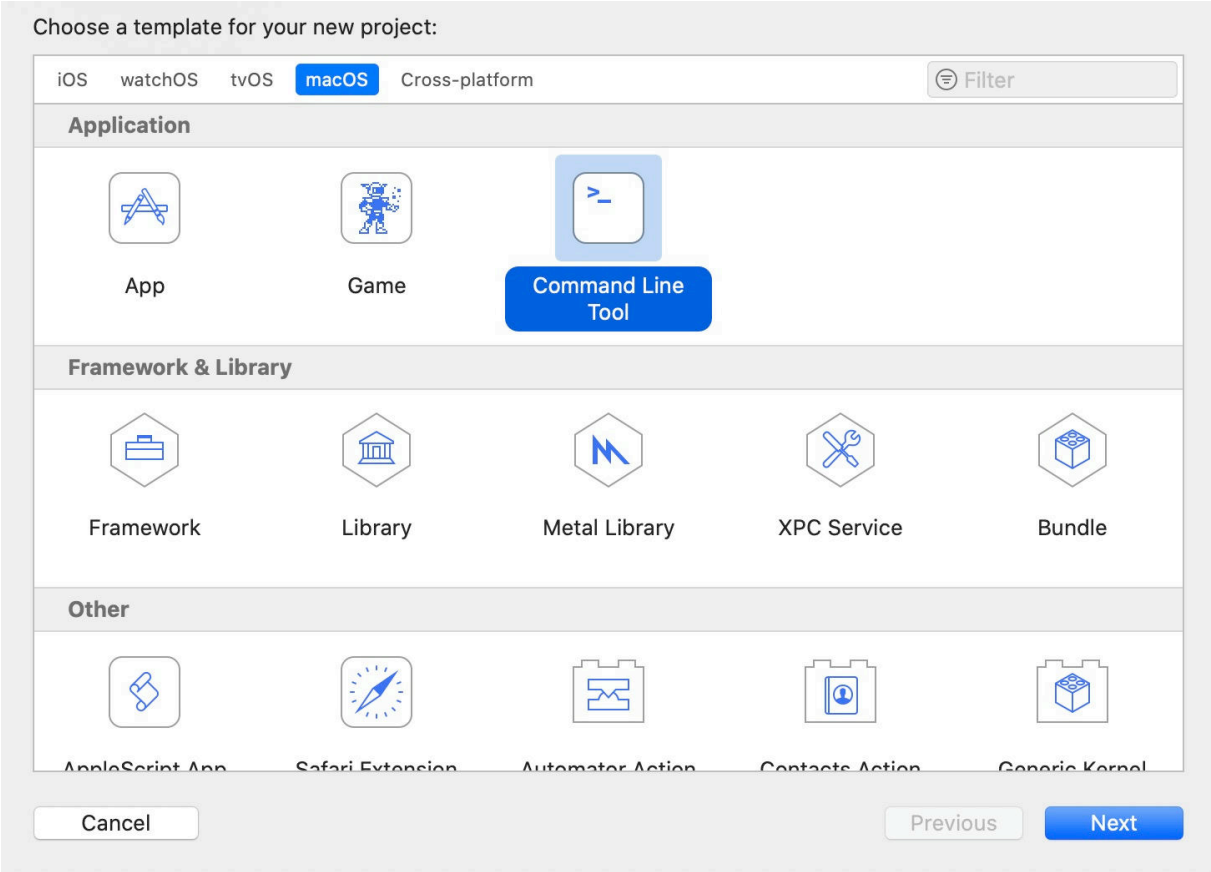



这个窗口是 Xcode 元件的安装引导．点击 `Install` 并输入当前用户密码即可．

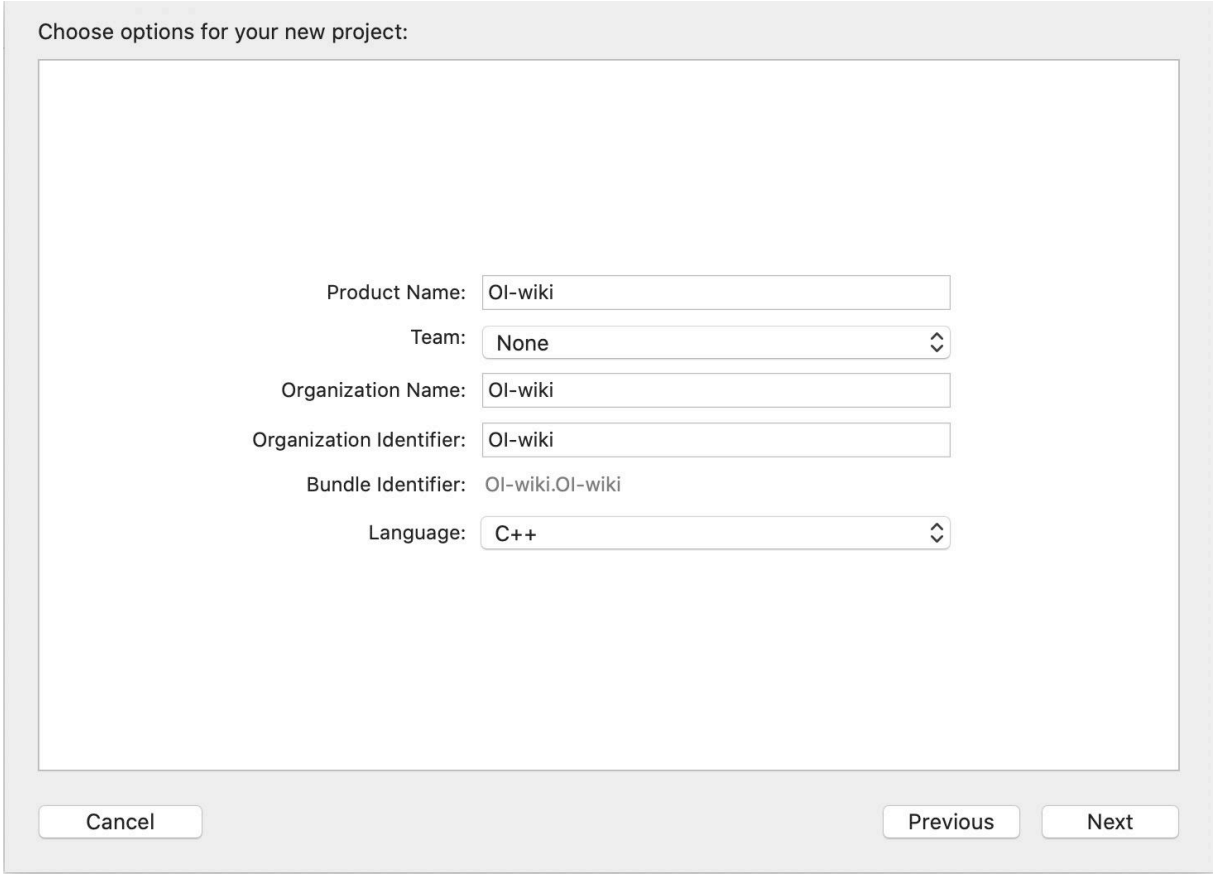
安装完毕后，界面左侧显示：




点击`Create a new Xcode project`（创建一个新的Xcode项目），然后选择上方`macOS`中的`Command Line Tool`（命令行工具），并点击右下角的`Next`.

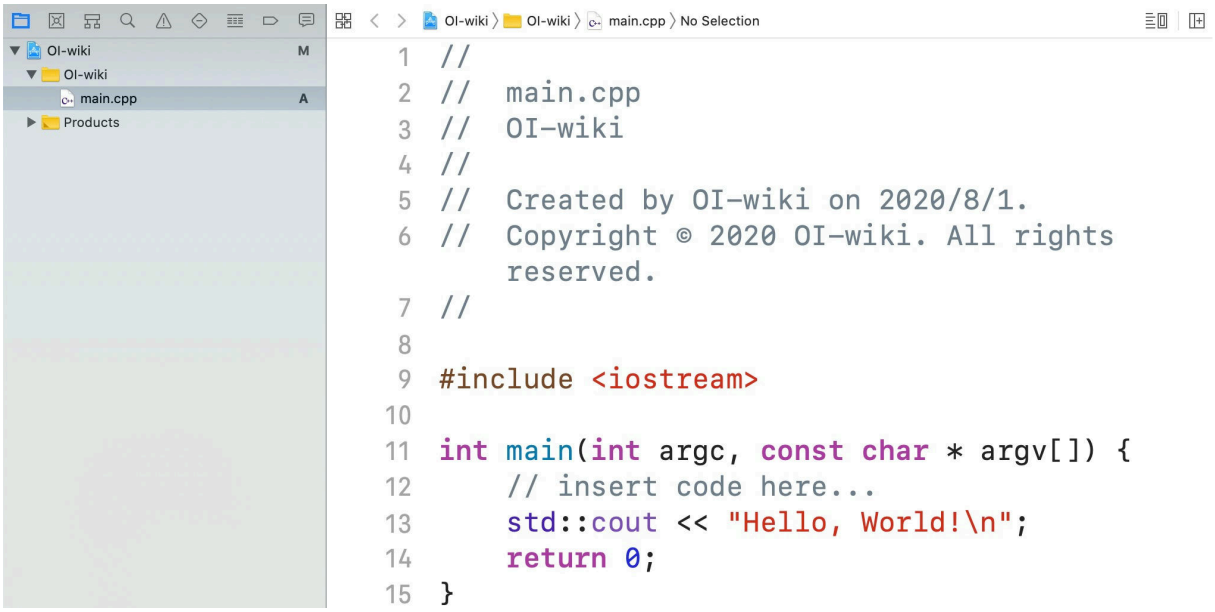



接下来，我们可以给项目命名，但最重要的是选择项目的语言．我们可以根据自己的需求，在最下方 Language 处选择 C 或者 C++：

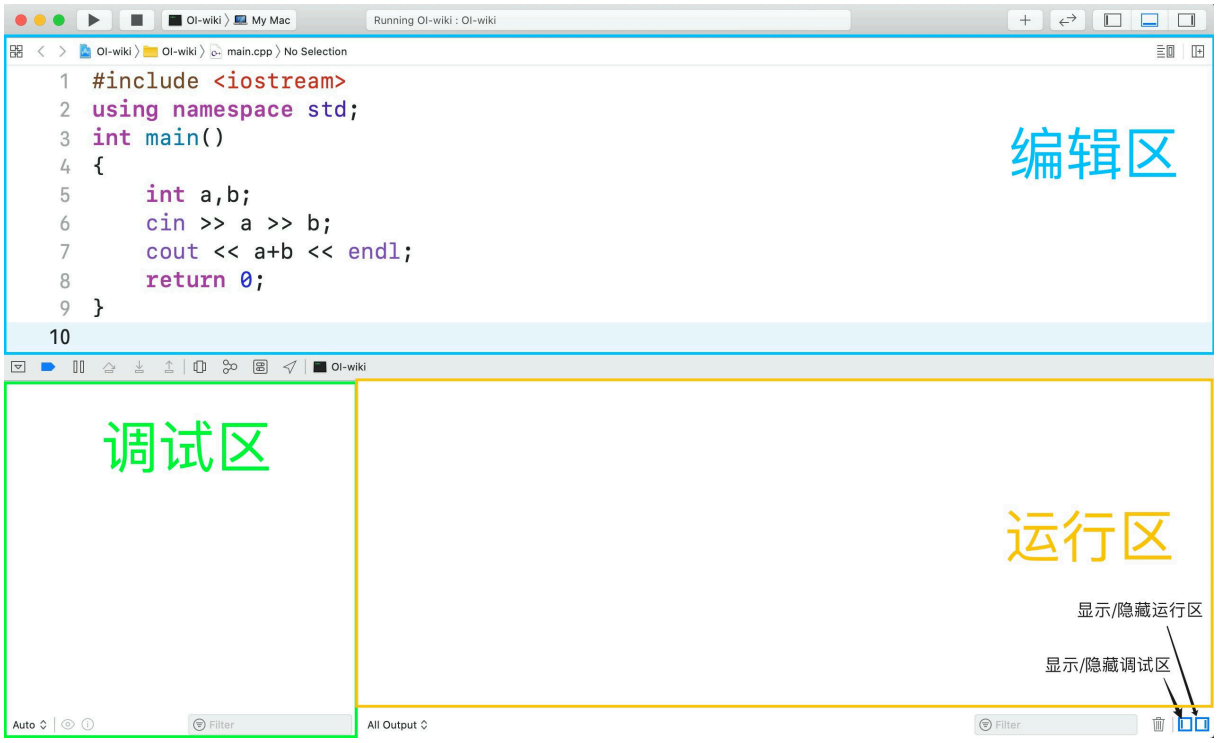



项目的目录可以根据需要选择．创建完毕后，Xcode 会自动打开这个项目，并自动创建一个 `main` 文件（C 语言的后缀为 `.c`，C++ 语言的后缀为 `.cpp`）．

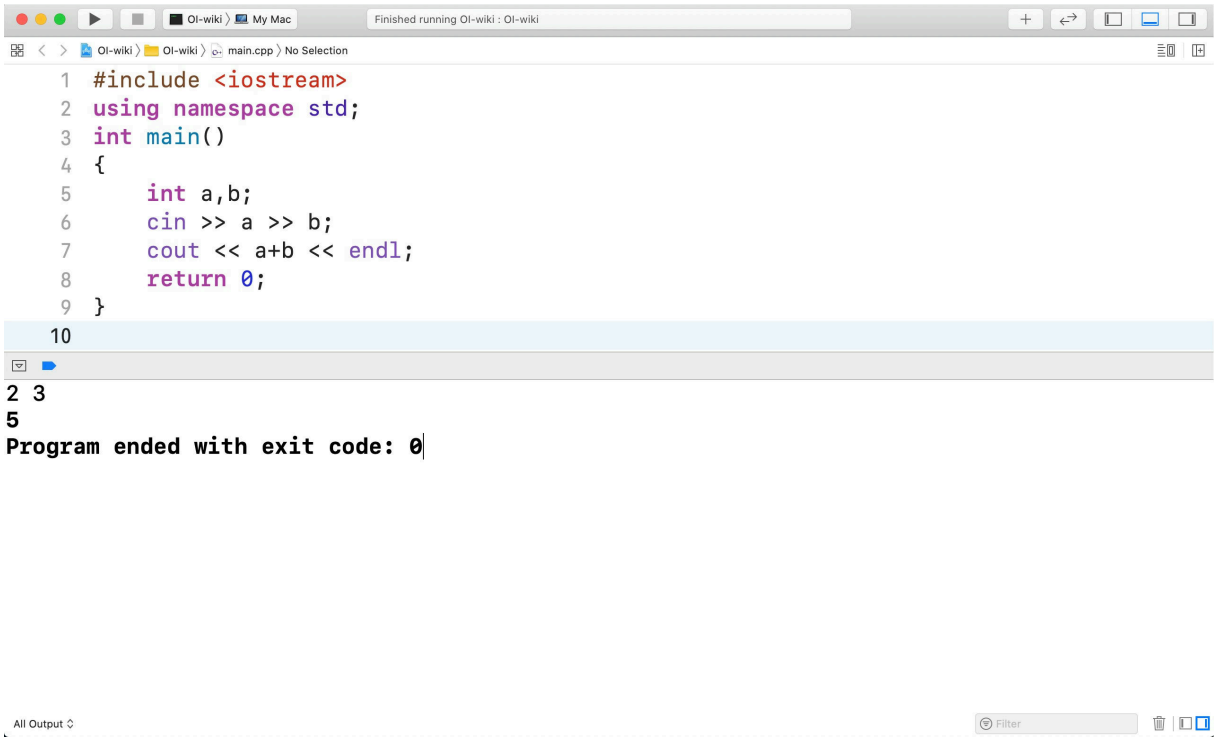
点击这个文件，就可以打开编辑区域：




编写代码后，可以按⌘B 编译（Build），⌘R 运行（Run）．运行后拖动，得到三个部分：




一般来说我们只使用【编辑区】和【运行区】. 若程序有输入，那么在【运行区】中进行输入之后，就可以得到输出. 界面呈现效果:




仿照这种方式，我们就可以运行任何的单个 C/C++ 程序.

## 万能头文件的使用

在编写代码过程中，我们可能会使用到很多头文件. 常用的解决方法是使用万能头文件.

我们在源代码第一行引入万能头文件，然而编译过程中却提示: `'bits/stdc++.h' file not found.` 即该头文件未找到.

```
1 #include<bits/stdc++.h>                    'bits/stdc++.h' file not found
```

这是因为在 macOS 上默认使用 libc++[3] 作为 C++ 标准库实现，而万能头 `bits/stdc++.h` 是 GNU libstdc++[4] 所独有的.

不过，我们可以手动编写一个万能头文件来使用.

### 步骤 1

打开终端（Terminal.app），前往 Xcode 存储头文件的文件夹，即:

```
cd /Applications/Xcode.app/Contents/Developer/Toolchains/
XcodeDefault.xctoolchain/usr/include/c++/v1
```

如果 Xcode 版本大于等于 12.5，那么

```
cd /Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/
Developer/SDKs/MacOSX.sdk/usr/include/c++/v1/
```

### 步骤 2

创建 `bits` 文件夹并进入:

```
mkdir bits
cd bits
```

用 vim 创建 stdc++.h 文件:

```
vim stdc++.h
```

界面如下:




接着，我们需要通过 vim 编辑文件. 敲击 i（insert）键盘即可进入插入/编辑模式（下方出现 `-- INSERT --`）:




将下面这段代码块复制并粘贴到终端中：

**万能头文件代码块**

```
// C++ includes used for precompiling -*- C++ -*-

// Copyright (C) 2003-2020 Free Software Foundation, Inc.
//
// This file is part of the GNU ISO C++ Library.  This library is free
// software; you can redistribute it and/or modify it under the
// terms of the GNU General Public License as published by the
// Free Software Foundation; either version 3, or (at your option)
// any later version.

// This library is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.

// Under Section 7 of GPL version 3, you are granted additional
// permissions described in the GCC Runtime Library Exception, version
// 3.1, as published by the Free Software Foundation.

// You should have received a copy of the GNU General Public License and
// a copy of the GCC Runtime Library Exception along with this program;
// see the files COPYING3 and COPYING.RUNTIME respectively.  If not, see
// <http://www.gnu.org/licenses/>.

/** @file stdc++.h
 *  This is an implementation file for a precompiled header.
```

```
27  */
28
29 // 17.4.1.2 Headers
30
31 // C
32 #ifndef _GLIBCXX_NO_ASSERT
33 #include <cassert>
34 #endif
35 #include <cctype>
36 #include <cerrno>
37 #include <cfloat>
38 #include <ciso646>
39 #include <climits>
40 #include <clocale>
41 #include <cmath>
42 #include <csetjmp>
43 #include <csignal>
44 #include <cstdarg>
45 #include <cstddef>
46 #include <cstdio>
47 #include <cstdlib>
48 #include <cstring>
49 #include <ctime>
50 #include <cwchar>
51 #include <cwctype>
52
53 #if __cplusplus >= 201103L
54 #include <ccomplex>
55 #include <cfenv>
56 #include <cinttypes>
57 #include <cstdbool>
58 #include <cstdint>
59 #include <ctgmath>
60 /* https://stackoverflow.com/a/25892335/15125422 */
61 #if defined(__GLIBCXX__) || defined(__GLIBCPP__)
62 #include <cstdalign>
63 #include <cuchar>
64 #endif
65 #endif
66
67 // C++
68 #include <algorithm>
69 #include <bitset>
70 #include <complex>
71 #include <deque>
72 #include <exception>
73 #include <fstream>
74 #include <functional>
75 #include <iomanip>
76 #include <ios>
77 #include <iosfwd>
78 #include <iostream>
```

```
#include <istream>
#include <iterator>
#include <limits>
#include <list>
#include <locale>
#include <map>
#include <memory>
#include <new>
#include <numeric>
#include <ostream>
#include <queue>
#include <set>
#include <sstream>
#include <stack>
#include <stdexcept>
#include <streambuf>
#include <string>
#include <typeinfo>
#include <utility>
#include <valarray>
#include <vector>

#if __cplusplus >= 201103L
#include <array>
#include <atomic>
#include <chrono>
#include <codecvt>
#include <condition_variable>
#include <forward_list>
#include <future>
#include <initializer_list>
#include <mutex>
#include <random>
#include <ratio>
#include <regex>
#include <scoped_allocator>
#include <system_error>
#include <thread>
#include <tuple>
#include <type_traits>
#include <typeindex>
#include <unordered_map>
#include <unordered_set>
#endif

#if __cplusplus >= 201402L
#include <shared_mutex>
#endif

#if __cplusplus >= 201703L
#include <any>
#include <charconv>
```

```
// #include <execution>
#include <filesystem>
#include <memory_resource>
#include <optional>
#include <string_view>
#include <variant>
#endif

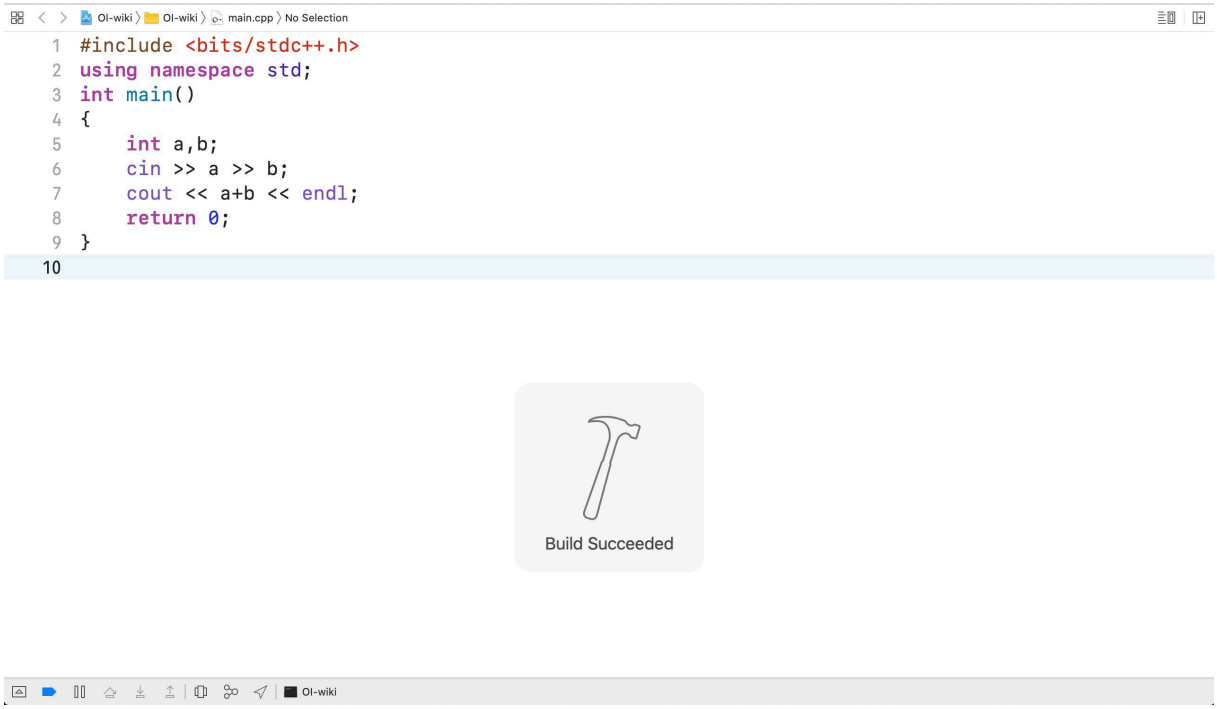
#if __cplusplus > 201703L
#include <bit>
#include <compare>
#include <concepts>
#include <numbers>
#include <ranges>
#include <span>
#include <stop_token>
// #include <syncstream>
#include <version>
#endif
```

该文件来源于 10.2.0 版本的 libstdc++[5] 并经少许修改以兼容 libc++.

按键盘左上角的 Esc 退出编辑模式，然后直接输入 :wq 并换行即可保存文件.

### 步骤 3

关闭终端，回到 Xcode. 重新按下 ⌘B/⌘R 进行编译，发现编译成功:




## 优缺点

优点：由苹果开发，适合 Mac 用户，界面齐全、美观.

缺点：Xcode 主要用来苹果程序的开发，对于竞赛来说功能冗余，安装包大小较大，而且仅能在 Mac 端上使用.

### 外部参考资料

[1] 快捷链接 [2] 苹果开发者下载页面 [3] libc++

[4] GNU libstdc++ [5] 10.2.0 版本的 libstdc++

## 3.2.12 GUIDE

GUIDE（GAIT Universal IDE）是由北航 GAIT 研究组开发的、专门为 NOI 选手设计的、支持 C/C++/Pascal 三种程序设计语言的小型集成开发环境.

> **Note**
>
> 自 2021 年 9 月 1 日起启用的 NOI Linux 2.0 不再包含 GUIDE. [17]

### 安装

#### Windows

参见 https://www.noi.cn/xw/2009-03-23/714714.shtml[2].

#### Linux

参见 https://www.noi.cn/xw/2009-03-23/714714.shtml[3] 或按照如下步骤安装.

##### 需要的动态库文件及包名

| 动态库 | Arch 包名 | Debian 包名 | Fedora 包名 | openSUSE x86 包名 | openSUSE x86_64 包名 |
|---|---|---|---|---|---|
| libpng12.so.0 | lib32-libpng12 | libpng12 | libpng12 | libpng12-0 | libpng12-0-32bit |
| libSM.so.6 | lib32-libsm | libsm6 | libSM | libSM6 | libSM6-32bit |
| libICE.so.6 | lib32-libice | libice6 | libICE | libICE6 | libICE6-32bit |
| libXi.so.6 | lib32-libxi | libxi6 | libXi | libXi6 | libXi6-32bit |
| libXrender.so.1 | lib32-libxrender | libxrender1 | libXrender | libXrender1 | libXrender1-32bit |
| libXrandr.so.2 | lib32-libxrandr | libxrandr | libXrandr | libXrandr2 | libXrandr2-32bit |
| libfreetype.so.6 | lib32-freetype2 | libfreetype6 | freetype | libfreetype6 | libfreetype6-32bit |
| libfontconfig.so.1 | lib32-fontconfig | libfontconfig1 | fontconfig | libfontconfig1 | libfontconfig1-32bit |
| libXext.so.6 | lib32-libxext | libxext6 | libXext | libXext6 | libXext6-32bit |
| libX11.so.6 | lib32-libx11 | libx11-6 | libX11 | libX11-6 | libX11-6-32bit |
| libz.so.1 | lib32-zlib | zlib1g | zlib | libz1 | libz1-32bit |
| libgthread-2.0.so.0 | lib32-glib2 | libglib2.0-0 | glib2 | libgthread-2_0-0 | libgthread-2_0-0-32bit |
| libglib-2.0.so.0 | lib32-glib2 | libglib2.0-0 | glib2 | libglib2_0-0 | libglib2_0-0-32bit |
| libstdc++.so.6 | lib32-gcc-libs | libstdc++6 | libstdc++ | libstdc++6 | libstdc++6-32bit |
| libgcc_s.so.1 | lib32-gcc-libs | lib32gcc1 | libgcc | libgcc_s1 | libgcc_s1 |
| librt.so.1 | lib32-glibc | libc6 | glibc | glibc | glibc-32bit |
| libpthread.so.0 | lib32-glibc | libc6 | glibc | glibc | glibc-32bit |
| libdl.so.2 | lib32-glibc | libc6 | glibc | glibc | glibc-32bit |

[17] NOI Linux 2.0 发布，将于 9 月 1 日起正式启用！ [1]

| | | | | | |
|---|---|---|---|---|---|
| libm.so.6 | lib32-glibc | libc6 | glibc | glibc | glibc-32bit |
| libc.so.6 | lib32-glibc | libc6 | glibc | glibc | glibc-32bit |

#### 在 Debian 或 Ubuntu 安装

```
sudo apt install -y libpng12 libsm6 libice6 libxi6 libxrender1 libxrandr libfreetype6 libfontconfig1 libxext6 libx11-6 zlib1g libglib2.0-0 libglib2.0-0 libstdc++6 lib32gcc1 libc6
wget -c http://download.noi.cn/T/noi/GUIDE-1.0.2-ubuntu.tar
tar -xvf GUIDE-1.0.2-ubuntu.tar
cd GUIDE-1.0.2-ubuntu
echo "install:\n\tinstall -Dm755 -t /usr/bin GUIDE\n\tinstall -Dm644 -t /usr/share/ lang_en.qm\n\tmkdir -p /usr/share/apis/ && cp -r apis/* /usr/share/apis/\n\tmkdir -p /usr/share/doc/GUIDE/ && mkdir -p /usr/share/doc/GUIDE/html/ && cp -r doc/*  /usr/share/doc/GUIDE/html/" > Makefile
sudo apt install -y checkinstall
sudo checkinstall --pkgname "GUIDE" --pkgversion "1.0.2" -y
```

#### 在 openSUSE 安装

按照 openSUSE/opi[4] 给出的方式安装 opi.

然后：（32 位用户自行删去 `-32bit`）

```
sudo opi checkinstall
sudo zypper install -n {libpng12-0,libSM6,libICE6,libXi6,libXrender1,libXrandr2,libfreetype6,libfontconfig1,libXext6,libX11-6,zlib1g,libglib2.0-0,libstdc++6,libgcc_s1,glibc}-32bit
wget -c http://download.noi.cn/T/noi/GUIDE-1.0.2-ubuntu.tar
tar -xvf GUIDE-1.0.2-ubuntu.tar
cd GUIDE-1.0.2-ubuntu
echo "install:\n\tinstall -Dm755 -t /usr/bin GUIDE\n\tinstall -Dm644 -t /usr/share/ lang_en.qm\n\tmkdir -p /usr/share/apis/ && cp -r apis/* /usr/share/apis/\n\tmkdir -p /usr/share/doc/GUIDE/ && mkdir -p /usr/share/doc/GUIDE/html/ && cp -r doc/*  /usr/share/doc/GUIDE/html/" > Makefile
sudo checkinstall --pkgname "GUIDE" --pkgversion "1.0.2" -y -rpmi
```

### 编辑文件

点击页面上方工具栏的「新文件」按钮（或者使用 `Ctrl` + `N` 快捷键）来创建一个新文件.

在默认情况下，GUIDE 的代码字体并非等宽字体，看上去非常不美观，因此需要在设置中更改字体.

在 编辑 -> 选项 -> 语法高亮设置 中，点击「全部字体」按钮，即可切换编辑器字体.

需要注意的是，对于未保存的新文件，字体仍然是默认字体. 因此建议在开始编辑前先保存文件（点击工具栏的「保存」按钮，或按下 `Ctrl` + `S` 快捷键），再进行编辑.

### 编译与运行

在编辑完源代码后，点击工具栏的「编译」按钮（或 `F7` 快捷键）进行编译.

> **更改编译选项**
>
> GUIDE 没有设置默认编译选项的功能，用户只能更改对某个文件的编译选项.
>
> 右键点击想要更改编译选项的文件的标签，选择 **设置编译命令** 选项，即可更改该文件的编译选项.

如果源代码正常编译，点击工具栏的「运行」按钮（或 Ctrl + F5 快捷键）即可运行程序.

### 调试

GUIDE 自带的调试功能存在很多 bug（如程序中途发生崩溃等），因此不推荐直接使用 GUIDE 的调试功能.

建议直接在 终端→3.4@p232 下使用 gdb 来进行调试.

[1] NOI Linux 2.0 发布，将于 9 月 1 日起正式启用！ 

[2] https://www.noi.cn/xw/2009-03-23/714714.shtml 

[3] https://www.noi.cn/xw/2009-03-23/714714.shtml 

[4] openSUSE/opi 

## 3.2.13 Sublime Text

### 简介

Sublime Text（以下简称 ST，后附数字作为版本区分，如无则各版本都适用）是一款轻量级的文本编辑器，支持多种语言的语法高亮及代码补全. 具有高度的可拓展性以及 Vim 模式，特别的热启动模式大幅减小了文件丢失的可能.

新版 NOI Linux 中支持版本为 ST3 最后一个版本 3.2.2[18]，故这里以 ST3 为主. 目前 ST4 正式版已经发布[19]，现在如仍使用 ST3 会提示更新.

ST4 与 ST3 的重要差别会有额外补充，在介绍中如果提及某项在 ST3 已有翻译则使用中文，否则为 ST4 中的英文.

### 安装

ST4 的安装方法参见 Sublime Text 4 的下载页面[3].

ST3 的安装方法参见 Sublime Text 3 的下载页面[4].

> **提示购买**
>
> ST 是收费软件，但有一个无限期的试用，试用并不会带来功能上的缺失，但会不时弹出弹窗提示激活.

### 插件与自定义

#### 汉化

ST 并不支持中文，如需中文需要安装汉化插件.

##### 安装插件管理器

打开 ST 后键入 Ctrl + Shift + P 唤出命令框，输入 Install 后回车（完整命令是 Install Package Control，不区分大小写），此时应该会看到左下角有一个 = 在不停的左右移动. Package Control 安装完成（或失败）后会有弹窗提示，具体的加载时间取决于网络.

如果完成的弹窗显示安装失败（与网络有关），则需要手动下载 Package Control[5] 并将下载好的文件放到 ST 的数据目录下的 \Installed Packages 文件夹中. 稍作等待，ST 会自动识别该插件（有时需要重启 ST）.

[18] NOI Linux 2.0 发布[1]
[19] Sublime Text 4 发布[2]

**ST 数据目录的路径**

Windows 下，如果在 ST 的 **安装目录** 下存在 `\data` 文件夹，会自动使用（或初始化）该文件夹作为数据目录.

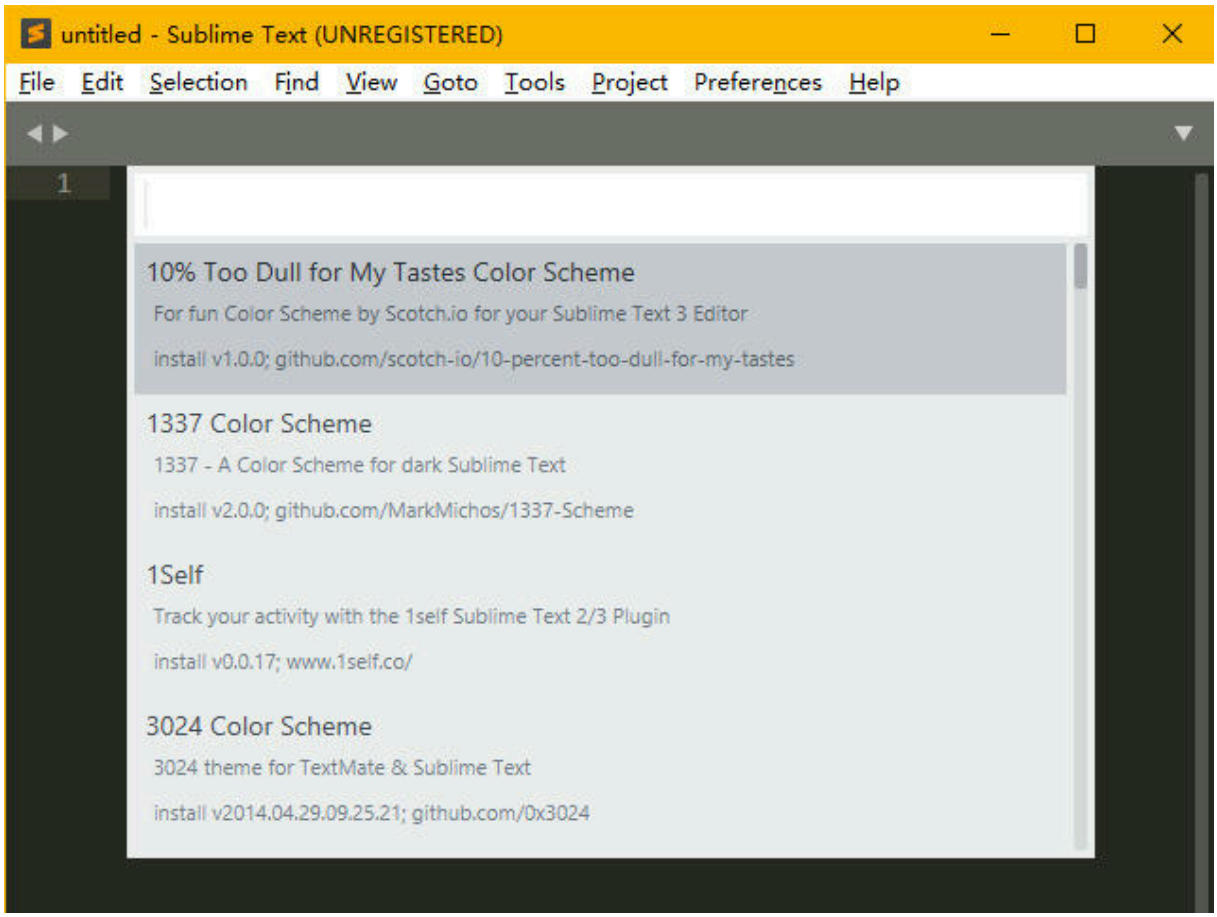
ST3 的路径一般为 `C:\Users\用户名\AppData\Roaming\Sublime Text 3`，ST4 一般为 `C:\Users\用户名\AppData\Roaming\Sublime Text`，ST 会先寻找对应版本的路径，如不存在则寻找更低版本的路径，如都不存在则新建并初始化.

在以 NOI Linux 所使用的 Ubuntu 20.04.1 中，ST3 的数据目录为 `$HOME/.config/sublime-text-3`，ST4 的数据目录为 `$HOME/.config/sublime-text`，使用的具体规则同 Windows 环境.

可以使用 `首选项->浏览插件目录...` 快速查看数据文件夹路径下的 `\Packages` 文件夹.

### 安装汉化插件

再次按下 `Ctrl` + `Shift` + `P` 输入 `Install` 后回车（完整命令是 `Package Contrl:Install Package`），等待加载完成，接下来应该是这个界面：




输入 `Chinese` 选择 ChineseLocalizations 并回车，等待安装完毕，完成后界面会自动切换为中文（如是 ST4，因为汉化插件未更新，会少一些新增的菜单项，但一般对编辑无影响）.

### 调整字体

进入 `首选项->设置`，在右边的用户设置中的花括号中添加一行 `"font_face": "字体名",`，ST 的设置使用 JSON 格式储存. 修改完成后保存，如果系统安装了对应字体会自动切换.

一般而言，如果单论对中文的显示的话，Microsoft Yahei Consolas 和 Microsoft YaHei Mono 是比较好的选择.

> ⚠ **Warning**
>
> 注意任何设置（包括插件设置）即使能也不要在左边的默认设置中修改，这是不被推荐的，如果软件（或插件）更新，默认设置会被重置.

### 安装插件与主题

安装插件与主题的方法与安装汉化插件一致.

输入 `Ctrl` + `Shift` + `P` 输入 `Install` 后回车，然后搜索插件/主题/配色即可.

插件推荐:

- BracketHighlighter : 对原版的括号高亮进行了增强，必备.
- C++ Snippets : ST 自带有 C++ 代码补全，格式为大括号不换行. 如果不习惯自带大括号换行的码风可以安装这个插件，同时增加了一部分补全.
- C++ 11 : 支持 C++ 11 标准高亮（ST4 中不需要）.
- SublimeAstyleFormatter : Astyle，用于格式化代码.
- Diffy : 按 `Ctrl` + `K` & `Ctrl` + `D` 即可快速比较第一视窗与第二视窗打开的文件的不同，比较方式为逐行比较.
- ConvertToUTF8 : 自动识别文件编码，并支持编码转换.
- SideBarEnhancements : 侧边栏增强，较为推荐.
- Transparency : 窗口透明化.

有其他需要可以尝试搜索.

一些主题：[20]

1337（单配色方案）、3024（单配色方案）、Grandson-of-Obsidian（单配色方案）、Seti_UI（单主题，额外包含 git 等格式的高亮，较为推荐）、Material Theme、Predawn、Agila、Materialize.

如果要编辑自己的配色方案，可以访问 tmTheme Editor[7].

如是 ST4 则可以在 `Preferences->Customize Color Scheme` 中调整配色方案或 `Preferences->Customize Scheme` 中调整主题.

### 开启 Vi Mode

ST 的开发者为 Vi 使用者提供了一个可选插件 Vintage，可模拟 Vi 的大部分功能.

#### 开启方式

Vintage 插件默认是禁用的. 可以通过 `首选项->设置` 在用户设置中，将 `"ignored_packages"` 一项中的 Vintage 删除并保存（不要整个删除，只删除 Vintage），ST 的状态栏左边就会出现 `INSERT MODE`，此时 Vintage 插件已开启.

或者按 `Ctrl` + `Shift` + `P`，然后输入 `Enable` 选择 `Package Control: Enable Package` 并回车，选择 Vintage 即可，该方法需要 Package Control.

#### 相关配置

如果想让上下左右键失效，可以在 `首选项->快捷键设置` 中添加:

```
{ "keys": ["left"], "command": ""},
{ "keys": ["right"], "command": ""},
{ "keys": ["up"], "command": ""},
{ "keys": ["down"], "command": ""},
```

如要使 ST 以命令模式启动，则可以在 `首选项->设置` 中添加:

[20] 便捷清新的文本编辑器 sublime[6]

```
"vintage_start_in_command_mode": true,
```

也可以通过快捷键设置将进入命令模式设置成任意键（具体详见 设置快捷键→3.2@p138）.

### 与 Vi 的不同

ST 的 Vintage 插件与 Vi 有一定不同，部分列于此处：

- 在插入模式下用选中不会进入可视模式，这时输入不会被识别为命令而是直接替换文本. 可视模式只有命令模式下才能进入.
- `r`、`R`、`zA`、`za`、`zi`、`z=`、`@` 与使用 `[`、`]` 或 `"` 键的命令不存在.
- 使用 `Ctrl`、`Shift` 和 `Alt` 键的快捷键会保留为 ST 设置的快捷键，如 `Ctrl` + `V` 不会进入可视模式而是正常粘贴.
- 命令行模式只保留了 `:e`、`:0`、`:$`、`:s`.
- 使用 `\` 与 `?` 命令会自动唤出搜索框，而不是直接在命令中键入单词进行搜索. 同时，数字将无法生效.
- `q` 宏录制命令会启动 ST 自带的宏录制，按 `Q` 后需要再按一个键启动录制，但录制的宏不会与该键绑定，需要按 `Ctrl` + `Shift` + `Q` 才能启动. 如果需要保存，需要 `工具->保存宏`，快捷键需要设置.
- 无法使用 `数字+.` 的组合.

## 设置

### 设置 ST

在 `首选项->设置` 中设置，这里列举部分较有用的选项：

```
{
  //字体大小
  "font_size": 11,

  //字体，可以不设置，默认为 Consolas
  "font_face": "",

  //Tab自动转换为空格
  "translate_tabs_to_spaces": true,

  //缩进宽度
  "tab_size": 4,

  //行高亮
  "highlight_line": true,

  //保存时自动在文件尾增加换行
  "trim_trailing_white_space_on_save": true,

  //在选择时查找自动只查找选择范围
  "auto_find_in_selection": true,

  //禁用了OI中不太可能用到的插件，可以自己调整
  "ignored_packages": [
    "ActionScript", "AppleScript", "ASP", "Batch File", "C#",
    "Clojure", "CSS", "D", "Diff", "Erlang", "Git Formats",
    "Go", "Graphviz", "Groovy", "Haskell", "HTML", "Java",
    "LaTeX", "Lisp", "Lua", "Makefile", "Matlab",
    "Objective-C", "OCaml", "Perl", "PHP", "Python",
    "R","Rails", "RestructuredText", "Ruby", "Rust",
```

```
    "Scala", "ShellScript", "SQL", "TCL", "Textile", "XML",
  ],

  //相对行号，可配合 Vintage 插件快速跳转
  "relative_line_numbers": false,
}
```

**设置快捷键**

在 `首选项->快捷键设置` 中设置，在左边找到需要修改的功能，然后复制到右边并修改按键即可.

例如，如果要把 `Ctrl` + `B` 的编译改为 `F9`（如果不令原有的快捷键失效，实际是增加一个触发方式），则可以在 `首选项->快捷键设置` 中添加:

```
//将build命令改为f9
{ "keys": ["f9"], "command": "build" },

//将原有的f9对应的行排序功能的快捷键改为shift+f9，由于大部分时候这个功能用不到，这一行也可以不添加
{ "keys": ["shift+f9"], "command": "sort_lines", "args": {"case_sensitive": false} },
```

**设置插件**

插件的设置可以在 `首选项->Package Setting->插件名` 中找到，做修改时请注意不要修改默认设置.

例如，这里给出 BracketHighlighter 的一些设置，在 `首选项->Package Setting->BracketHighlighter->Bracket Setting` 中修改:

```
{
  //在匹配的括号之间行的行首显示一条线，可以快速找到括号的范围
  "content_highlight_bar": true,

  //在小地图中显示匹配的括号
  "show_in_minimap": true,

  //忽略匹配范围限制
  "ignore_threshold": true,

  //style高亮样式，bold为块高亮，underline为加粗下划线，outline为外围一圈
  //color为颜色，默认设置中已经包含了所有支持的颜色
  //icon为在侧边栏显示的标志
  "bracket_styles": {
    "default": {"icon": "dot", "color": "region.yellowish", "style": "bold",},
    "unmatched": {"icon": "question", "color": "region.redish", "style": "outline",},
    "curly": {"icon": "curly_bracket", "color": "region.purplish",},
    "round": {"icon": "round_bracket", "color": "region.yellowish",},
    "square": {"icon": "square_bracket", "color": "region.bluish",},
    "angle": {"icon": "angle_bracket", "color": "region.orangish",},
    "tag": {"icon": "tag", "color": "region.orangish",},
    "c_define": {"icon": "hash", "color": "region.yellowish",},
    "single_quote": {"icon": "single_quote", "color": "region.greenish",},
    "double_quote": {"icon": "double_quote", "color": "region.greenish",},
    "regex": {"icon": "star", "color": "region.greenish",}
  }
```

```
27 }
```

### 修改与添加

有时候，插件的某些地方可能并不尽如人意，或想对插件进行汉化，这时就需要对插件做一些修改.

插件存放的位置是数据目录下的`\Installed Packages`文件夹.

里面的文件以`.sublime-package`为后缀，实际上为`.zip`格式，可以使用解压缩软件打开.

例如，如果要修改自动补全，可以打开 ST 的 **安装目录** `\Packages\C++` 插件中的`\Snippets\*.sublime-snippet`文件修改，如要 **增添** 自动补全，可以安装 C++ Snippets 并在其中修改或添加文件（或新建一个插件，但不能直接添加进自带的 C++ 插件，否则无法被识别）. 保存任何修改时 **必须** 关闭 ST，且请提前做好备份，否则可能出现文件丢失.

当然，任何增添都可以放在数据目录路径下的`\Packages\User\`下，这总是被支持的.

例如，一个文件模板的补全可以这么写:

```
<snippet>
  <description>template_code</description> <!-- 这里的内容是补全内容的预览 -->
  <content><![CDATA[#include <cstdio>
using namespace std;

int main() {
  freopen("${1:file name}.in", "r", stdin);
  freopen("$1.out", "w", stdout);
  ${0:/* code */}
  fclose(stdin);
  fclose(stdout);
  return 0;
}]]></content>
  <tabTrigger>code</tabTrigger> <!-- 这里的内容是补全的触发文本 -->
  <scope>source.c++</scope> <!-- 这里的内容是补全适用语言 -->
</snippet>
```

以下列出部分文件后缀以及其用途，具体的插件开发教程详见 社区文档[8] 和 官方文档[9]:

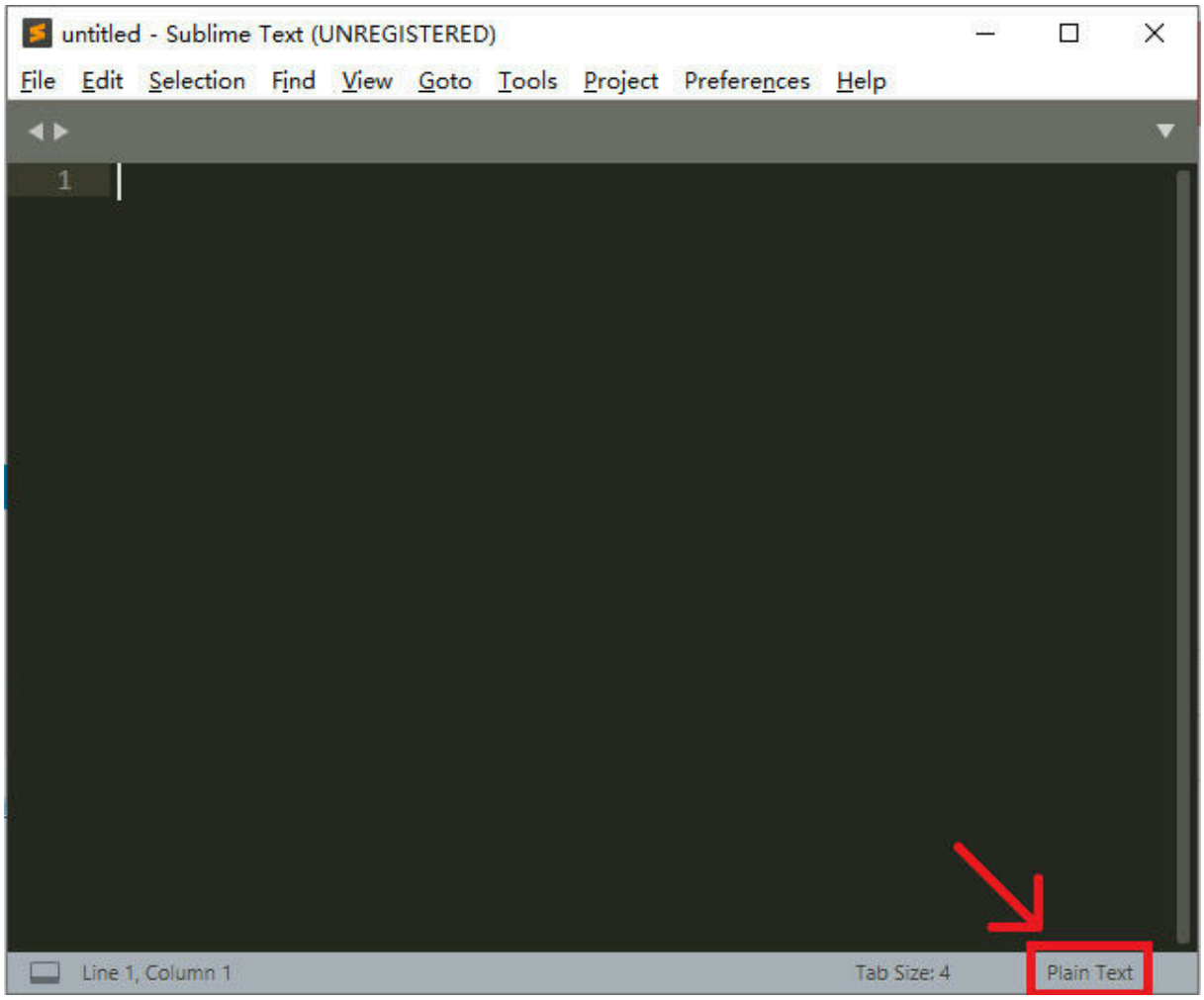
| 后缀名 | 用途 |
|---|---|
| .sublime-build | 编译系统文件 |
| .sublime-completions | 文件名补全列表（一般为头文件） |
| .sublime-syntax | 语法高亮文件 |
| .sublime-settings | 设置文件 |
| .tmPreferences | 首选项中的列表文件 |
| .sublime-keymap | 快捷键设置文件 |
| .sublime-snippet | 代码补全文件 |
| .sublime-commands | 命令定义文件 |
| .sublime-menu | ST UI 文件，包括侧边栏以及顶部菜单栏（汉化的主要对象） |

由于插件更新会直接覆盖原文件，所以建议备份更改的文件.

## 编辑

### 设置语法

按 Ctrl + Shift + P 后输入语法即可，或者按右下角的 Plain Text 然后修改为需要的语言，同时在 视图->语法 中也可以设置.




### 快捷键

ST 有复合快捷键，如 Ctrl + K & Ctrl + Backspace 表示先按 Ctrl + K 再按 Ctrl + Backspace .

部分快捷键:

| 按键 | 命令 |
| --- | --- |
| Ctrl + X | 剪切当前行 |
| Ctrl + Shift + K | 删除行 |
| Ctrl + Enter | 在下方插入行 |
| Ctrl + Shift + Enter | 在上方插入行 |
| Ctrl + Shift + Up | 行上移 |
| Ctrl + Shift + Down | 行下移 |
| Ctrl + L | 选择行，重复以向下选择多行 |

| | |
|---|---|
| Ctrl + D | 选择词，重复以选择多个相同词，并进入多重选择模式（用于快速批量更改） |
| Ctrl + M | 跳转到匹配的括号 |
| Ctrl + Shift + M | 选择括号内的内容（不包括括号），重复以包括括号 |
| Ctrl + K & Ctrl + K | 删至行尾（复合快捷键，建议使用 Vim 模式代替） |
| Ctrl + K & Ctrl + Backspace | 删至行首（复合快捷键，建议使用 Vim 模式代替） |
| Ctrl + ] | 缩进当前（选择的）行 |
| Ctrl + [ | 取消缩进当前（选择的）行 |
| Ctrl + Shift + D | 复制当前行，并插入在下一行 |
| Ctrl + J | 合并下一行与当前行 |
| Ctrl + Shift + V | 粘贴并缩进（用于整段粘贴代码） |
| Ctrl + K & Ctrl + Shift + V | 从历史粘贴（复合快捷键，建议修改为 Ctrl + Alt + V ） |
| Ctrl + Alt + Down | 光标下移，并保留当前行光标（进入多重选择模式） |
| Ctrl + Alt + Up | 光标上移，并保留当前行光标（进入多重选择模式） |
| Ctrl + R | 跳至文件中的任意符号（函数或类型定义） |
| Ctrl + Shift + R | 跳至项目中的任意符号（函数或类型定义） |
| Ctrl + P | 跳至任意文件（曾经打开过或在项目中且存在的文件） |
| ~ | 转换选择内容的大小写 |

### 自动补全

ST 有丰富的补全功能，可能的补全内容会在光标下方显示，按 Tab 或 Enter 进行补全（ST4 中，如进行一个非 Snippet 类型的补全，接下来再按 Tab 可继续选择为以该补全为子串的补全）.

Snippet 类型的补全一般会有一些编辑块，补全后会自动选择为替换文本，如果是 `for` 等含有多个编辑块的复杂补全，编辑完成后再次按 Tab 完成下一个编辑块，此时要在编辑块中触发补全需要使用 Enter （在 ST4 中可继续使用 Tab ）.

如果没有自动补全，请如下修复：

1. 检查是否切换了语言，ST 默认新建文件为 Plain Text.
2. 进入 首选项->设置 然后添上两行：

```
"auto_complete": true,
"auto_match_enabled": true,
```

### 多重选择

按住 Ctrl 并用鼠标单击即可在屏幕上增加光标， Ctrl + Alt + Up 或 Ctrl + Alt + Down 可以在相邻两行直接增加光标，任何编辑性质的操作会同时应用至所有光标.

### 查找与替换

Ctrl + F 为查找， F3 为查找下一个 Shift + F3 为查找上一个， Ctrl + H 为替换.

五个查找选项分别为正则表达式匹配、大小写敏感、全字匹配、循环查找、在选段中查找.

建议在首选项中将 `"auto_find_in_selection"` 设置为 `true`. 这样在选择超过一个词时使用查找会自动只在选段中查找.

## 演示

### 热启动

尝试在 ST 中键入一些内容，并直接把整个 ST 关闭，ST 会直接关闭且没有任何提示，再打开 ST 时只要不对电脑进行数据还原就不会丢失任何数据.

### 多重选择

如果要把以下代码中的所有 `bok` 改为 `book`，只需将光标放置在任意一个 `bok` 中，长按 Ctrl + D 即可快速选中.

```
int check() {
  book[1] = 1, book[2] = 1, book[3] = 1, bok[1] = 1, bok[2] = 1, bok[3] = 1,
  bok[4] = 1, bok[5] = 1;
}
```

如果要将下列文件中的所有等号后面改成 `"good"`，只需在将光标放置于第一行的 `aaa` 前，并按五次 Ctrl + Alt + Down，再然后按下 Ctrl + D 并键入 `good` 即可.

或选中 `"a` 并按住 Ctrl + D，然后按 Right、Ctrl + D，之后键入即可.

```
s[1] = "aaa";
s[2] = "aab";
s[3] = "aac";
s[4] = "good";
s[5] = "aae";
s[6] = "aaf";
```

如要为下列所有 `a + b` 加上括号，只需选择一个 `a + b`，按住 Ctrl + D 并键入 ( 即可（如选择一定区域，则任意左括号键入会为该区域两边添加匹配的括号）.

```
a + b*a + b*a + b
```

### 查找与替换

如果要将下列文件中的所有等号后面改成 `"good"`，也可以用 Ctrl + H 使用替换，打开正则，输入 `".*"`，并替换成 `"good"` 即可.

```
s[1] = "aaa";
s[2] = "aab";
s[3] = "aac";
s[4] = "good";
s[5] = "aae";
s[6] = "aaf";
```

如要为以下代码添加分号，只需使用选择区域替换，打开正则，输入 `\n`，并替换成 `;\n` 即可.

```
int main() {  int a, b  cin >> a >> b  cout << a + b  return 0}
```

### 宏录制

如要为以下代码添加分号，可以按 Ctrl + q 启动宏录制接下来依次按 End、;、Down 再按 Ctrl + q 结束宏录制（中途左下角不会全程显示正在录制，但确实在录制），接下来重复 Ctrl + Shift + q 即可.

```
int main() {
  int a, b
  cin >> a >> b
```

```
    cout << a + b
    return 0
}
```

**如已开启 Vintage 插件**

执行一次 Ctrl + Shift + q 后，可以 Esc 进入命令模式，输入 .. 即可（. 命令可以重复 ST3 命令）

关于宏的保存与绑定按键详见 社区文档[10].

## 编译与运行

ST 的编译环境已经配置好了，可以直接使用.

Windows 环境下需要将 g++ 所在目录添加到环境变量中，并重启 ST.

### 编译

直接按 Ctrl + B 编译，第一次使用会需要选择编译系统，选择 C++ Single File（编译）或 C++ Single File - Run（编译并运行）.

#### 修改编译选项

ST 默认的编译选项为 g++ "${file}" -o "${file_path}/${file_base_name}"，如果要修改编译选项，可以新建一个编译系统.

进入 工具->编译系统->新建编译系统... 然后在大括号中间输入:

```
// 编译选项可以自己调整
// 编译并运行
"shell_cmd": "g++ -Wall \"${file}\" -o \"${file_path}/${file_base_name}.exe\" && \"${file_path}/${file_base_name}.exe\"",

// 这一行可以让ST3图形化显示报错，如果习惯了看g++返回的信息可以去掉
"file_regex": "^(..[^:]*):([0-9]+):?([0-9]+)?:? (.*)$",
```

保存后按 Ctrl + Shift + B 切换编译系统就可以使用了，这里的配置是编译并在外部 CMD 运行.

保存的文件为数据目录路径下的 \Packages\User\编译系统名.sublime-build 可以反复修改.

### 运行

如果编译时选择 C++ Single File - Run（即编译后运行）或配置了自动运行，那么在下方弹出的编译信息窗口应该不会有任何显示（除非编译错误），因为 ST 的编译信息窗口实际上是一个终端，可以直接输入数据.

运行结束后会提示程序的运行时间，其计时为从按下 Ctrl + B 到全部 CMD 命令运行结束的时间，也就是说包括编译的时间和输入的时间，以及如果在外部 CMD 运行还包括 CMD 开启关闭的时间.

**Warning**

这个窗口无法输入 F6 或 Ctrl + Z，如果运行读入到文件末尾的程序请使用文件输入，或配置在外部 CMD 运行.

### 调试

可以安装插件使 ST 支持图形化 gdb 调试，但不建议依赖插件进行 gdb 调试.

更好的做法是在配置编译系统时加上相关命令启动 gdb，在外部进行命令行调试.

### 杂项

- 把文件夹拖进 ST 中并按 Ctrl + K & Ctrl + B 开启侧边栏，从而快速切换文件.
- 善用跳转功能，尤其是 Ctrl + P 进行文件间跳转与 Ctrl + R 进行函数跳转.
- ST 支持 git[21].
- ST 的所有配置储存在数据目录下，可以随意打包，但注册信息无法在多台电脑上使用.

### 外部链接

- 使用命令行调试→3.4@p232
- Sublime Text 3 官方文档[12]
- Sublime Text 社区文档[13]

### 参考资料与注释

[1] NOI Linux 2.0 发布 

[2] Sublime Text 4 发布 

[3] Sublime Text 4 的下载页面 

[4] Sublime Text 3 的下载页面 

[5] Package Control 

[6] 便捷清新的文本编辑器 sublime 

[7] tmTheme Editor 

[8] 社区文档 

[9] 官方文档 

[10] 社区文档 

[11] Sublime Text Git 集成 

[12] Sublime Text 3 官方文档 

[13] Sublime Text 社区文档 

## 3.2.14 CP Editor

**Authors:** zarttic, xk2013

### 简介

CP Editor[1] 专为算法竞赛设计，不像其它 IDE 主要是为了开发设计的. 它可以帮助你自动化编译、运行、测试，从而让你专注于算法设计. 它甚至可以从各种算法竞赛网站上获取样例，将代码提交到 Codeforces[2] 上!

### 下载与安装

参见 安装 | CP Editor[3].

### 基础配置

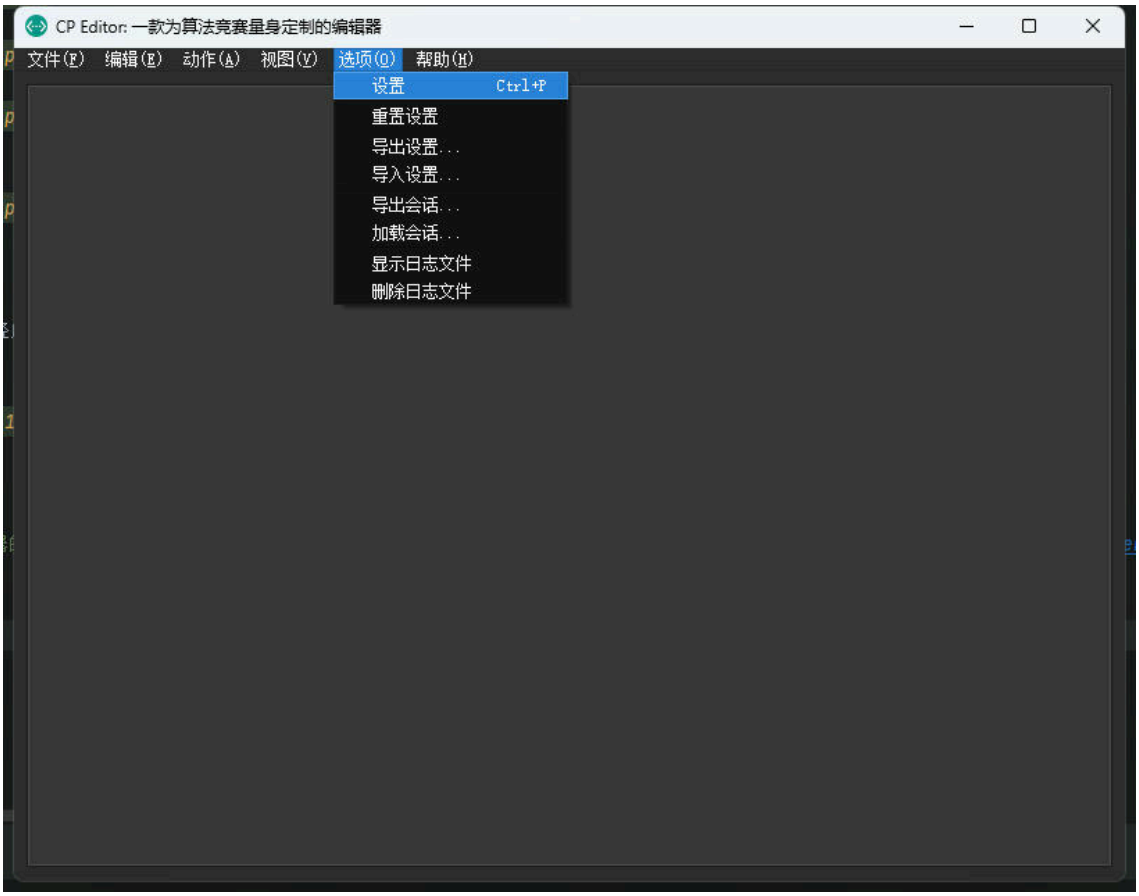
> CP Editor 内部没有集成编译器, 需要自己安装配置编译器, 如有需要请参考本站关于编译器安装相关的文章[22]，当然，如果你在下载时选择了带有 `with-gcc-<GCC 版本号>-llvm-<LLVM 版本号>` 后缀的安装包，你就可以使用 CP Editor 自带的编译器，路径在 `{安装目录}/mingw64/bin/`.

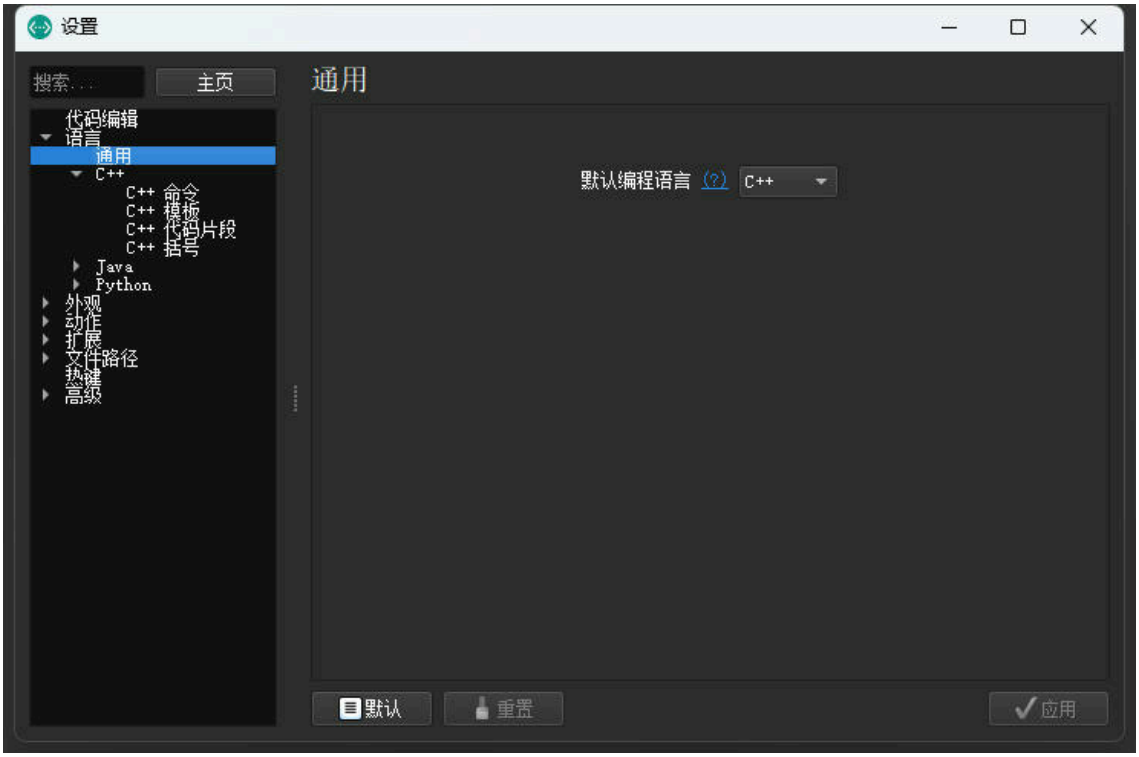
- 设置默认语言

  编辑器默认的语言为 `C++`.

[21] Sublime Text Git 集成[11]

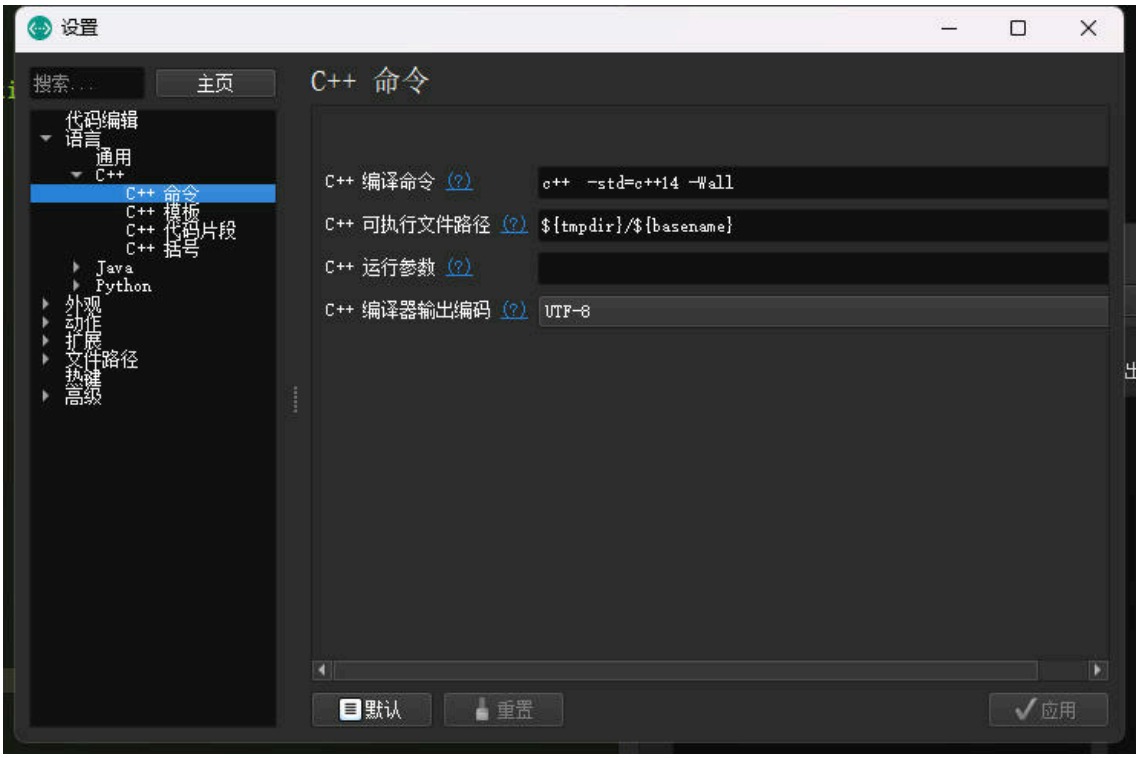
[22] 编译器 - OI Wiki→3.6@p241



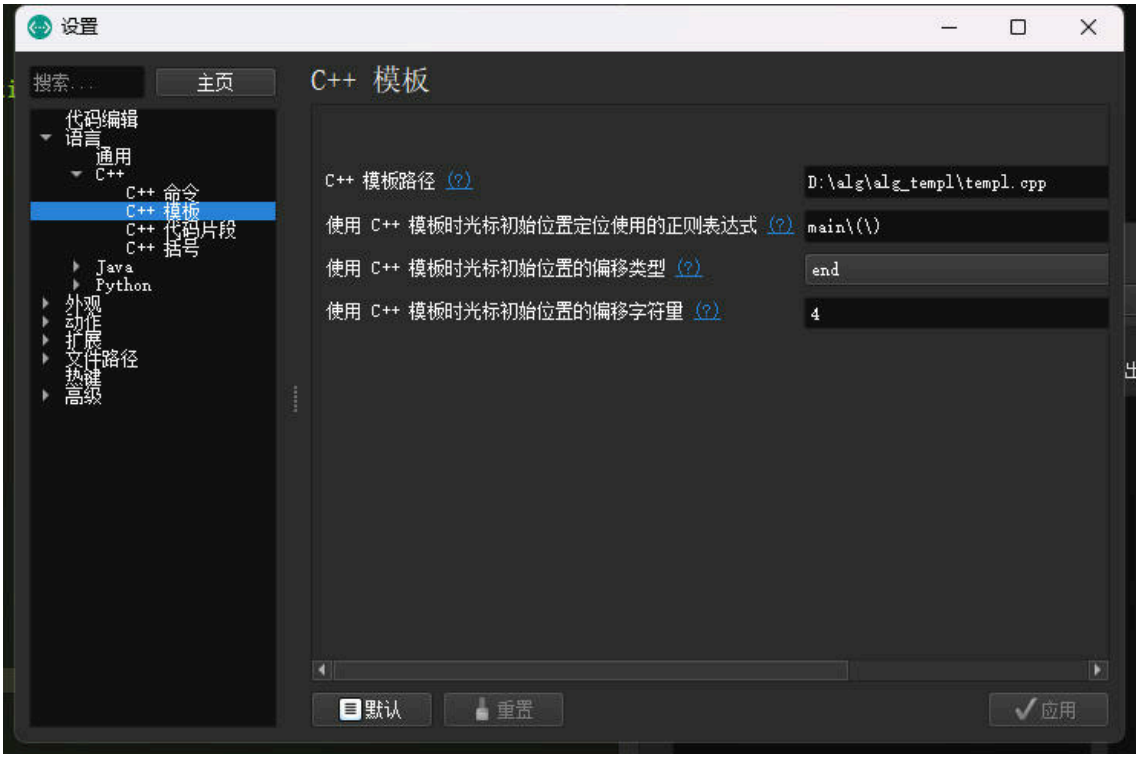

- 设置 `C++` 命令

  需要设置一些必要的编译命令，这个要根据编译器来设定.




- 设置模板

  新建文件的时候会自动初始化的模板，需要注意的是 CP Editor 需要的是一个 `xxx.cpp` 的文件作为模板文件.




完成了以上的基本操作你就可以使用最基本的功能了.

## 基本功能

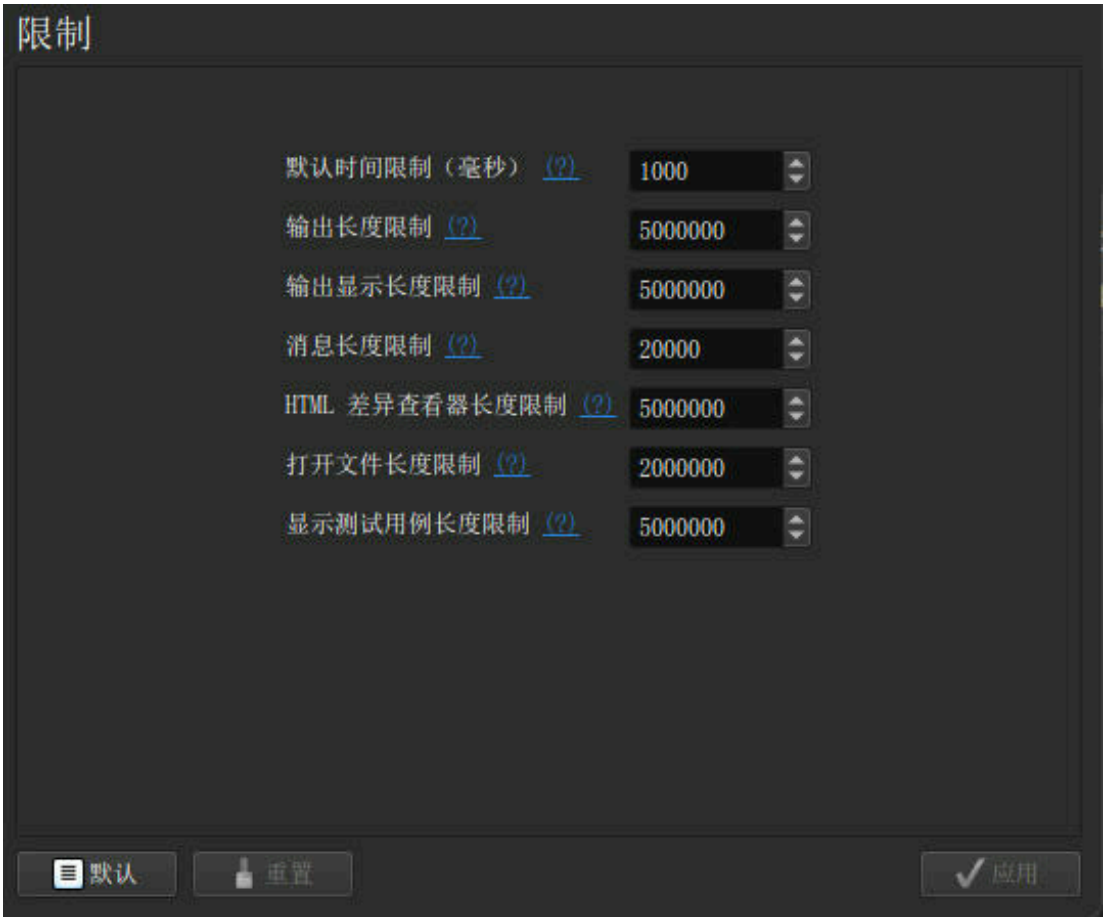
- 快捷键

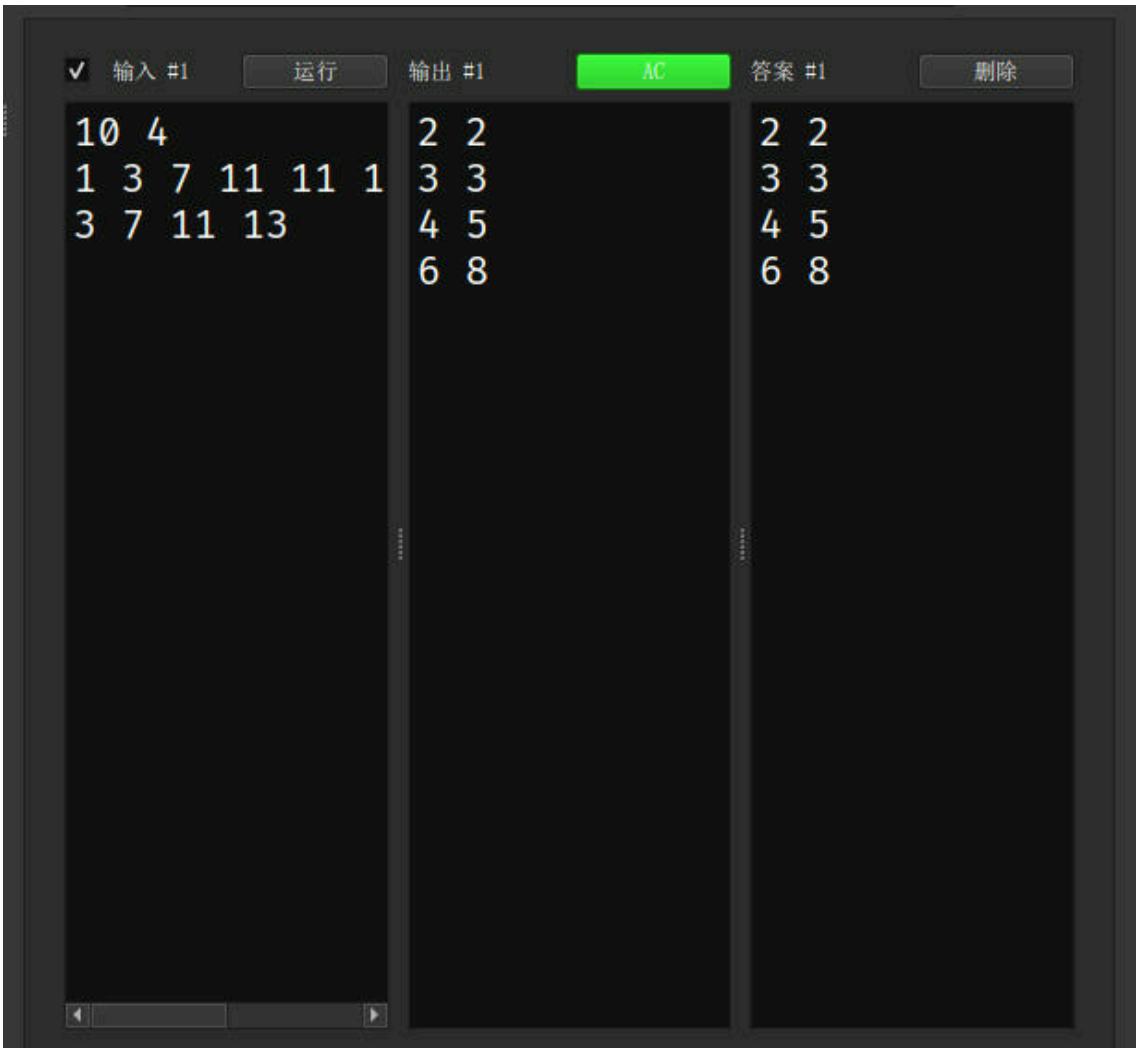
| 命令 | 操作 |
| --- | --- |
| Ctrl + Shift + C | 编译. |
| Ctrl + Shift + R | 编译并运行. |
| Ctrl + R | 运行. |
| Ctrl + Alt + D | 在终端中运行. |
| Ctrl + K | 终止所有进程. |
| Ctrl + Shift + I | 格式化代码. |

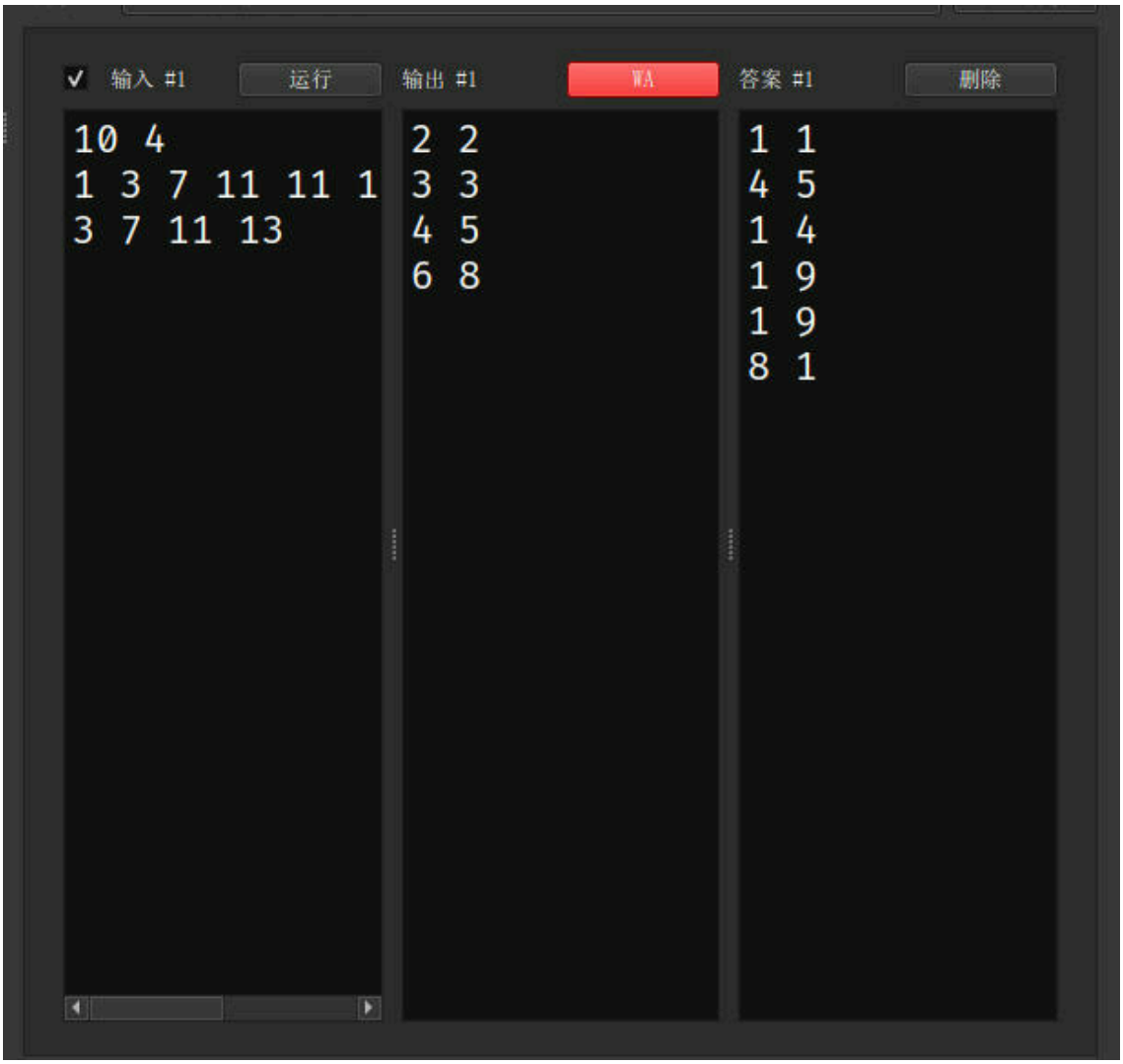
具体可以查阅 官方文档[4].

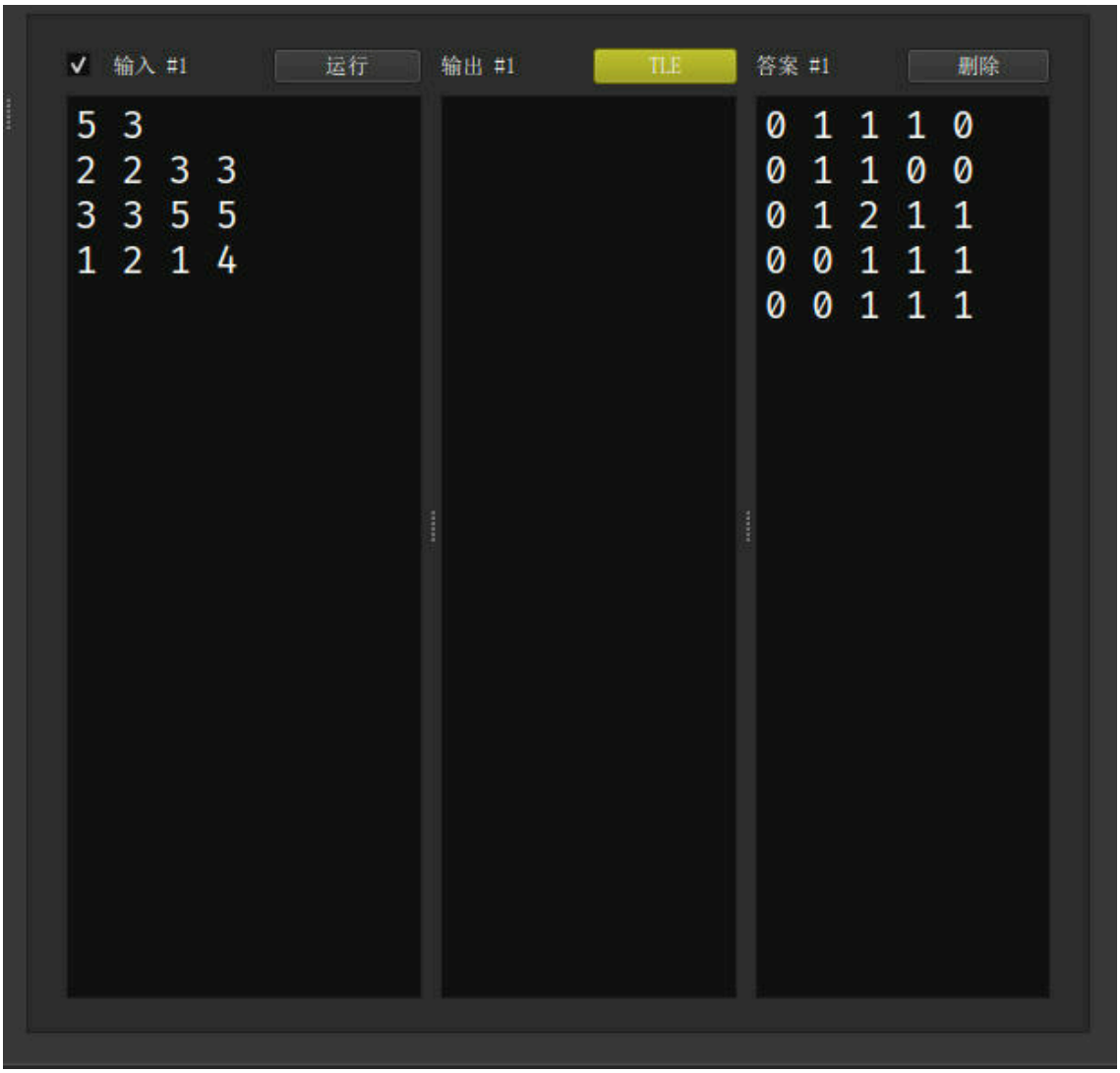
- 样例测试

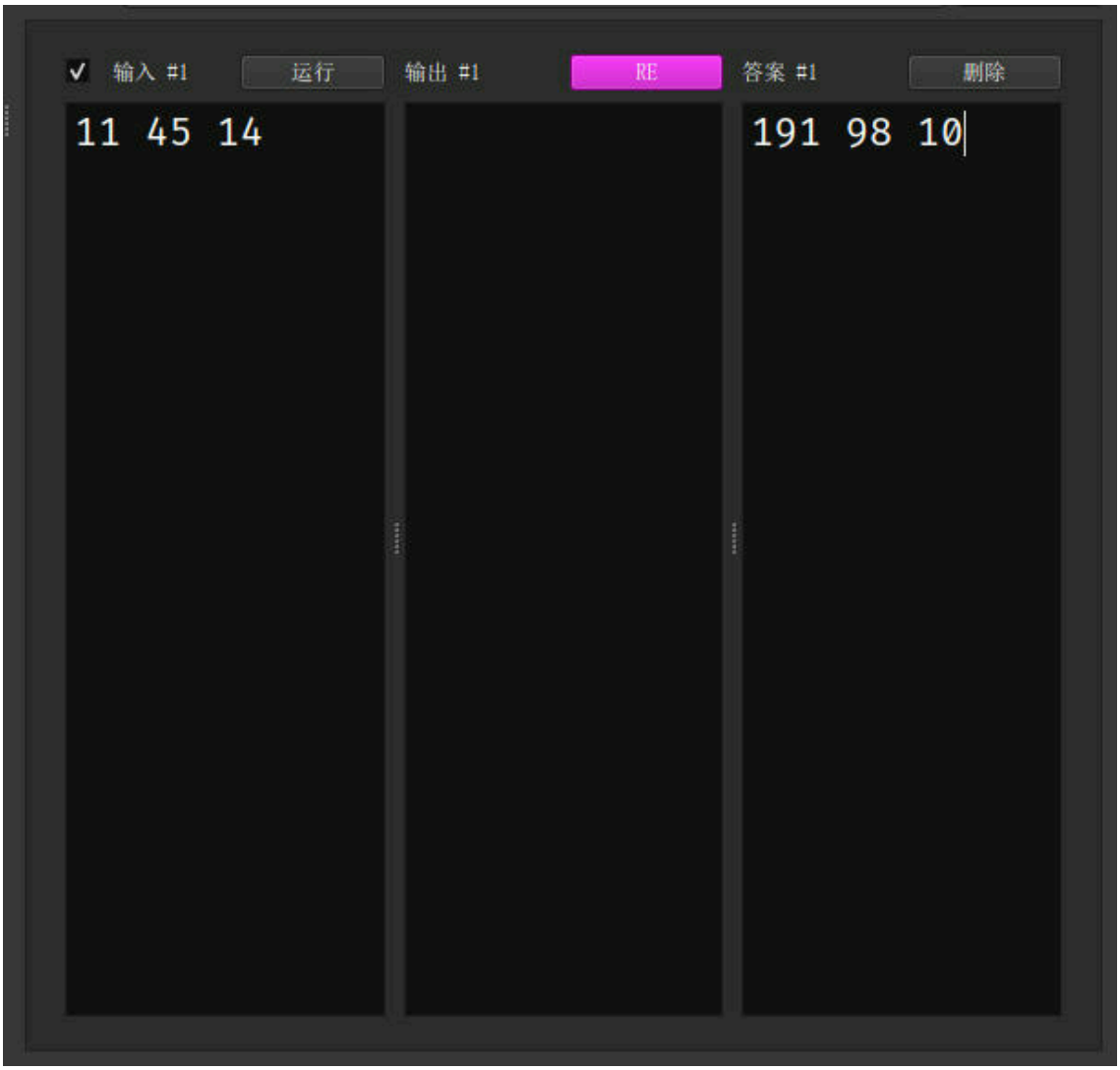
可以把题面的样例复制下来，由 CP Editor 自动评测，而且还可以设置时间限制！




输入 #1
运行
输出 #1
AC
答案 #1
删除
10 4
1 3 7 11 11 1
3 7 11 13
2 2
3 3
4 5
6 8
2 2
3 3
4 5
6 8


输入 #1
运行
输出 #1
WA
答案 #1
删除
10 4
1 3 7 11 11 1
3 7 11 13
2 2
3 3
4 5
6 8
1 1
4 5
1 4
1 9
1 9
8 1


输入 #1
运行
输出 #1
TLE
答案 #1
删除
5 3
2 2 3 3
3 3 5 5
1 2 1 4
0 1 1 1 0
0 1 1 0 0
0 1 2 1 1
0 0 1 1 1
0 0 1 1 1




## 参考资料

[1] CP Editor

[2] Codeforces

[3] 安装 | CP Editor

[4] 官方文档

### 3.2.15 Code::Blocks

#### 简介

Code::Blocks 是一个使用 C++ 开发的开源集成开发环境（IDE），采用 wxWidgets 作为图形界面库．该项目始于 2001 年，目前由官方社区维护，主要用于 C、C++ 和 Fortran 等编程语言的开发．[23]

优点：

- **轻量和高效**：Code::Blocks 资源占用少且启动迅速，适合资源有限的环境以及偏好轻量级 IDE 的开发者，也适合入门级开发者学习和使用．
- **跨平台兼容性**：支持 Windows、Linux 和 macOS 等多种操作系统，提供了一致的用户体验，使得开发者可以无缝地在不同平台上工作．

[23] Code::Blocks - 维基百科[1]

- **广泛的编译器支持**：支持包括 GCC、MSVC (Microsoft Visual C++)、Digital Mars 和 Borland C++ 5.5 在内的多种编译器，允许开发者根据项目需求选择最合适的编译工具.

缺点：

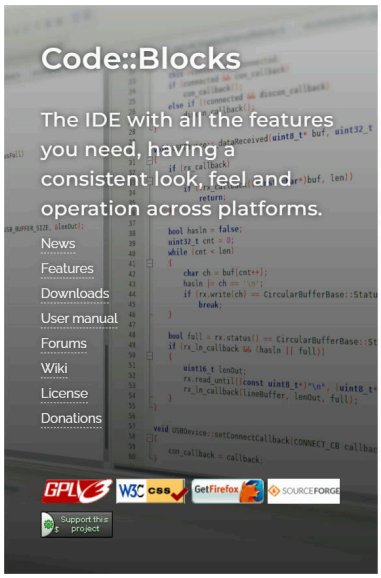
- **功能范围有限**：相比于 CLion 或 Eclipse 等 IDE，内置功能和工具较为基础，可能不足以满足复杂项目的需求.
- **插件生态较弱**：尽管支持插件扩展功能，但第三方插件数量和质量有限，插件生态相对较弱.

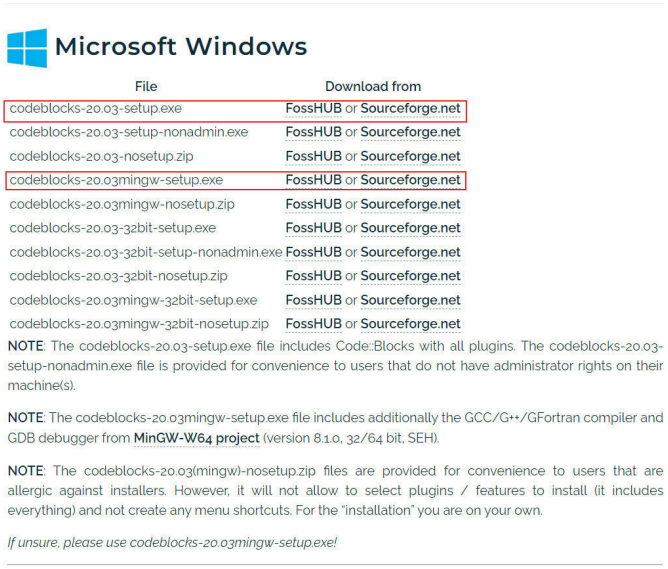
## 安装

参见 Code::Blocks 官方网站[2]，选择下载二进制安装程序（Binary Release），或者下载源代码编译安装（Source Code），然后根据需求和操作系统选择合适的安装程序，按照安装向导完成安装即可.

**下载包含 MinGW 的安装包**

对于 Windows 用户，如果不希望手动配置编译器，建议下载包含 MinGW 的安装程序，例如 `codeblocks-xxxxmingw-setup.exe`，该版本已经包括了 GCC 编译器，无需额外安装和配置即可开始开发 C 和 C++ 项目.




## 配置

如果安装时选择了不包含 MinGW 的安装程序，或者需要使用其他编译器，则需要手动安装和配置编译器，然后设置 Code::Blocks 以使用该编译器.

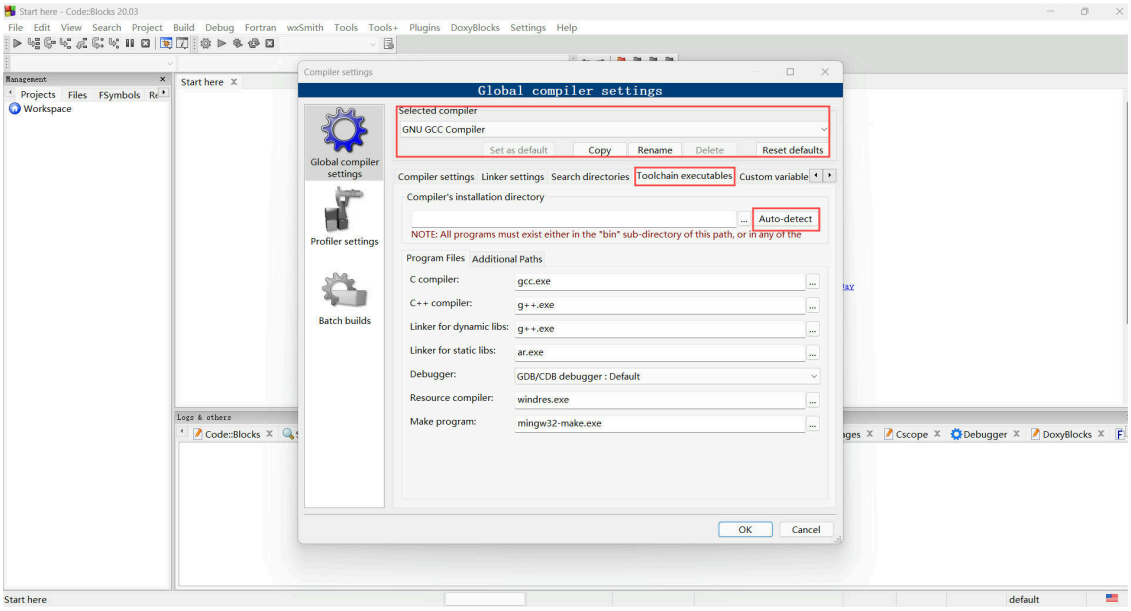
### 工具链安装

参考本站的 编译器→3.6@p241 安装指南，下载并安装你需要的编译器.

### 工具链设置

当第一次启动 Code::Blocks 时，软件会自动扫描系统中已安装的编译器，如果没有找到编译器，可以通过以下步骤手动添加：

1. 打开 Code::Blocks，点击菜单栏的 `Settings -> Compiler`，打开编译器设置对话框（如下图所示）.




2. 在 `Selected compiler` 下拉框中选择需要配置的编译器，例如 `GNU GCC Compiler`.
3. 在 `Toolchain executables` 选项卡中，单击 `Auto-detect` 按钮，Code::Blocks 将自动扫描系统中已安装的编译器.
4. 如果自动扫描失败，你可以手动设置编译器路径. 在 `Compiler's installation directory` 中输入编译器的安装路径，例如 `C:\MinGW\bin`.
5. 设置完成后，点击 `OK` 保存设置，现在你可以使用该编译器来编译和运行项目.

## 使用

Code::Blocks 内置项目管理器，支持用户自定义构建项目，你可以在 `Project -> Build options` 中设置编译选项，选择编译器、编译选项、链接选项等，也可以在 `Project -> Properties` 中设置项目属性，例如项目名称、路径、文件列表等.

> **配置 Makefile**
>
> Code::Blocks 默认不需要编写 Makefile，如果需要使用自定义的 Makefile，可以在 `Project -> Properties` 中勾选 `This is a custom Makefile` 选项，然后在 `Project -> Build options` 中设置 Makefile 的路径.

## 创建项目

Code::Blocks 支持的编程语言包括 C、C++ 和 Fortran 等，当启动 Code::Blocks 后，可以通过 `File -> New -> Project` 创建新项目，选择项目类型和模板，然后按照向导的指示，设置项目名称、路径、编译器等，最后点击 `Finish` 完成项目创建.

Code::Blocks 也支持单文件的编译和运行，可以通过 `File -> New -> File` 创建新文件，编写代码并保存后，点击工具栏上的 `Build and run` 按钮，或者按下 `F9` 键，自动编译和运行当前文件.

## 构建和运行

以一个简单的 Console Application 项目为例，接下来介绍如何构建和运行项目：

1. 项目创建完成后，你会看到一个默认的 `main.cpp` 文件，你可以在该文件中编写代码，然后保存文件.
2. 编写完代码后，点击工具栏上的 `Build and run` 按钮，或者按下 `F9` 键，Code::Blocks 将自动编译和运行项目.

3. 编译和运行后，输出窗口中会显示程序的输出结果，你可以在输出窗口中查看程序的输出，根据需要调整代码.
4. 如果只需要编译项目，而不运行，可以点击工具栏上的 `Build` 按钮，或者按下 `Ctrl + F9` 键，Code::Blocks 将只编译项目，不运行程序.

### 调试

Code::Blocks 内置了调试器，你可以在 `Debug` 菜单中设置和启动调试器，帮助你定位和解决程序中的错误.

同理，以一个简单的 Console Application 项目为例，接下来介绍如何调试项目：

1. **设置断点**：在需要调试的代码行左侧单击鼠标左键，设置断点，程序将在断点处停止执行.
2. **启动调试器**：点击工具栏上的 `Debug` 按钮，或者按下 `F8` 键，Code::Blocks 将自动编译并启动调试器.
3. **调试程序**：在调试器中，你可以单步执行程序，查看变量值、调用栈等，帮助你定位和解决程序中的错误.
4. **停止调试**：调试完成后，你可以点击工具栏上的 `Stop` 按钮，或者按下 `Shift + F8`，停止调试器.

## 自定义设置

Code::Blocks 提供了丰富的设置选项，可以帮助调整编辑器的行为，以下是一些常用的设置：

### 界面设置

1. 在 `Settings -> Editor` 中，可以设置编辑器的字体、颜色、缩进、自动补全等选项.
2. 在 `Settings -> Environment` 中，可以设置 Code::Blocks 的全局行为，例如自动保存、自动备份、自动提示等.
3. 在 `View` 菜单中，可以调整编辑器的布局，例如打开/关闭文件浏览器、工具栏、状态栏、输出窗口等.

### 插件设置

Code::Blocks 支持插件来扩展功能，可以通过 `Settings -> Plugins` 查看和安装可用的插件，例如 DoxyBlocks、wxSmith 等，以下是一些常用的插件：

- **DoxyBlocks**：著名的文档生成工具 Doxygen 的集成插件，可以直接在 Code::Blocks 中生成项目文档.
- **wxSmith**：用于开发 wxWidgets 应用程序的插件，提供了可视化的界面设计工具，允许快速创建和布局 GUI 界面，简化开发流程.
- **Thread Search**：支持多线程搜索的插件，可以在项目中快速搜索和替换符号和文本，适用于大型项目的开发.

插件的安装和使用方法请参考 Code::Blocks 的插件文档[3]，根据插件的需求和功能，选择合适的插件安装和使用.

> ⚠ **注意**
>
> Code::Blocks 的插件相对单一和基础，且大部分插件已经集成到软件中，第三方插件的数量和质量有限，建议根据实际需求选择合适的插件.

### 快捷键设置

你可以通过 `Settings -> Editor -> Keyboard shortcuts` 选项卡查看和修改快捷键设置，根据自己的习惯调整快捷键.

以下是一些常用的快捷键：

| 功能 | 快捷键 |
| --- | --- |
| 新建文件 | `Ctrl + Shift + N` |

| | |
|---|---|
| 打开文件 | `Ctrl + O` |
| 保存当前文件 | `Ctrl + S` |
| 全部保存 | `Ctrl + Shift + S` |
| 关闭当前文件 | `Ctrl + W` |
| 关闭所有文件 | `Ctrl + Shift + W` |
| 构建和运行当前项目 | `F9` |
| 只构建当前项目 | `Ctrl + F9` |
| 只编译当前项目 | `Ctrl + Shift + F9` |
| 运行当前项目 | `Ctrl + F10` |
| 调试：开始/继续 | `F8` |
| 调试：停止 | `Shift + F8` |
| 调试：下一步 | `F7` |
| 调试：进入 | `Shift + F7` |
| 调试：跳出 | `Ctrl + F7` |
| 调试：切换断点 | `F5` |
| 查找 | `Ctrl + F` |
| 查找并替换 | `Ctrl + R` |
| 转到指定行 | `Ctrl + G` |
| 转到匹配的括号 | `Ctrl + B` |
| 全屏切换 | `F11` |
| 开关所有折叠 | `Ctrl + Shift + -` |
| 展开所有折叠 | `Ctrl + Shift + +` |
| 选择下一个匹配项 | `Ctrl + E` |
| 选择跳到下一个匹配项 | `Ctrl + Shift + E` |

### 参考资料与注释

[1] Code::Blocks - 维基百科  [2] Code::Blocks 官方网站  [3] Code::Blocks 的插件文档 

# 3.3 评测工具

## 3.3.1 评测工具简介

> **Authors:** Ir1d, HeRaNO, NachtgeistW, i-Yirannn, bear-good, ranwen, CoelacanthusHex, billchenchina, Tiger3018, Xeonacid, renbaoshuo

**评测软件** 是用于本地测试分数的软件．使用者在将代码提交到 OJ 前，可以使用评测软件对自己的程序估分．

本部分介绍了以下几种评测软件：

- Arbiter→3.3.2@p224
- CCR Plus→3.3.4@p230
- Cena→3.3.3@p230
- Lemon→3.3.5@p231

## 3.3.2 Arbiter

> **Authors:** Ir1d, HeRaNO, NachtgeistW, i-Yirannn, bear-good, ranwen, CoelacanthusHex, billchenchina, Tiger3018, Xeonacid, Cryflmind

### Arbiter

**Arbiter** 为北京航空航天大学为 NOI Linux 开发的评测工具，现已用于各大 NOI 系列程序设计竞赛的评测. 据吕凯风在 2016 年冬令营上的讲稿《下一代测评系统》，Arbiter 是由北京航空航天大学的团队（GAIT）在尹宝林老师的带领下开发完成的.

在 NOI Linux 更新到 2.0 版本后，Arbiter 也用 Qt 5.12.8 重新编译，并发布为 Arbiter 2.0. 因为之后的测评环境均使用 NOI Linux 2.0，因此以下介绍使用的 Arbiter 版本均为 NOI Linux 2.0 中自带的 Arbiter 2.0.

此测评软件仅能在 NOI Linux 下找到. 二进制文件位置为 `/usr/local/arbiter/local/arbiter_local`.

### 使用方法

#### 配置程序

配置选手源程序文件夹和选手名单. 选手文件夹如 NOIP 格式创建:

```
players/
| -- day1
|    | -- <contestant_1's ID>
|    |     | -- <problem_1>
|    |     |   `-- <problem_1>.c/cpp/pas
|    |     | -- <problem_2>
|    |     |   `-- <problem_2>.c/cpp/pas
|    |     | ...
|    |     | -- <problem_x>
|    |     |    `-- <problem_x>.c/cpp/pas
|    | -- <contestant_2's ID>
|    |     | -- <problem_1>
|    |     ...
|    ...
| -- day2
|    | -- <contestant_1's ID>
|    |     | -- <problem_1>
|    |     |   `-- <problem_1>.c/cpp/pas
|    |     | -- <problem_2>
|    |     |   `-- <problem_2>.c/cpp/pas
|    |     | ...
|    |     | -- <problem_x>
|    |     |    `-- <problem_x>.c/cpp/pas
|    | -- <contestant_2's ID>
|    |     | -- <problem_1>
|    |     ...
|    ...
...
```

其中，`day<x>` 中的 `<x>` 是场次编号，`<contestant_x's ID>` 指的是选手编号，形如 `<省份>-<编号>`，例如 HL-001，JL-125 等等；`<problem_x>` 指的是题目名称. 在自测时可以使用字母、短线（即 `-`）和数字的组合作为选手编号.

选手名单格式如下:

```
<contestant_1's ID>,<contestant_1's name>
<contestant_2's ID>,<contestant_2's name>
...
```

其中，`<contestant_x's name>` 表示选手姓名．保存这个文件为纯文本文件或 csv 文件，可以使用 `UTF-8` 编码．

选手名单也可以在启动 Arbiter 后手动添加．

接下来配置测试数据．每组数据的命名格式如下：

```
<problem_x><y>.in <problem_x><y>.ans
```

其中，`<y>` 是数据编号，编号从 1 开始．默认测试数据后缀名是 `.ans`，选手输出的后缀名是 `.out`，不能混淆．

如果需要将之前生成的 out 格式修改为 ans 格式，在 NOI Linux 2.0 中可以使用 `rename` 命令批量修改，而在 Windows 中可以使用 `ren` 命令批量修改．我们将在后面介绍这些命令的用法．

不用将每题的测试数据放置在各题的文件夹里，只需要放在一起即可．

然后开始测评文件夹的配置．

左下角「显示应用程序」-「全部」-「Arbiter_local」，启动 Arbiter．

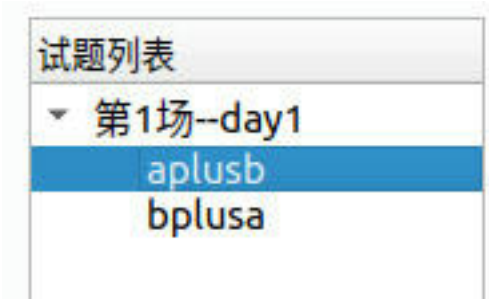



点击 OPEN 可以打开已经建立的比赛，之后需选择对应比赛文件夹下的 `setup.cfg` 文件；点击 NEW 可以新建一个竞赛，并设置名称和比赛目录．注意，需要在用户 **主目录下** 新建一个文件夹，然后选择其为比赛目录，如果在桌面上建立比赛目录的话无法测评．出现这种问题很有可能是因为比赛文件夹路径中不能包含中文．




在左边试题概要里「右键」-「添加考试」，再在考试标签上「右键」-「添加试题」，新建出试题即可.

单击考试左边的向下箭头即可全部显示，单击试题标签对试题名称进行修改，改为题目的英文名称，同时修改题目时间与空间限制和比较方式. 比较方式十分不推荐用「全文完全直接比较」，对于 Windows 下制作的数据十分不友好. 可以根据题目自主选择比较器，但是需要注意必须选择一个比较器，否则测评结果将是 `No Score.`.




点击「文件」-「保存」. 该操作不可省略，否则程序将不会生成题目配置文件. 注意每一次对题目配置的修改都要保存.

此时，打开考试文件夹，会发现有如下内容.

```
<name>/
| -- data
| -- evaldata
```

```
| -- filter
| -- final
| -- players
| -- result
| -- tmp
`-- day1.info
`-- player.info
`-- setup.cfg
`-- task1_1.info
`-- task1_2.info
`-- team.info
```

`filter` 文件夹放置了一些比较器；`result` 文件夹存放选手的测评结果；`tmp` 文件夹是测评时的缓存文件夹. 其中 `day<x>.info` 为场次配置文件，`<x>` 为场次编号；`task<x>_<y>.info` 文件为题目配置文件，`<x>` 为场次编号，`<y>` 为题目序号.

把已经建好的选手程序文件夹放在 `players/` 目录下，注意最外层应按照考试日建立相应的 `day<x>` 文件夹. 将所有测试数据（不放在文件夹里）放在 `evaldata` 中. 如果使用了自定义校验器，则需要将自定义校验器放在 `filter` 中.

**正式测评**

点开「试题评测」标签，会出现如下页面：




如果选手名单已经建立了，直接选择右边的「导入名单」进行导入. 如果人数较少，可以选择右边的「添加选手」进行导入.

导入后的页面如图.




示例中的编号是 `HL-001`，程序会自动识别出「所属」一栏．如果不是 NOIP 规范的编号是识别不出来的.

把测评第 0 场变为测评第 1 场（或者其他场次）．然后选择右边的全选（或选择指定的选手），再选择下面的评测选定选手，选择要测评的题目（或全部试题），最后等待测评结束即可.

测试点详细信息需要在 `result` 文件夹下查看，文件夹下会有选手的结果文件夹，结果文件的后缀名为 `.result`，用纯文本方式查看即可．如果出现 `No score file.` 的错误，可以检查测评时是否生成了 `/tmp/_eval.score` 文件.

### 自定义校验器的编写

反编译其他校验器，可以知道运行自定义校验器的命令是 `<problem>_e <in> <out> <ans>`．后三个参数分别代表输入，选手输出和答案文件．最终的评分结果需写入 `/tmp/_eval.score` 文件中，第一行是测评信息，第二行是分数，10 分为满分.

编译后自定义校验器的名称必须为 `<problem>_e`，其中 `<problem>` 为题目名称．在配置题目时选择自定义校验器，然后选择需要的自定义校验器即可.

在试题管理中题目配置的地方将提交方式由源代码改为答案文件，然后选择自定义校验器，可以测试提交答案题.

### 注意事项

已确认需要注意的内容:

- 需要注意及时保存比赛，否则操作时可能闪退．为了确保不会闪退可以尝试多次保存比赛，或进行一次修改时就保存比赛．
- 没有进行过评测时不要点击上面的成绩统计，否则将会导致 Arbiter 直接闪退．
- 由于 Linux 运行时栈限制，如果要开无限栈，应在终端先输入 `ulimit -s unlimited` 后执行 `arbiter_local` 打开测评器，否则可能出现 `Exceeding memory limit` 的问题．
- 对于正式测评，在题目准备时需要让所有题目空间限制一致．测评时将命令中的 `unlimited` 换为题目空间限制的千字节数（KiB），如：题目空间限制为 512 MiB，则命令为 `ulimit -s $((512 * 1024))`．导致这一问题的主要原因是直接启动 Arbiter，其父进程为 GNOME，子进程继承了父进程的栈空间限制．
- 软件的工作目录不建议包含空格，若包含空格的话很可能会导致创建比赛时所有的默认校验器都无法拷贝进 filter 目录中（即 filter 目录为空）．此时进行评测会出现全部爆 0 的情况，同时生成的 result 文件中可以看到 `Compile Failed.` 的提示．
- 查看代码时提示「未找到答案文件」指的是没有找到选手的源代码．

存疑的内容：

- 很容易死机，如大量测评时移动鼠标会导致死机．
- 不定时闪退（一部分原因是没有及时保存比赛）．
- 修改比较方式后有概率会出现修改失败的情况，即比较方式修改后未被应用．
- 配置时需要注意权限问题，但确保使用同一用户建立比赛，拷贝数据和进行测评的情况下不会出现权限问题．

### 漏洞

由于长期缺乏维护，系统存在一些漏洞，如可以使用 `#pragma G++ optimize("O2")` 和 `__attribute__((__optimize__("-O2")))` 等．可以使用 gcc-plugins-for-oi[1] 在编译期实现对这些命令的检测．

### 评价

Arbiter 1.0.2 在开发完成后就一直没有实质性更新，导致测评体验极差，UI 脱离现代审美．在 NOI Linux 1.4.1 中，它和 NOI Linux 自带的 GUIDE 一样沦为选手与教练疯狂吐槽的对象．在 NOI Linux 2.0 中，除了比较器移除了源代码和软件整体使用 Qt 5 重新编译外，并没有很大的变化，一些稳定性问题仍未得到解决．

**附：ren 和 rename 命令的使用方法**

在 Windows 操作系统中自带了一个修改文件名称的命令：`ren`．

命令语法如下：

```
ren [<drive>:][<path>]<filename1> <filename2>
```

如果我们需要对当前工作目录下的所有的文件进行修改，比如将所有的 out 文件修改为 ans 文件，可以执行如下命令：

```
ren *.out *.ans
```

如果是在 NOI Linux 2.0 环境中进行此类修改，似乎目前比较好用的是 `rename` 命令，但它不是 NOI Linux 2.0 环境内自带的命令，所以你要先进行安装：

```
sudo apt install rename
```

注：如果执行后提示 `E: Unable to locate package package_name`，你需要先执行这个命令：`sudo apt-get update`

安装完成后，就可以正常使用 `rename` 命令了．`rename` 命令的使用类似于直接的文本替换，其在 NOI Linux 2.0 环境下的命令语法如下：

```
rename 's/<修改前的文本>/<修改后的文本>/' <filename>
```

其中 `<filename>` 可以使用通配符 `*`，也可以指定其中一类文件（比如 `*.out`）．

请注意在引号内末尾还有一个 `/`，如果少写了一个 `/`，`rename` 命令将会报错：`Substitution replacement not terminated at (user-supplied code)`．

此时如果我们需要对当前工作目录下的所有的文件进行修改，比如将所有的 out 文件修改为 ans 文件，可以这么写：

```
rename 's/\.out/\.ans/' *
```

其中 `\.` 表示对 `.` 进行转义．

（温馨提示：如果少写了 `\.`，假如你的文件里有个 `outtest.out`，这条命令执行过后文件将会被修改成 `anstest.out`）

类似的，如果你需要对所有名为 `atmost<x>.ans` 的文件进行统一修改（其中 `<x>` 代表测试点编号），将它们都修改为 `test<x>.ans`，不妨这么写：

```
rename 's/atmost/test/' *.ans
```

## 外部参考资料

[1] gcc-plugins-for-oi



### 3.3.3 Cena

> **Authors:** Ir1d, HeRaNO, NachtgeistW, i-Yirannn, bear-good, ranwen, CoelacanthusHex, billchenchina, Tiger3018, Xeonacid

#### Cena

**Cena** 是由刘其帅和李子星使用 Pascal 语言编写的开源评测工具，是流传最广泛的本地评测工具．Cena 最初开源于 Google Code 平台，由于不明原因 Google 删除了 Cena 项目．目前可以在 Web Archive[1] 上找到 Cena 的官网．

Cena 对权限的限制不是很明确，测试的时候可以读测点 AC．

Cena 的源代码托管于 oi-archive/cena[2]．

## 外部参考资料

[1] Web Archive



[2] oi-archive/cena



### 3.3.4 CCR Plus

> **Authors:** Ir1d, HeRaNO, NachtgeistW, i-Yirannn, bear-good, ranwen, CoelacanthusHex, billchenchina, Tiger3018, Xeonacid

### CCR Plus

CCR Plus 是一款适用于 NOI 系列比赛的开源的跨平台测评环境，使用 Qt 编写，目前支持 Windows 和 Linux.

源代码托管于 sxyzccr/CCR-Plus[1].

## 外部参考资料

[1] sxyzccr/CCR-Plus



## 3.3.5 Lemon

> **Authors:** Ir1d, HeRaNO, NachtgeistW, i-Yirannn, bear-good, ranwen, CoelacanthusHex, billchenchina, Tiger3018, Xeonacid

### Lemon

> ⚠ **Warning**
>
> macOS 下 Lemon 可能会出现内存测试不准确的情况，因为 macOS 缺少部分 Linux 的监测工具，且 Lemon-Linux 也没有针对 macOS 进行优化.

**Lemon** 是 zhipeng-jia 编写的开源评测工具，源代码托管于 zhipeng-jia/project-lemon[1].

#### 可直接运行的版本

- Ir1d 提供了一份 Linux 下编译好的版本，源代码托管于 FreestyleOJ/Project_lemon[2].
- （已停止维护）Menci 提供了一份更新的版本，源代码托管于 Menci/Lemon[3].
- （已停止维护）Dust1404 维护了一份支持子文件夹和单题测试等功能的版本，源代码托管于 Dust1404/Project_LemonPlus[4].
- iotang 和 Coelacanthus 维护了一份支持子文件夹和单题测试等功能的版本，源代码托管于 Project-LemonLime/Project_LemonLime[5].

#### 自行编译

Ubuntu：

```
sudo apt update
sudo apt install qt5-default build-essential git -y
git clone --depth=1 https://github.com/Menci/Lemon.git
cd lemon
# 可以修改 -j 后面的数字来调整 make job 的线程数
./make -j2
sudo install -Dm755 -t /usr/bin/ Lemon
```

如要编译 LemonLime，请参阅 LemonLime 的 编译手册[6].

#### 数据格式

首先打开 lemon 选择「新建试题」，然后打开新建试题的文件夹.

题目和数据应该如以下格式所示：

```
├── data
│   ├── gendata.py
│   ├── product
│   │   ├── product100.in
```

```
|   |   ├── product100.out
|   |   ├── product10.in
|   |   ├── product10.out
|   |   ├── product11.in
...
```

当所有试题添加完成后，回到 lemon 选择「自动添加试题」. 此时题目和数据点将显示在 lemon 当中.

### 外部参考资料

[1] zhipeng-jia/project-lemon 

[2] FreestyleOJ/Project_lemon 

[3] Menci/Lemon 

[4] Dust1404/Project_LemonPlus 

[5] Project-LemonLime/Project_LemonLime 

[6] 编译手册 

## 3.4 命令行

**Authors:** StudyingFather, ayalhw, qinyihao, CoderOJ, mcendu, Libaray

虽然图形界面能做的事情越来越多，但有很多高阶操作仍然需要使用命令行来解决.

本页面将简要介绍命令行的一些使用方法.

### 基础

Windows 自带的命令行界面有两个.「命令提示符」（`cmd`）是其中较为古老的一个，功能也相对简单. PowerShell 是较新的一个命令行界面，自带的功能丰富，但相对臃肿. 两个界面都可以在开始菜单中找到.

类 Unix 系统（包含 macOS 和 Linux，以下称为 Unix）分为有图形界面和无图形界面两种情况. 如果系统有图形界面（如使用 macOS 或者在 Linux 下安装了 GNOME、KDE 等图形界面），则命令行一般可以通过名为「终端」（Terminal 或 Console）的程序打开. 没有图形界面的系统会在启动完成后自动进入命令行.

Windows 下的命令行长这样：

```
C:\Users\chtholly>
```

在命令行上输入的指令会显示在 `>` 以后.

```
C:\Users\chtholly>echo "Hello World!"
```

Unix 下的命令行长这样（以 Debian/Ubuntu 为例，其它系统的命令行大体类似）：

```
chtholly@seniorious:~$
```

在命令行上输入的指令会显示在 `$` 以后.

```
chtholly@seniorious:~$ echo "Hello World!"
```

如果在 Unix 下使用 `root` 登录命令行，那么 `$` 会被替换成 `#`：

```
root@seniorious:~# apt-get install gcc
```

命令行的 `>`，`$` 或 `#` 之前会显示一个路径，这个路径就是工作目录（working directory），或者当前目录. 在 Unix 下当前目录有时会显示成类似 `~/folder` 的形式，最开头的 `~` 就是当前登录的用户的主目录. 用户 `chtholly` 的主目录在不同系统下的位置是不同的；在 Linux 下，其主目录位于 `/home/chtholly`，而在 macOS 下，其主目录位于 `/Users/chtholly`.

# 语法和常用命令[1]

## 文件系统相关

先介绍文件系统里描述位置的两种方式，相对路径和绝对路径.

- 相对路径: 用相对当前路径的位置关系来描述位置. 例如当前路径为`~/folder`, 则`./a.cpp`实际上指的就是`~/folder/a.cpp`这个文件. **随着当前路径的变化，相对路径描述的位置也可能发生改变**.
- 绝对路径: 用完整的路径来描述位置. 例如`~/folder/a.cpp`就是一个绝对路径的例子. **绝对路径描述的位置不随当前路径的变化而改变**.

  Windows/Unix 用`.`代表当前目录，`..`代表当前目录的父目录. 特别地，在 Unix 下，用`~`表示用户主目录（注意`~`由 shell 展开，因此在其他地方可能不可用）.

在 Unix 下，使用`pwd`命令可以打印当前的目录（在 Windows PowerShell 中也有此命令，但在 Windows 命令提示符中无此命令，详情见下面的提示）. 在 Windows/Unix 中，`cd <目录>`命令都可以切换当前的目录. 例如，`cd folder`会切换到当前目录的`folder`子目录；`cd ..`会切换到当前目录的父目录.

**对 Windows 命令提示符的特别提示**

在 Windows 命令提示符中并没有`pwd`命令，但可以用没有任何参数的`cd`命令近似代替.

同时，需要注意的是，在 Windows 命令提示符下使用`cd`命令切换目录，如果 **切换到的目录的盘符与当前目录的盘符不同**, 则当前目录不会改变. 你可以再敲一遍切换到的路径的盘符，也可以使用`cd /d <目录>`命令来同时切换盘符. 两种方法对应的命令行界面如下:

```
C:\Users\Libaray>cd D:\Codes
C:\Users\Libaray>D:
D:\Codes>
```

```
C:\Users\Libaray>cd /d D:\Codes
D:\Codes>
```

在 Windows 下，使用`dir`命令可以列出当前目录的文件列表. 在 Unix 下，列出文件列表的命令是`ls`. 特别的，在 PowerShell 下，可以使用与 Unix 相同的`ls`命令.

在 Windows 下，使用`md <目录>`或者`mkdir <目录>`命令创建一个新目录，使用`rd <目录>`或者`rmdir <目录>`命令删除一个目录. 在 Unix 下，这两个命令分别是`mkdir`和`rmdir`. 需要注意的是，**使用`rd`或是`rmdir`删除一个目录前，这个目录必须是空的**. 如果想要删除非空目录（和该目录下的所有文件）的话，Unix 下可以执行`rm -r <目录>`命令，Windows 下可以执行`rd /s <目录>`命令.

## 重定向机制

> 我编译了一个程序，它从标准输入读入，并输出到标准输出. 然而输入文件和输出文件都很大，这时候能不能想办法把输入重定向到指定的输入文件，输出重定向到指定的输出文件呢?

使用如下命令即可实现.

```
$ command < input > output
```

例如，`./prog < 1.in > 1.out`这个命令就将让`prog`这个程序从当前目录下的`1.in`中读入数据，并将程序输出覆盖写入到`1.out`.

[1]刘汝佳《算法竞赛入门经典（第 2 版）》附录 A 开发环境与方法

> ⚠ **Warning**
>
> `1.out` 原本的内容会被覆盖，如果想要在原输出文件末尾追加写入，请使用 `>>`，即 `./prog >> 1.out` 的方式做输出重定向

注意，PowerShell 只支持输出重定向，不支持输入重定向.

事实上，大多数 OJ 都采用了这样的重定向机制. 选手提交的程序采用标准输入输出，通过重定向机制，就可以让选手的程序从给定的输入文件读入数据，输出到指定的输出文件，再进行文件比较就可以评测了.

## 执行程序

对于一个可执行程序或是批处理脚本，只需在命令行里直接输入它的文件名即可执行它.

当然，执行一个文件时，命令行并不会把所有目录下的文件都找一遍. 环境变量 `PATH` 描述了命令行搜索路径的范围，命令行会在 `PATH` 中的路径寻找目标文件.

对于 Windows 系统，**当前目录也在命令行的默认搜索范围内**. 例如 Windows 系统中，输入 `hello` 命令就可以执行当前目录下的 `hello.exe`. 但是在 PowerShell 中，PowerShell 默认不会从当前目录寻找可执行文件（这与在 Unix 的行为一致），因而在 PowerShell 中需要使用相对路径或绝对路径调用当前目录下的可执行文件，例如 `.\hello.exe`，否则，你将看到以下报错：

```
PS> hello
hello: The term 'hello' is not recognized as a name of a cmdlet,
function, script file, or executable program.
Check the spelling of the name, or if a path was included, verify that
the path is correct and try again.

Suggestion [3,General]: The command hello was not found, but does exist
in the current location. PowerShell does not load commands from the
current location by default. If you trust this command, instead type:
".\hello". See "get-help about_Command_Precedence" for more details.
```

在 Unix 系统中，**当前目录并不在命令行的默认搜索范围内**，所以执行当前目录下的 `hello` 程序的命令就变成了 `./hello`:

```
$ hello
hello: command not found
$ ./hello
Hello World!
```

## 总结

上面介绍的用法只是命令行命令的一小部分，还有很多命令没有涉及到. 在命令行里输入帮助命令 `help`，可以查询所有基本命令以及它们的用途.

下面给出 Windows 系统和 Unix 系统的命令对照表，以供参考.

| 分类 | Windows 系统 | Unix 系统 |
| --- | --- | --- |
| 文件列表 | `dir` | `ls` |
| 切换目录 | `cd` | `cd` |
| 建立目录 | `md` | `mkdir` |
| 删除目录 | `rd` | `rmdir` |
| 比较文件 | `fc` | `diff` |

| 复制文件 | `copy` | `cp` |
|---|---|---|
| 移动文件 | `move` | `mv` |
| 文件改名 | `ren` | `mv` |
| 删除文件 | `del` | `rm` |

## 使用命令行编译/调试

参见：命令行编译与调试→3.5@p237.

## 命令行使用技巧

### 自动补全

补全是 Shell 提供的基本功能之一，主要用于减少命令行使用中的输入量和 typo 概率.

一般情况下，使用补全的快捷键一般是 `Tab`，按下后 Shell 会根据已输入的字符补全信息.

不同的 Shell 提供了能力不尽相同的补全能力.

以下是常见 Shell 的补全能力[2]:

| Shell | 补全能力（补全范围） |
|---|---|
| cmd（Windows 的传统控制台） | 文件路径 |
| PowerShell | 文件路径、PATH 中的命令名、内建命令名、函数名、命令参数，支持模糊匹配，自动纠错 |
| Bash | 文件路径、PATH 中的命令名、内建命令名、函数名、命令参数 |
| Zsh | 文件路径、PATH 中的命令名、内建命令名、函数名、命令参数，支持模糊匹配，自动纠错和建议 |
| Fish | 文件路径、PATH 中的命令名、内建命令名、函数名、命令参数，支持模糊匹配，补全时可显示参数功能，自动纠错和建议 |

> **Note**
>
> PowerShell 的部分功能需要 PSReadline Module 载入或者位于 PowerShell ISE 中.
>
> Bash 的补全功能一般需要一个名为 `bash-completions` 的包才能获得完整功能，部分软件的补全文件由软件包自带.
>
> Zsh 完整的补全功能需要配合用户预定义的文件（一般随 Zsh 包或对应软件包安装）.
>
> Fish 在默认配置下提供良好完整的补全功能，但仍有部分官方未覆盖到的软件的补全文件由软件自行提供.

### 帮助文档

一般来说，命令行下的程序都附有「帮助」，Windows 下一般使用 `command /?` 或者 `command -?` 获取，Unix-like（例如 Linux）上一般使用 `command --help` 或者 `command -h` 获取（但是 BSD 下的「帮助」往往过分简略而难以使用）.

此外，在 Unix-like 系统上，还有可通过 `man command` 获取的「手册」(manual)，相比「帮助」一般更为详细.

### built-in time 和 GNU time

测试程序运行时间时，我们通常可以使用 `time` 命令.

[2] Comparison_of_command_shells#Interactive_features[1]

但是这个命令实际上在系统中有两个对应的命令：一个是部分 Shell（例如 Bash）内建的命令，一个是 GNU time（是一个单独的软件）．这两个之间存在一些差异．

一般在 Bash 中直接使用 `time` 调用的是 Bash 内建的版本，我们可以使用 `TIMEFORMAT` 环境变量控制其输出格式，例如将其设为 `%3lR` 即可输出三位精度的实际运行时间，`%3lU` 即可输出三位精度的用户空间运行时间．[3]

如果想要调用 GNU 版本的 time，则需使用 `\time` 或者 `/usr/bin/time` 调用，但是它的输出格式并不易读，我们可以附加 `-p` 参数（即为 `\time -p`）来获得易读的输出．

## 管道

假设我们现在有两个程序 A 和 B，都用标准输入输出，如何让 A 的输出重定向到 B 的输入？

我们可以使用上文中提到的重定向的方式，先把 A 的输出重定向到一个临时文件，在把 B 的输入重定向到这个临时文件上．

但这个方法很低效，不仅需要创建新的文件，磁盘 IO 的操作也可能成为瓶颈，而且两个程序不能同时运行，必须等 A 跑完了才能开始跑 B．有没有更好的方法？

有，那就是 **管道**，使用起来也非常简单，如下操作即可：

```
$ A | B
```

这会在内存创建一个管道，然后两个程序被同时启动．程序 A 每次要输出被重定向到这个管道中，而这个管道本身不会存储数据（其实有一个很小的缓冲区）．在 B 读取之前，A 的输出操作会被阻塞，等到 B 把数据读入以后，A 的输出才能继续进行．这样优美地解决了上述的问题，没有磁盘 IO 操作，两份代码同时运行，也没有额外消耗很多的内存储存中间结果．

### 命名管道

有时候我们不只是要把一个程序的输出重定向到另一个的输入．比如在做 IO 交互题的时候，经常需要将 A 的输出重定向到 B 的输入，B 的输出重定向到 A 的输出，这个时候用上文提到的普通管道就无能为力了．而重定向到文件，有无法让两个程序同时运行．这个时候就需要一个长得像文件的管道——命名管道．

在 Unix 系统中，可以使用如下命令创建命名管道（以命名为 `my_pipe` 举例）：

```
$ mkfifo my_pipe
```

这个时候使用 `ls` 命令列出当前目录下的文件，会发现多了一个 `my_pipe|` 的文件．这就创建了一个命名管道，文件名后的 `|` 代表这是一个管道文件．然后就可以像文件的重定向一样向这个管道中读写了．

通过命名管道，我们可以通过这样的方式让两个程序交互：

```
$ mkfifo input output
$ ./checker > input < output # 这里一定要把 > input 写在前面，不然 shell 会先打开 output 管道，而这个管道现在并没有东西，会阻塞 checker 的运行．
$ ./my_code < input > output
```

使用完后，可以像普通文件一样用 `rm` 命令删除命名管道．

## 参考资料与注释

[1] Comparison_of_command_shells#Interactive_features [2] https://unix.stackexchange.com/a/70655

[3] https://unix.stackexchange.com/a/70655[2]

# 3.5 命令行编译与调试

**Authors:** CoelacanthusHex, qinyihao, StudyingFather, ksyx, NachtgeistW, CoderOJ, Enter-tainer, mcendu, Tiphereth-A, ayalhw, CCXXXI, Early0v0, HeRaNO, ouuan, swiftqwq, Xeonacid, xiaofu-15191

阅读本节之前，请先安装 GCC 和 gdb，具体方法参见 编译器→3.6@p241 一文.

## 命令行使用 g++ 编译 cpp 文件

### 手动编译

在命令行下输入 `g++ a.cpp` 就可以编译 `a.cpp` 这个文件了（Windows 系统需提前把编译器所在目录加入到 `PATH` 中）.

编译过程中可以加入一些编译选项:

- `-o <文件名>`：指定编译器输出可执行文件的文件名.
- `-g`：在编译时添加调试信息（使用 gdb 调试时需要）.
- `-Wall`：显示所有编译警告信息.
- `-O1`，`-O2`，`-O3`，`-Ofast`：对编译的程序进行优化，越往后的优化级别表示采用的优化手段越多（开启优化会影响使用 gdb 调试）.
- `-DDEBUG`：在编译时定义 `DEBUG` 符号（符号可以随意更换，例如 `-DONLINE_JUDGE` 定义了 `ONLINE_JUDGE` 符号）.
- `-UDEBUG`：在编译时取消定义 `DEBUG` 符号.
- `-lm`，`-lgmp`: 链接某个库（此处是 math 和 gmp，具体使用的名字需查阅库文档，但一般与库名相同）.

**Note**

在 Unix 下，如使用了标准 C 库里的 math 库（`math.h`），则需在编译时添加 `-lm` 参数. [1]

**如何开大栈空间?**

在 Windows 下，可以使用编译选项 `-Wl,--stack=536870912` 将栈空间开大到 512 MB，其中等号后面的数字为 **字节数**.

在 Unix 下，使用 `ulimit -s [num]` 将 **当前终端** 的栈空间调为 `[num]` KiB.

### 使用 GNU Make 的内置规则[2]

对于名为 `qwq.c/cpp` 的 C/C++ 程序源代码，可以使用 `make qwq` 自动编译成对应名为 `qwq` 的程序.

如需添加额外的编译选项，可使用 `export CFLAGS="xxx"`（C 程序）或 `export CXXFLAGS="xxx"`（C++ 程序）指定. 如需添加额外的预编译选项，可使用 `export CPPFLAGS="xxx"` 指定. 上述设置方法也可以写做类似 `CFLAGS="xxx" CPPFLAGS="xxx" make qwq` 来指定单次命令执行中使用的环境变量.

### Sanitizers

### 介绍

sanitizers 是一种集成于编译器中，用于调试 C/C++ 代码的工具，通过在编译过程中插入检查代码来检查代码运行时出现的内存访问越界、未定义行为等错误.

它分为以下几种:

- AddressSanitizer[3]：检测对堆、栈、全局变量的越界访问，无效的释放内存、内存泄漏（实验性）.

[1] Why do you have to link the math library in C?[1]
[2] Catalogue of Built-In Rules[2]

- ThreadSanitizer[4]：检测多线程的数据竞争.
- MemorySanitizer[5]：检测对未初始化内存的读取.
- UndefinedBehaviorSanitizer[6]：检测未定义行为.

### 使用方式

最新版本的 clang++、g++ 以及 MSVC（部分支持）均已内置 sanitizers，但功能和使用方法有所不同，这里以 clang++ 为例，它的使用方法如下：

```
$ clang++ -fsanitize=<name> test.cc
```

其中 `<name>` 即为要启用的功能（一个 sanitizer 可理解为一些功能的集合），例如：

```
$ clang++ -fsanitize=memory test.cc # 启用 MemorySanitizer
$ clang++ -fsanitize=signed-integer-overflow test.cc # 启用有符号整型溢出检测
```

之后直接像平常一样运行可执行文件即可，如果 sanitizer 检测到错误，则会输出到 `stderr` 流，例如：

```
$ ./a.out
test.cc:3:5: runtime error: signed integer overflow: 2147483647 + 1 cannot be
represented in type 'int'
```

> **Warning**
>
> Windows 下的 g++ 不支持 sanitizers，需要使用 修改过后的 MinGW64[7] 或使用其它编译器.
>
> MSVC 从 16.0 截至版本 17.14 仅支持 AddressSanitizer.

### 时间/内存代价

显而易见，这些调试工具会严重拖慢代码的运行时间和增大所用内存，以下为使用它们的时间/内存代价：

| 名称 | 所增大内存倍数 | 所增大时间倍数 |
| --- | --- | --- |
| AddressSanitizer | N/A | 2 |
| ThreadSanitizer | 5~15 | 5~10 |
| MemorySanitizer | N/A | 3 |
| UndefinedBehaviorSanitizer | N/A | N/A |

## 命令行使用 gdb 调试

```
$ g++ a.cpp -o a -g
$ gdb ./a
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04.2) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
```

[3]https://clang.llvm.org/docs/AddressSanitizer.html[3]
[4]https://clang.llvm.org/docs/ThreadSanitizer.html[4]
[5]https://clang.llvm.org/docs/MemorySanitizer.html[5]
[6]https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html[6]

```
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.
--Type <RET> for more, q to quit, c to continue without paging--
```

按下 c 继续. 接着提示 `Reading symbols from [filename]...`, 出现 `(gdb)` 字样时, 就可以输入命令调试了.

以下是按照分类列表的常用命令:

### gdb 基础命令

| 命令 | 描述 |
| --- | --- |
| `help` | 显示帮助信息 |
| `quit` | 退出 gdb |
| `file [filename]` | 加载要调试的程序 `[filename]` |

### 运行控制命令

| 命令 | 描述 |
| --- | --- |
| `run` | 运行程序，直到遇到断点或程序结束 |
| `continue` | 继续运行，直到遇到断点或程序结束 |
| `next` | 单步执行，遇到函数调用则进入函数 |
| `step` | 单步执行，遇到函数调用则进入函数 |
| `finish` | 运行到当前函数返回为止，然后停下来等待命令 |
| `until [num]` | 运行到指定行号 `[num]` 为止，然后停下来等待命令 |
| `break [num]` | 在第 `[num]` 行设置断点，程序运行到该行时停下来等待命令 |
| `condition [id] [p]` | 设置编号为 `[id]` 的断点条件，只有满足表达式 `[p]` 条件时，断点被启用 |
| `ignore [id] [num]` | 忽略前 `[num]` 次触发断点 |
| `delete [id]` | 删除指定编号的断点 |
| `disable [id]` | 禁用指定编号的断点 |
| `enable [id]` | 启用指定编号的断点 |
| `list` | 列出源代码，接着上次的位置往下列，每次列 10 行 |
| `list [num]` | 列出以第 `[num]` 行为中间行的源代码 |
| `list [func-name]` | 列出某个函数为中间行的源代码 |
| `call [function]` | 调用函数，并打印返回值 |

`break [num]` 会输出断点的编号，也可以使用 `break [func-name]` 设置函数断点;

你也可以使用 `break [num] [p]` 在设置断点时实现与 `condition [id] [p]` 接近的效果.

### 栈帧命令

| 命令 | 描述 |
| --- | --- |
| `info args` | 查看函数的参数 |
| `backtrace` | 查看各级函数调用及参数 |
| `frame` | 选择栈帧 |
| `up` | 向上移动一级栈帧 |
| `down` | 向下移动一级栈帧 |

## 变量命令

| 命令 | 描述 |
| --- | --- |
| `print [p]` | 打印表达式 `[p]` 的值，通过表达式可以修改变量的值 |
| `display [p]` | 每次暂停时打印表达式 `[p]` 的值，但不进入函数 |
| `watch [var]` | 监视变量 `[var]` 的值，当变量被写入时，会自动打印出来并暂停 |
| `rwatch [var]` | 监视变量 `[var]` 的值，当变量被读取时，会自动打印出来 |
| `awatch [var]` | 当变量 `[var]` 被修改或写入时，会自动打印出来并暂停 |
| `set [assignment]` | 执行赋值语句 |

`display` 和 `print` 指令都支持控制输出格式，其方法是在命令后紧跟 `/` 与格式字符，例如 `print/display [var]`（按照十进制打印变量 `[var]` 的值），支持的格式字符有：

| 格式字符 | 对应格式 |
| --- | --- |
| `d` | 按十进制格式显示变量 |
| `x` | 按十六进制格式显示变量 |
| `a` | 按十六进制格式显示变量 |
| `t` | 按二进制格式显示变量 |
| `c` | 按字符格式显示变量 |
| `f` | 按浮点数格式显示变量 |
| `u` | 按十进制格式显示无符号整型 |
| `o` | 按八进制格式显示变量 |

## 信息命令

| 命令 | 描述 |
| --- | --- |
| `info breakpoints` | 列出所有断点 |
| `info locals` | 列出当前栈帧的局部变量 |
| `info args` | 列出当前栈帧的函数参数 |
| `info threads` | 列出所有线程 |
| `info program` | 显示程序的当前状态 |
| `info registers` | 显示当前寄存器的值 |
| `info frame` | 显示当前栈帧的信息 |

## 其他命令

| 命令 | 描述 |
| --- | --- |
| `enable pretty-printer` | 启用 pretty-printer，可以以人类可读的方式打印 STL 容器 |
| `checkpoint`[7] | 创建检查点，可以回滚到检查点 |
| `restart [num]`[8] | 回滚到第 `[num]` 个检查点 |
| `save breakpoints [filename]` | 保存断点到文件 |
| `source [filename]` | 导入断点文件 |

[7] 与检查点相关的指令仅适用于 GNU/Linux 平台．详见 GDB 官方手册[8]．
[8] 与检查点相关的指令仅适用于 GNU/Linux 平台．详见 GDB 官方手册[9]．

> **提示**
>
> gdb 调试时的命令大多都可以被简写为可以唯一确定的字母缩写，例如 `breakpoint` 简写为 `b`，`step` 简写为 `s`，`info args` 简写为 `i ar`．详见 `help` 命令．

## 参考资料与注释


[1] Why do you have to link the math library in C?

[2] Catalogue of Built-In Rules

[3] https://clang.llvm.org/docs/AddressSanitizer.html

[4] https://clang.llvm.org/docs/ThreadSanitizer.html

[5] https://clang.llvm.org/docs/MemorySanitizer.html

[6] https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[7] 修改过后的 MinGW64

[8] GDB 官方手册

[9] GDB 官方手册


# 3.6 编译器

本页面主要介绍了各系统下各类编译器/解释器的安装步骤．

## GCC

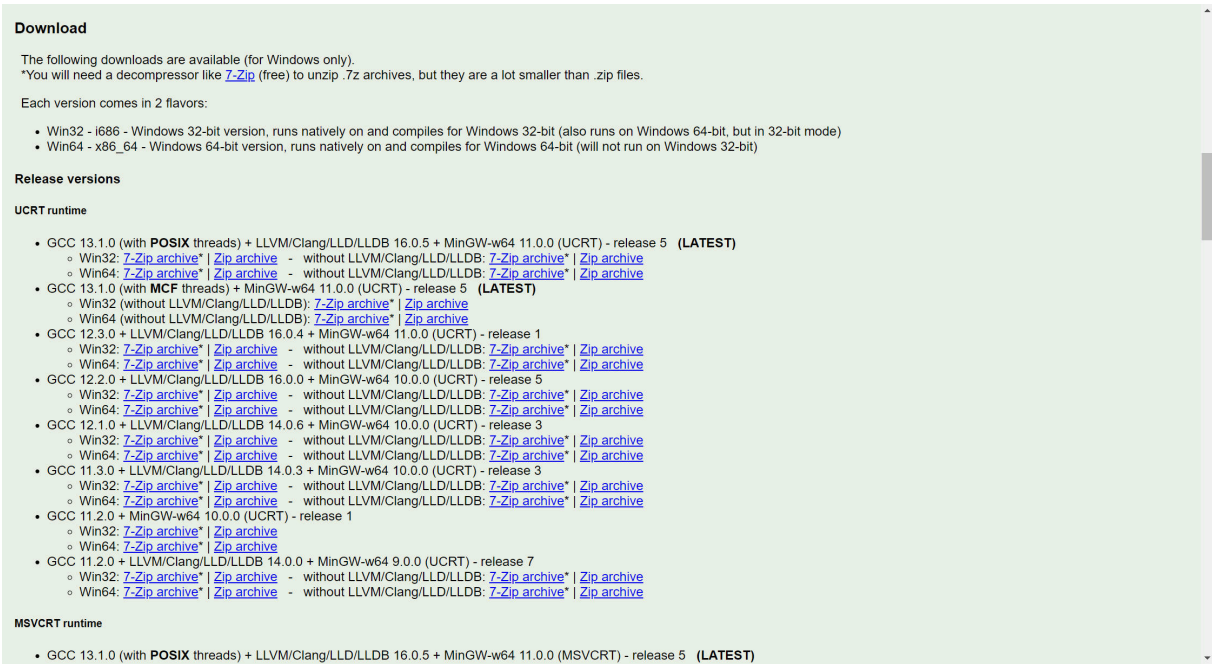
### Windows

#### 手动下载安装

访问 MinGW-w64[1] 的下载页面，有多个构建版本．方便起见，我们使用由 WinLibs 提供的构建版本．

首先前往 WinLibs[2] 下载最新的安装包，选择合适的版本，本文选择了 GCC 12.3.0 + LLVM/Clang/LLD/LLDB 16.0.4 + MinGW-w64 11.0.0 (UCRT)：

默认会附带安装 LLVM Clang，如果不想安装，你也可以选择右边的 without LLVM/Clang/LLD/LLDB．



**Download**

The following downloads are available (for Windows only).
*You will need a decompressor like 7-Zip (free) to unzip .7z archives, but they are a lot smaller than .zip files.

Each version comes in 2 flavors:

- Win32 - i686 - Windows 32-bit version, runs natively on and compiles for Windows 32-bit (also runs on Windows 64-bit, but in 32-bit mode)
- Win64 - x86_64 - Windows 64-bit version, runs natively on and compiles for Windows 64-bit (will not run on Windows 32-bit)

**Release versions**

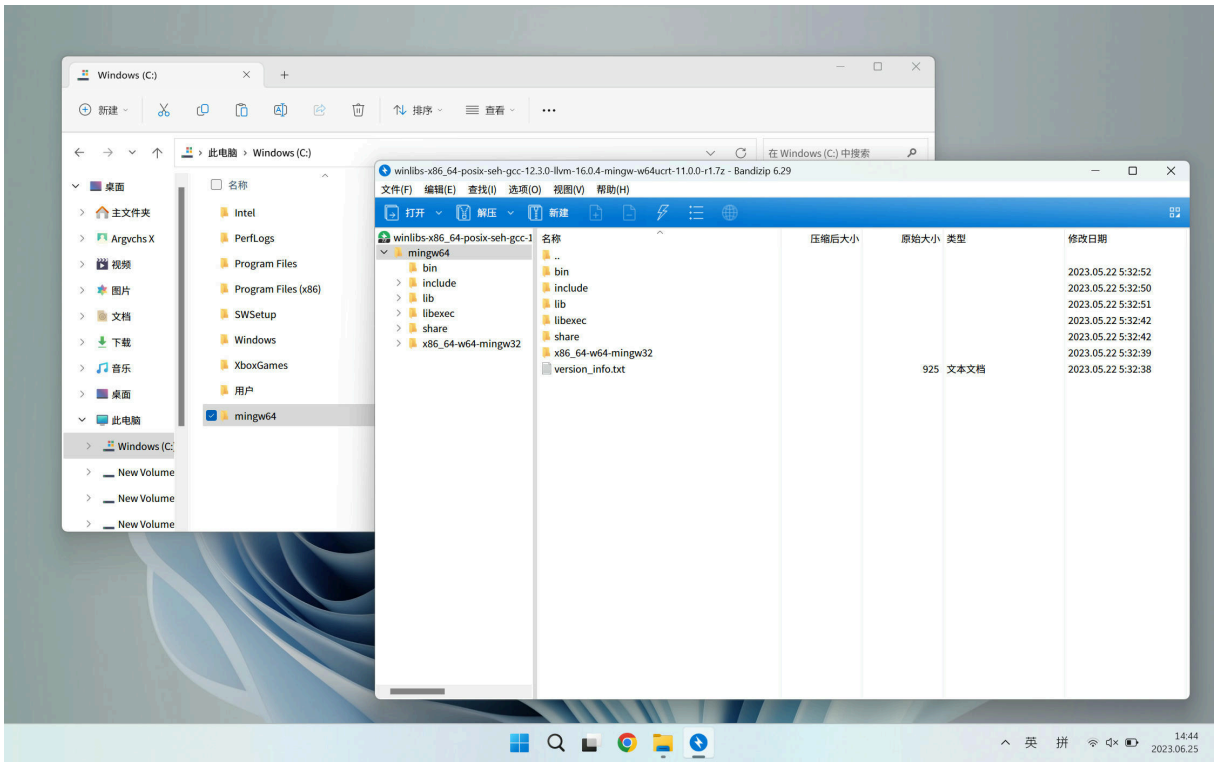
**UCRT runtime**

- GCC 13.1.0 (with **POSIX** threads) + LLVM/Clang/LLD/LLDB 16.0.5 + MinGW-w64 11.0.0 (UCRT) - release 5 **(LATEST)**
  - Win32: 7-Zip archive* | Zip archive - without LLVM/Clang/LLD/LLDB: 7-Zip archive* | Zip archive
  - Win64: 7-Zip archive* | Zip archive - without LLVM/Clang/LLD/LLDB: 7-Zip archive* | Zip archive
- GCC 13.1.0 (with **MCF** threads) + MinGW-w64 11.0.0 (UCRT) - release 5 **(LATEST)**
  - Win32 (without LLVM/Clang/LLD/LLDB): 7-Zip archive* | Zip archive
  - Win64 (without LLVM/Clang/LLD/LLDB): 7-Zip archive* | Zip archive
- GCC 12.3.0 + LLVM/Clang/LLD/LLDB 16.0.4 + MinGW-w64 11.0.0 (UCRT) - release 1
  - Win32: 7-Zip archive* | Zip archive - without LLVM/Clang/LLD/LLDB: 7-Zip archive* | Zip archive
  - Win64: 7-Zip archive* | Zip archive - without LLVM/Clang/LLD/LLDB: 7-Zip archive* | Zip archive
- GCC 12.2.0 + LLVM/Clang/LLD/LLDB 16.0.0 + MinGW-w64 10.0.0 (UCRT) - release 5
  - Win32: 7-Zip archive* | Zip archive - without LLVM/Clang/LLD/LLDB: 7-Zip archive* | Zip archive
  - Win64: 7-Zip archive* | Zip archive - without LLVM/Clang/LLD/LLDB: 7-Zip archive* | Zip archive
- GCC 12.1.0 + LLVM/Clang/LLD/LLDB 14.0.6 + MinGW-w64 10.0.0 (UCRT) - release 3
  - Win32: 7-Zip archive* | Zip archive - without LLVM/Clang/LLD/LLDB: 7-Zip archive* | Zip archive
  - Win64: 7-Zip archive* | Zip archive - without LLVM/Clang/LLD/LLDB: 7-Zip archive* | Zip archive
- GCC 11.3.0 + LLVM/Clang/LLD/LLDB 14.0.3 + MinGW-w64 10.0.0 (UCRT) - release 3
  - Win32: 7-Zip archive* | Zip archive - without LLVM/Clang/LLD/LLDB: 7-Zip archive* | Zip archive
  - Win64: 7-Zip archive* | Zip archive - without LLVM/Clang/LLD/LLDB: 7-Zip archive* | Zip archive
- GCC 11.2.0 + MinGW-w64 10.0.0 (UCRT) - release 1
  - Win32: 7-Zip archive* | Zip archive
  - Win64: 7-Zip archive* | Zip archive
- GCC 11.2.0 + LLVM/Clang/LLD/LLDB 14.0.0 + MinGW-w64 9.0.0 (UCRT) - release 7
  - Win32: 7-Zip archive* | Zip archive - without LLVM/Clang/LLD/LLDB: 7-Zip archive* | Zip archive
  - Win64: 7-Zip archive* | Zip archive - without LLVM/Clang/LLD/LLDB: 7-Zip archive* | Zip archive

**MSVCRT runtime**

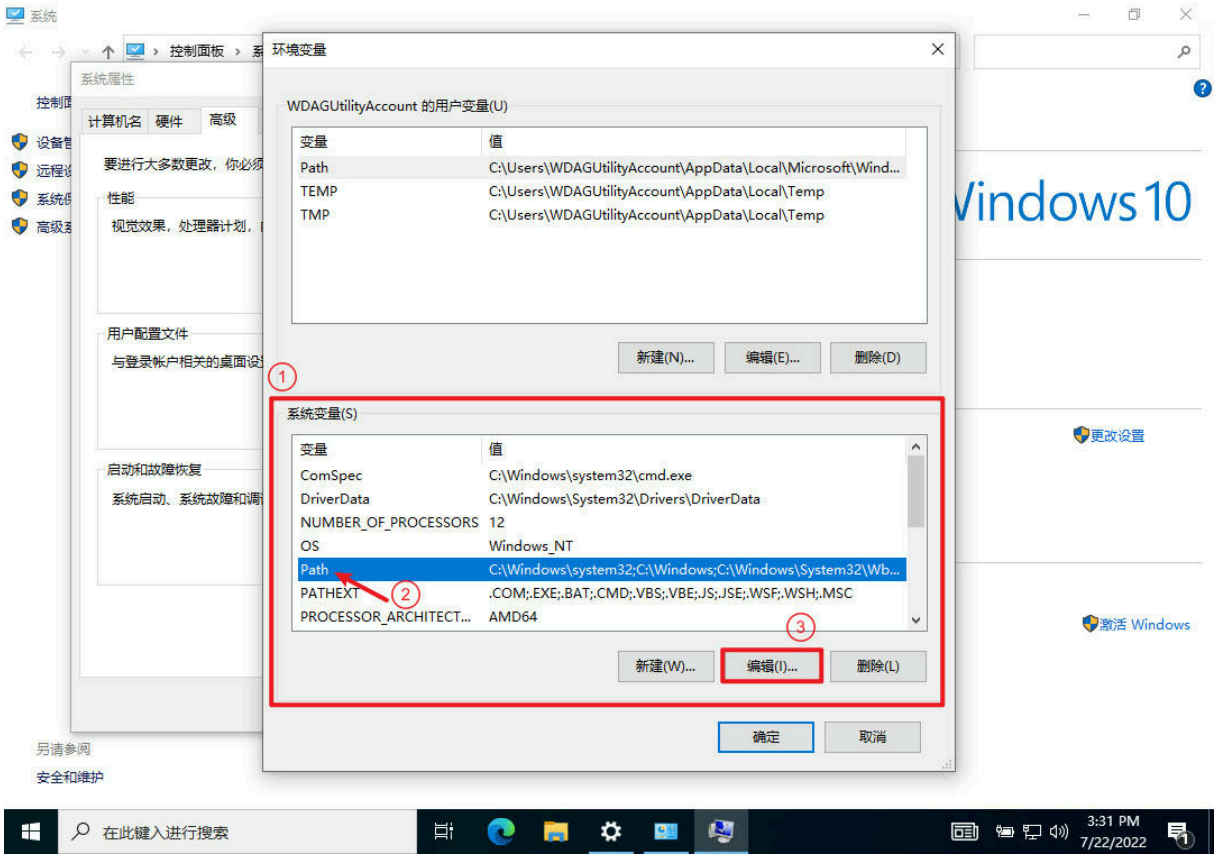
- GCC 13.1.0 (with **POSIX** threads) + LLVM/Clang/LLD/LLDB 16.0.5 + MinGW-w64 11.0.0 (MSVCRT) - release 5 **(LATEST)**

下载好后将其解压到电脑中的某个位置，教程中将其解压到了 C 盘的根目录．目录名中最好不要包含非英文字符和空格，否则可能会在后期导致一些问题．

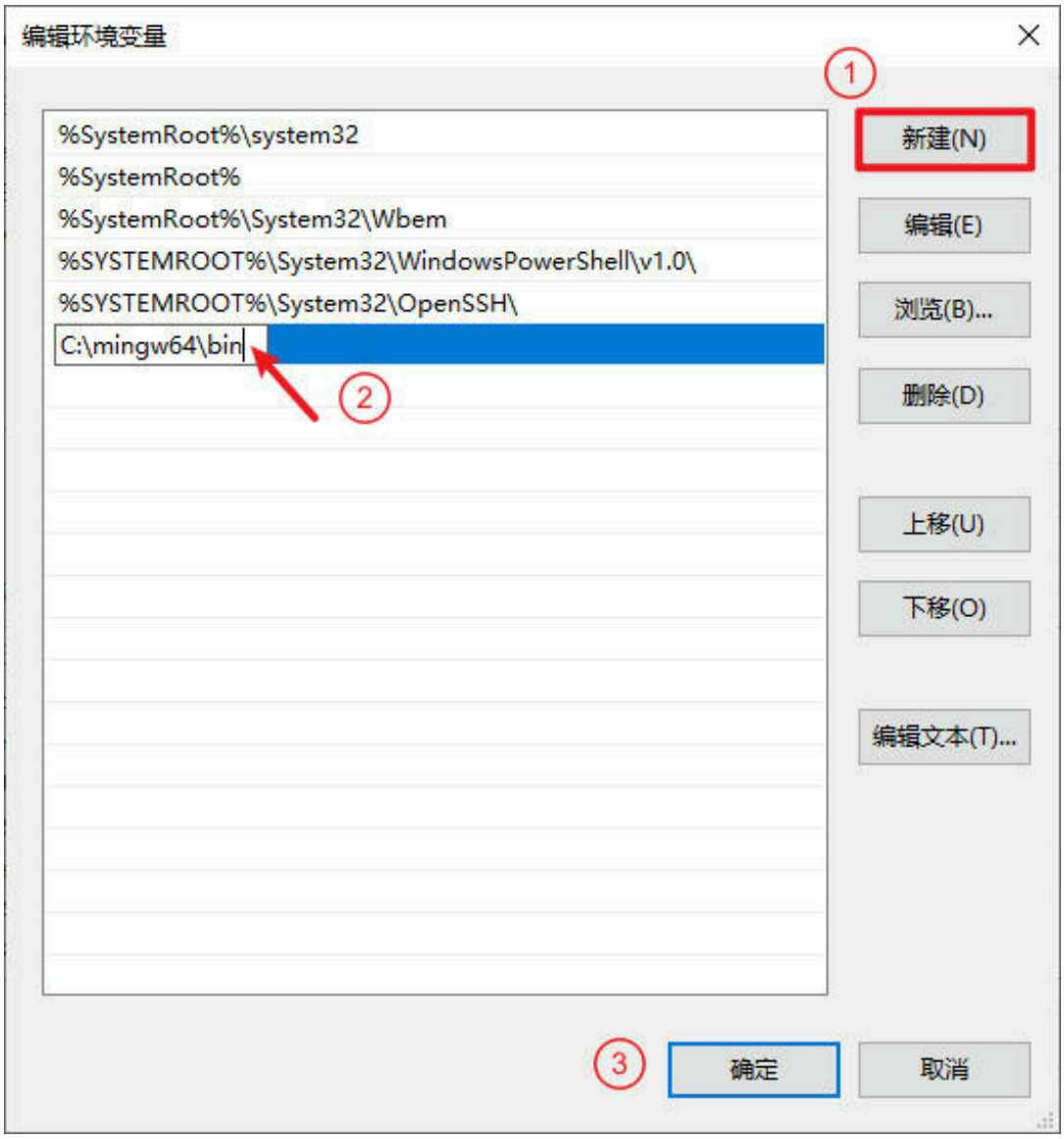



接下来我们需要将编译器的可执行文件目录添加到系统环境变量中，这样在编译时就不需要指定编译器的路径了，方便使用．上方我们将 MinGW 解压到了 `C:\mingw64` 目录中，那么可执行文件所在的目录就是 `C:\mingw64\bin`.

按下 Windows 徽标 + R 组合键，输入 `rundll32.exe sysdm.cpl,EditEnvironmentVariables`，打开系统环境变量设置窗口，并在「系统变量」一节中选中名为「Path」的变量，然后点击「编辑」按钮：

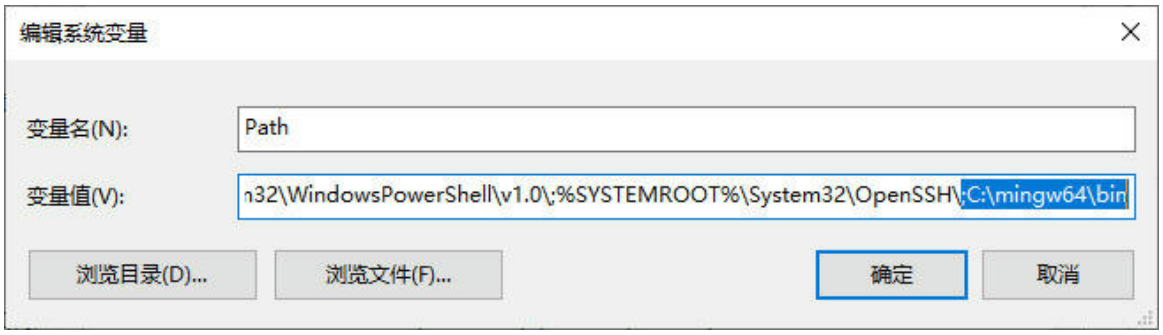



在编辑窗口中点击右侧的「新建」按钮，为「Path」变量新建一个条目，并填入上文中记录下的可执行文件所在的目录（教程中为 `C:\mingw64\bin`）.



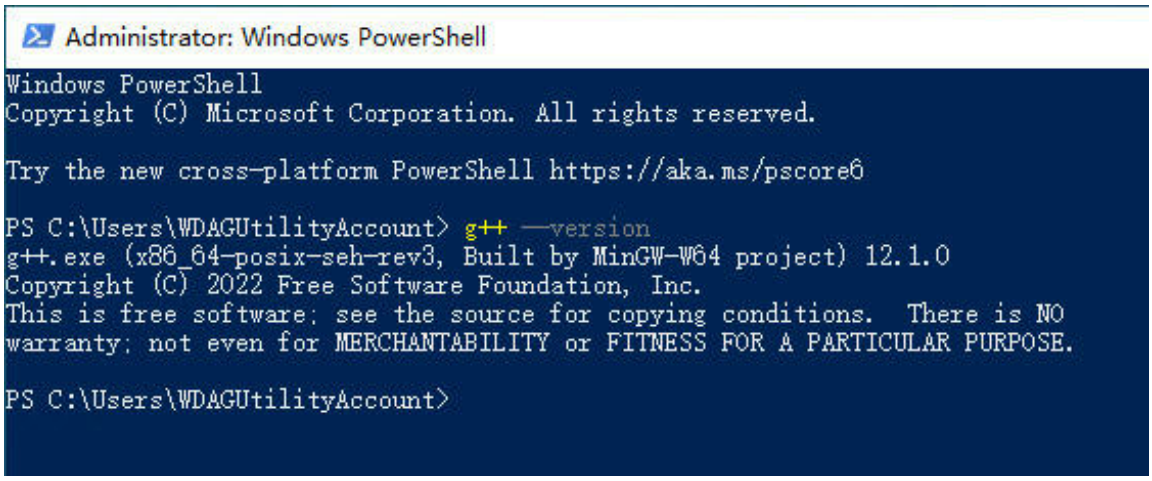

**对部分老版本系统的提示**

部分老版本系统只能手动修改变量的文本值，那么需要在变量的值的末尾插入一个**半角分号**，再将可执行文件所在的目录粘贴到这个半角分号的后面，如图所示：




完成后一路点击「确定」按钮退出即可.

接下来打开终端，输入 `g++ --version` 并按下回车，如果出现如图所示的提示则代表安装成功.




### MSYS2 安装

MSYS2 提供多个彼此独立的开发环境．如果不确定应该选择哪个，MSYS2 官方建议使用 UCRT64．

打开 PowerShell，运行以下命令安装 MSYS2：

```
winget install MSYS2.MSYS2
```

安装完成后，在开始菜单中搜索并打开 MSYS2 UCRT64 终端．MSYS2 是滚动发行版，首先完整更新系统：

```
pacman -Suy
```

如果更新过程中提示关闭所有 MSYS2 进程，确认后重新打开 MSYS2 UCRT64 终端，并再次执行 `pacman -Suy`，直到更新完成．

然后安装 UCRT64 环境下的完整 GNU 工具链：

```
pacman -S ${MINGW_PACKAGE_PREFIX}-toolchain
```

在 UCRT64 环境中，`MINGW_PACKAGE_PREFIX` 的值为 `mingw-w64-ucrt-x86_64`，因此上述命令会安装 GCC、GDB、Make 等工具．安装完成后，可运行以下命令验证：

```
gcc --version
g++ --version
gdb --version
```

**为什么是 UCRT64**

UCRT64 使用 GCC、UCRT 和 libstdc++，也是 MSYS2 在不确定如何选择环境时推荐使用的环境．CLANG64 同样使用 UCRT，但默认使用 LLVM/Clang、LLD 和 libc++．MINGW64 使用较旧的 MSVCRT，目前已被 MSYS2 列为 legacy 环境．MSYS 环境则主要用于运行依赖 POSIX 兼容层的 Unix 工具，不适合用来生成普通的原生 Windows 竞赛程序．

UCRT 能提高与 MSVC 在 C 运行时层面的兼容性，但这并不意味着 GCC/MinGW 与 MSVC 的 C++ ABI 或对象文件、静态库可以直接互换．

> ⚠ **不要修改全局 Path**
>
> 不建议将 `C:\msys64\ucrt64\bin` 添加到 Windows 的全局 `Path`. MSYS2 UCRT64 终端启动时已经会为当前环境设置正确的 `PATH`; 直接使用对应的 MSYS2 终端可以避免不同环境的工具链和运行库相互混用.
>
> 需要注意的是，在不把 UCRT64 的 `bin` 目录加入 Windows 全局 `Path` 的情况下，也不应直接在普通的 PowerShell 或 CMD 中调用 MSYS2 安装的 GCC. 例如，GCC 在编译过程中会启动 `cc1.exe` 等内部程序，而这些程序依赖的 DLL 位于 UCRT64 环境的 `bin` 目录中；如果没有由 MSYS2 UCRT64 终端预先设置好环境变量，就可能因找不到依赖 DLL 而启动失败. 因此，使用这套 GCC 工具链时，应从 MSYS2 UCRT64 终端或继承了该终端环境变量的终端中运行编译命令.

#### Scoop 安装

打开 PowerShell，运行以下脚本:

```
Set-ExecutionPolicy RemoteSigned -Scope CurrentUser
irm get.scoop.sh | iex
scoop install mingw-winlibs
```

### Linux

#### Debian/Ubuntu

首先先更新软件包列表:

```
sudo apt update
```

再使用命令直接安装即可:

```
sudo apt install g++
```

#### Arch Linux

使用命令直接安装即可:

```
sudo pacman -Syu gcc
```

#### openSUSE

使用命令直接安装即可:

```
sudo zypper in gcc-c++
```

### macOS

首先更新包管理器:

```
brew upgrade
brew update
```

再使用命令直接安装即可:

```
brew install gcc
```

## JDK

JDK 的发行版有很多，以下介绍两种:

- OpenJDK 中的 Eclipse Temurin[3]1: 参见 Install Eclipse TemurinTM | Adoptium[8].

- Oracle JDK：可参见 JDK Installation Guide（JDK 17）[9].

## Python 3

Python 的实现也有很多[2]，以 CPython 3 为例，参见 Download Python | Python.org[11].

## LLVM

### Windows

> **LLVM 在 Windows 上的坑**
>
> 由于 LLVM 在 Windows 上缺失标准库，所以你仍需安装 MSVC 或 GCC.

#### 直接安装

访问 LLVM[12] 的下载页面，选择 LLVM-*-win64.exe 下载.

如果你的网络质量不佳，你也可以选择访问 清华大学开源软件镜像站[13] 进行下载.

打开 .exe 文件，安装时勾选 Add LLVM to system PATH for current user，随后一直点击下一步即可安装完成.

打开终端，输入 `clang++ --version` 并回车，出现

```
clang version 15.0.1
Target: x86_64-pc-windows-msvc
Thread model: posix
InstalledDir: <omitted>
```

类似物即代表成功.

#### MSYS2 安装

MSYS2 中既可以在 UCRT64 环境里额外安装 Clang，也可以直接使用以 LLVM 为默认工具链的 CLANG64 环境.

##### UCRT64 环境

如果平时主要使用 GNU GCC，只是偶尔需要 Clang，可以在 MSYS2 UCRT64 终端中安装：

```
pacman -S ${MINGW_PACKAGE_PREFIX}-clang ${MINGW_PACKAGE_PREFIX}-llvm
```

如果还需要 `clangd`、`clang-tidy` 等开发工具，再安装：

```
pacman -S ${MINGW_PACKAGE_PREFIX}-clang-tools-extra
```

UCRT64 属于 GCC 系环境，因此这里安装的 Clang 默认仍使用 GNU 链接器和 libstdc++，可以继续使用 `bits/stdc++.h` 等 libstdc++ 提供的内容. 安装完成后，可运行以下命令验证：

```
clang --version
clang++ --version
```

##### CLANG64 环境

如果希望使用以 LLVM 为核心的完整环境，可以从开始菜单打开 MSYS2 CLANG64 终端. 如果此前还没有更新 MSYS2，先执行：

```
pacman -Suy
```

---

[1] Eclipse Temurin[4] 即为原 AdoptOpenJDK[5]，后者已于 2021 年 7 月移交至 Eclipse 基金会[6]. 具体可见 本声明[7].
[2] Alternative Python Implementations | Python.org[10]

如果更新过程中提示关闭所有 MSYS2 进程，重新打开 MSYS2 CLANG64 终端后再次执行该命令．随后安装当前环境的完整工具链：

```
pacman -S ${MINGW_PACKAGE_PREFIX}-toolchain
```

在 CLANG64 环境中，`MINGW_PACKAGE_PREFIX` 的值为 `mingw-w64-clang-x86_64`；该工具链包含 Clang、LLVM、LLD、LLDB、libc++、Make 等组件．CLANG64 终端会自动设置当前环境所需的 `PATH`，同样不建议把 `C:\msys64\clang64\bin` 添加到 Windows 的全局 `Path`．

#### Scoop 安装

打开 PowerShell，运行以下脚本：

```
Set-ExecutionPolicy RemoteSigned -Scope CurrentUser
irm get.scoop.sh | iex
scoop install llvm
```

### Linux

#### openSUSE

使用命令直接安装即可：

```
sudo zypper in llvm clang
```

## MSVC (Visual Studio)

访问 下载 Visual Studio[14] 页面，找到「下载」一节中的「社区」部分，点击「免费下载」．下载完成后打开安装器选择「Community 2022 安装」．在随后弹出来的窗口中仅选择「使用 C++ 的桌面开发」，然后单击安装．

如果你不想安装完整的 Visual Studio，可以滚动到下方「所有下载」一节，在「用于 Visual Studio 的工具」中找到「Visual Studio 生成工具」，点击后方的「下载」．下载完成后打开安装器，按照提示步骤选择「使用 C++ 的桌面开发」后安装即可．也可以使用 PortableBuildTools[15] 工具以仅安装 MSVC 编译器．

[1] MinGW-w64

[2] WinLibs

[3] Eclipse Temurin

[4] Eclipse Temurin

[5] AdoptOpenJDK

[6] Eclipse 基金会

[7] 本声明

[8] Install Eclipse TemurinTM | Adoptium

[9] JDK Installation Guide（JDK 17）

[10] Alternative Python Implementations | Python.org

[11] Download Python | Python.org

[12] LLVM

[13] 清华大学开源软件镜像站

[14] 下载 Visual Studio

[15] PortableBuildTools

# 3.7 WSL (Windows 10)

> **Authors:** GoodCoder666, Ir1d, H-J-Granger, NachtgeistW, StudyingFather, Enter-tainer, abc1763613206, Anti-Li, shenyouran, Chrogeek, SukkaW, Henry-ZHR, Early0v0, andylizf, tootal, Marcythm, CoelacanthusHex, indevn, qinyihao, peasoft




本章主要介绍了在 Windows 系统下使用 Windows Subsystem for Linux 运行 Linux 环境的方法.

## 引言[1]

现在大部分学校的竞赛练习环境都是构建在 Windows 系操作系统上，但是在 NOI 系列赛中，已经用上了 NOI Linux 这个 Ubuntu 操作系统的修改版.

NOI 竞赛（自 2021 年 9 月 1 日）的环境要求如下.[2]

| 类别 | 软件或模块 | 版本 | 备注说明 |
|---|---|---|---|
| 系统 | Linux 内核 | `5.4.0-42-generic` | 64 位 x86 (AMD64) |
| 语言环境 | GCC（`gcc` 和 `g++`） | `9.3.0` | C 和 C++ 编译器 |
| | FPC | `3.0.4` | Pascal 编译器（注：自 2022 年起，NOI 系列竞赛不再支持 Pascal 语言） |
| | Python 2 | `2.7` | 非竞赛语言 |
| | Python 3 | `3.8` | 非竞赛语言 |
| 调试工具 | GDB | `9.1` | |
| | DDD | `3.3.12` | GDB 的 GUI 前端 |
| 集成开发环境（IDE） | Code::Blocks | `20.03` | C/C++ IDE |
| | Lazarus | `2.0.6` | Pascal IDE |
| | Geany | `1.36` | C/C++/Pascal（轻量级）IDE |
| 文本编辑工具 | Visual Studio Code | `1.54.3` | |
| | GNU Emacs | `26.3` | |
| | gedit | `3.36.2` | |
| | Vim | `8.1` | |
| | Joe | `4.6` | |
| | nano | `4.8` | |
| | Sublime Text | `3.2.2` | |
| 其它软件 | Firefox | `79.0` | 浏览器 |
| | Midnight Commander (`mc`) | `4.8.24` | 文件管理器 |
| | xterm (uxterm) | `3.5.3` | 终端 |
| | Arbiter-local | `1.02` | 程序评测工具单机版 |

考场环境与一般环境会有一系列差异:

- 命令行上的操作和图形界面上的操作会有差异.

[1]洛谷日报 #6[1]
[2]NOI Linux 2.0 发布，将于 9 月 1 日起正式启用！[2]

- Linux 和 Windows 的差异，如对于大小写的敏感性差异.
- 不同编译器的行为（MSVC 和 GCC）和不同版本的编译器（Windows 上和 Linux 上的 GCC，32 位和 64 位的 Linux GCC，GCC 7 和 GCC 8 等）的行为，如对变量初始化和对数组下标越界的处理会有差异.
- 不同评测系统（洛谷和 Arbiter）的超时检查和内存限制检查会有差异.

这有可能导致一系列的尴尬情况：

- 想用 `Ctrl` + `C` 复制，结果退出了程序.
- 平时 AC 的程序模板到了 Linux 上就 WA.

为了防止考场上出现此类尴尬情况，我们必须要提前熟悉 Linux 系统的操作方法.

虽然 NOI 的官网已经放出了 NOI Linux 的 ISO 镜像，虚拟机的配置较为麻烦. 且由于 NOI Linux 默认自带图形界面，无法保证在低配系统上流畅运行.

Windows 10 在一周年更新时推出了 Linux 子系统（WSL），在 2020 年 5 月更新中升级到了 WSL 2. 截至 2020 年 6 月 1 日，WSL 已支持安装 Ubuntu、openSUSE Leap、Kali、Debian 等主流 Linux 分发版. 但 WSL 并不支持 NOI 评测用的 Arbiter.

**什么是 Linux 子系统（WSL）**[3]

适用于 Linux 的 Windows 子系统（英语：Windows Subsystem for Linux，简称 WSL）是一个为在 Windows 10、Windows 11 与 Windows Server 2019 上能够原生运行 Linux 二进制可执行文件（ELF 格式）的兼容层.

WSL 可让开发人员按原样运行 GNU/Linux 环境 - 包括大多数命令行工具、实用工具和应用程序 - 且不会产生虚拟机开销.

WSL 仅在 64 位 Windows 10 版本 1607 及以上、Windows 11 和 Windows Server 2019/2022 中可用.

## 启用 WSL[3]

### 自动安装

**Warning**

本部分适用于 Windows 10 版本 2004 及更高版本（内部版本 19041 及更高版本）或 Windows 11.

如果你正在使用 2004 以下版本或你的电脑不支持虚拟化，请阅读下面的手动安装一节.

如果你正在使用 1607 以下版本的 Windows 10，你的系统不支持 WSL.

1. 以管理员身份打开 Windows PowerShell（右击「开始」按钮，选择 Windows PowerShell（管理员）或 Windows 终端（管理员））
2. 输入 `wsl --install`，并等待所有组件自动安装完成. 期间你可能需要重启你的计算机来启用必要的 Windows 功能.
3. 安装完成后，你可以在「开始」菜单或 Windows 终端的标签页中找到你安装的发行版.
4. 接下来，请转到下面「配置分发版」一节完成其他设置.

**提示**

如果你的 `wsl --install` 进度条很慢或多次失败，请在 GitHub 上的 release 页面[5] 下载后缀名为 `.msi` 的离线安装包.

[3]安装 WSL, Microsoft Docs[4]

## 手动安装[4]

> **Warning**
>
> 下面介绍手动安装 WSL 的步骤. 如果你已经完成了自动安装，请跳过此部分.

### 启用适用于 Linux 的 Windows 子系统

在安装适用于 WSL 的任何 Linux 分发版之前，必须在下述两种方法中选择一种，以确保启用「适用于 Linux 的 Windows 子系统」可选功能:

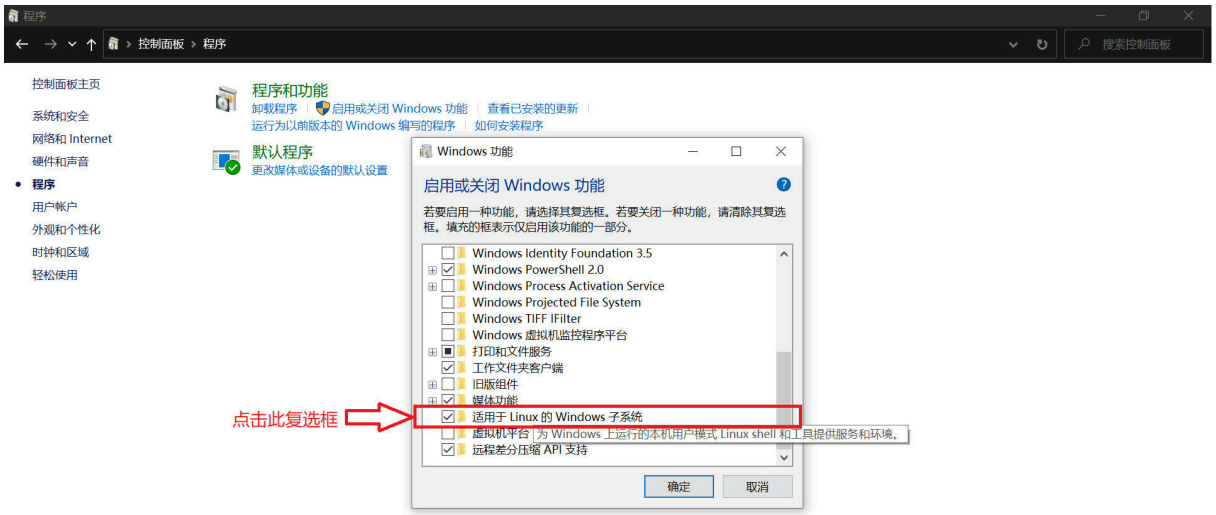
使用命令行:

1. 以管理员身份打开 PowerShell 并运行:

```
Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Windows-Subsystem-Linux
# 如果你只想要使用 WSL 1 请跳过此步骤
Enable-WindowsOptionalFeature -Online -FeatureName VirtualMachinePlatform
```

2. 出现提示时，重启计算机.

使用图形界面:




1. 打开「控制面板」
2. 访问「程序和功能」子菜单「打开或关闭 Windows 功能」
3. 选择「适用于 Linux 的 Windows 子系统」与「虚拟机平台」
4. 点击确定
5. 重启

### 安装内核更新包

如果你想要使用 WSL 1, 请跳过此步骤.

下载 适用于 x64 计算机的 WSL2 Linux 内核更新包[7] 并安装.

[4] 旧版 WSL 的手动安装步骤[6]

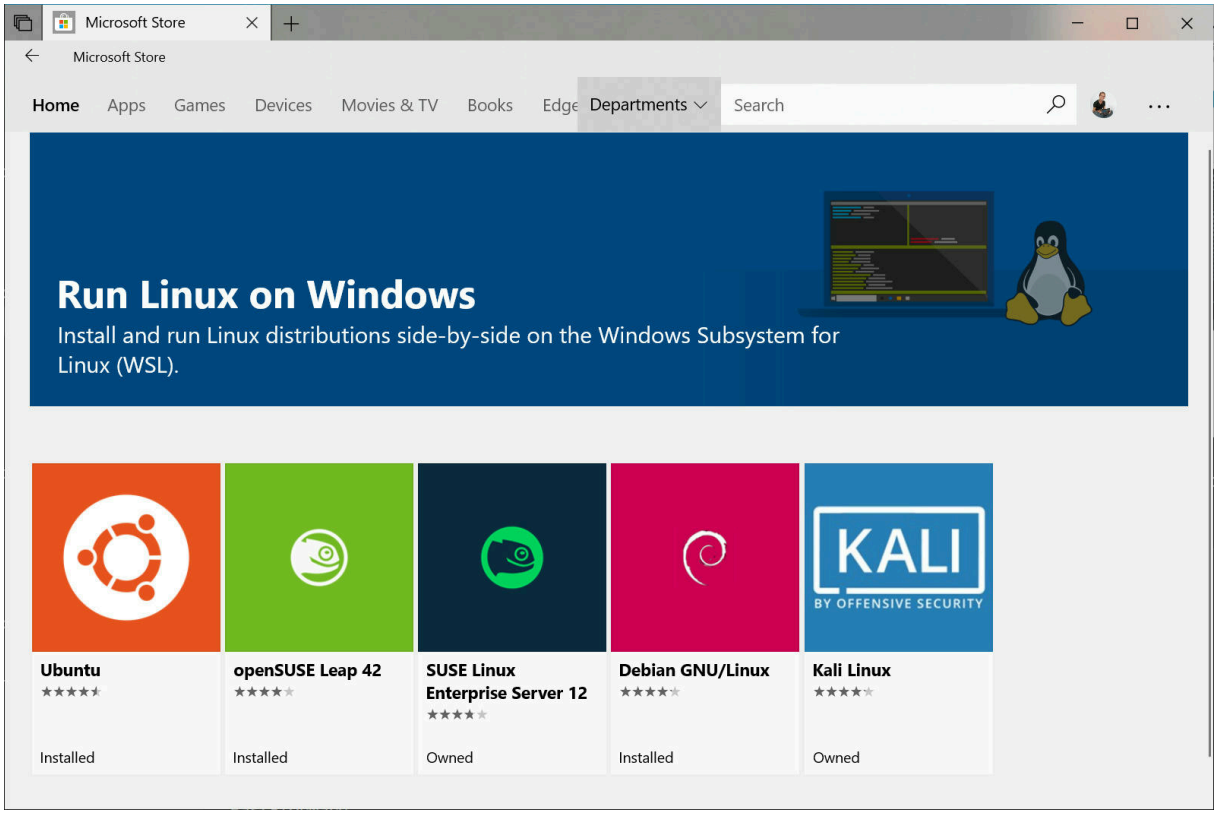
### 设置 WSL 默认版本

绝大部分情况下，建议使用 WSL 2. WSL 1 与 WSL 2 的区别，请见 比较 WSL 2 和 WSL 1[8]

> **关于 systemd**
>
> WSL 1 完全不支持 systemd（这意味着一些需要 systemd 的功能无法实现或需要其他替代方案）. WSL 2 已经内建对 systemd 的支持. 如果需要使用 systemd，而当前运行的发行版没有配置为启用 systemd，可参考 WSL 中的高级设置配置[9].

```
# 将 WSL 默认版本设置为 WSL 2
wsl --set-default-version 2
```

### 安装 WSL 分发版




进入 Microsoft Store，搜索「Ubuntu」，然后选择「Ubuntu」，点击「安装」进行安装. 也可打开 Ubuntu 的商店页面[10].

> **Warning**
>
> Microsoft Store 的 Ubuntu 随着 Ubuntu 的更新而更新，因此内容可能会有所改变. 如果想获取稳定的 Ubuntu 长期支持版，可以在 Microsoft Store 安装 Ubuntu 的 LTS 版本.

由于 Microsoft Store 在国内下载速度不稳定，推荐访问 Ubuntu 官网[11] 获取最新发布的 LTS 版本，或前往 Ubuntu Release[12] 获取先前版本，下载后缀名为 `.wsl` 的文件后双击即可，对于其他版本，也可在 WSL/Distribution.json[13] 中找到对应网址.

[5] WSL-Ubuntu 维基，ubuntu wiki[14]

## 配置分发版 [5]

本章以 Windows 自动安装的 Ubuntu 为例.

### 运行 Ubuntu

打开「开始」菜单找到 Ubuntu 并启动，或使用 `wsl` 命令从 Windows 命令行启动.

可以为 Ubuntu 创建应用程序磁贴或固定至任务栏，以在下次方便地打开.

### 初始化

第一次运行 Ubuntu，需要完成初始化.

```
    Installing, this may take a few minutes...
```

等待一两分钟时间，系统会提示创建新的用户账户.

```
    Please create a default UNIX user account. The username does not need to match your Windows username.
    For more information visit: https://aka.ms/wslusers
    Enter new UNIX username: chtholly
```

输入完用户名以后会提示输入密码. 在 Linux 中，输入密码时屏幕上不显示文字属于正常现象.

```
    Enter new UNIX password:
```

设置好账户名和密码后，WSL 就安装完成了.

```
    Installation successful!
    To run a command as administrator (user "root"), use "sudo <command>".
    See "man sudo_root" for details.

    chtholly@SENIORIOUS:~$
```

## 基础配置

初次安装好的系统不附带任何 C/C++ 编译器，需要手动配置环境.

```
$ gcc
The program 'gcc' is currently not installed. You can install it by typing:
sudo apt install gcc
$ g++
The program 'g++' is currently not installed. You can install it by typing:
sudo apt install g++
```

### 更换为国内软件源

Ubuntu 默认的软件源在国外，可以换成国内镜像以加快速度. 请参考 MirrorZ 的 Ubuntu 帮助页面[15]，选择与自己系统版本相匹配的配置（可使用 `lsb_release -a` 查看 Ubuntu 版本），并按页面给出的命令操作.

> **⚠ 使用与自己系统版本匹配的软件源**
>
> 除非你知道你在做什么，否则不要使用与自己的系统版本不匹配的源!

换源后运行以下命令更新软件:

```
$ sudo su # 执行这行指令后，终端提示符会从 $ 变成 #，执行下文的命令前注意关注提示符
[sudo] xxx 的密码:
```

```
# apt update
# apt upgrade -y
```

### 安装中文环境

```
# apt install language-pack-zh-hans -y
# apt install fontconfig -y
# apt install fonts-noto-cjk fonts-wqy-microhei fonts-wqy-zenhei -y # 中文字体
# dpkg-reconfigure locales
```

此时会进入一个设置菜单，不用管，直接回车.

下一个菜单中选择 `zh_CN.UTF-8` 回车.

```
    Default locale for the system environment:

                None
                C.UTF-8
                en_US.UTF-8
               [zh_CN.UTF-8]

           <Ok>            <Cancel>
```

之后关闭 WSL 并重启，系统就会变成中文.

再依次输入下列命令，把 `man` 帮助页替换为中文. [6]

```
# apt install manpages-zh
# sed -i 's|/usr/share/man|/usr/share/man/zh_CN|g' /etc/manpath.config
```

可以用 `man help` 测试.

### 安装编译环境 [7]

```
# apt install -y build-essential vim ddd gdb fpc emacs gedit anjuta lazarus
```

GUIDE 的安装请参考 Debian 或 Ubuntu 下 GUIDE 的安装→3.2.12@p200.

这里安装的是基础 + NOI 官方要求的环境，如有需要可以用 `sudo apt install <程序名>` 来安装其它软件包. 若想安装其他版本可以参考 Debian 官方的 包管理手册[18].

以下为一个示例程序:

```
$ vim cpuid.cpp
...
$ g++ -Wall cpuid.cpp -o cpuid
$ ./cpuid
AMD Ryzen 5 1400 Quad-Core Processor
```

> **Note**
>
> Linux 环境下可执行文件可不带扩展名，运行方式参见上方命令.

[6] Ubuntu 的 man 命令帮助如何设置中文版，Frank 看庐山，2017-06-09[16]

[7] Run Bash on Ubuntu on Windows, Mike Harsh, 2016-05-30, Windows Blog[17]

# 进阶操作

## 使用 WSLg 运行 GUI 程序

如果你使用 Windows 10 19044 及以上版本或 Windows 11，则可以使用 WSL 2 提供的集成的桌面体验. 该功能允许你直接安装并启动 Linux 桌面程序而无须其他配置.

参见 在适用于 Linux 的 Windows 子系统上运行 Linux GUI 应用[19]

## 安装图形环境，并使用远程桌面连接

如果你使用的版本尚不支持 WSLg, 可以尝试使用以下指南开启图形界面功能.

以下以 Xfce 为例.

如果只想安装 Xfce，可以执行以下命令:

```
$ sudo apt install xfce4 tightvncserver -y
```

如果除 Xfce 外想要更多的软件，可以执行以下命令:

```
$ sudo apt install xubuntu-desktop -y
```

图形环境文件较大，下载解包需要一定时间.

配置 xrdp:

```
$ sudo apt install xrdp -y
$ echo "xfce4-session" >~/.xsession
$ sudo service xrdp restart
```

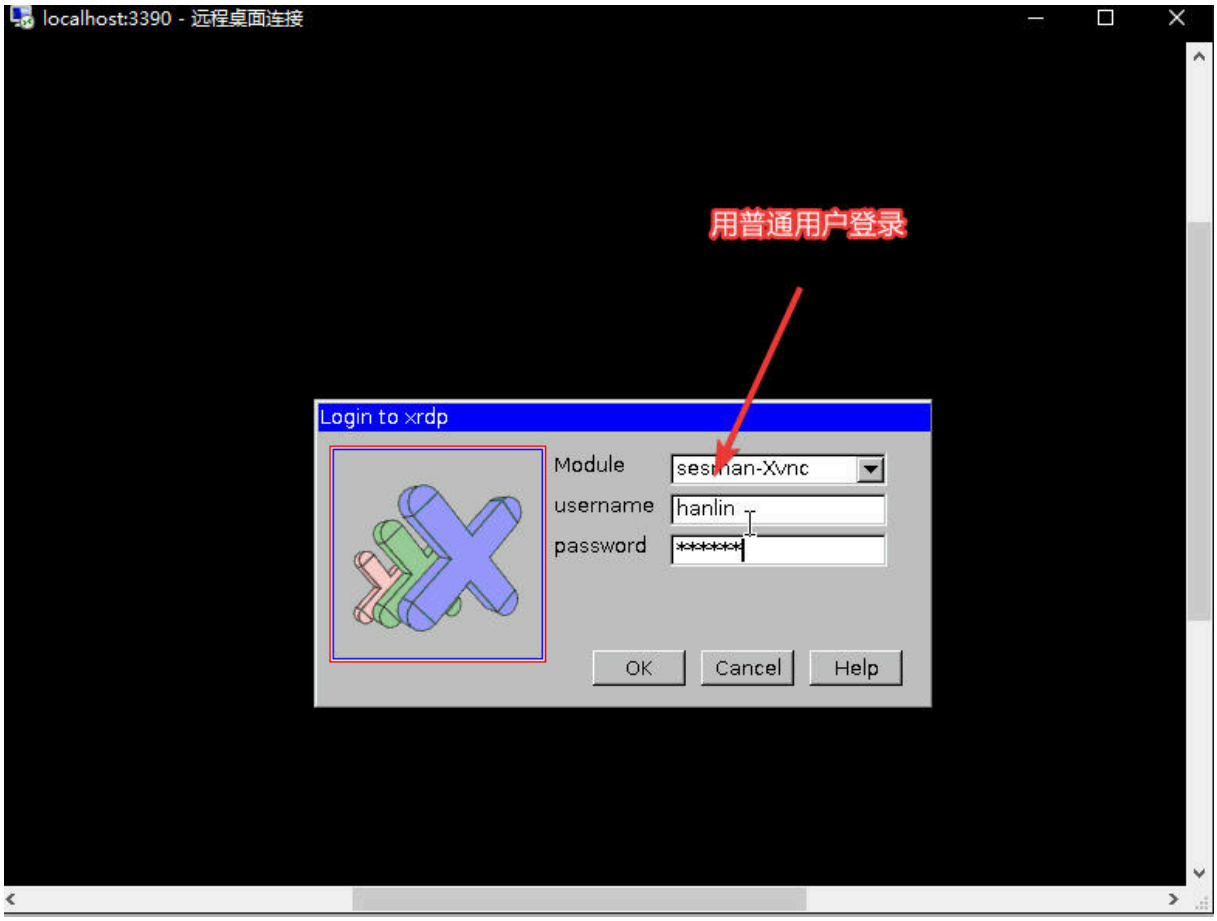
为了防止和计算机原有的远程桌面冲突，需要更换默认端口.

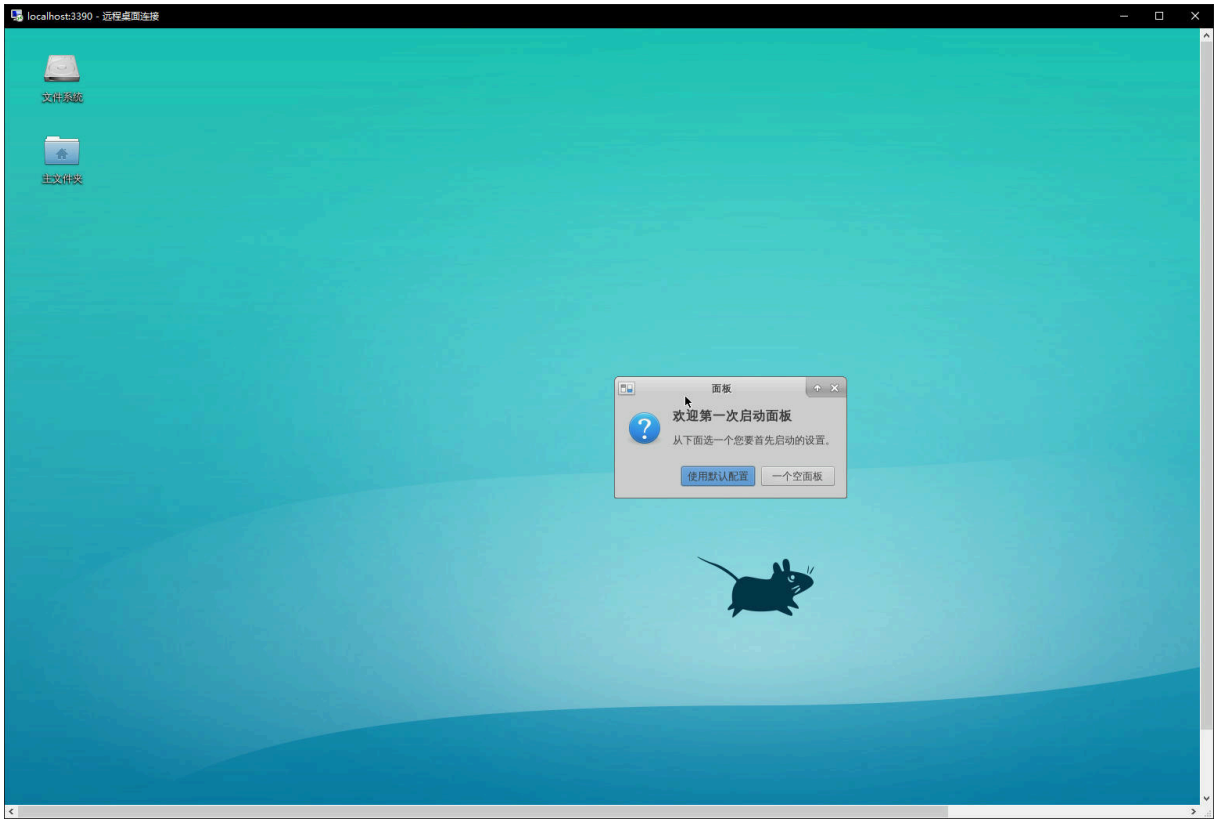



运行命令 `sudo sed -i 's/port=[0-9]\{1,5\}/port=otherport/' /etc/xrdp/xrdp.ini`，其中 `otherport` 为其他端口（如 `3390`）.

```
[globals]
...
port=3390
```

运行 `sudo service xrdp restart`，然后去开始菜单，用 `localhost:otherport` 来访问.


localhost:3390 - 远程桌面连接
用普通用户登录
Login to xrdp
Module
sesman-Xvnc
username
hanlin
password
OK
Cancel
Help


localhost:3390 - 远程桌面连接
面板
欢迎第一次启动面板
从下面选一个您要首先启动的设置。
使用默认配置
一个空面板

### 使用 Xming 连接

进入 Ubuntu 环境，安装 xterm：

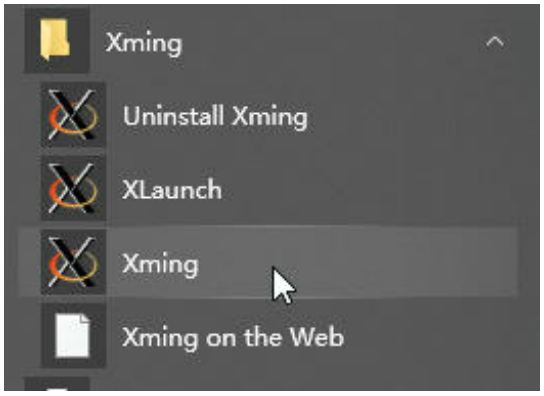
```
# apt install xterm -y
```

退出 Ubuntu.

从 Xming X Server 下载地址[20] 下载最新的 Xming Server，然后安装：

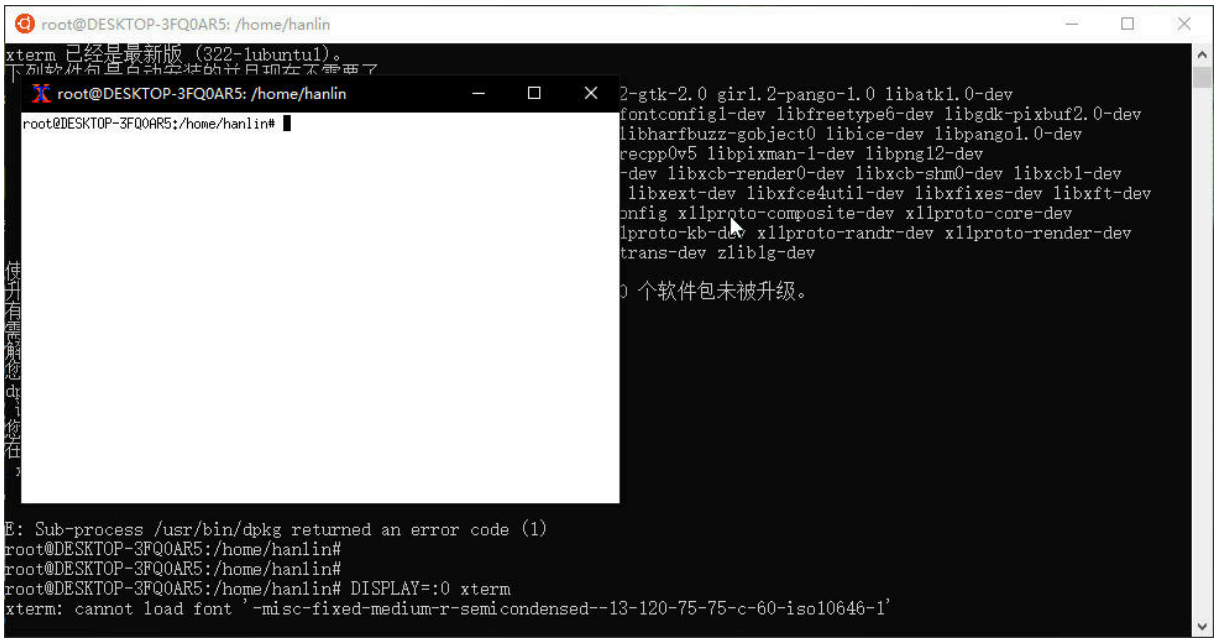



如果安装完后忘记勾选 Launch Xming，需在开始菜单里打开 Xming：



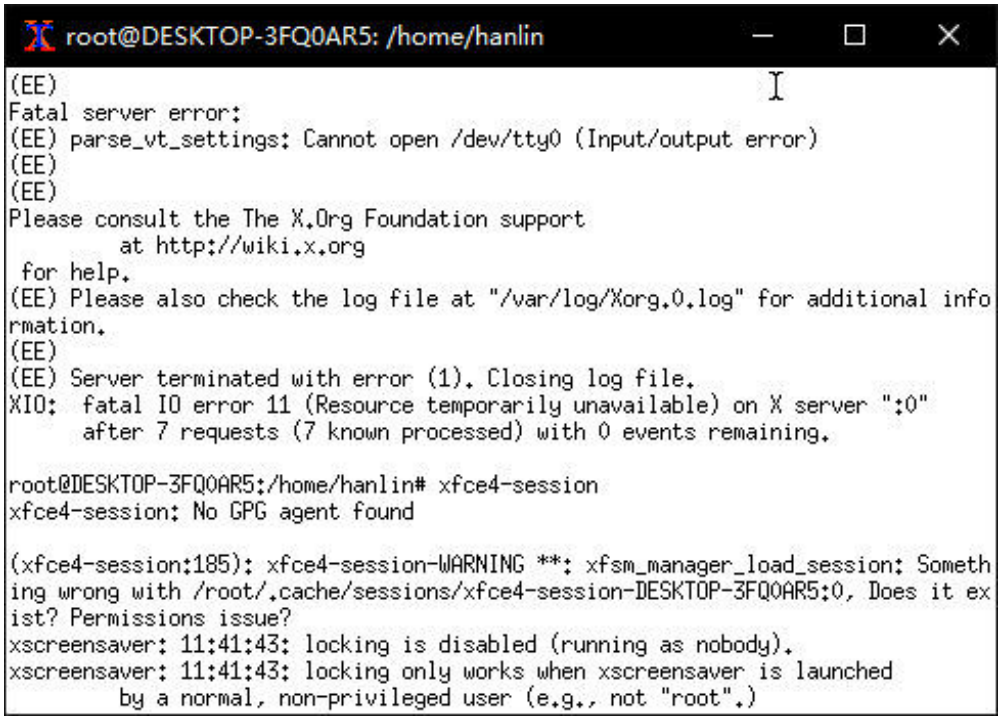

之后再回到 Ubuntu，键入如下指令：

```
$ DISPLAY=:0 xterm
```

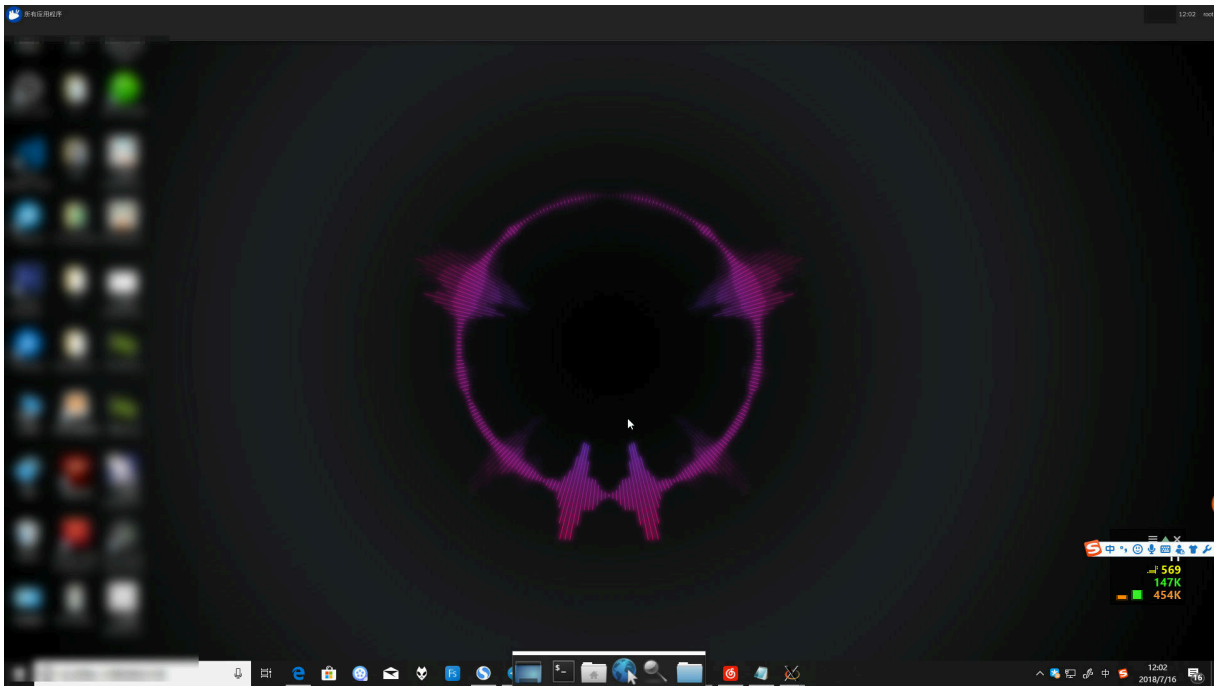



如果使用了 xfce4，可以在弹出的窗口中使用如下命令激活 xfce4：

```
1 $ xfce4-session
```




运行结果如图.（在 Xming 中使用 Ctrl + C 就可以退出该界面.）



## WSL 与 Windows 文件的互访问

Windows 下的硬盘被自动挂载至 Linux 环境下的 `/mnt` 文件夹下. 如 C 盘在 WSL 下的路径为 `/mnt/c`

```
PS C:\Users\chtholly> bash
/mnt/c/Users/chtholly$ echo "Hello world!" > hello
/mnt/c/Users/chtholly$ exit
PS C:\Users\chtholly> cat hello
Hello world!
PS C:\Users\chtholly> echo "Welcome!" > welcome
PS C:\Users\chtholly> bash
/mnt/c/Users/chtholly$ cat welcome
Welcome!
```

另外, 也可以从文件管理器访问 WSL 目录. 在安装 WSL 后, 可以在资源管理器的侧边栏中发现 Linux 项, 在其中可以访问所有安装的发行版中的文件.

同样, 也可以在资源管理器的路径或运行 (Win+R) 中直接输入 `\\wsl$` 来转到 WSL 的目录.

也可以直接使用诸如 `\\wsl$\Ubuntu\home\` 的路径访问其子文件夹.

## 配合 Visual Sudio Code 进行编辑

如果习惯在 Windows 环境下使用 Visual Studio Code→3.2.3@p147 进行代码编辑, 可以安装 VS Code 中的 `Remote - WSL` 插件, 更方便地对 WSL 系统中的文件进行编辑.

通过 `Remote - WSL`, 可以在 Windows 下的 VS Code 界面中直接对 WSL 子系统进行操作, 更加方便地编辑子系统目录下的文件、更方便地使用终端进行调试.

通过在 WSL 中直接键入 `code .`, 可以在该目录下直接唤出 Visual Studio Code, 对于该目录下的文件进行编辑.

同时, 可以通过类似 `code filename` 的命令, 对于指定文件进行编辑.

在插件 `Remote - WSL` 的 Getting Started 页面, 包含对于编辑操作的详细简介.

同时, 也可以参考 Visual Studio Code 的官方文档中关于 WSL 的内容 (Remote development in WSL[21]), 这篇文章包含从 WSL 安装到配合插件使用的全流程的更详细的介绍.

## WSL1 升级为 WSL2

> **Warning**
>
> 请确认已经完成前面 WSL1 的安装步骤.

执行命令 `wsl -l -v` 可以看到 WSL 版本号是 1，需要执行升级，才能到 2.

1. 启用「虚拟机平台」功能

   使用 PowerShell 以管理员身份运行:

   ```
   dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /
   all /norestart
   ```

   然后 **重启电脑**.

2. 下载 Linux 内核更新包
   - x64[22] 的内核更新包.
   - ARM64/AArch64[23] 的内核更新包.
3. 设置分发版版本

   执行命令：`wsl --set-version <分发版名称> <版本号>`

   如：将 Ubuntu 18.04 设置为 WSL 2 的命令为 `wsl --set-version Ubuntu-18.04 2`

   这一步比较耗时，执行完成后通过命令 `wsl -l -v` 来检查升级是否成功.

## FAQ

参见：常见问题[24]，WSL 2 常见问题解答[25]

- 如何在子系统下进行 xxx?

  可以用自带命令行，或者使用图形界面. 比如说 vim，在命令行中键入 `man vim`，会给出一份详尽的使用方法. 亦可使用 `vim --help`.

  关于命令行，可阅读 命令行→3.4@p232

- 对系统资源的占用量?

  这个系统和 Windows 10 共用 Host，所以理论上是比虚拟机占用小的.

## 外部链接

- 关于适用于 Linux 的 Windows 子系统[26]
- Ubuntu 软件仓库 - MirrorZ Help[27]
- Dev on Windows with WSL（在 Windows 上用 WSL 优雅开发）[28]
- GitHub 上的 Awesome-WSL[29]
- 排查适用于 Linux 的 Windows 子系统问题[30]
- WSL1 升级为 WSL2[31]

## 参考资料与注释

[1] 洛谷日报 #6

[2] NOI Linux 2.0 发布，将于 9 月 1 日起正式启用!

[3] 什么是 Linux 子系统（WSL）

[4] 安装 WSL, Microsoft Docs

[5] GitHub 上的 release 页面

[6] 旧版 WSL 的手动安装步骤

[7] 适用于 x64 计算机的 WSL2 Linux 内核更新包

[8] 比较 WSL 2 和 WSL 1

[9] WSL 中的高级设置配置

[10] Ubuntu 的商店页面

[11] Ubuntu 官网

[12] Ubuntu Release

[13] WSL/Distribution.json

[14] WSL-Ubuntu 维基，ubuntu wiki

[15] MirrorZ 的 Ubuntu 帮助页面

[16] Ubuntu 的 man 命令帮助如何设置中文版，Frank 看庐山，2017-06-09

[17] Run Bash on Ubuntu on Windows, Mike Harsh, 2016-05-30, Windows Blog

[18] 包管理手册

[19] 在适用于 Linux 的 Windows 子系统上运行 Linux GUI 应用

[20] Xming X Server 下载地址

[21] Remote development in WSL

[22] x64

[23] ARM64/AArch64

[24] 常见问题

[25] WSL 2 常见问题解答

[26] 关于适用于 Linux 的 Windows 子系统

[27] Ubuntu 软件仓库 - MirrorZ Help

[28] Dev on Windows with WSL（在 Windows 上用 WSL 优雅开发）

[29] GitHub 上的 Awesome-WSL

[30] 排查适用于 Linux 的 Windows 子系统问题

[31] WSL1 升级为 WSL2

# 3.8 Special Judge

**Authors:** Xeonacid, NachtgeistW, 2014CAIS01, sshwy, Chrogeek, Menci, yzy-1

本页面主要介绍部分评测工具/OJ 的 spj 编写方法.

## 简介

**Special Judge**（简称：spj，别名：checker）是当一道题有多组解时，用来判断答案合法性的程序.

**Warning**

spj 还应当判断文件尾是否有多余内容，及输出格式是否正确（如题目要求数字间用一个空格隔开，而选手却使用了换行）. 但是，目前前者只有 Testlib 可以方便地做到这一点，而后者几乎无人去特意进行这种判断.

判断浮点数时应注意 NaN. 不合理的判断方式会导致输出 NaN 即可 AC 的情况.

在对选手文件进行读入操作时应该要检查是否正确读入了所需的内容，防止造成 spj 的运行错误.（部分 OJ 会将 spj 的运行错误作为系统错误处理）

**Note**

以下均以 C++ 作为编程语言，以「要求标准答案与选手答案差值小于 1e-3，文件名为 num，单个测试点满分为 10 分」为例.

## Testlib

参见：Testlib/简介→3.9.1@p269，Testlib/Checker→3.9.6@p278

Testlib 是一个 C++ 的库，用于辅助出题人使用 C++ 编写算法竞赛题.

必须使用 Testlib 作为 spj 的评测工具/OJ：Codeforces、洛谷、UOJ 等.

可以使用 Testlib 作为 spj 的评测工具/OJ：LibreOJ (Lyrio[1])、Lemon、牛客网等.

SYZOJ 2 所需的修改版 Testlib 托管于 pastebin[2][1]，但此修改版并未修改交互模式. syzoj/testlib[4] 处托管了一份可以在 SYZOJ 2 上使用交互模式的 Testlib.

Lemon 所需的修改版 Testlib 托管于 GitHub - GitPinkRabbit/Testlib-for-Lemons[5]. 注意此版本 Testlib 注册 checker 时应使用 `registerLemonChecker()`，而非 `registerTestlibCmd()`. 此版本继承自 matthew99 的旧版[6]，添加了一些 Testlib 的新功能. 如果你使用 LemonLime，则可以使用原生的 Testlib.

DOMJudge 所需的修改版 Testlib 托管于 cn-xcpc-tools/testlib-for-domjudge[7]. 此版本 Testlib 同时可作为 Special Judge 的 checker 和交互题的 interactor.

Arbiter 所需的修改版 Testlib 托管于 testlib-for-arbiter[8].

其他评测工具/OJ 大部分需要按照其 spj 编写格式修改 Testlib，并将 testlib.h 与 spj 一同上传；或将 testlib.h 置于 include 目录.

```cpp
#include "testlib.h"
//
#include <cmath>

int main(int argc, char *argv[]) {
  /*
   * inf: 输入
   * ouf: 选手输出
   * ans: 标准输出
   */
  registerTestlibCmd(argc, argv);

  double pans = ouf.readDouble(), jans = ans.readDouble();

  if (abs(pans - jans) < 1e-3)
    quitf(_ok, "Good job\n");
  else
    quitf(_wa, "Too big or too small, expected %f, found %f\n", jans, pans);
}
```

## Lemon

> **Note**
>
> Lemon 有现成的修改版 Testlib→3@p137，建议使用 Testlib.
>
> LemonLime 最新版已经支持使用原版 Testlib 编写评测器，如果你使用 LemonLime，建议使用 Testlib.

```cpp
#include <cmath>
#include <cstdio>

int main(int argc, char* argv[]) {
  /*
   * argv[1]: 输入
```

[1] LibreOJ 支持 testlib 检查器啦！[3]

```
    * argv[2]: 选手输出
    * argv[3]: 标准输出
    * argv[4]: 单个测试点分值
    * argv[5]: 输出最终得分 (0 ~ argv[4])
    * argv[6]: 输出错误报告
    */
  FILE* fin = fopen(argv[1], "r");
  FILE* fout = fopen(argv[2], "r");
  FILE* fstd = fopen(argv[3], "r");
  FILE* fscore = fopen(argv[5], "w");
  FILE* freport = fopen(argv[6], "w");

  double pans, jans;
  fscanf(fout, "%lf", &pans);
  fscanf(fstd, "%lf", &jans);

  if (abs(pans - jans) < 1e-3) {
    fprintf(fscore, "%s", argv[4]);
    fprintf(freport, "Good job\n");
  } else {
    fprintf(fscore, "%d", 0);
    fprintf(freport, "Too big or too small, expected %f, found %f\n", jans,
            pans);
  }
}
```

## Cena

```
#include <cmath>
#include <cstdio>

int main(int argc, char* argv[]) {
  /*
    * FILENAME.in: 输入
    * FILENAME.out: 选手输出
    * argv[1]: 单个测试点分值
    * argv[2]: 标准输出
    * score.log: 输出最终得分 (0 ~ argv[1])
    * report.log: 输出错误报告
    */
  FILE* fin = fopen("num.in", "r");
  FILE* fout = fopen("num.out", "r");
  FILE* fstd = fopen(argv[2], "r");
  FILE* fscore = fopen("score.log", "w");
  FILE* freport = fopen("report.log", "w");

  double pans, jans;
  fscanf(fout, "%lf", &pans);
  fscanf(fstd, "%lf", &jans);

  if (abs(pans - jans) < 1e-3) {
    fprintf(fscore, "%s", argv[1]);
    fprintf(freport, "Good job\n");
  } else {
```

```
    fprintf(fscore, "%d", 0);
    fprintf(freport, "Too big or too small, expected %f, found %f\n", jans,
            pans);
  }
}
```

## CCR

```cpp
#include <cmath>
#include <cstdio>

int main(int argc, char* argv[]) {
  /*
   * stdin: 输入
   * argv[2]: 标准输出
   * argv[3]: 选手输出
   * stdout:L1: 输出最终得分比率 (0 ~ 1)
   * stdout:L2: 输出错误报告
   */
  FILE* fout = fopen(argv[3], "r");
  FILE* fstd = fopen(argv[2], "r");

  double pans, jans;
  fscanf(fout, "%lf", &pans);
  fscanf(fstd, "%lf", &jans);

  if (abs(pans - jans) < 1e-3) {
    printf("%d\n", 1);
    printf("Good job\n");
  } else {
    printf("%d\n", 0);
    printf("Too big or too small, expected %f, found %f\n", jans, pans);
  }
}
```

## Arbiter

```cpp
#include <cmath>
#include <cstdio>

int main(int argc, char* argv[]) {
  /*
   * argv[1]: 输入
   * argv[2]: 选手输出
   * argv[3]: 标准输出
   * /tmp/_eval.score:L1: 输出错误报告
   * /tmp/_eval.score:L2: 输出最终得分
   */
  FILE* fout = fopen(argv[2], "r");
  FILE* fstd = fopen(argv[3], "r");
  FILE* fscore = fopen("/tmp/_eval.score", "w");

  double pans, jans;
```

```
  fscanf(fout, "%lf", &pans);
  fscanf(fstd, "%lf", &jans);

  if (abs(pans - jans) < 1e-3) {
    fprintf(fscore, "Good job\n");
    fprintf(fscore, "%d", 10);
  } else {
    fprintf(fscore, "Too big or too small, expected %f, found %f\n", jans,
            pans);
    fprintf(fscore, "%d", 0);
  }
}
```

## HUSTOJ

```
#include <cmath>
#include <cstdio>

#define AC 0
#define WA 1

int main(int argc, char* argv[]) {
  /*
   * argv[1]: 输入
   * argv[2]: 标准输出
   * argv[3]: 选手输出
   * exit code: 返回判断结果
   */
  FILE* fin = fopen(argv[1], "r");
  FILE* fout = fopen(argv[3], "r");
  FILE* fstd = fopen(argv[2], "r");

  double pans, jans;
  fscanf(fout, "%lf", &pans);
  fscanf(fstd, "%lf", &jans);

  if (abs(pans - jans) < 1e-3)
    return AC;
  else
    return WA;
}
```

## QDUOJ

相较之下，QDUOJ 略为麻烦．它带 spj 的题目没有标准输出，只能把 std 写进 spj，待跑出标准输出后再判断．

```
#include <cmath>
#include <cstdio>

#define AC 0
#define WA 1
#define ERROR -1

```

```
double solve(...) {
  // std
}

int main(int argc, char* argv[]) {
  /*
   * argv[1]: 输入
   * argv[2]: 选手输出
   * exit code: 返回判断结果
   */
  FILE* fin = fopen(argv[1], "r");
  FILE* fout = fopen(argv[2], "r");

  double pans, jans;
  fscanf(fout, "%lf", &pans);

  jans = solve(...);
  if (abs(pans - jans) < 1e-3)
    return AC;
  else
    return WA;
}
```

## HDOJ

HDOJ 和 QDUOJ 的情况基本一致，也需要在 spj 中实现 std 后与选手输出比较．但与 QDUOJ 不同的是，HDOJ 会比较答案与 spj 输出在标准输出的内容后给出最终结果．因此，上传输出时仅需上传 spj 在正确时的输出即可．

HDOJ 需上传 Windows 下编译后的二进制文件，而非源代码．

```
#include <cmath>
#include <cstdio>

double solve(FILE* fin) {
  // std, read input from fin
}

int main(int argc, char* argv[]) {
  /*
   * argv[1]: 输入
   * stdin: 选手输出
   */
  FILE* fin = fopen(argv[1], "r");

  double pans, jans;
  if (scanf("%lf", &pans) != 1) {
    printf("WA\n");
    goto finish;
  }

  jans = solve(fin);
  if (abs(pans - jans) < 1e-3)
    printf("AC\n");
```

```
  else
    printf("WA\n");

finish:
  fclose(fin);
  return 0;
}
```

对应的答案文件为：

```
AC
```

## SYZOJ 2

> **Note**
>
> SYZOJ 2 有现成的修改版 Testlib→3@p137，建议使用 Testlib.
>
> LibreOJ 的最新版本已不再基于 SYZOJ，而是基于 Lyrio[9]. Lyrio 支持使用原版 Testlib 编写评测器，这也是更加通用且推荐的做法.

```
#include <cmath>
#include <cstdio>

int main(int argc, char* argv[]) {
  /*
   * in: 输入
   * user_out: 选手输出
   * answer: 标准输出
   * code: 选手代码
   * stdout: 输出最终得分 (0 ~ 100)
   * stderr: 输出错误报告
   */
  FILE* fin = fopen("input", "r");
  FILE* fout = fopen("user_out", "r");
  FILE* fstd = fopen("answer", "r");
  FILE* fcode = fopen("code", "r");

  double pans, jans;
  fscanf(fout, "%lf", &pans);
  fscanf(fstd, "%lf", &jans);

  if (abs(pans - jans) < 1e-3) {
    printf("%d", 100);
    fprintf(stderr, "Good job\n");
  } else {
    printf("%d", 0);
    fprintf(stderr, "Too big or too small, expected %f, found %f\n", jans,
            pans);
  }
}
```

## 牛客网

> **Note**
>
> 牛客网有现成的修改版 Testlib→3@p137，建议使用 Testlib.

参见：如何在牛客网出 Special Judge 的编程题[10]

```cpp
#include <cmath>
#include <cstdio>

#define AC 0
#define WA 1

int main(int argc, char* argv[]) {
  /*
   * input: 输入
   * user_output: 选手输出
   * output: 标准输出
   * exit code: 返回判断结果
   */
  FILE* fin = fopen("input", "r");
  FILE* fout = fopen("user_output", "r");
  FILE* fstd = fopen("output", "r");

  double pans, jans;
  fscanf(fout, "%lf", &pans);
  fscanf(fstd, "%lf", &jans);

  if (abs(pans - jans) < 1e-3)
    return AC;
  else
    return WA;
}
```

## DOMJudge

> **Note**
>
> DOMJudge 支持任何语言编写的 spj，参见：problemarchive.org output validator 格式[11].
>
> DOMJudge 有现成的修改版 Testlib→3@p137，建议使用 Testlib.

DOMJudge 使用的 Testlib 及导入 Polygon 题目包方式的文档：https://github.com/cn-xcpc-tools/testlib-for-domjudge[12]

DOMJudge 的 默认比较器[13] 自带了浮点数带精度比较，只需要在题目配置的 `validator_flags` 中添加 `float_tolerance 1e-3` 即可.

```cpp
#include <cmath>
#include <cstdio>

#define AC 42
#define WA 43
char reportfile[50];
```

```
7
8  int main(int argc, char* argv[]) {
9    /*
10    * argv[1]: 输入
11    * argv[2]: 标准输出
12    * argv[3]: 评测信息输出的文件夹
13    * stdin: 选手输出
14    */
15   FILE* fin = fopen(argv[1], "r");
16   FILE* fstd = fopen(argv[2], "r");
17   sprintf(reportfile, "%s/judgemessage.txt", argv[3]);
18   FILE* freport = fopen(reportfile, "w");
19
20   double pans, jans;
21   scanf("%lf", &pans);
22   fscanf(fstd, "%lf", &jans);
23
24   if (abs(pans - jans) < 1e-3) {
25     fprintf(freport, "Good job\n");
26     return AC;
27   } else {
28     fprintf(freport, "Too big or too small, expected %f, found %f\n", jans,
29             pans);
30     return WA;
31   }
32 }
```

也可以使用 Kattis Problem Tools 提供的头文件 validate.h[14] 编写，以实现更加复杂的功能.

## 参考资料


[1] Lyrio

[2] pastebin

[3] LibreOJ 支持 testlib 检查器啦!

[4] syzoj/testlib

[5] GitHub - GitPinkRabbit/Testlib-for-Lemons

[6] matthew99 的旧版

[7] cn-xcpc-tools/testlib-for-domjudge

[8] testlib-for-arbiter

[9] Lyrio

[10] 如何在牛客网出 Special Judge 的编程题

[11] problemarchive.org output validator 格式

[12] https://github.com/cn-xcpc-tools/testlib-for-domjudge

[13] 默认比较器

[14] validate.h


# 3.9 Testlib

## 3.9.1 Testlib 简介

**Authors:** Xeonacid, sshwy

如果你正在使用 C++ 出一道算法竞赛题目，Testlib 是编写相关程序（generator, validator, checker, interactor）时的优秀辅助工具. 它是俄罗斯和其他一些国家的出题人的必备工具，许多比赛也都在用它: ROI、ICPC 区域赛、所有 Codeforces round.....

Testlib 库仅有 `testlib.h` 一个文件，使用时仅需在所编写的程序开头添加 `#include "testlib.h"` 即可.

Testlib 的具体用途:

- 编写 Generator→3.9.3@p273，即数据生成器.
- 编写 Validator→3.9.4@p276，即数据校验器，判断生成数据是否符合题目要求，如数据范围、格式等.
- 编写 Interactor→3.9.5@p277，即交互器，用于交互题.
- 编写 Checker→3.9.6@p278，即 Special Judge→3.8@p261.

Testlib 与 Codeforces 开发的 Polygon[1] 出题平台完全兼容.

`testlib.h` 在 2005 年移植自 `testlib.pas`，并一直在更新. Testlib 与绝大多数编译器兼容，如 VC++ 和 GCC g++，并兼容 C++11.

**本文主要翻译自 Testlib - Codeforces[2]. `testlib.h` 的 GitHub 存储库为 MikeMirzayanov/testlib[3].**

### 外部参考资料

[1] Polygon  [2] Testlib - Codeforces  [3] MikeMirzayanov/testlib 

## 3.9.2 通用

本页面介绍 Testlib checker/interactor/validator 的一些通用状态/对象/函数、一些用法及注意事项. 请在阅读其他页面前完整阅读本页面的内容.

### 通用状态

| 结果 | Testlib 别名 | 含义 |
|---|---|---|
| Ok | `_ok` | 答案正确. |
| Wrong Answer | `_wa` | 答案错误. |
| Presentation Error | `_pe` | 答案格式错误. 注意包括 Codeforces 在内的许多 OJ 并不区分 PE 和 WA. |
| Partially Correct | `_pc(score)` | 答案部分正确. 仅限于有部分分的测试点，其中 `score` 为一个正整数，从 0（没分）到 100（可能的最大分数）.（`quitf+_pc` 只是为了兼容旧的 pascal-testlib，如果想要输出部分分，建议使用 `quitp`[1]） |
| Fail | `_fail` | validator 中表示输入不合法，不通过校验. checker 中表示程序内部错误、标准输出有误或选手输出比标准输出更优，需要裁判/出题人关注.（也就是题目锅了） |

通常用程序的返回值表明结果，但是也有一些其他方法：创建一个输出 xml 文件、输出信息到 stdout 或其他位置……这些都通过下方函数表中的 `quitf` 函数来完成.

### 通用对象

| 对象 | 含义 |
|---|---|
| `inf` | 输入文件流 |
| `ouf` | 选手输出流 |
| `ans` | 参考输出流 |

### 通用函数

非成员函数:

| 调用 | 含义 |
|---|---|

[1]issue 链接[1]

| | |
|---|---|
| `void registerTestlibCmd(int argc, char* argv[])` | 注册程序为 checker |
| `void registerInteraction(int argc, char* argv[])` | 注册程序为 interactor |
| `void registerValidation()/void registerValidation(int argc, char* argv[])` | 注册程序为 validator |
| `void registerGen(int argc, char* argv[], int randomGeneratorVersion)` | 注册程序为 generator`randomGeneratorVersion` 推荐为 `1` |
| `void quit(TResult verdict, string message)/void quitf(TResult verdict, string message, ...)` | 结束程序，返回 `verdict`，输出 `message` |
| `void quitif(bool condition, TResult verdict, string message, ...)` | 如果 `condition` 成立，调用 `quitf(verdict, message, ...)` |
| `void quitp(F points, string message, ...)` | 结束程序，返回部分分．大部分 OJ（如洛谷、UOJ）的 `points` 需要提供一个 $[0,1]$ 内的实数，表示得分百分比，还有部分 OJ（如 Lyrio）的 `points` 需要提供一个 $[0,100]$ 的实数（OJ 会自动舍弃小数部分），表示百分制下的测试点得分 |

流成员函数：

| 调用 | 含义 |
|---|---|
| `char readChar()` | 读入一个字符 |
| `char readChar(char c)` | 读入一个字符，必须为 `c` |
| `char readSpace()` | 等同于 `readChar(' ')` |
| `string readToken()/string readWord()` | 读入一个串，到空白字符（空格、Tab、EOLN 等）停止 |
| `string readToken(string regex)/string readWord(string regex)` | 读入一个串，必须与 `regex` 匹配 |
| `long long readLong()` | 读入一个 64 位整数 |
| `long long readLong(long long L, long long R)` | 读入一个 64 位整数，必须在 $[L,R]$ 之间 |
| `vector<long long> readLongs(int n, long long L, long long R)` | 读入 $N$ 个 64 位整数，必须均在 $[L,R]$ 之间 |
| `int readInt()/int readInteger()` | 读入一个 32 位整数 |
| `int readInt(int L, int R)/int readInteger(L, R)` | 读入一个 32 位整数，必须在 $[L,R]$ 之间 |
| `vector<int> readInts(int n, int L, int R)/ vector<int> readIntegers(int n, int L, int R)` | 读入 $N$ 个 32 位整数，必须均在 $[L,R]$ 之间 |
| `double readReal()/double readDouble()` | 读入一个双精度浮点数 |
| `double readReal(double L, double R)/double readDouble(double L, double R)` | 读入一个双精度浮点数，必须在 $[L,R]$ 之间 |
| `double readStrictReal(double L, double R, int minPrecision, int maxPrecision)/double readStrictDouble(double L, double R, int minPrecision, int maxPrecision)` | 读入一个双精度浮点数，必须在 $[L,R]$ 之间，小数位数必须在 $[minPrecision, maxPrecision]$ 之间，不得使用指数计数法等非正常格式 |
| `string readString()/string readLine()` | 读入一行（包括换行符），同时将流指针指向下一行的开头 |
| `string readString(string regex)/string readLine(string regex)` | 读入一行，必须与 `regex` 匹配 |
| `void readEoln()` | 读入 EOLN（在 Linux 环境下读入 `LF`，在 Windows 环境下读入 `CR LF`） |

| | |
|---|---|
| `void readEof()` | 读入 EOF |
| `void quit(TResult verdict, string message)/void quitf(TResult verdict, string message, ...)` | 结束程序，若 `Stream` 为 `ouf` 返回 `verdict`，否则返回 `_fail`；输出 `message` |
| `void quitif(bool condition, TResult verdict, string message, ...)` | 如果 `condition` 成立，调用 `quitf(verdict, message, ...)` |

未完待续……

## 极简正则表达式

上面的输入函数中的一部分允许使用「极简正则表达式」特性，如下所示：

- 字符集．如 `[a-z]` 表示所有小写英文字母，`[^a-z]` 表示除小写英文字母外任何字符．
- 范围．如 `[a-z]{1,5}` 表示一个长度在 $[1,5]$ 范围内且只包含小写英文字母的串．
- 「或」标识符．如 `mike|john` 表示 `mike` 或 `john` 其一．
- 「可选」标识符．如 `-?[1-9][0-9]{0,3}` 表示 $[-9999,9999]$ 范围内的非零整数（注意那个可选的负号）．
- 「重复」标识符．如 `[0-9]*` 表示零个或更多数字，`[0-9]+` 表示一个或更多数字．
- 注意这里的正则表达式是「贪婪」的（「重复」会尽可能匹配）．如 `[0-9]?1` 将不会匹配 `1`（因为 `[0-9]?` 将 `1` 匹配上，导致模板串剩余的那个 `1` 无法匹配）．

## 首先 include testlib.h

请确保 testlib.h 是你 include 的 **第一个** 头文件，Testlib 会重写/禁用（通过名字冲突的方式）一些与随机有关的函数（如 `random()`），保证随机结果与环境无关，这对于 generator 非常重要，generator 页面 →3.9.3@p273 会详细说明这一点．

## 使用项别名

推荐给 `readInt/readInteger/readLong/readDouble/readWord/readToken/readString/readLine` 等的有限制调用最后多传入一个 `string` 参数，即当前读入的项的别名，使报错易读．例如使用 `inf.readInt(1, 100, "n")` 而非 `inf.readInt(1, 100)`，报错信息将为 `FAIL Integer parameter [name=n] equals to 0, violates the range [1, 100]`．

## 使用 `ensuref/ensure()`

这两个函数用于检查条件是否成立（类似于 `assert()`）．例如检查 $x_i \neq y_i$，我们可以使用

```
ensuref(x[i] != y[i], "Graph can't contain loops");
```

还可以使用 C 风格占位符如

```
ensuref(s.length() % 2 == 0,
        "String 's' should have even length, but s.length()=%d",
        int(s.length()));
```

它有一个简化版 `ensure()`，我们可以直接使用 `ensure(x> y)` 而不添加说明内容（也不支持添加说明内容），如果条件不满足报错将为 `FAIL Condition failed: "x > y"`．很多情况下不加额外的说明的这种报错很不友好，所以我们通常使用 `ensuref()` 并加以说明内容，而非使用 `ensure()`．

> ⚠ **Warning**
>
> 注意全局与成员 `ensuref/ensure()` 的区别
>
> 全局函数 `::ensuref/ensure()` 多用于 generator 和 validator 中，如果检查失败将统一返回 `_fail`．

> 成员函数 `InStream::ensuref/ensure()` 一般用于判断选手和参考程序的输出是否合法．当 `InStream` 为 `ouf` 时，返回 `_wa`；为 `inf`（一般不在 checker 中检查输入数据，这应当在 validator 中完成）或 `ans` 时，返回 `_fail`．详见 Checker - 编写 readAns 函数→3.9.6@p278．

**本文主要翻译并综合自 Testlib - Codeforces[2] 系列．`testlib.h` 的 GitHub 存储库为 MikeMirzayanov/testlib[3]．**

[1] issue 链接 [2] Testlib - Codeforces [3] MikeMirzayanov/testlib

### 3.9.3 Generator

Generator，即数据生成器．当数据很大，手造会累死的时候，我们就需要它来帮助我们自动造数据．

#### 简单的例子

生成两个 $[1, n]$ 范围内的整数：

```cpp
// clang-format off

#include "testlib.h"
#include <iostream>

using namespace std;

int main(int argc, char* argv[]) {
  registerGen(argc, argv, 1);
  int n = atoi(argv[1]);
  cout << rnd.next(1, n) << " ";
  cout << rnd.next(1, n) << endl;
}
```

#### 为什么要使用 Testlib?

有人说写 generator 不需要用 Testlib，它在这没什么用．实际上这是个不正确的想法．一个好的 generator 应该满足这一点：**在任何环境下对于相同输入它给出相同输出**．写 generator 就避免不了生成随机值，平时我们用的 `rand()` 或 C++11 的 `mt19937/uniform_int_distribution`，当操作系统不同、使用不同编译器编译、不同时间运行等，它们的输出都可能不同（对于非常常用的 `srand(time(nullptr))`，这是显然的），而这就会给生成数据带来不确定性．

需要注意的是，一旦使用了 Testlib，就不能再使用标准库中的 `srand()`，`rand()` 等随机数函数，否则在编译时会报错．因此，**请确保所有与随机相关的函数均使用 Testlib 而非标准库提供的．**

而 Testlib 中的随机值生成函数则保证了相同调用会输出相同值，与 generator 本身或平台均无关．另外，它给生成各种要求的随机值提供了很大便利，如 `rnd.next("[a-z]{1,10}")` 会生成一个长度在 $[1, 10]$ 范围内的串，每个字符为 `a` 到 `z`，很方便吧！

#### Testlib 能做什么?

在一切之前，先执行 `registerGen(argc, argv, 1)` 初始化 Testlib（其中 `1` 是使用的 generator 版本，通常保持不变），然后我们就可以使用 `rnd` 对象来生成随机值．随机数种子取自命令行参数的哈希值，对于某 generator `g.cpp`，`g 100`(Unix-Like) 和 `g.exe "100"`(Windows) 将会有相同的输出，而 `g 100 0` 则与它们不同．

`rnd` 对象的类型为 `random_t`，你可以建立一个新的随机值生成对象，不过通常你不需要这么做．

该对象有许多有用的成员函数，下面是一些例子：

| 调用 | 含义 |
| --- | --- |
| `rnd.next(4)` | 等概率生成一个 $[0,4)$ 范围内的整数 |
| `rnd.next(4, 100)` | 等概率生成一个 $[4,100]$ 范围内的整数 |
| `rnd.next(10.0)` | 等概率生成一个 $[0,10.0)$ 范围内的浮点数 |
| rnd.next("one \| two \| three") | 等概率从 `one`,`two`,`three` 三个串中返回一个 |
| `rnd.wnext(4, t)` | `wnext()` 是一个生成不等分布（具有偏移期望）的函数[2]，$t$ 表示调用 `next()` 的次数，并取生成值的最大值．例如 `rnd.wnext(3, 1)` 等同于 `max({rnd.next(3), rnd.next(3)})`；`rnd.wnext(4, 2)` 等同于 `max({rnd.next(4), rnd.next(4), rnd.next(4)})`．如果 $t<0$，则为调用 $-t$ 次，取最小值；如果 $t=0$，等同于 `next()`． |
| `rnd.any(container)` | 等概率返回一个具有随机访问迭代器（如 `std::vector` 和 `std::string`）的容器内的某一元素的引用 |

附：关于 `rnd.wnext(i,t)` 的形式化定义：

$$\mathrm{wnext}(i,t)=\begin{cases}\mathrm{next}(i) & t=0\\ \max(\mathrm{next}(i),\mathrm{wnext}(i,t-1)) & t>0\\ \min(\mathrm{next}(i),\mathrm{wnext}(i,t+1)) & t<0\end{cases}$$

另外，不要使用 `std::random_shuffle()`，请使用 Testlib 中的 `shuffle()`，它同样接受一对迭代器．它使用 `rnd` 来打乱序列，即满足如上「好的 generator」的要求．

## 示例：生成一棵树

下面是生成一棵树的主要代码，它接受两个参数——顶点数和伸展度．例如，当 $n=10,t=1000$ 时，可能会生成链；当 $n=10,t=-1000$ 时，可能会生成菊花．

```
#define forn(i, n) for (int i = 0; i < int(n); i++)

registerGen(argc, argv, 1);

int n = atoi(argv[1]);
int t = atoi(argv[2]);

vector<int> p(n);

/* 为节点 1..n-1 设置父亲 */
forn(i, n) if (i > 0) p[i] = rnd.wnext(i, t);

printf("%d\n", n);

/* 打乱节点 1..n-1 */
vector<int> perm(n);
forn(i, n) perm[i] = i;
shuffle(perm.begin() + 1, perm.end());

/* 根据打乱的节点顺序加边 */
vector<pair<int, int>> edges;
for (int i = 1; i < n; i++)
  if (rnd.next(2))
    edges.push_back(make_pair(perm[i], perm[p[i]]));
  else
```

[2]事实上，当 `i` 为浮点数时，`rnd.wnext(i, t)` 服从 $[0,i)$ 上的 Beta 分布[1]：当 $t>0$ 时，服从 $i\cdot\mathrm{Beta}(t+1,1)$；当 $t<0$ 时，服从 $i\cdot\mathrm{Beta}(1,t+1)$．

```
26      edges.push_back(make_pair(perm[p[i]], perm[i]));
27
28  /* 打乱边 */
29  shuffle(edges.begin(), edges.end());
30
31  for (int i = 0; i + 1 < n; i++)
32    printf("%d %d\n", edges[i].first + 1, edges[i].second + 1);
```

### 一次性生成多组数据

跟不使用 Testlib 编写的时候一样，每次输出前重定向输出流就好，不过 Testlib 提供了一个辅助函数 `startTest(test_index)`，它帮助你将输出流重定向到 `test_index` 文件.

### 一些注意事项

- 严格遵循题目的格式要求，如空格和换行，注意文件的末尾应有一个换行.
- 对于大数据首选 `printf` 而非 `cout`，以提高性能.（不建议在使用 Testlib 时关闭流同步）
- 不使用 UB（Undefined Behavior，未定义行为），如本文开头的那个示例，输出如果写成 `cout << rnd.next(1, n) << " " << rnd.next(1, n) << endl;`，则 `rnd.next()` 的调用顺序没有定义.

### 新特性：解析命令行参数

在之前，我们通常使用类似 `int n = atoi(argv[3]);` 的代码，但是这样并不好. 有以下几点原因:

- 不存在第三个命令行参数的时候是不安全的;
- 第三个命令行参数可能不是有效的 32 位整数.

现在，你可以这样写：`int n = opt<int>(3)`. 与此同时，你也可以使用 `int64_t m = opt<int64_t>(1);`，`bool t = opt<bool>(2);` 和 `string s = opt(4);` 等.

另外，testlib 同时也支持命名参数. 如果有很多参数，这样 `g 10 20000 a true` 的可读性就会比 `g -n10 -m200000 -t=a -increment` 差.

在这种情况下，现在你可以在 generator 中使用以下代码:

```
1  int n = opt<int>("n");
2  long long n = opt<long long>("m");
3  string t = opt("t");
4  bool increment = opt<bool>("increment");
```

你可以自由地混合使用按下标和按名称读取参数的方式.

支持的用于编写命名参数的方案有以下几种:

- `--key=value` 或 `-key=value`;
- `--key value` 或 `-key value`——如果 `value` 不是新参数的开头（不以连字符 `-` 开头或一个/两个连字符后没有跟随字母）;
- `--k12345` 或 `-k12345`——如果 key k 是一个字母，且后面是一个数字;
- `-prop` 或 `--prop`——启用 bool 属性.

下面是一些例子:

```
1  g1 -n1
2  g2 --len=4 --s=oops
3  g3 -inc -shuffle -n=5
4  g4 --length 5 --total 21 -ord
```

### 更多示例

可以在 GitHub[2] 中找到.

**本文主要翻译自 Генераторы на testlib.h - Codeforces[3]．新特性翻译自 Testlib: Opts—parsing command line options[4]．`testlib.h` 的 GitHub 存储库为 MikeMirzayanov/testlib[5]．**

[1] Beta 分布

[2] GitHub

[3] Генераторы на testlib.h - Codeforces

[4] Testlib: Opts—parsing command line options

[5] MikeMirzayanov/testlib

## 3.9.4 Validator

前置知识：通用→3.9.2@p270

本页面将简要介绍 validator 的概念与用法.

### 概述

Validator（中文：校验器）用于检验造好的数据的合法性．当造好一道题的数据，又担心数据不合法（不符合题目的限制条件：上溢、图不连通、不是树……）时，出题者通常会借助 validator 来检查．[3]

因为 Coderforces 支持 hack 功能，所以所有 Codeforces 上的题目都必须要有 validator．UOJ 也如此．Polygon→3.10@p283 内建了对 validator 的支持.

### 使用方法

直接在命令行输入 `./val` 即可．数据通过 stdin 输入．如果想从文件输入可 `./val < a.in`.

若数据没有问题，则什么都不会输出且返回 0；否则会输出错误信息并返回一个非 0 值.

### 提示

- 写 validator 时，不能对被 validate 的数据做任何假设，因为它可能包含任何内容．因此，出题者要对各种不合法的情况进行判断（使用 Testlib 会大大简化这一流程）.
  - 例如，输入一个点数为 $n$ 的树，主要工作是判断 $n$ 是否符合范围和判断输入的是树与否．但是切不可在判断过 $n$ 范围之后就不对接下来输入的边的起点与终点的范围进行判断，否则可能会导致 validator RE.
  - 即使不会 RE 也不应该不判断，这会导致你的报错不正确．如上例，如果不判断，报错可能会是「不是一棵树」，但是正确的报错应当是「边起点/终点不在 $[1,n]$ 之间」.
- 不能对选手的读入方式做任何假设．因此，必须保证能通过 validate 的数据完全符合输入格式.
  - 例如，选手可能逐字符地读入数字，在数字与数字之间只读入一个空格．所以在编写 validator 时，数据中的每一个空白字符都要在 validator 中显式地读入（如空格和换行）.
- 结束时不要忘记 `inf.readEof()`.
- 如果题目开放 hack（或者说，validator 的错误信息会给别人看），请使报错信息尽量友好.
  - 读入变量时使用「项别名」.
  - 在判断使用的表达式不那么易懂时，使用 ensuref 而非 ensure.

### 示例

以下是 CF Gym 100541A - Stock Market[2] 的 validator：

```
#include "testlib.h"

int main(int argc, char* argv[]) {
  registerValidation(argc, argv);
  int testCount = inf.readInt(1, 10, "testCount");
```

[3] Validators with testlib.h - Codeforces[1]

```
  inf.readEoln();

  for (int i = 0; i < testCount; i++) {
    int n = inf.readInt(1, 100, "n");
    inf.readSpace();
    inf.readInt(1, 1000000, "w");
    inf.readEoln();

    for (int i = 0; i < n; ++i) {
      inf.readInt(1, 1000, "p_i");
      if (i < n - 1) inf.readSpace();
    }
    inf.readEoln();
  }

  inf.readEof();
}
```

## 外部链接

- Validator 的更多示例[3]
- `testlib.h` 的 GitHub 存储库 MikeMirzayanov/testlib[4]

## 参考资料与注释

[1] Validators with testlib.h - Codeforces 

[2] CF Gym 100541A - Stock Market 

[3] Validator 的更多示例 

[4] `testlib.h` 的 GitHub 存储库 MikeMirzayanov/testlib 

## 3.9.5 Interactor

Interactor，即交互器，用于交互题与选手程序交互．交互题的介绍见 题型介绍 - 交互题→2.3.1@p76.

> **Note**
>
> Testlib 仅支持 Codeforces 形式交互题，即两程序交互．不支持 NOI 形式的选手编写函数与其他函数交互.

请在阅读下文前先阅读 通用→3.9.2@p270.

Testlib 为 interactor 提供了一个特殊的流 `std::fstream tout`，它是一个 log 流，你可以在 interactor 中向它写入，并在 checker 中用 `ouf` 读取.

在 interactor 中，我们从 `inf` 读取题目测试数据，将选手程序（和标程）的标准输入写入 `stdout`（在线），从 `ouf` 读选手输出（在线），从 `ans` 读标准输出（在线）.

如果 interactor 返回了 ok 状态，checker（如果有的话）将接管工作，检查答案合法性.

## 用法

Windows:

```
interactor.exe <Input_File> <Output_File> [<Answer_File> [<Result_File> [-appes]]],
```

Linux:

```
./interactor.out <Input_File> <Output_File> [<Answer_File> [<Result_File> [-appes]]],
```

### 简单的例子

**题目**

interactor 随机选择一个 $[1, 10^9]$ 范围内的整数，你要写一个程序来猜它，你最多可以询问 50 次一个 $[1, 10^9]$ 范围内的整数.

interactor 将返回：

`1`：询问与答案相同，你的程序应当停止询问.

`0`：询问比答案小.

`2`：询问比答案大.

注意在此题中我们不需要 `ans`，因为我们不需要将标准输出与其比较；而在其他题中可能需要这么做.

```cpp
int main(int argc, char** argv) {
  registerInteraction(argc, argv);
  int n = inf.readInt();  // 选数
  cout.flush();           // 刷新缓冲区
  int left = 50;
  bool found = false;
  while (left > 0 && !found) {
    left--;
    int a = ouf.readInt(1, 1000000000);  // 询问
    if (a < n)
      cout << 0 << endl;
    else if (a > n)
      cout << 2 << endl;
    else
      cout << 1 << endl, found = true;
    cout.flush();
  }
  if (!found) quitf(_wa, "couldn't guess the number with 50 questions");
  ouf.readEof();
  quitf(_ok, "guessed the number with %d questions!", 50 - left);
}
```

**本文主要翻译自 Interactors with testlib.h - Codeforces[1]. `testlib.h` 的 GitHub 存储库为 MikeMirzayanov/testlib[2].**

### 外部参考资料

[1] Interactors with testlib.h - Codeforces

[2] MikeMirzayanov/testlib

## 3.9.6 Checker

Checker，即 Special Judge→3.8@p261，用于检验答案是否合法. 使用 Testlib 可以让我们免去检验许多东西，使编写简单许多.

Checker 从命令行参数读取到输入文件名、选手输出文件名、标准输出文件名，并确定选手输出是否正确，并返回一个预定义的结果：

请在阅读下文前先阅读 通用→3.9.2@p270.

## 简单的例子

**题目**

给定两个整数 $a, b$（$-1000 \le a, b \le 1000$），输出它们的和.

这题显然不需要 checker 对吧，但是如果一定要的话也可以写一个：

```cpp
#include "testlib.h"

int main(int argc, char* argv[]) {
  registerTestlibCmd(argc, argv);

  int pans = ouf.readInt(-2000, 2000, "sum of numbers");

  // 假定标准输出是正确的，不检查其范围
  // 之后我们会看到这并不合理
  int jans = ans.readInt();

  if (pans == jans)
    quitf(_ok, "The sum is correct.");
  else
    quitf(_wa, "The sum is wrong: expected = %d, found = %d", jans, pans);
}
```

## 编写 readAns 函数

假设你有一道题输入输出均有很多数，如：给定一张 DAG，求 $s$ 到 $t$ 的最长路并输出路径（可能有多条，输出任一）.

下面是一个 **不好** 的 checker 的例子.

### 不好的实现

```cpp
#include "testlib.h"
//
#include <map>
#include <vector>
using namespace std;

map<pair<int, int>, int> edges;

int main(int argc, char* argv[]) {
  registerTestlibCmd(argc, argv);
  int n = inf.readInt();  // 不需要 readSpace() 或 readEoln()
  int m = inf.readInt();  // 因为不需要在 checker 中检查标准输入合法性
                          //  （有 validator）
  for (int i = 0; i < m; i++) {
    int a = inf.readInt();
    int b = inf.readInt();
    int w = inf.readInt();
    edges[make_pair(a, b)] = edges[make_pair(b, a)] = w;
  }
  int s = inf.readInt();
```

```
  int t = inf.readInt();

  // 读入标准输出
  int jvalue = 0;
  vector<int> jpath;
  int jlen = ans.readInt();
  for (int i = 0; i < jlen; i++) {
    jpath.push_back(ans.readInt());
  }
  for (int i = 0; i < jlen - 1; i++) {
    jvalue += edges[make_pair(jpath[i], jpath[i + 1])];
  }

  // 读入选手输出
  int pvalue = 0;
  vector<int> ppath;
  vector<bool> used(n);
  int plen = ouf.readInt(2, n, "number of vertices");  // 至少包含 s 和 t 两个点
  for (int i = 0; i < plen; i++) {
    int v = ouf.readInt(1, n, format("path[%d]", i + 1).c_str());
    if (used[v - 1])  // 检查每条边是否只用到一次
      quitf(_wa, "vertex %d was used twice", v);
    used[v - 1] = true;
    ppath.push_back(v);
  }
  // 检查起点终点合法性
  if (ppath.front() != s)
    quitf(_wa, "path doesn't start in s: expected s = %d, found %d", s,
          ppath.front());
  if (ppath.back() != t)
    quitf(_wa, "path doesn't finish in t: expected t = %d, found %d", t,
          ppath.back());
  // 检查相邻点间是否有边
  for (int i = 0; i < plen - 1; i++) {
    if (edges.find(make_pair(ppath[i], ppath[i + 1])) == edges.end())
      quitf(_wa, "there is no edge (%d, %d) in the graph", ppath[i],
            ppath[i + 1]);
    pvalue += edges[make_pair(ppath[i], ppath[i + 1])];
  }

  if (jvalue != pvalue)
    quitf(_wa, "jury has answer %d, participant has answer %d", jvalue, pvalue);
  else
    quitf(_ok, "answer = %d", pvalue);
}
```

这个 checker 主要有两个问题：

1. 它确信标准输出是正确的．如果选手输出比标准输出更优，它会被判成 WA，这不太妙．同时，如果标准输出不合法，也会产生 WA．对于这两种情况，正确的操作都是返回 Fail 状态．
2. 读入标准输出和选手输出的代码是重复的．在这道题中写两遍读入问题不大，只需要一个 `for` 循环；但是如果有一道题输出很复杂，就会导致你的 checker 结构混乱．重复代码会大大降低可维护性，让你在 debug 或修改格式时变得困难．

读入标准输出和选手输出的方式实际上是完全相同的，这就是我们通常编写一个用流作为参数的读入函数的原因.

### 好的实现

```cpp
// clang-format off

#include "testlib.h"
#include <map>
#include <vector>
using namespace std;

map<pair<int, int>, int> edges;
int n, m, s, t;

// 这个函数接受一个流，从其中读入
// 检查路径的合法性并返回路径长度
// 当 stream 为 ans 时，所有 stream.quitf(_wa, ...)
// 和失败的 readXxx() 均会返回 _fail 而非 _wa
// 也就是说，如果输出非法，对于选手输出流它将返回 _wa，
// 对于标准输出流它将返回 _fail
int readAns(InStream& stream) {
  // 读入输出
  int value = 0;
  vector<int> path;
  vector<bool> used(n);
  int len = stream.readInt(2, n, "number of vertices");
  for (int i = 0; i < len; i++) {
    int v = stream.readInt(1, n, format("path[%d]", i + 1).c_str());
    if (used[v - 1]) {
      stream.quitf(_wa, "vertex %d was used twice", v);
    }
    used[v - 1] = true;
    path.push_back(v);
  }
  if (path.front() != s)
    stream.quitf(_wa, "path doesn't start in s: expected s = %d, found %d", s,
                 path.front());
  if (path.back() != t)
    stream.quitf(_wa, "path doesn't finish in t: expected t = %d, found %d", t,
                 path.back());
  for (int i = 0; i < len - 1; i++) {
    if (edges.find(make_pair(path[i], path[i + 1])) == edges.end())
      stream.quitf(_wa, "there is no edge (%d, %d) in the graph", path[i],
                   path[i + 1]);
    value += edges[make_pair(path[i], path[i + 1])];
  }
  return value;
}

int main(int argc, char* argv[]) {
  registerTestlibCmd(argc, argv);
  n = inf.readInt();
  m = inf.readInt();
  for (int i = 0; i < m; i++) {
```

```
51      int a = inf.readInt();
52      int b = inf.readInt();
53      int w = inf.readInt();
54      edges[make_pair(a, b)] = edges[make_pair(b, a)] = w;
55    }
56    int s = inf.readInt();
57    int t = inf.readInt();
58
59    int jans = readAns(ans);
60    int pans = readAns(ouf);
61    if (jans > pans)
62      quitf(_wa, "jury has the better answer: jans = %d, pans = %d\n", jans,
63            pans);
64    else if (jans == pans)
65      quitf(_ok, "answer = %d\n", pans);
66    else  // (jans < pans)
67      quitf(_fail, ":( participant has the better answer: jans = %d, pans = %d\n",
68            jans, pans);
69  }
```

注意到这种写法我们同时也检查了标准输出是否合法，这样写 checker 让程序更短，且易于理解和 debug. 此种写法也适用于输出 YES（并输出方案什么的），或 NO 的题目.

**Note**

对于某些限制的检查可以用 `InStream::ensure/ensuref()` 函数更简洁地实现. 如上例第 23 至 25 行也可以等价地写成如下形式:

```
1 stream.ensuref(!used[v - 1], "vertex %d was used twice", v);
```

**Warning**

请在 `readAns` 中避免调用 **全局** 函数 `::ensure/ensuref()`，这会导致在某些应判为 WA 的选手输出下返回 `_fail`，产生错误.

## 建议与常见错误

- 编写 `readAns` 函数，它真的可以让你的 checker 变得很棒.
- 读入选手输出时永远限定好范围，如果某些变量忘记了限定且被用于某些参数，你的 checker 可能会判定错误或 RE 等.
  - 反面教材

```
1  // ....
2  int k = ouf.readInt();
3  vector<int> lst;
4  for (int i = 0; i < k; i++)  // k = 0 和 k = -5 在这里作用相同（不会进入循环体）
5    lst.push_back(ouf.readInt());
6  // 但是我们并不想接受一个长度为 -5 的 list，不是吗?
7  // ....
8  int pos = ouf.readInt();
9  int x = A[pos];
10 // 可能会有人输出 -42, 2147483456 或其他一些非法数字导致 checker RE
```

  - 正面教材

```cpp
// ....
int k = ouf.readInt(0, n);  // 长度不合法会立刻判 WA 而不会继续 check 导致 RE
vector<int> lst;
for (int i = 0; i < k; i++) lst.push_back(ouf.readInt());
// ....
int pos = ouf.readInt(0, (int)A.size() - 1);  // 防止 out of range
int x = A[pos];
// ....
```

- 使用项别名.
- 和 validator 不同，checker 不用特意检查非空字符．例如对于一个按顺序比较整数的 checker，我们只需判断选手输出的整数和答案整数是否对应相等，而选手是每行输出一个整数，还是在一行中输出所有整数等格式问题，我们的 checker 不必关心.

### 使用方法

通常我们不需要本地运行它，评测工具/OJ 会帮我们做好这一切．但是如果需要的话，以以下格式在命令行运行：

```
./checker <input-file> <output-file> <answer-file> [<report-file> [<-appes>]]
```

### 一些预设的 checker

很多时候我们的 checker 完成的工作很简单（如判断输出的整数是否正确，输出的浮点数是否满足精度要求），Testlib[1] 已经为我们给出了这些 checker 的实现，我们可以直接使用.

一些常用的 checker 有：

- ncmp：按顺序比较 64 位整数.
- rcmp4：按顺序比较浮点数，最大可接受误差（绝对误差或相对误差）不超过 $10^{-4}$（还有 rcmp6，rcmp9 等对精度要求不同的 checker，用法和 rcmp4 类似）.
- wcmp：按顺序比较字符串（不带空格，换行符等非空字符）.
- yesno：比较 YES 和 NO，大小写不敏感.

**本文主要翻译自 Checkers with testlib.h - Codeforces[2]．`testlib.h` 的 GitHub 存储库为 MikeMirzayanov/testlib[3].**

### 外部参考资料

[1] Testlib

[2] Checkers with testlib.h - Codeforces

[3] MikeMirzayanov/testlib

## 3.10 Polygon

**Authors:** ouuan, NachtgeistW

本页面将简要介绍多人协作出题平台 Polygon.

## 简介

### 什么是 Polygon

网址：Index Page - Polygon[1]

Polygon 是一个支持多人协作的出题平台，功能非常完善．官网描述为「Polygon 的使命是为创建编程竞赛题目提供平台.」

在 Codeforces (CF) 出题必须使用 Polygon．在其它地方出题，尤其是多人合作出题时，使用 Polygon 也是不错的选择．

### 优点

- 有版本管理系统，多人合作时不会乱成一团，也不需要互相传文件．
- 出题系统完善，validator、generator、checker、solutions 环环相扣，输出自动生成．
- 可以为 solutions 设置标签，错解 AC、正解未 AC 都会警告，方便地逐一卡掉错解．
- 可以方便地对拍，拍出来的数据可以直接添加到题目数据中．
- 发现问题可以提 issue，而不会被消息刷屏却一直没有 fix．
- 为日后出 CF 做准备．
- ……

## 题目列表

题目列表中会显示一道题目的基本信息，如题面、题解撰写情况、数据生成情况以及 std、validator 和 checker 的设置．

可以双击题目列表的 "Name" 这一栏来写上 note，比如需要提醒自己做的事（need to add more tests/need to write tutorial），或者是这道题预订的 score distribution，可以根据自己的需要随意填写，当然也可以空着．

"Rev." 中的 "x/y" 的 x 指当前题目版本，y 指 package 的版本．如果两者不一样 y 会显示为红色．

"Edit session" 中的 "Start" 是指你的账号还没有看过这道题，"Continue (x) Discard" 是指你的账号处于这个题目的第 x 个版本，点击 "Start" 或 "Continue (x)" 就会进入题目管理界面，点击 "Discard" 会 **不可恢复地** 撤销你的所有更改，回到没有看过这题的状态．

如果你的账号上有一道题的更改没有提交，题目列表中这一整行就会变红．

## 题目管理

Polygon 的大部分功能都不需要学，能看懂英文就基本能用了．

> ⚠ **Warning**
>
> 题面不能使用 Markdown，只能用 TeX．

- Invocation 是用来测试 solution 的．
- Stress 是用来对拍的．
- 数据在 Tests 中用 generator 造，generator 在 Files 中上传．

### General Info

在这个页面中可以设置题目的时间限制、空间限制、题目类型、题目的标签、所属比赛．

在页面的最下方的 "statement sketch" 和 "tutorial sketch" 可以编辑题面、题解的 **草稿**，注意这两项不会出现在正式的题面、题解上．

### Statement

这个页面是用来写题面和题解的．还可以通过 "Review" 按钮来查看题面、validator 与 checker，一般用于审核．

题面和题解都需要使用 TeX 的语法，不能使用 Markdown．例如，需要使用 `\textbf{text}` 而不是 `**text**`．但 Polygon 支持的实际上是 TeX 的一个非常小的子集，具体可以自己尝试．

可以通过最上方的 "In HTML" 链接查看渲染后的题面，通过 "Tutorial in HTML" 查看渲染后的题解．

如果需要在题面中添加图片，需要先在下面的 "Statement Resource Files" 中上传图片，然后在题面中加上 `\includegraphics{filename.png}`.

### Files

"Source Files" 是用来存放 **除了 solutions 外** 的其它代码的，如 validator、checker、generator，如果是 IO 式交互题还有 interactor.

如果这些代码需要 include 其它文件，例如 Tree-Generator[2]，需要放在 "Resource Files" 中.

grader 式交互参见 官方教程[3].

### Checker

testlib.h 提供了一些内置的 checker，在选择框中有简要介绍，也可以选择后再点 "View source" 查看源码.

如果需要自己编写 checker，请参考 checker 教程→3.9.6@p278.

下面的 "Checker tests" 是通过 "Add test" 添加若干组输出以及对应的期望评测结果，然后点击 "Run tests" 就可以测试 checker 是否正确返回了评测结果.

### Interactor

仅 IO 式交互题需要，请参考 interactor 教程→3.9.5@p277.

### Validator

validator 用来检测数据合法性，编写请参考 validator 教程→3.9.4@p276.

下面的 "Validator tests" 类似于 "Checker tests"，需要提供输入和期望是否合法，用来测试 validator.

### Tests

这个页面是用来管理数据的.

在 Polygon 上，推荐的做法是使用少量 **带命令行参数** 的 generator→3.9.3@p273 来生成数据，而不是写一堆 generator 或者每生成一组数据都修改 generator. 并且，只需要生成输入，输出会自动生成.

"Testset" 就是一个测试集，如果是给 CF 出题需要手动添加 "pretests" 这个 Testset，并且 "pretests" 需要是 "tests" 的子集.

"Add Test" 是手动添加一组数据，一般用于手动输入样例或较小的数据. 虽然可以通过文件上传数据，但这是 **不推荐的**，数据应该要么是手动输入的要么是使用 generator 在某个参数下生成的.

如果勾选了 "Use in statements"，这组数据就会成为样例，自动加在题面里. 如果需要题面里显示的不是样例的输入输出（一般用于交互题），就可以点 "If you want to specify custom content of input or output data for statements click here"，然后输入你想显示在题面中的输入输出.

Tests 页面的下方是用来输入生成数据的脚本的，如 `generator-name [params] > test-index`. 可以使用 `generator-name [params] > $`，就不用手动指定测试点编号了.

可以参考 Polygon 提供的教程[4] 使用 Freemarker 来批量生成脚本.

"Preview Tests" 可以预览生成的数据.

### Stresses

这个页面是用来对拍的.

点击 "Add Stress" 就可以添加一组对拍，"Script pattern" 是一个生成数据的脚本，其中可以使用 "[10..100]" 之类的来表示在一个范围内随机选择.

然后运行对拍，如果拍出错就会显示 "Crashed"，并且可以一键把这组数据加到 Tests 中.

### Solution Files

这个页面是用来放解这道题的代码的，可以是正解也可以是错解．将错解传上来可以便捷地卡掉它们，也可以提醒自己需要卡掉它们．

### Invocations

这个页面是用来运行 solutions 的．

选择代码和测试点就可以运行了，之后可以在列表里点进去（"View"）查看详细信息．

评测状态 "FL" 表示评测出错了，一般是数据没有过 validate 或者 validator/checker/interactor 之类的 RE 了．"RJ" 有两种情况，一种是出现了 "FL"，另一种是这份代码第一个测试点就没有通过．

如果用时在时限的一半到两倍之间，会用黄色标识出来．

如果数据中存在变量没有达到最小值或最大值，会在最下方提醒．

### Issues

用来提 Issue 的地方．

### Packages

Package 包含了一道题的全部信息，在出 CF 时是 CF 评测的依据（例如，如果赛时要修锅，更新了 package 才会影响到 CF），其它时候可以用来导出．

"Verify" 是测试所有 solution 都符合标签（AC、WA、TLE），并且 checker 通过 checker tests，validator 通过 validator tests．

### Manage access

管理题目权限．

## 侧边栏

第一栏会显示一些基本信息，如果有哪里不符合规范（如 tests 没有包含 pretests、有重复的测试点）就会显示为黄色，鼠标移上去会显示具体信息．

"View changes" 可以看修改的历史记录．需要注意的是 "switch" 不能用来回退到某一个版本，只能在某个版本的基础上进行不产生冲突的修改，而这实际上是没有意义的，所以 switch 相当于是只读的．

"Update Working Copy" 是获取（他人的）更新．

"Commit Changes" 是提交你的更新．

commit 时如果有不合规范、需要警告的地方会列出来．

# 比赛管理

如果要出一场比赛，可以通过 "New Contest" 来创建比赛，就可以更加方便地管理题目．

比赛管理页面的题目列表右上角的 "Add problems?" 是把一道已有的题目加到比赛里．

侧边栏的 "New problem" 是新建一道题目加到比赛里．

上面的 "Manage problem access" 是查看每道题的权限，下面的 "Manage developers list" 是管理有这场比赛的权限的人．通过 "New problem" 创建一道题以及添加一个新的 developer 时会自动添加权限，但通过 "Add problems?" 加进来的题不会给已有的 developer 权限．

侧边栏还可以预览所有题面、所有题解、所有 validator & checker，下载整个比赛的 package，给题目重新编号．

## 冲突解决

在多人合作使用 Polygon 命题时，如果当前修改的题目版本与远程的题目版本不同，并且在从远程获取最新版本或提交更新时，修改的文件无法自动合并，就会发生冲突（Conflicted）.

发生冲突后，Polygon 会在冲突题目选项中提供 "Resolve conflicts" 选项，用户可以在里面对冲突文件进行冲突解决.

## 外部参考资料

[1] Index Page - Polygon

[2] Tree-Generator

[3] 官方教程

[4] Polygon 提供的教程

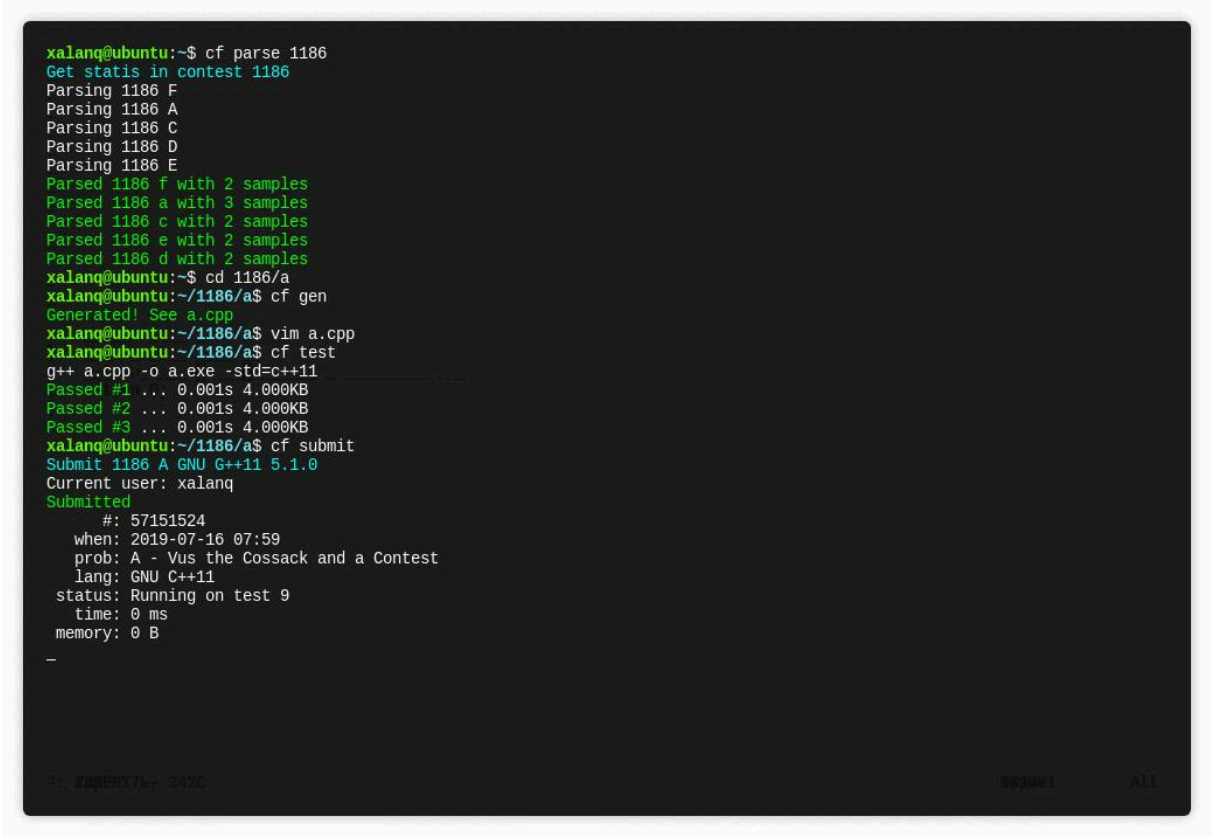
# 3.11 OJ 工具

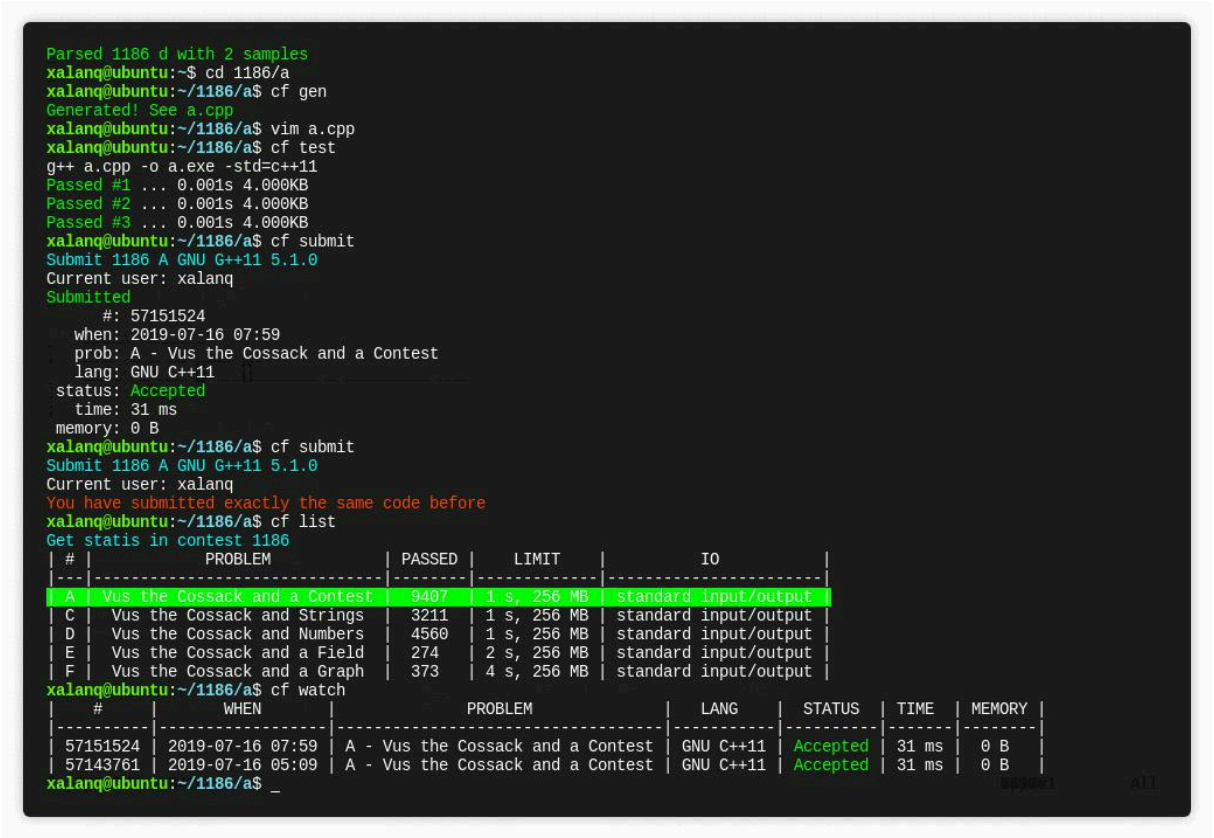
本页面将介绍一些 OJ 工具.

## cf-tool

cf-tool 是 Codeforces 的命令行界面的跨平台（支持 Windows、Linux、macOS）工具，支持很多常用操作.

源码托管在 xalanq/cf-tool[1] 上.






## 特点

- 支持 Codeforces 中的所有编程语言.
- 支持 Contests 和 Gym.
- 提交代码.
- 动态刷新提交后的情况.
- 拉取问题的样例.
- 本地编译和测试样例.
- 拉取某人的所有代码.
- 从指定模板生成代码（包括时间戳，作者等信息）.
- 列出某场比赛的所有题目的整体信息.
- 用默认的网页浏览器打开题目页面、榜单、提交页面等.
- 丰富多彩的命令行.

## 下载

前往 cf-tool/releases[2] 下载最新版.

之后的更新可以直接使用 `upgrade` 命令获取.

## 使用

将下载好的可执行文件 `cf`（或者 `cf.exe`）放置到合适的位置后（见常见问题的第二条），然后打开命令行，用 `cf config` 命令来配置一下用户名、密码和代码模板.

## 使用举例

以下简单模拟一场比赛的流程.

```
cf race 1136
```

要开始打 1136 这场比赛了！其中 1136 可以从比赛的链接获取，比方说这个例子的比赛链接就为 https://codeforces.com/contest/1136[3].

如果比赛还未开始，则该命令会进行倒计时．比赛已开始或倒计时完后，工具会自动用默认浏览器打开比赛的所有题目页面，并拉取样例到本地．

`cd 1136/a`

进入 A 题的目录，此时该目录下会包含该题的样例．

`cf gen`

用默认模板生成一份代码，在这里不妨设为 `a.cpp`．

`vim a.cpp`

用 Vim 写代码（或者用其他的编辑器或 IDE 进行）．

`cf test`

编译并测试样例．

`cf submit`

提交代码．

`cf list`

查看当前比赛各个题目的信息．

`cf stand`

用浏览器打开榜单，查看排名．

### 常见问题

1. 我双击了这个程序但是没啥效果

   cf-tool 是命令行界面的工具，你应该在终端里运行这个工具．

2. 我无法使用 `cf` 这个命令

   你应该将 `cf` 这个程序放到一个已经加入到系统变量 PATH 的路径里（比如说 Linux 里的 `/usr/bin/`）．

   不明白的话请直接搜索「PATH 添加路径」．

3. 如何加一个新的测试数据

   新建两个额外的测试数据文件 `inK.txt` 和 `ansK.txt`（K 是包含 0~9 的字符串）．

4. 怎样在终端里启用 tab 补全命令

   使用这个工具 Infinidat/infi.docopt_completion[4] 即可．

   注意：如果有一个新版本发布（尤其是添加了新命令），你应该重新运行 `docopt-completion cf`．

## Codeforces Visualizer

官网：Codeforces Visualizer[5]

源码托管在 sjsakib/cfviz[6] 上．

这个网站有三个功能：

- 用炫酷的图表来可视化某个用户的各种信息（比如通过题目的难度分布）．
- 对比两个用户．
- 计算一场比赛的 Rating 预测．

## Competitive Companion

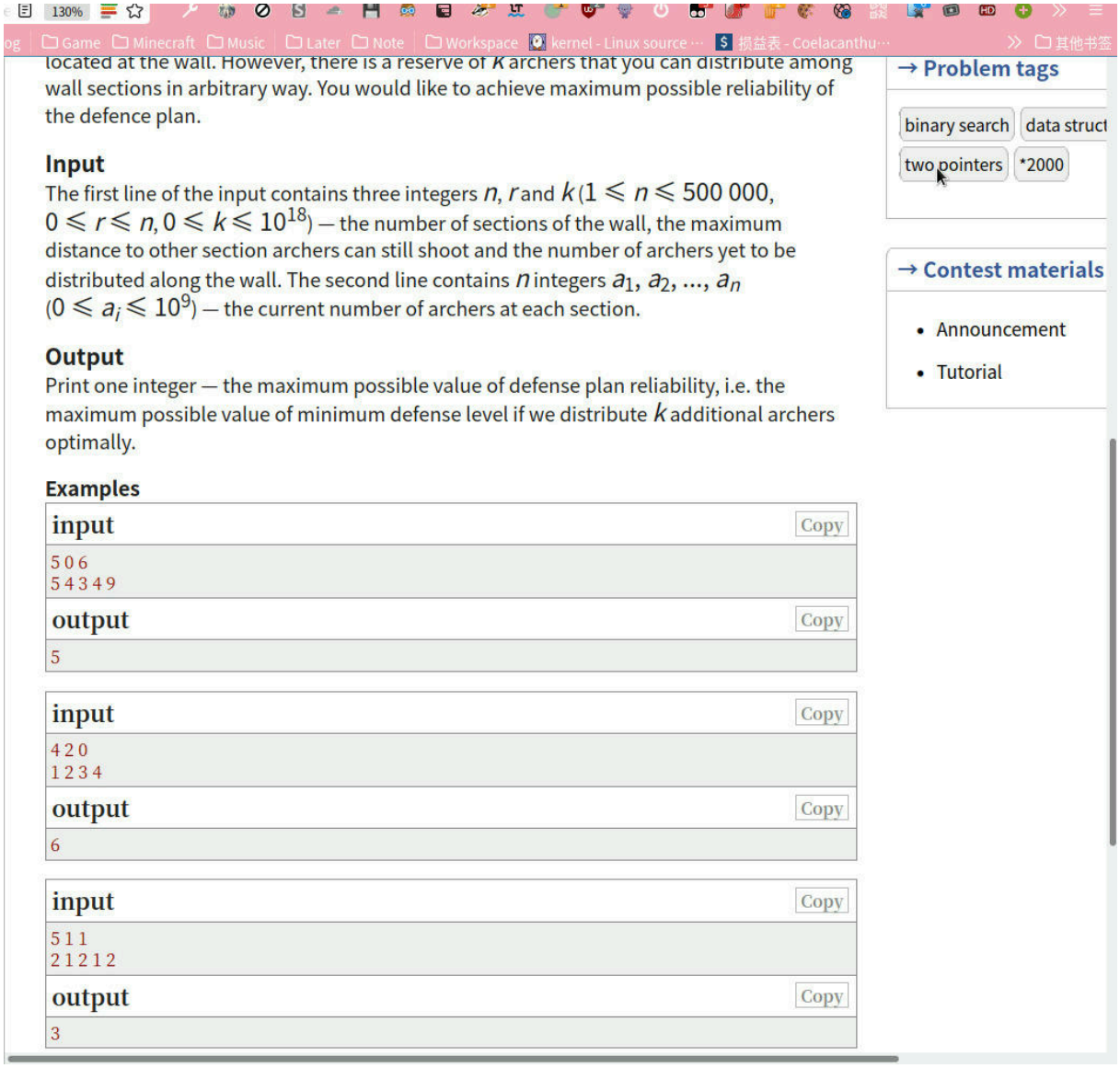
这个工具是一个浏览器插件，用于解析网页里面的测例数据．它支持解析几乎所有的主流 oj 平台（比如 Codeforces、AtCoder）．使用这个插件后，再也不用手动复制任何的测例数据．

源码托管在 jmerle/competitive-companion[7] 上．

使用方法：

- 在谷歌或者火狐浏览器上安装插件．该工具会将解析到的测例数据以 JSON 格式的形式发到指定的端口．
- 在本地安装任何可以从端口监听读取数据的工具即可，可以参考 官方给出的示例[8]．

图片演示：




使用 zqxyz73[9] 同学的 bytetools[10] 完成演示．

## ac-predictor

ac-predictor 是一个在 atcoder rating 更新前提前知道比赛 rating 变化的插件．

这个工具是一个 tampermonkey 脚本，所以你需要首先安装 tampermonkey[11]．

完成后来到 greasyfork[12]，点击安装即可．

安装完成后，比赛的排行榜界面会显示每个用户的 rating 变化预测.

这个工具有一个经由 GoodCoder666[13] 汉化的版本，点击 这里[14] 以安装.

## 外部参考资料


[1] xalanq/cf-tool

[2] cf-tool/releases

[3] https://codeforces.com/contest/1136

[4] Infinidat/infi.docopt_completion

[5] Codeforces Visualizer

[6] sjsakib/cfviz

[7] jmerle/competitive-companion

[8] 官方给出的示例

[9] zqxyz73

[10] bytetools

[11] tampermonkey

[12] greasyfork

[13] GoodCoder666

[14] 这里


# 3.12 LaTeX 入门

## 介绍

### 什么是 LaTeX

LaTeX（读作/lɑːtɛx/或/leɪtɛx/）是一个让你的文档看起来更专业的排版系统，而不是文字处理器. 它尤其适合处理篇幅较长、结构严谨的文档，并且十分擅长处理公式表达. 它是免费的软件，对大多数操作系统都适用.

LaTeX 基于 TeX（Donald Knuth 在 1978 年为数字化排版设计的排版系统）. TeX 是一种电脑能够处理的低级语言，但大多数人发现它很难使用. LaTeX 正是为了让它变得更加易用而设计的. 目前 LaTeX 的版本是 LaTeX 2e.

如果你习惯于使用微软的 Office Word 处理文档, 那么你会觉得 LaTeX 的工作方式让你很不习惯. Word 是典型的「所见即所得」的编辑器，你可以在编排文档的时候查看到最终的排版效果. 但使用 LaTeX 时你并不能方便地查看最终效果，这使得你专注于内容而不是外观的调整.

一个 LaTeX 文档是一个以 `.tex` 结尾的文本文件，可以使用任意的文本编辑器编辑，比如 Notepad，但对于大多数人而言，使用一个合适的 LaTeX 编辑器会使得编辑的过程容易很多. 在编辑的过程中你可以标记文档的结构. 完成后你可以进行编译——这意味着将它转化为另一种格式的文档. 它支持多种格式，但最常用的是 PDF 文档格式.

### 在开始之前

下面列出在本文中使用到的记号：

- 希望你实施的操作会被打上一个箭头 →；
- 你输入的字符会被装进代码块中；
- 菜单命令与按钮的名称会被标记为 **粗体**.

### 一些概念

如果需要编写 LaTeX 文档，你需要安装一个「发行版」，常用的发行版有 TeX Live[1]、MikTeX[2] 和适用于 macOS 用户的 MacTeX（实际上是 TeX Live 的 macOS 版本），至于 CTeX[3] 则现在不推荐使用. TeX Live 和 MacTeX 带有几乎所有的 LaTeX 宏包；而 MikTeX 只带有少量必须的宏包，其他宏包将在需要时安装.

TeX Live 和 MikTeX 都带有 TeXworks 编辑器，你也可以安装功能更多的 TeXstudio 编辑器，或者自行配置 Visual Studio Code 或 Notepad++ 等编辑器．下文所使用的编辑器是运行在 Windows 7 上的 TeXworks.

大部分发行版都带有多个引擎，如 pdfTeX 和 XeTeX．对于中文用户，推荐使用 XeTeX 以获得 Unicode 支持.

TeX 有多种格式，如 Plain TeX 和 LaTeX．现在一般使用 LaTeX 格式．所以，你需要使用与你所使用的格式打包在一起的引擎．如对于 pdfTeX，你需要使用 pdfLaTeX，对于 XeTeX 则是 XeLaTeX.

扩展阅读：TeX 引擎、格式、发行版之介绍[4].

### 环境配置

对于 Windows 用户，你需要下载 TeX Live 或 MikTeX．国内用户可以使用 清华大学 TUNA 镜像站[5]，请点击页面右侧的「获取下载链接」按钮，并选择「应用软件」标签下的「TeX 排版系统」即可下载 TeX Live 或 MikTeX 的安装包，其中 TeX Live 的安装包是一个 ISO 文件，需要挂载后以管理员权限执行 `install-tl-advanced.bat`.

对于 macOS 用户，清华大学 TUNA 镜像站同样提供 MacTeX 和 macOS 版 MikTeX 的下载.

对于 Linux 用户，如果使用 TeX Live，则同样下载 ISO 文件，执行 `install-tl` 脚本；如果使用 MikTeX，则按照 官方文档[6] 进行安装.

## 文档结构

### 基本要素

→ 打开 TeXworks.

一个新的文档会被自动打开.

→ 进入 **Format** 菜单，选择 **Line Numbers**.

行号并不是要素，但它可以帮助你比较代码与屏幕信息，找到错误.

→ 进入 **Format** 菜单，选择 **Syntax Coloring**，然后选择 **LaTeX**.

语法色彩会高亮代码，使得代码更加易读.

→ 输入以下文字：

```
\documentclass[a4paper, 12pt]{article}

\begin{document}
  A sentence of text.
\end{document}
```

`\documentclass` 命令必须出现在每个 LaTeX 文档的开头．花括号内的文本指定了文档的类型．**article** 文档类型适合较短的文章，比如期刊文章和短篇报告．其他文档类型包括 **report**（适用于更长的多章节的文档，比如博士生论文），**proc**（会议论文集），**book** 和 **beamer**．方括号内的文本指定了一些选项——示例中它设置纸张大小为 A4，主要文字大小为 12pt.

`\begin{document}` 和 `\end{document}` 命令将你的文本内容包裹起来．任何在 `\begin{document}` 之前的文本都被视为前导命令，会影响整个文档．任何在 `\end{document}` 之后的文本都会被忽视.

空行不是必要的，但它可以让长的文档更易读.

→ 按下 **Save** 按扭；→ 在 **Libraries>Documents** 中新建一个名为 **LaTeX course** 文件夹；→ 将你的文档命名为 **Doc1** 并将其保存为 **TeX document** 放在这个文件夹中.

将不同的 LaTeX 文档放在不同的目录下，在编译的时候组合多个文件是一个很好的想法.

→ 确保 typeset 菜单设置为了 **xeLaTeX**．→ 点击 **Typeset** 按扭.

这时你的源文件会被转换为 PDF 文档，这需要花费一定的时间．在编译结束后，TeXworks 的 PDF 查看器会打开并预览生成的文件．PDF 文件会被自动地保存在与 TeX 文档相同的目录下．

### 处理问题

如果在你的文档中存在错误，TeXworks 无法创建 PDF 文档时，**Typeset** 按扭会变成一个红叉，并且底部的终端输出会保持展开．这时：

→ 点击 **Abort typesetting** 按扭．→ 阅读终端输出的内容，最后一行可能会给出行号表示出现错误的位置．→ 找到文档中对应的行并修复错误．→ 再次点击 **Typeset** 按扭尝试编译源文件．

### 添加文档标题

`\maketitle` 命令可以给文档创建标题．你需要指定文档的标题．如果没有指定日期，就会使用现在的时间，作者是可选的．

→ 在 `\begin{document}` 和 命令后紧跟着输入以下文本：

```
\title{My First Document}
\author{My Name}
\date{\today}
\maketitle
```

你的文档现在长成了这样：

```
\documentclass[a4paper, 12pt]{article}

\begin{document}
  \title{My First Document}
  \author{My Name}
  \date{\today}
  \maketitle

  A sentence of text.
\end{document}
```

→ 点击 **Typeset** 按扭，核对生成的 PDF 文档．

要点笔记：

- `\today` 是插入当前时间的命令．你也可以输入一个不同的时间，比如 `\date{November 2013}`．
- **article** 文档的正文会紧跟着标题之后在同一页上排版．**report** 会将标题置为单独的一页．

### 章节

如果需要的话，你可能想将你的文档分为章（Chatpers）、节（Sections）和小节（Subsections）．下列分节命令适用于 **article** 类型的文档：

- `\section{...}`
- `\subsection{...}`
- `\subsubsection{...}`
- `\paragraph{...}`
- `\subparagraph{...}`

花括号内的文本表示章节的标题．对于 **report** 和 **book** 类型的文档我们还支持 `\chapter{...}` 的命令．

→ 将 "A sentence of text." 替换为以下文本：

```
\section{Introduction}
This is the introduction.
```

```
\section{Methods}

\subsection{Stage 1}
The first part of the methods.

\subsection{Stage 2}
The second part of the methods.

\section{Results}
Here are my results.
```

你的文档会变成

```
\documentclass[a4paper, 12pt]{article}

\begin{document}
  \title{My First Document}
  \author{My Name}
  \date{\today}
  \maketitle

  \section{Introduction}
  This is the introduction.

  \section{Methods}

  \subsection{Stage 1}
  The first part of the methods.

  \subsection{Stage 2}
  The second part of the methods.

  \section{Results}
  Here are my results.
\end{document}
```

→ 点击 **Typeset** 按扭，核对 PDF 文档．应该是长这样的：

My First Document

My Name

August 16, 2019

## 1 Introduction

This is the introduction.

## 2 Methods

### 2.1 Stage 1

The first part of the methods.

### 2.2 Stage 2

The second part of the methods.

## 3 Results

Here are my results.

### 创建标签

你可以对任意章节命令创建标签, 这样他们可以在文档的其他部分被引用. 使用 `\label{labelname}` 对章节创建标签. 然后输入 `\ref{labelname}` 或者 `\pageref{labelname}` 来引用对应的章节.

→ 在 `\subsection{Stage 1}` 下面另起一行, 输入 `\label{sec1}`. → 在 **Results** 章节输入 `Referring to section \ref{sec1} on page \pageref{sec1}.`

你的文档会变成这样:

```
\documentclass[a4paper, 12pt]{article}

\begin{document}
  \title{My First Document}
  \author{My Name}
  \date{\today}
  \maketitle

  \section{Introduction}
  This is the introduction.

  \section{Methods}

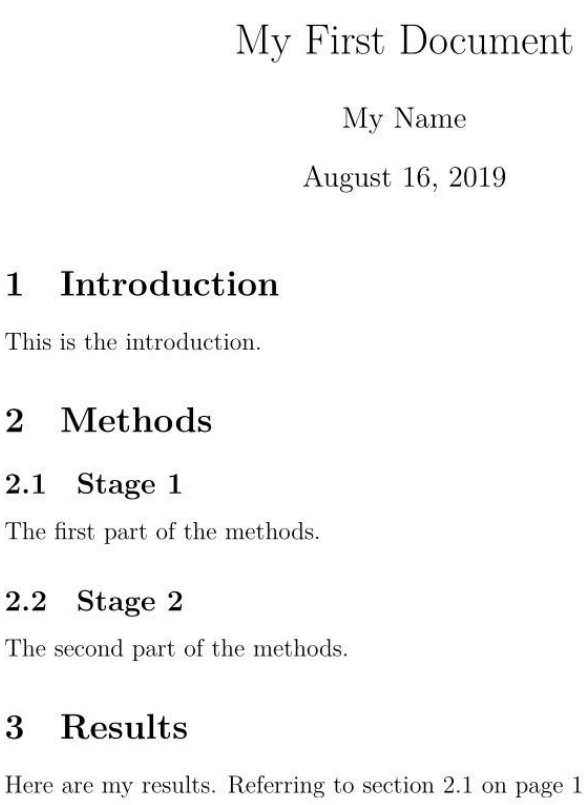
  \subsection{Stage 1}
  \label{sec1} The first part of the methods.

  \subsection{Stage 2}
  The second part of the methods.

  \section{Results}
```

```
  Here are my results. Referring to section \ref{sec1} on page \pageref{sec1}
\end{document}
```

→ 编译并检查 PDF 文档（你可能需要连续编译两次）：



My First Document

My Name

August 16, 2019

1 Introduction

This is the introduction.

2 Methods

2.1 Stage 1

The first part of the methods.

2.2 Stage 2

The second part of the methods.

3 Results

Here are my results. Referring to section 2.1 on page 1

## 生成目录（TOC）

如果你使用分节命令，那么可以容易地生成一个目录．使用 `\tableofcontents` 在文档中创建目录．通常我们会在标题的后面建立目录．

你可能也想更改页码为罗马数字（i,ii,iii）．这会确保文档的正文从第 1 页开始．页码可以使用 `\pagenumbering{...}` 在阿拉伯数字和罗马数字见切换．

→ 在 `\maketitle` 之后输入以下内容：

```
\pagenumbering{roman}
\tableofcontents
\newpage
\pagenumbering{arabic}
```

`\newpage` 命令会另起一个页面，这样我们就可以看到 `\pagenumbering` 命令带来的影响了．你的文档的前 14 行长这样：

```
\documentclass[a4paper, 12pt]{article}

\begin{document}

\title{My First Document}
\author{My Name}
\date{\today}
```

```
\maketitle

\pagenumbering{roman}
\tableofcontents
\newpage
\pagenumbering{arabic}
```

→ 编译并核对文档（可能需要多次编译，下文不赘述）.

文档的第一页长这样:

My First Document

My Name

August 16, 2019

Contents

1 Introduction 1

2 Methods 1
2.1 Stage 1 . . . . . 1
2.2 Stage 2 . . . . . 1

3 Results 1

第二页:

## 1 Introduction

This is the introduction.

## 2 Methods

### 2.1 Stage 1

The first part of the methods.

### 2.2 Stage 2

The second part of the methods.

## 3 Results

Here are my results. Referring to section 2.1 on page 1

# 文字处理

## 中文字体支持

阅读本文学习 LaTeX 的人，首要学会的自然是 LaTeX 的中文字体支持．事实上，让 LaTeX 支持中文字体有许多方法．在此我们仅给出最 **简洁** 的解决方案：使用 CTeX 宏包．只需要在文档的前导命令部分添加：

```
\usepackage[UTF8]{ctex}
```

就可以了．在编译文档的时候使用 `xelatex` 命令，因为它是支持中文字体的．

## 字体效果

LaTeX 有多种不同的字体效果，在此列举一部分：

```
\textit{words in italics} \textsl{words slanted} \textsc{words in smallcaps} \textbf{words
in bold} \texttt{words in teletype} \textsf{sans serif words} \textrm{roman
words} \underline{underlined words}
```

效果如下：

*words in italics.*
*words slanted*
WORDS IN SMALLCAPS
**words in bold**
`words in teletype`
sans serif words
roman words
underlined words

→ 在你的文档中添加更多的文本并尝试各种字体效果．

## 彩色字体

为了让你的文档支持彩色字体，你需要使用包（package）．你可以引用很多包来增强 LaTeX 的排版效果．包引用的命令放置在文档的前导命令的位置（即放在 `\begin{document}` 命令之前）．使用 `\usepackage[options]{package}` 来引用包．其中 **package** 是包的名称，而 **options** 是指定包的特征的一些参数．

使用 `\usepackage{color}` 后，我们可以调用常见的颜色：

red, green, blue, cyan, magenta, yellow, white

使用彩色字体的代码为

```
{\color{colorname}text}
```

其中 **colorname** 是你想要的颜色的名字，**text** 是你的彩色文本内容．注意到示例效果中的黄色与白色是有文字背景色的，这个我们同样可以使用 Color 包中的 `\colorbox` 命令来达到．用法如下：

```
\colorbox{colorname}{text}
```

→ 在 `\begin{document}` 前输入 `\usepackage{color}`．→ 在文档内容中输入 `{\color{red}fire}`．→ 编译并核对 PDF 文档内容．

单词 fire 应该是红色的．

你也可以添加一些参数来调用更多的颜色，甚至自定义你需要的颜色．但这部分超出了本书的内容．如果想要获取更多关于彩色文本的内容请阅读 LaTeX Wikibook 的 Colors 章节[7]．

### 字体大小

接下来我们列举一些 LaTeX 的字体大小设定命令：

```
normal size words {\tiny tiny words} {\scriptsize scriptsize words}
{\footnotesize footnotesize words} {\small small words} {\large large words}
{\Large Large words} {\LARGE LARGE words} {\huge huge words}
```

效果如下：




→ 尝试为你的文本调整字体大小．

### 段落缩进

LaTeX 默认每个章节第一段首行顶格，之后的段落首行缩进．如果想要段落顶格，在要顶格的段落前加 `\noindent` 命令即可．如果希望全局所有段落都顶格，在文档的某一位置使用 `\setlength{\parindent}{0pt}` 命令，之后的所有段落都会顶格．

### 列表

LaTeX 支持两种类型的列表：有序列表（enumerate）和无序列表（itemize）．列表中的元素定义为 `\item`．列表可以有子列表．

→ 输入下面的内容来生成一个有序列表套无序列表：

```
\begin{enumerate}
  \item First thing

  \item Second thing
    \begin{itemize}
      \item A sub-thing

      \item Another sub-thing
    \end{itemize}

  \item Third thing
\end{enumerate}
```

→ 编译并核对 PDF 文档．

列表长这样：

1. First thing
2. Second thing
   - A sub-thing
   - Another sub-thing
3. Third thing

可以使用方括号参数来修改无序列表头的标志. 例如, `\item[-]` 会使用一个杠作为标志, 你甚至可以使用一个单词, 比如 `\item[One]`.

下面的代码:

```
\begin{itemize}
  \item[-] First thing

  \item[+] Second thing
    \begin{itemize}
      \item[Fish] A sub-thing

      \item[Plants] Another sub-thing
    \end{itemize}

  \item[Q] Third thing
\end{itemize}
```

生成的效果为

- First thing
+ Second thing
  Fish A sub-thing
  Plants Another sub-thing
Q Third thing

## 注释和空格

我们使用 % 创建一个单行注释, 在这个字符之后的该行上的内容都会被忽略, 直到下一行开始.

下面的代码:

```
It is a truth universally acknowledged% Note comic irony
in the very first sentence , that a single man in possession of a good fortune,
must be in want of a wife.
```

生成的结果为

It is a truth universally acknowledgedin the very first sentence , that a single man in possession of a good fortune, must be in want of a wife.

多个连续空格在 LaTeX 中被视为一个空格. 多个连续空行被视为一个空行. 空行的主要功能是开始一个新的段落. 通常来说, LaTeX 忽略空行和其他空白字符, 两个反斜杠 (`\\`) 可以被用来换行.

→ 尝试在你的文档中添加注释和空行.

如果你想要在你的文档中添加空格, 你可以使用 `\vspace{...}` 的命令. 这样可以添加竖着的空格, 高度可以指定. 如 `\vspace{12pt}` 会产生一个空格, 高度等于 12pt 的文字的高度.

### 特殊字符

下列字符在 LaTeX 中属于特殊字符:

```
# $ % ^ & _ { } ~ \
```

为了使用这些字符, 我们需要在他们前面添加反斜杠进行转义:

```
\# \$ \% \^{} \& \_ \{ \} \~{}
```

注意在使用 `^` 和 `~` 字符的时候需要在后面紧跟一对闭合的花括号, 否则他们就会被解释为字母的上标, 就像 `\^ e` 会变成 ê. 上面的代码生成的效果如下:

# $ % ^ & _ { } ~

注意, 反斜杠不能通过反斜杠转义 (不然就变成了换行了), 使用 `\textbackslash` 命令代替.

→ 输入代码来在你的文档中生成下面内容:

Item #A\642 costs $8 & is sold at a ^ 10% profit.

询问专家或者查看本页面的 源代码[8] 获取帮助.

## 表格

表格 (tabular) 命令用于排版表格. LaTeX 默认表格是没有横向和竖向的分割线的——如果你需要, 你得手动设定. LaTeX 会根据内容自动设置表格的宽度. 下面的代码可以创一个表格:

```
\begin{tabular}{...}
```

省略号会由定义表格的列的代码替换:

- `l` 表示一个左对齐的列;
- `r` 表示一个右对齐的列;
- `c` 表示一个向中对齐的列;
- `|` 表示一个列的竖线;

例如, `{lll}` 会生成一个三列的表格, 并且保存向左对齐, 没有显式的竖线; `{|l|l|r|}` 会生成一个三列表格, 前两列左对齐, 最后一列右对齐, 并且相邻两列之间有显式的竖线.

表格的数据在 `\begin{tabular}` 后输入:

- `&` 用于分割列;
- `\\` 用于换行;
- `\hline` 表示插入一个贯穿所有列的横着的分割线;
- `\cline{1-2}` 会在第一列和第二列插入一个横着的分割线.

最后使用 `\end{tabular}` 结束表格. 举一些例子:

```
\begin{tabular}{|l|l|}
  Apples       & Green  \\
  Strawberries & Red    \\
  Orange       & Orange \\
\end{tabular}

\begin{tabular}{rc}
  Apples              & Green  \\
  \hline
  Strawberries        & Red    \\
  \cline{1-1} Oranges & Orange \\
\end{tabular}

\begin{tabular}{|r|l|}
  \hline
  8              & here's \\
  \cline{2-2} 86 & stuff  \\
  \hline
  \hline
  2008           & now    \\
  \hline
\end{tabular}
```

效果如下:

| Apples | Green |
|---|---|
| Strawberries | Red |
| Orange | Orange |

| | |
|---|---|
| Apples | Green |
| Strawberries | Red |
| Oranges | Orange |

| | |
|---|---|
| 8 | here's |
| 86 | stuff |
| 2008 | now |

## 实践

尝试画出下列表格:

| Item | Quantity | Price($) |
|---|---|---|
| Nails | 500 | 0.34 |
| Wooden boards | 100 | 4.00 |
| Bricks | 240 | 11.50 |

| | Year | | |
|---|---|---|---|
| City | 2006 | 2007 | 2008 |
| London | 45789 | 46551 | 51298 |
| Berlin | 34549 | 32543 | 29870 |
| Paris | 49835 | 51009 | 51970 |

## 图表

本章介绍如何在 LaTeX 文档中插入图表. 这里我们需要引入 **graphicx** 包. 图片应当是 PDF, PNG, JPEG 或者 GIF 文件. 下面的代码会插入一个名为 myimage 的图片:

```
\begin{figure}[h]
  \centering
  \includegraphics[width=1\textwidth]{myimage}
  \caption{Here is my image}
  \label{image-myimage}
\end{figure}
```

`[h]` 是位置参数, **h** 表示把图表近似地放置在这里（如果能放得下）. 有其他的选项: **t** 表示放在页面顶端; **b** 表示放在页面的底端; **p** 表示另起一页放置图表. 你也可以添加一个 **!** 参数来强制放在参数指定的位置（尽管这样排版的效果可能不太好）.

`\centering` 将图片放置在页面的中央. 如果没有该命令会默认左对齐. 使用它的效果是很好的, 因为图表的标题也是居中对齐的.

`\includegraphics{...}` 命令可以自动将图放置到你的文档中, 图片文件应当与 TeX 文件放在同一目录下.

`[width=1\textwidth]` 是一个可选的参数, 它指定图片的宽度——与文本的宽度相同. 宽度也可以以厘米为单位. 你也可以使用 `[scale=0.5]` 将图片按比例缩小（示例相当于缩小一半）.

`\caption{...}` 定义了图表的标题. 如果使用了它, LaTeX 会给你的图表添加「Figure」开头的序号. 你可以使用 `\listoffigures` 来生成一个图表的目录.

`\label{...}` 创建了一个可以供你引用的标签.

### 实践

→ 在你文档的前导命令中添加 `\usepackage{graphicx}`. → 找到一张图片, 放置在你的 **LaTeX course** 文件夹下. → 在你想要添加图片的地方输入以下内容:

```
\begin{figure}[h!]
  \centering
  \includegraphics[width=1\textwidth]{ImageFilename}
  \caption{My test image}
\end{figure}
```

将 **ImageFilename** 替换为你的文件的名字（不包括后缀）. 如果你的文件名有空格, 就使用双引号包裹, 比如 `"screen 20"`.

→ 编译并核对文件.

## 公式

使用 LaTeX 的主要原因之一是它可以方便地排版公式. 我们使用数学模式来排版公式.

### 插入公式

你可以使用一对 `$` 来启用数学模式，这可以用于撰写行内数学公式. 例如 `$1+2=3$` 的生成效果是 $1+2=3$.

如果你想要行间的公式，可以使用 `$$...$$`（现在我们推荐使用 `\[...\]`，因为前者可能产生不良间距）. 例如，`$$1+2=3$$` 的生产效果为

$$1+2=3$$

如果是生成带标号的公式，可以使用 `\begin{equation}...\end{equation}`. 例如:

```
\begin{equation}
  1+2=3
\end{equation}
```

生成的效果为:

$$1+2=3 \tag{1}$$

数字 6 代表的是章节的编号，仅当你的文档有设置章节时才会出现，比如 **report** 类型的文档.

使用 `\begin{eqnarray}...\end{eqnarray}` 来撰写一组带标号的公式. 例如:

```
\begin{eqnarray}
  a & = & b + c \\
  & = & y - z
\end{eqnarray}
```

生成的效果为

$$\begin{aligned} a &= b+c && (1)\\ &= y-z && (2) \end{aligned}$$

要撰写不标号的公式就在环境标志的后面添加 `*` 字符，如 `{equation*}`，`{eqnarray*}`.

> ⚠ **Warning**
>
> 可以发现，使用 `eqnarray` 时，会出现等号周围的空隙过大之类的问题.
>
> 可以使用 `amsmath` 宏包中的 `align` 环境:
>
> ```
> \usepackage{amsmath}
> ...
> \begin{align}
>   a & = b + c \\
>     & = y - z
> \end{align}
> ```

> 或在行间公式中使用 `aligned` 环境．它们的名字后面加上星号后，公式就不带标号了．
>
> 详见 更多阅读→3@p137 中第一篇资料的「4.4 多行公式」.

## 数学符号

尽管一些基础的符号可以直接键入，但大多数特殊符号需要使用命令来显示.

本书只是数学符号使用的入门教程，LaTeX Wikibook 的数学符号章节是另一个更好更完整的教程．如果想要了解更多关于数学符号的内容请移步．如果你想找到一个特定的符号，可以使用 Detexfiy[9]，它可以识别手写字符.

### 上标和下标

上标（Powers）使用 `^` 来表示，比如 `$n^2$` 生成的效果为 $n^2$.

下标（Indices）使用 `_` 表示，比如 `$2_a$` 生成的效果为 $2_a$.

如果上标或下标的内容包含多个字符，请使用花括号包裹起来．比如 `$b_{a-2}$` 的效果为 $b_{a-2}$.

### 分数

分数使用 `\frac{numerator}{denominator}` 命令插入．比如 `$$\frac{a}{3}$$` 的生成效果为

$$\frac{a}{3}$$

分数可以嵌套．比如 `$$\frac{y}{\frac{3}{x}+b}$$` 的生成效果为

$$\frac{y}{\frac{3}{x}+b}$$

### 根号

我们使用 `\sqrt{...}` 命令插入根号．省略号的内容由被开根的内容替代．如果需要添加开根的次数，使用方括号括起来即可.

例如 `$$\sqrt{y^2}$$` 的生成效果为

$$\sqrt{y^2}$$

而 `$$\sqrt[x]{y^2}$$` 的生成效果为

$$\sqrt[x]{y^2}$$

### 求和与积分

使用 `\sum` 和 `\int` 来插入求和式与积分式．对于两种符号，上限使用 `^` 来表示，而下限使用 `_` 表示.

`$$\sum_{x=1}^5 y^z$$` 的生成效果为

$$\sum_{x=1}^{5} y^z$$

而 `$$\int_a^b f(x)$$` 的生成效果为

$$\int_a^b f(x)$$

### 希腊字母

我们可以使用反斜杠加希腊字母的名称来表示一个希腊字母．名称的首字母的大小写决定希腊字母的形态．例如

- `$\alpha$`=$\alpha$
- `$\beta$`=$\beta$

- `$\delta, \Delta$`=$\delta, \Delta$
- `$\pi, \Pi$`=$\pi, \Pi$
- `$\sigma, \Sigma$`=$\sigma, \Sigma$
- `$\phi, \Phi, \varphi$`=$\phi, \Phi, \varphi$
- `$\psi, \Psi$`=$\psi, \Psi$
- `$\omega, \Omega$`=$\omega, \Omega$

### 实践

→ 撰写代码来生成下列公式：

$$
\begin{aligned}
e &= mc^2 && (1)\\
\pi &= \frac{c}{d} && (2)\\
\frac{d}{dx}e^x &= e^x && (3)\\
\frac{d}{dx}\int_0^\infty f(s)ds &= f(x) && (4)\\
f(x) &= \sum_i 0^\infty \frac{f^{(i)}(0)}{i!}x^i && (5)\\
x &= \sqrt{\frac{x_i}{z}y} && (6)
\end{aligned}
$$

如果需要帮助，可以查看本页面的 源代码[10].

## 参考文献

### 介绍

LaTeX 可以轻松插入参考文献以及目录．本文会介绍如何使用另一个 BibTeX 文件来存储参考文献.

### BibTeX 文件类型

BibTeX 文件包含了所有你想要在你文档中引用的文献．它的文件后缀名为 `.bib`．它的名字应设置为你的 TeX 文档的名字．`.bib` 文件是文本文件．你需要将你的参考文献按照下列格式输入：

```
@article{
    Birdetal2001,
    Author = {Bird, R. B. and Smith, E. A. and Bird, D. W.},
    Title = {The hunting handicap: costly signaling in human foraging strategies},
    Journal = {Behavioral Ecology and Sociobiology},
    Volume = {50},
    Pages = {9-19},
    Year = {2001}
}
```

每一个参考文献先声名它的文献类型（reference type）．示例中使用的是 @article，其他的类型包括 @book，@incollection 用于引用一本书的中的章节，@inproceedings 用于引用会议论文．可以 在此[11] 查看更多支持的类型.

接下来的花括号内首先要列出一个引用键值（citation key）．必须保证你引用的文献的引用键值是不同的．你可以自定义键值串，不过使用第一作者名字加上年分会是一个表义清晰的选择.

接下来的若干行包括文献的若干信息，格式如下：

```
Field name = {field contents},
```

你可以使用 LaTeX 命令来生成特殊的文字效果．比如意大利斜体可以使用 `\emph{Rattus norvegicus}`.

对于需要大写的字母，请用花括号包裹起来．BibTeX 会自动把标题中除第一个字母外所有大写字母替换为小写．比如 `Dispersal in the contemporary United States` 的生成效果为 Dispersal in the contemporary united states, 而`Dispersal in the contemporary {U}nited {S}tates` 的生成效果为 Dispersal in the contemporary United States.

你可以手写 BibTeX 文件，也可以使用软件来生成.

## 插入文献列表

使用下列命令在文档当前位置插入文献列表：

```
\bibliographystyle{plain}
\bibliography{references}
```

参考文献写在 `references.bib` 里.

## 参考文献标注

使用 `\cite{citationkey}` 来在你想要引用文献的地方插入一个标注. 如果你不希望在正文中插入一个引用标注，但仍想要在文献列表中显示这次引用，使用 `\nocite{citationkey}` 命令.

想要在引用中插入页码信息，使用方括号：`\cite[p. 215]{citationkay}`.

要引用多个文献，使用逗号分隔：`\cite{citation01,citation02,citation03}`.

## 引用格式

### 数字标号引用

LaTeX 包含了多种行内数字标号引用的格式：

**Plain** 方括号包裹数字的形式，如 [1]．文献列表按照第一作者的字母表顺序排列．每一个作者的名字是全称.

**Abbrv** 与 **plain** 是相同的，但作者的名字是缩写.

**Unsrt** 与 **plain** 是相同的，但文献列表的排序按照在文中引用的先后顺序排列.

**Alpha** 与 **plain** 一样，但引用的标注是作者的名字与年份组合在一起，不是数字，如 [*Kop*10].

### 作者日期引用

如果你想使用作者日期的引用，使用 **natbib** 包．它使用 `\citep{...}` 命令来生成一个方括号标注，如 [*Koppe*, 2010]，使用 `\citet{...}` 来生成一个标注，只把年份放到方括号里，如 *Koppe*[2010]．在此[12] 查看它的更多用法.

Natbib 包也有三种格式：**plainnat**，**abbrvnat** 和 **unsrtnat**，他们与 **plain**，**abbrv** 和 **unsrt** 的效果是一样的.

### 其他引用格式

如果你需要使用不同的格式，你需要在同一个文件夹下创建一个格式文件（`.bst` 文件），引用这个格式的时候使用它的文件名调用 `\bibliographystyle{...}` 命令实现.

## 实践

→ 在同一文件夹下新建一个同名的 BibTeX 文件，用正确的格式输入参考文献的信息．→ 切换到 TeX 文档，并使用 `\cite`，`\bibliographystyle` 和 `\bibliograph` 命令来引用文献．→ 编译 TeX 文件．→ 切换到 BibTeX 文件，并编译（点击 **Typeset** 按扭）→ 切换到 TeX 文件并编译它 **两次**，然后核对 PDF 文档.

## 更多阅读

- 一份（不太）简短的 LATEX 2ε 介绍 https://github.com/CTeX-org/lshort-zh-cn/releases/download/v6.02/lshort-zh-cn.pdf[13] 或 112 分钟了解 LaTeX 2ε.
- LaTeX Project http://www.latex-project.org/[14] Official website - has links to documentation, information about installing LATEX on your own computer, and information about where to look for help.
- LaTeX Wikibook http://en.wikibooks.org/wiki/LaTeX/[15] Comprehensive and clearly written, although still a work in progress. A downloadable PDF is also available.
- Comparison of TeX Editors on Wikipedia http://en.wikipedia.org/wiki/Comparison_of_TeX_editors[16] Information to help you to choose which L A TEX editor to install on your own computer.
- TeX Live http://www.tug.org/texlive/[17]"An easy way to get up and running with the TeX document production system". Available for Unix and Windows (links to MacTeX for MacOSX users). Includes the TeXworks editor.
- Workbook Source Files http://edin.ac/17EQPM1[18] Download the .tex file and other files needed to compile this workbook.

**本文译自 http://www.docs.is.ed.ac.uk/skills/documents/3722/3722-2014.pdf[19], 依据其他文献略有修改.**

## 外部参考资料

[1] TeX Live

[2] MikTeX

[3] CTeX

[4] TeX 引擎、格式、发行版之介绍

[5] 清华大学 TUNA 镜像站

[6] 官方文档

[7] Colors 章节

[8] 源代码

[9] Detexfiy

[10] 源代码

[11] 在此

[12] 在此

[13] https://github.com/CTeX-org/lshort-zh-cn/releases/download/v6.02/lshort-zh-cn.pdf

[14] http://www.latex-project.org/

[15] http://en.wikibooks.org/wiki/LaTeX/

[16] http://en.wikipedia.org/wiki/Comparison_of_TeX_editors

[17] http://www.tug.org/texlive/

[18] http://edin.ac/17EQPM1

[19] http://www.docs.is.ed.ac.uk/skills/documents/3722/3722-2014.pdf

# 3.13 Git

**Note**

本页面将着重介绍 Git 这一版本控制系统，与 GitHub 相关的内容，请参考 GitHub 帮助[1] 和 如何参与 - OI Wiki→1.3@p17.

Git 是目前使用最广泛的版本控制系统之一. **OI Wiki** 也使用了 Git 作为版本控制系统.

## 安装

参见 Git - Downloads[2].

# 配置

Git 根据配置文件的应用范围，将配置文件分为不同的等级，其中较常用的有两个级别[1]：

1. 适用于当前用户的全局配置文件，该用户操作本系统上的所有仓库时都会查询该配置文件.
2. 适用于当前仓库的配置文件.

当多个配置文件对同一个选项作出设置的时候，局部设置会自动覆盖全局设置. 因此如果需要在某个仓库应用特定的设置的话，只需更改该仓库下的特定设置即可，不会对全局设置造成影响.

修改配置文件需要用到 `git config` 命令.

## 设置用户信息

安装 Git 后，第一件事情就是设置你的用户名和邮箱. 这些信息在每次提交时都会用到.

```
$ git config --global user.name "OI Wiki"
$ git config --global user.email oi-wiki@example.com
```

> **Note**
>
> 这里给出的用户名和邮箱仅供演示. 您在根据本页面的内容配置时，请记得将这里的用户名和邮箱改成自己的信息.

这里的 `--global` 表示修改的是全局配置，即该设置对当前用户下的所有仓库均有效. 如果不添加 `--global` 选项，则会默认修改当前仓库下的配置文件.

如果想要修改某个仓库的特定设置，只需在该仓库下执行不带 `--global` 的命令即可.

## 配置编辑器

```
$ git config --global core.editor emacs
```

执行如上命令可以将编辑器更改为 Emacs→3.2.2@p142.

在 Windows 下，Git 的默认编辑器可以在安装 Git 时选择（见前文）. 之后若要修改，在 Git Bash 里输入如上命令，将编辑器名换成编辑器的绝对路径，运行命令即可.

## 显示配置

可以通过 `git config -l` 列出当前已经设置的所有配置参数. 使用 `git config --global -l` 可以列出所有全局配置.

# 仓库操作基础

## 新建 Git 仓库

新建一个 Git 仓库非常简单，只需在想要建立仓库的文件夹输入如下命令：

```
$ git init
```

Git 将在当前文件夹新建一个 `.git` 文件夹，一个仓库就这样建好了.

如果想把一个仓库克隆到自己的电脑上（比如将 OI Wiki 的代码拷贝到本地上进行编辑），采用 `git clone` 命令即可.

```
$ git clone https://github.com/OI-wiki/OI-wiki
```

[1] 事实上 Git 还有一个针对系统上每一个用户及系统上所有仓库的通用配置文件，该配置文件覆盖范围最广，等级在用户配置文件之上. 因为该配置实践中较少使用，这里不再展开.

**远程仓库的链接**

这里给出的仓库链接是 HTTP(S) 链接，也即我们采用了 HTTP(S) 方式连接到远程仓库.

事实上, 连接到远程仓库的方式还有多种. 其中使用 ssh 连接到远程仓库的方法更为方便和安全, 在「远程仓库的管理」部分我们会简单介绍使用 ssh 连接到远程仓库的方法.

这样，被克隆的仓库的内容就会被储存到当前文件夹下一个与仓库同名的新文件夹. 在本例中，当前文件夹下会出现一个名为 `OI-wiki` 的新文件夹.

## 跟踪文件

在对仓库的文件做出了一些更改后，这些更改需要被纳入到版本管理当中去.

使用 `git status` 命令可以查看当前仓库文件的状态.

举个例子，在一个空仓库中新增了一个 `README.md` 文件后，执行 `git status` 命令的效果如下:

```
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        README.md

nothing added to commit but untracked files present (use "git add" to track)
```

这里的 Untracked files 指的是 Git 之前没有纳入版本跟踪的文件. 如果文件没有纳入版本跟踪, 对该文件的更改不会被 Git 记录.

执行 `git add <文件>` 命令可以将指定的文件纳入到版本跟踪中.

```
$ git add README.md # 将这个文件纳入到版本跟踪中
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   README.md
```

这时 `README.md` 已经纳入了版本跟踪，放入了暂存区. 接下来只需执行 `git commit` 命令就可以提交这次更改了.

但在进行这一工作之前，让我们先对 `README.md` 做点小更改.

```
$ vim README.md # 随便更改点东西
$ git status
On branch master

No commits yet

Changes to be committed:
```

```
  (use "git rm --cached <file>..." to unstage)

        new file:   README.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore -- <file>..." to discard changes in working directory)

        modified:   README.md
```

你会发现 README.md 同时处于暂存区和非暂存区. 实际上, 是否处于暂存区是对于更改而言的, 而不是对于文件而言的, 所以对 README.md 的前一次更改已被纳入暂存区, 而后一次更改还没有. 如果这时候执行 git commit 命令，只有处于暂存区的更改会被提交，而非暂存区的更改，则不会被提交.

Git 给了一条提示，执行 git add README.md 就可以将非暂存区的更改放入暂存区了.

**一次性将所有更改放入暂存区**

git add 命令会将对指定的文件的更改放入暂存区中.

在多数情况下，用户更期望一次性将所有更改都放入暂存区中，这时候可以应用 git add -A 命令. 该命令会将所有更改（包括未被纳入版本跟踪的文件，不包括被忽略的文件）放入暂存区.

如果只需更新已被纳入版本跟踪的文件，而不将未纳入版本跟踪的文件加入暂存区，可以使用 git add -u.

**忽略文件**

有些时候我们并不希望将一些文件（如可执行文件等）纳入到版本跟踪中. 这时候可以在仓库根目录下创建 .gitignore 文件，在该文件里写下想要忽略的文件. Git 将不会将这些文件纳入到版本跟踪中.

例如，*.exe 将自动忽略仓库里的所有扩展名为 .exe 的文件.

现在将非暂存区的文件加入暂存区，将所有更改一并提交（commit）.

```
$ git add README.md
$ git commit # 接下来会弹出编辑器页面，你需要写下 commit 信息
[master (root-commit) f992763] initial commit
 1 file changed, 2 insertions(+)
 create mode 100644 README.md
```

现在重点观察一下这一次 commit 的信息.

master 表示当前位于 master 分支（关于分支的问题，下文将会详细介绍），f992763 表示本次提交的 SHA-1 校验和的前几位，后面则是本次提交的信息.

需要特别关注的是这里的 SHA-1 校验码，每个校验码都与某个时刻仓库的一个快照相对应. 利用这一特性我们可以访问历史某个时刻的仓库快照，并在该快照上进行更改.

接下来两行则详细说明了本次更新涉及的文件更改.

另外，commit 过程中可以利用几个参数来简化提交过程:

- -a: 在提交前将所有已跟踪的文件的更改放入暂存区. 需要注意的是未被跟踪的文件（新创建的文件）不会被自动加入暂存区，需要用 git add 命令手动添加.
- -m: 该参数后跟提交信息，表示以该提交信息提交本次更改. 例如 git commit -m "fix: typo" 会创建一条标题为 fix: typo 的 commit.

### 查看提交记录

使用 `git log` 命令可以查看仓库的提交历史记录.

可以看到，提交历史里记录了每次提交时的 SHA-1 校验和，提交的作者，提交时间和 commit 信息.

```
$ git log
commit ae9dd3768a405b348bc6170c7acb8b6cb5fe333e (HEAD -> master)
Author: OI Wiki <oi-wiki@example.com>
Date:   Sun Sep 13 00:30:18 2020 +0800

    feat: update README.md

commit f99276362a3c260d439364c505a7a06859f34bf9
Author: OI Wiki <oi-wiki@example.com>
Date:   Sun Sep 13 00:06:07 2020 +0800

    initial commit
```

## 分支管理

为什么版本管理中需要分支管理呢？答案主要有两点：

1. 直接更改主分支不仅会使历史记录混乱，也可能会造成一些危险的后果.
2. 通过分支, 我们可以专注于当前的工作. 如果我们需要完成两个不同的工作, 只需开两个分支即可, 两个分支间的工作互不干扰.

在 Git 中，简单来说，分支就是指向某个快照的指针. 每次提交时，Git 都会为这次提交创建一个快照，并将当前分支的指针移动到该快照.

另外还有一个 HEAD 指针，它指向当前所在的分支.

切换分支的过程，简单来说就是将 HEAD 指针，从指向当前所在的分支，改为指向另外一个分支. 在这一过程中，Git 会自动完成文件的更新，使得切换分支后仓库的状态与目标分支指向的快照一致.

### 分支的创建

利用 `git branch` 命令可以创建分支，`git switch` 命令可以切换分支，`git switch -c` 命令可以创建分支并切换到这个新分支.

```
$ git switch -c dev # 创建一个叫做 dev 的新分支并切换当前分支到 dev
Switched to branch 'dev'
$ git branch # 查看分支列表
  master
* dev
```

`dev` 前面的星号代表该仓库的当前分支为 `dev`，接下来对这个仓库的更改都将记录在这个分支上.

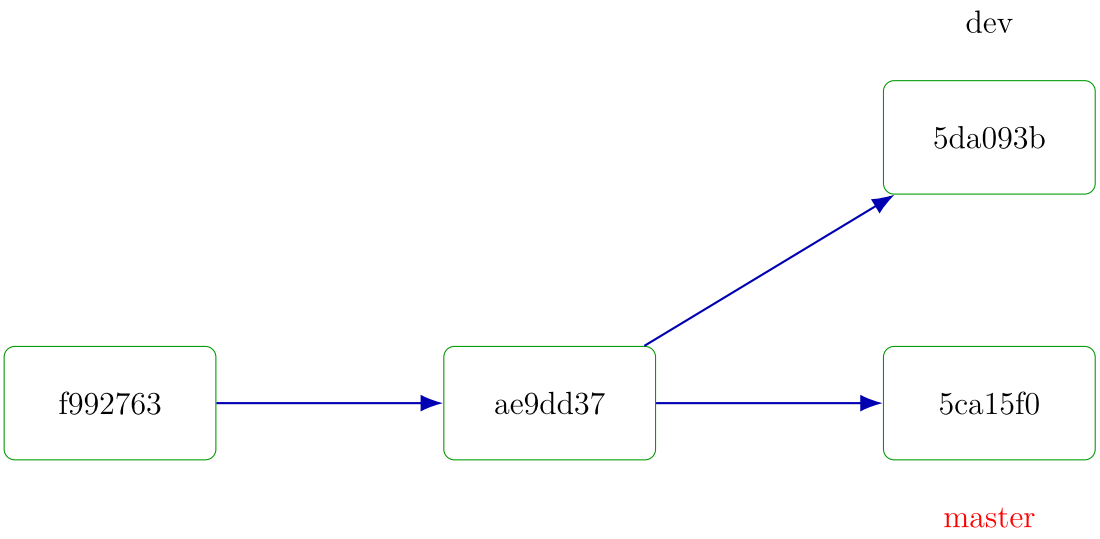
试着创建一个新文件 `aplusb.cpp`.

```
$ vim aplusb.cpp
$ git add aplusb.cpp
$ git commit -m "feat: add A+B Problem code"
[dev 5da093b] feat: add A+B Problem code
 1 file changed, 7 insertions(+)
 create mode 100644 aplusb.cpp
```

现在切换回 `master` 分支，这时候文件夹中没有了 `aplusb.cpp`，一切都回到了刚刚创建 `dev` 分支时的状态. 这时候可以在 `master` 分支上继续完成其他的工作.

```
$ git switch master
Switched to branch 'master'
$ vim README.md # 对 README 做些小改动
$ git commit -a -m "feat: update README.md"
[master 5ca15f0] feat: update README.md
 1 file changed, 1 insertion(+), 1 deletion(-)
```

下面用一张图来解释刚才的操作过程.




`master` 分支被标红，表明在这几次操作后，它是当前分支（即 HEAD 指向的位置）.

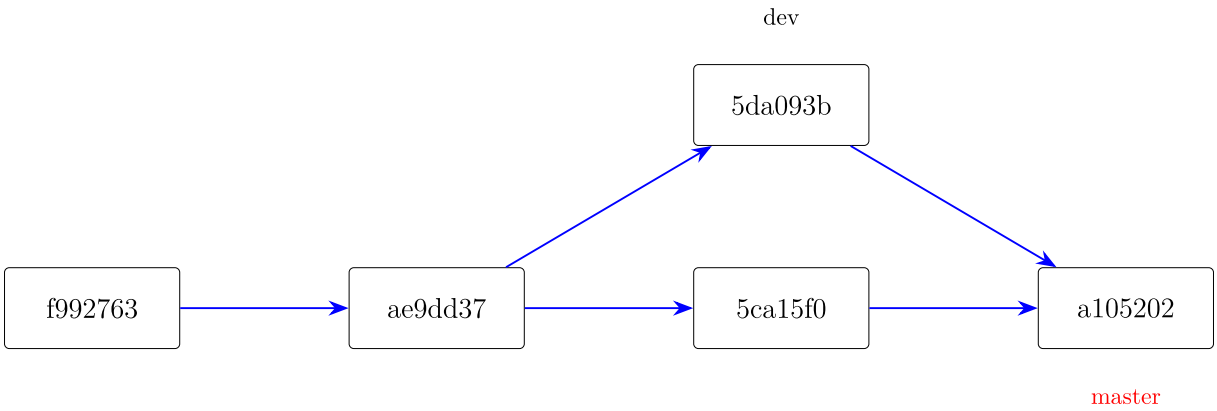
- 最开始时 `master` 指向 `ae9dd37` 这一快照.
- 接下来在 `master` 所在的位置创建了一个新的 dev 分支，该分支一开始和 master 指向相同位置.
- 在 `dev` 分支上作了一些更改（创建了 `aplusb.cpp`），进行了一次提交，本次提交后，`dev` 分支指向 `5da093b` 这一快照.
- 切换回 `master` 分支后，因为 `master` 分支还指向 `ae9dd37`，还没有创建 `aplusb.cpp`，因此仓库中没有这一文件.
- 接下来在 `master` 分支上进行更改（更新了 `README.md`），进行了一次提交，`master` 分支指向了 `5ca15f0` 这一快照.

### 分支的合并

当一个分支上的工作已经完成，就可以将这些工作合并到另外一个分支上去.

还是接着上面这个例子，`dev` 分支的工作已经完成，通过 `git merge` 命令可以将该分支合并到当前分支（`master`）上:

```
$ git merge dev
Merge made by the 'recursive' strategy.
 aplusb.cpp | 7 +++++++
 1 file changed, 7 insertions(+)
 create mode 100644 aplusb.cpp
```




这次合并具体是怎么执行的呢?

在合并之前，`master` 指向 `5ca15f0`，而 `dev` 指向 `5da093b`，这两个状态并不在一条链上.

Git 会找到这两个状态的最近公共祖先（在上图中是 `ae9dd37`），并对这三个快照进行一次合并. 三个快照合并的结果作为一个新的快照，并将当前分支指向这一快照.

合并过程本身也是一次提交，不过与常规提交不同的是，合并提交有不止一个前驱提交，它是多个提交状态合并后的结果.

在合并完成后，`dev` 分支就完成了它的使命，这时候可以利用下面的命令删除 `dev` 分支:

```
$ git branch -d dev # 对于未合并的分支，可以使用 -D 参数强制删除
```

不过合并过程并非总是这么顺利, 在某些情况下, 合并过程可能会出现冲突, 这个问题接下来会讲到.

## 解决合并冲突

如果在两个分支中, 对同一个文件的同一部分进行了不同的更改, Git 就无法自动合并这两个分支, 也就是发生了合并冲突.

接着上面的例子，假如你在合并后的 `master` 分支的基础上，新开了一个 `readme-refactor` 分支，准备重写一份自述文件. 但因为一些疏忽，你同时更改了 `readme-refactor` 和 `master` 分支的自述文件.

刚开始自述文件是这样的:

```
# This is a test repo.

This repo includes some c++ codes.
```

在 `readme-refactor` 分支下的自述文件是这样的:

```
# Code Library

This repo includes some c++ codes.
```

在 `master` 分支下的自述文件是这样的:

```
# This is a code library.

This repo includes some c++ codes.
```

这时候运行 `git merge readme-refactor` 命令，Git 提示出现了合并冲突.

执行一下 `git status` 命令，可以查看是哪些文件引发了冲突.

```
$ git status
```

```
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:      README.md

no changes added to commit (use "git add" and/or "git commit -a")
```

如何解决冲突?对于每个发生了合并冲突的文件, Git 都会在这些文件中加入标准的冲突解决标记. 比如这个例子中的 `README.md` 文件，打开后它长这个样子:

```
<<<<<< HEAD
# This is a code library.
=======
# Code Library
>>>>>> readme-refactor

This repo includes some c++ codes.
```

`=======` 作为分界线将两个分支的内容隔开，`<<<<<< HEAD` 标记和 `=======` 之间的部分是 HEAD 指针（`master` 分支）的内容，而 `=======` 和 `>>>>>> readme-refactor` 标记之间的部分是 `readme-refactor` 分支的内容.

通过编辑文本来处理冲突，删除这些冲突标记，保存文件，将这些文件纳入暂存区后提交，就可以解决合并冲突了.

```
$ git add README.md # 将发生冲突的文件纳入暂存区
$ git commit
[master fe92c6b] Merge branch readme-refactor into master
```

## 其他合并方式

默认情况下，Git 采用 Merge（合并）的方式合并两个分支. 使用该方法将分支 B 并入分支 A 时，会将 B 分支的所有 commit 并入 A 分支的提交历史中.

除此以外，Git 还提供了两种合并分支的方式：Squash（压缩）和 Rebase（变基）.

### Squash（压缩）

使用 Squash 方式将分支 B 并入分支 A 时，在 B 分支上的所有更改会被合并为一次 commit 提交到 A 分支.

在 `git merge` 中加入 `--squash` 参数即可使用 Squash 方式进行分支合并.

```
$ git merge <branch> --squash
```

需要注意的是，在执行上述命令后，Git 只会将 B 分支的所有更改存入 A 分支的缓冲区内，接下来还需要执行一次 `git commit` 命令完成合并工作.

使用 Squash 方式合并可以简化 commit 记录，但是会丢失具体到每一次 commit 的信息（每次 commit 的提交者，每次 commit 的更改等等），只留下合并为一个整体的信息（每次 commit 的提交者会以 "Co-authored-by" 的形式在提交信息中列出）. 但如果是在 GitHub 上进行 Squash and Merge，原有的信息都可以在 Pull Request 中查看.

### Rebase（变基）

使用 Rebase 方式将分支 B 并入分支 A 时，在 B 分支上的每一次 commit 都会单独添加到 A 分支，而不再像 Merge 方式那样创建一个合并 commit 来合并两个分支的内容[2].

首先，切换到 B 分支，接下来将 B 分支变基到 A 分支：

```
$ git checkout B
$ git rebase A
```

现在切回到 A 分支，再执行一次 `git merge` 命令，即可完成将 B 分支的内容合并到 A 分支的工作.

```
$ git checkout A
$ git merge B
```

使用 Rebase 完成合并可以让提交历史线性化，在适当的场景下正确地使用 Rebase 可以达到比 Merge 更好的效果. 但是这样做会改变提交历史，在进行 Rebase 时和 Rebase 后再进行相关合并操作时都会增加出现冲突的可能，如果操作不当可能反而会使提交历史变得杂乱. 因此，如果对 Rebase 操作没有充分的了解，不建议使用.

## 管理远程仓库

在本地完成更改后，你可能会需要将这些更改推送到 GitHub 等 Git 仓库托管平台上. 托管在这些平台上的仓库就归属于远程仓库的范畴——你可以从这些仓库中获取信息，也可以将你作出的更改推送到远程仓库上. 与其他人的协作往往离不开远程仓库，因此学会管理远程仓库很有必要.

### 远程仓库的查看

使用 `git remote` 命令可以查看当前仓库的远程仓库列表.

如果当前仓库是克隆来的，那么应该会有一个叫做 origin 的远程仓库，它的链接就是克隆时用的链接.

```
$ git remote
origin
```

如果要查看某个远程仓库的详细信息的话，可以这样操作：

```
$ git remote show origin
* remote origin
  Fetch URL: git@github.com:OI-wiki/OI-wiki.git
  Push  URL: git@github.com:OI-wiki/OI-wiki.git
  HEAD branch: master
  Remote branches:
    git             tracked
    master          tracked
  ...
```

### 远程仓库的配置

执行 `git remote add <name> <url>` 命令可以添加一个名字为 `name`，链接为 `url` 的远程仓库.

执行 `git remote rename <oldname> <newname>` 可以将名字为 `oldname` 的远程仓库改名为 `newname`.

执行 `git remote rm <name>` 可以删除名字为 `name` 的远程仓库.

执行 `git remote get-url <name>` 可以查看名字为 `name` 的远程仓库的链接.

执行 `git remote set-url <name> <newurl>` 可以将名字为 `name` 的远程仓库的链接更改为 `newurl`.

[2] Pro Git Book[3] 中提供了可视化的 Rebase 过程图，借助图片读者可以更好地理解 Rebase 的机制.

### 从远程仓库获取更改

在远程仓库中，其他人可能会推送一些更改，执行 `git fetch` 命令可以将这些更改获取到本地.

```
$ git fetch <remote-name> # 获取 <remote-name> 的更改
```

需要注意的是，`git fetch` 命令只会获取远程仓库的更改，而不会将这些更改合并到本地仓库中. 如果需要将这些更改进行合并，可以使用 `git pull` 命令. 在默认情况下，`git pull` 相当于 `git fetch` 后 `git merge FETCH_HEAD`.

```
$ git pull <remote-name> <branch> # 获取 <remote-name> 的更改，然后将这些更改合并到 HEAD
```

### 将更改推送到远程仓库

当你完成了一些更改之后，使用 `git push` 命令可以将这些更改推送到远程仓库.

```
$ git push <remote> <from>:<to> # 将本地 <from> 分支的更改推送至 <remote> 的 <to> 分支
```

根据远程仓库的要求，可能会要求你输入远程仓库账户的用户名和密码.

需要注意的是，你的更改能成功推送，需要满足两个条件：你拥有向这个仓库（分支）的写入权限，且你的这个分支比远程仓库的相应分支新（可以理解为没有人在你进行更改的这段时间进行了推送）. 当远程分支有当前分支没有的新更改时，可以执行 `git pull` 命令完成合并再提交.

如果你需要强制将本地分支的更改推送到远程仓库的话，可以加入 `-f` 参数. 此时 **远程仓库的提交历史会被本地的提交历史覆盖**，因此该命令应谨慎使用. 更好的选择是使用 `--force-with-lease` 参数，该参数仅在远程仓库没有更新时才会进行覆盖. 需要注意的是，此处「更新」是相对于上一次 fetch 而言的，如果使用了 VS Code 提供的 Auto Fetch 功能，可能会没有注意到更新而使 `--force-with-lease` 和 `-f` 一样危险.

### 追踪远程分支

通过将一个本地分支设定为追踪远程分支，可以方便地查看本地分支与远程分支的差别，并能简化与远程分支交互时的操作.

在开始追踪前，你需要先执行 `git fetch <remote-name>` 将远程仓库的信息抓取到本地.

接下来执行 `git switch <remote-branch>`，会在本地自动创建名字为 `<remote-branch>` 的新分支，并设定该分支自动追踪相应的远程分支.

> **Note**
>
> 需要注意，只有当本地不存在该分支，且恰好只有一个远程分支的名字与该分支匹配时，Git 才会自动创建该分支且设定其追踪相应的远程分支.

这时候执行 `git status` 命令，会提示当前分支与远程分支之间的差别.

因为设定了本地分支追踪的远程分支，向远程分支推送的命令也被简化了. 只需要执行 `git push` 命令，在本地分支上作出的更改就能被推送至其追踪的远程分支.

对于本地已有的分支，设定其对应的远程追踪分支也很容易. 只需在当前分支下执行 `git branch -u <remote-name>/<remote-branch>`，就可以设定当前的本地分支追踪 `<remote-name>/<remote-branch>` 这一远程分支.

### 使用 ssh 连接

与 HTTP(S) 相比，使用 ssh 连接到远程仓库更为方便安全.

在使用 ssh 连接到远程仓库之前，需要先在本地添加 ssh 密钥．接下来需要将本地添加的 ssh 密钥的**公钥**上传到远程仓库账户．

考虑到本文主要是给 **OI Wiki** 的贡献者提供一个使用 Git 的教程，这里直接给出 GitHub Docs 提供的教程[4]，供各位读者参考．

完成以上步骤后，你就可以通过 ssh 连接到远程仓库了．下面就是一条通过 ssh 连接 clone **OI Wiki** 仓库的命令：

```
$ git clone git@github.com:OI-wiki/OI-wiki.git
```

将更改推送至远程仓库的过程与使用 HTTP(S) 连接类似．但使用 ssh 连接可以免去验证远程仓库账号密码的过程．

## Git GUI Tools

对于不熟悉命令行的同学，纯命令行的 Git 的上手难度可能会偏高，而借助 GUI 工具可以一定程度上降低 Git 的上手难度．此外，相比于命令行，GUI 工具在查看 diff 以及 log 时在体验上有一定程度的提高．

Git 本身自带有 GUI，市面上也有很多优秀的 Git GUI 工具，例如针对 Windows 用户的 TortoiseGit[3]，支持 Windows 和 Mac 的 Sourcetree[4]等．

这里简单介绍一下 TortoiseGit 的使用．下载并安装好 TortoiseGit 之后，在本地仓库的目录下，单击鼠标右键，在右键菜单中就可以看到 Git 的各个功能．

[3] TortoiseGit[5]
[4] Sourcetree[6]


Pull...
Fetch...
Push...
Diff
Diff with previous version
Show log
Show RefLog
Browse References
Daemon
Revision graph
Repo-browser
Check for modifications
Rebase...
Stash changes
Stash Pop
Stash List
Bisect start
Resolve...
Rename...
Delete
Delete (keep local)
Revert...
Clean up...
Switch/Checkout...
Merge...
Create Branch...
Create Tag...
Export...
Add...
Submodule Add...
Create Patch Serial...
Apply Patch Serial...
Settings
Help
About
查看(V)
排序方式(O)
分组依据(P)
刷新(E)
自定义文件夹(F)...
粘贴(P)
粘贴快捷方式(S)
撤消 重命名(U) Ctrl+Z
在 Windows 终端 中打开
Git GUI Here
Git Bash Here
授予访问权限(G)
Git Sync...
Git Commit -> "git"...
TortoiseGit
新建(W)
属性(R)

详细的使用方法这里不再赘述，可以参考官网里的使用文档或者通过搜索引擎学习，例如 TortoiseGit Manual[7].

很多 GUI 工具都有官方中文支持，例如 Git Desktop 以及 TortoiseGit. 但是还是会有部分翻译看起来较为变扭，推荐使用英文版本.

## 外部链接

- Git Reference[8]
- Pro Git Book[9]
- Learn Git Branching[10]

## 参考资料与注释

[1] GitHub 帮助 

[2] Git - Downloads 

[3] Pro Git Book 

[4] GitHub Docs 提供的教程 

[5] TortoiseGit 

[6] Sourcetree 

[7] TortoiseGit Manual 

[8] Git Reference 

[9] Pro Git Book 

[10] Learn Git Branching 

# 第四章

# 语言基础

# 4.1 语言基础简介

本章将会介绍编程相关的知识，包括 C++ 从入门到进阶教程和一些其它语言的简介.

程序是算法与数据结构的载体，是解决 OI 问题的工具.

在 OI 中，最常用的编程语言是 C++.

学习编程是学习 OI 最基础的部分.

# 4.2 C++ 基础

## 4.2.1 Hello, World!

### 环境配置

工欲善其事，必先利其器.

### 集成开发环境

IDE 操作较为简单，一般入门玩家会选用 IDE 来编写代码. 在竞赛中最常见的是 Dev-C++→3.2.8@p171（如果考试环境是 Windows 系统，一般也会提供这一 IDE）.

### 编译器

#### Windows

推荐使用 GNU 编译器. 需要去 MinGW Distro[1] 下载 MinGW 并安装. 此外 Windows 下也可以选择 Microsoft Visual C++ 编译器[2]，需要去 Visual Studio 页面[3] 下载安装.

#### macOS

在终端中执行：

```
xcode-select --install
```

#### Linux

使用 `g++ -v` 来检查是否安装过 `g++`.

使用如下命令可以安装：

```
sudo apt update && sudo apt install g++
```

### 在命令行中编译代码

熟练之后也有玩家会使用更灵活的命令行来编译代码，这样就不依赖 IDE 了，而是使用自己熟悉的文本编辑器编写代码.

```
g++ test.cpp -o test -lm
```

`g++` 是 C++ 语言的编译器（C 语言的编译器为 `gcc`），`-o` 用于指定可执行文件的文件名，编译选项 `-lm` 用于链接数学库 `libm`，从而使得使用 `math.h` 的代码可以正常编译运行.

注：C++ 程序不需要 `-lm` 即可正常编译运行. 历年 NOI/NOIP 试题的 C++ 编译选项中都带着 `-lm`，故这里也一并加上.

### 第一份代码

通过这样一个示例程序来展开 C++ 入门之旅吧～

注：请在编写前注意开启英文输入法.

C++ 语言

```cpp
#include <iostream>  // 引用头文件

int main() {                      // 定义 main 函数
  std::cout << "Hello, world!";  // 使用标准命名空间中的 cout 函数
  return 0;  // 返回 0，结束 main 函数. 编译器一般会自动加上这一行，一般可以省略
}
```

C 语言

```c
#include <stdio.h>  // 引用头文件

int main() {                // 定义 main 函数
  printf("Hello, world!");  // 输出 Hello, world!
  return 0;                 // 返回 0，结束 main 函数
}
```

注意：C 语言在这里仅做参考，C++ 基本兼容 C 语言，并且拥有许多新的功能，可以让选手在赛场上事半功倍. 具体请见 C++ 与其他常用语言区别→4.5@p470

### 外部参考资料

[1] MinGW Distro

[2] Microsoft Visual C++ 编译器

[3] Visual Studio 页面

## 4.2.2 C++ 语法基础

### 代码框架

如果你不想深究背后的原理，初学时可以直接将这个「框架」背下来：

```cpp
#include <cstdio>
#include <iostream>

int main() {
  // do something...
  return 0;
}
```

**什么是 include?**

`#include` 其实是一个预处理命令，意思为将一个文件「放」在这条语句处，被「放」的文件被称为头文件. 也就是说，在编译时，编译器会「复制」头文件 `iostream` 中的内容，「粘贴」到 `#include <iostream>` 这条语句处. 这样，你就可以使用 `iostream` 中提供的 `std::cin`、`std::cout`、`std::endl` 等对象了.

如果你学过 C 语言，你会发现目前我们接触的 C++ 中的头文件一般都不带 `.h` 后缀，而那些 C 语言中的头文件 `xx.h` 都变成了 `cxx`，如 `stdio.h` 变成了 `cstdio`. 因为 C++ 为了和 C 保持兼容，都直接使用了 C 语言中的头文件，为了区分 C++ 的头文件和 C 的头文件，使用了 `c` 前缀.

一般来说，应当根据你需要编写的 C++ 程序的需要来确定你要 `#include` 哪些头文件. 但如果你 `#include` 了多余的头文件，只会增加编译时间，几乎不会对运行时间造成影响. 目前我们只接触到了 `iostream` 和 `cstdio` 两个头文件，如果你只需要 `scanf` 和 `printf`，就可以不用 `#include <iostream>`.

可以 `#include` 自己写的头文件吗？答案是，可以.

你可以自己写一个头文件，如：`myheader.h`．然后，将其放到和你的代码相同的目录里，再 `#include "myheader.h"` 即可．需要注意的是，自定义的头文件需要使用引号而非尖括号．当然，你也可以使用编译命令 `-I <header_file_path>` 来告诉编译器在哪找头文件，就不需要将头文件放到和代码相同的目录里了．

**什么是 `main()`?**

可以理解为程序运行时就会执行 `main()` 中的代码．

实际上，`main` 函数是由系统或外部程序调用的．如，你在命令行中调用了你的程序，也就是调用了你程序中的 `main` 函数（在此之前先完成了全局 变量→4.2.3@p328 的构造）．

最后的 `return 0;` 表示程序运行成功．默认情况下，程序结束时返回 0 表示一切正常，否则返回值表示错误代码（在 Windows 下这个错误代码的十六进制可以通过 Windows Error Codes 网站[1] 进行查询）．这个值返回给谁呢？其实就是调用你写的程序的系统或外部程序，它会在你的程序结束时接收到这个返回值．如果不写 `return` 语句的话，程序正常结束默认返回值也是 0．

在 C 或 C++ 中，程序的返回值不为 0 会导致运行时错误（RE）．

## 注释

在 C++ 代码中，注释有两种写法：

1. 行内注释

   以 `//` 开头，行内位于其后的内容全部为注释．

2. 注释块

   以 `/*` 开头，`*/` 结尾，中间的内容全部为注释，可以跨行．

注释对程序运行没有影响，可以用来解释程序的意思，还可以在让某段代码不执行（但是依然保留在源文件里）．

在工程开发中，注释可以便于日后维护、他人阅读．

在 OI 中，很少有人写许多注释，但注释可以便于在写代码的时候理清思路，或者便于日后复习．而且，如果要写题解、教程的话，适量的注释可以便于读者阅读，理解代码的意图．希望各位同学能养成写注释的好习惯．

## 输入与输出

### `cin` 与 `cout`

```cpp
#include <iostream>

int main() {
  int x, y;                          // 声明变量
  std::cin >> x >> y;                // 读入 x 和 y
  std::cout << y << std::endl << x;  // 输出 y，换行，再输出 x
  return 0;                          // 结束主函数
}
```

**什么是变量?**

可以参考 变量→4.2.3@p328 页面．

**什么是 std?**

std 是 C++ 标准库所使用的 **命名空间**. 使用命名空间是为了避免重名.

关于命名空间的详细知识, 可以参考 命名空间→4.4.2@p413 页面.

### scanf 与 printf

scanf 与 printf 其实是 C 语言提供的函数. 大多数情况下, 它们的速度比 cin 和 cout 更快, 并且能够方便地控制输入输出格式.

**读入输出优化**

cin/cout 和 scanf/prinf 的具体差别和读入输出优化, 请参考 读入、输出优化→2.6.1@p103 页面.

```cpp
#include <cstdio>

int main() {
  int x, y;
  scanf("%d%d", &x, &y);   // 读入 x 和 y
  printf("%d\n%d", y, x);  // 输出 y，换行，再输出 x
  return 0;
}
```

其中, %d 表示读入/输出的变量是一个有符号整型（int 型）的变量.

类似地:

1. %s 表示字符串.
2. %c 表示字符.
3. %lf 表示双精度浮点数 (double).
4. %lld 表示长整型 (long long). 根据系统不同, 也可能是 %I64d.
5. %u 表示无符号整型 (unsigned int).
6. %llu 表示无符号长整型 (unsigned long long), 也可能是 %I64u.

除了类型标识符以外, 还有一些控制格式的方式. 许多都不常用, 选取两个常用的列举如下:

1. %ld 表示长度为 1 的整型. 在读入时, 即使没有空格也可以逐位读入数字. 在输出时, 若指定的长度大于数字的位数, 就会在数字前用空格填充. 若指定的长度小于数字的位数, 就没有效果.
2. %.6lf, 用于输出, 保留六位小数.

这两种运算符的相应地方都可以填入其他数字, 例如 %.3lf 表示保留三位小数.

**「双精度浮点数」,「长整型」是什么**

这些表示变量的类型. 和上面一样, 会留到 变量→4.2.3@p328 中统一讲解.

**为什么 scanf 中有 & 运算符?**

在这里, & 实际上是取址运算符, 返回的是变量在内存中的地址. 而 scanf 接收的参数就是变量的地址. 具体可能要在 指针→4.2.6.4@p356 才能完全清楚地说明, 现在只需要记下来就好了.

**什么是 \n?**

\n 是一种 **转义字符**, 表示换行.

转义字符用来表示一些无法直接输入的字符，如由于字符串字面量中无法换行而无法直接输入的换行符，由于有特殊含义而无法输入的引号，由于表示转义字符而无法输入的反斜杠.

常用的转义字符有：

1. `\t` 表示制表符.
2. `\\` 表示 `\`.
3. `\"` 表示 `"`.
4. `\0` 表示空字符，用来表示 C 风格字符串的结尾.
5. `\r` 表示回车. Linux 中换行符为 `\n`，Windows 中换行符为 `\r\n`. 在 OI 中，如果输出需要换行，使用 `\n` 即可. 但读入时，如果使用逐字符读入，可能会由于换行符造成一些问题，需要注意. 例如，`gets` 将 `\n` 作为字符串结尾，这时候如果换行符是 `\r\n`，`\r` 就会留在字符串结尾.
6. 特殊地，`%%` 表示 `%`，只能用在 `printf` 或 `scanf` 中，在其他字符串字面量中只需要简单使用 `%` 就好了.

**什么是字面量?**

「字面量」是在代码里直接作为一个值的程序段，例如 `3` 就是一个 `int` 字面量，`'c'` 就是一个 char 字面量. 我们上面写的程序中的 `"hello world"` 也是一个字符串字面量.

不加解释、毫无来由的字面量又被称为「魔术数」(magic number)，如果代码需要被人阅读的话，这是一种十分不被推荐的行为.

## 一些扩展内容

### C++ 中的空白字符

在 C++ 中，所有空白字符（空格、制表符、换行），多个或是单个，都被视作是一样的.（当然，引号中视作字符串的一部分的不算.）

因此，你可以自由地使用任何代码风格（除了行内注释、字符串字面量与预处理命令必须在单行内），例如：

```cpp
#include <iostream>

 int

    main(){
int/**/x, y;  std::cin
>> x >>y;
                std::cout <<
          y  <<std::endl
     << x

          ;

    return       0;     }
```

当然，这么做是不被推荐的.

一种也被广泛使用但与 OI Wiki 要求的码风不同的代码风格：

```cpp
#include <iostream>

int main()
```

```
{
    int x, y;

    std::cin >> x >> y;
    std::cout << y << std::endl << x;

    return 0;
}
```

### `#define` 命令

`#define` 是一种预处理命令，用于定义宏，本质上是文本替换．例如：

```
#include <iostream>
#define n 233

// n 不是变量，而是编译器会将代码中所有 n 文本替换为 233，但是作为标识符一部分的
// n 的就不会被替换，如 fn 不会被替换成 f233，同样，字符串内的也不会被替换

int main() {
  std::cout << n;  // 输出 233
  return 0;
}
```

**什么是标识符?**

标识符就是可以用作变量名的一组字符．例如，`abcd` 和 `abc1` 都是合法的标识符，而 `1a` 和 `c+b` 都不是合法的标识符．

标识符由英文字母、下划线开头，中间只允许出现英文字母、下划线和数字．值得注意的是，关键字（如 `int`,`for`,`if`）不能用作标识符．

**什么是预处理命令?**

预处理命令就是预处理器所接受的命令，用于对代码进行初步的文本变换，比如 文件包含操作 `#include` 和 处理宏 `#define` 等，对 GCC 而言，默认不会保留预处理阶段的输出 `.i` 文件．可以用 `-E` 选项保留输出文件．

宏可以带参数，带参数的宏可以像函数一样使用：

```
#include <iostream>
#define sum(x, y) ((x) + (y))
#define square(x) ((x) * (x))

int main() {
  std::cout << sum(1, 2) << ' ' << 2 * sum(3, 5) << std::endl;  // 输出 3 16
}
```

但是带参数的宏和函数有区别．因为宏是文本替换，所以会引发许多问题．如：

```
#include <iostream>
#define sum(x, y) x + y
// 这里应当为 #define sum(x, y) ((x) + (y))
#define square(x) ((x) * (x))

```

```cpp
int main() {
  std::cout << sum(1, 2) << ' ' << 2 * sum(3, 5) << std::endl;
  // 输出为 3 11，因为 #define 是文本替换，后面的语句被替换为了 2 * 3 + 5
  int i = 1;
  std::cout << square(++i) << ' ' << i;
  // 输出未定义，因为 ++i 被执行了两遍
  // 而同一个语句中多次修改同一个变量是未定义行为（有例外）
}
```

使用 `#define` 是有风险的（由于 `#define` 作用域是整个程序，因此可能导致文本被意外地替换，需要使用 `#undef` 及时取消定义），因此应谨慎使用．较为推荐的做法是：使用 `const` 限定符声明常量，使用函数代替宏．

但是，在 OI 中，`#define` 依然有用武之处（以下两种是不被推荐的用法，会降低代码的规范性）：

1. `#define int long long+signed main()`．通常用于避免忘记开 long long 导致的错误，或是调试时排除忘开 long long 导致错误的可能性．（也可能导致增大常数甚至 TLE，或者因为爆空间而 MLE）
2. `#define For(i, l, r) for (int i = (l); i <= (r); ++i)`、`#define pb push_back`、`#define mid ((l + r) / 2)`，用于减短代码长度．

不过，`#define` 也有优点，比如结合 `#ifdef` 等预处理指令有奇效，比如：

```cpp
#ifdef LINUX
// code for linux
#else
// code for other OS
#endif
```

可以在编译的时候通过 `-DLINUX` 来控制编译出的代码，而无需修改源文件．这还有一个优点：通过 `-DLINUX` 编译出的可执行文件里并没有其他操作系统的代码，那些代码在预处理的时候就已经被删除了．

`#define` 还能使用 `#`、`##` 运算符，极大地方便调试．

### 外部参考资料

[1] Windows Error Codes 网站 

## 4.2.3 变量

### 数据类型

C++ 的类型系统由如下几部分组成：

1. 基础类型（括号内为代表关键词/代表类型）
    1. 无类型/`void` 型（`void`）
    2. （C++11 起）空指针类型（`std::nullptr_t`）
    3. 算术类型
        1. 整数类型（`int`）
        2. 布尔类型/`bool` 型（`bool`）
        3. 字符类型（`char`）
        4. 浮点类型（`float`,`double`）
2. 复合类型[1]

[1]包括数组类型、引用类型、指针类型、类类型、函数类型等．由于本篇文章是面向初学者的，故不在本文做具体介绍．具体请参阅 类型 - cppreference.com[1]

### 布尔类型

一个 `bool` 类型的变量取值只可能为两种：`true` 和 `false`.

一般情况下，一个 `bool` 类型变量占有 1 字节（一般情况下，1 字节 =8 位）的空间.

**Tip**

可通过头文件 `<climits>`(C++)/`<limits.h>`(C) 中的宏常量 `CHAR_BIT` 获取字节的位数.

**C 语言的布尔类型**

另请参阅 C++ 与其他常用语言的区别 - bool→4.5@p470.

C 语言最初是没有布尔类型的，直到 C99 时才引入 `_Bool` 关键词作为布尔类型，其被视作无符号整数类型.

**Note**

C 语言的 `bool` 类型从 C23 起不再使用整型的零与非零值定义，而是定义为足够储存 `true` 和 `false` 两个常量的类型.

为方便使用，`stdbool.h` 中提供了 `bool`,`true`,`false` 三个宏，定义如下：

```
#define bool _Bool
#define true 1
#define false 0
```

这些宏于 C23 中移除，并且 C23 起引入 `true`,`false` 和 `bool` 作为关键字，同时保留 `_Bool` 作为替代拼写形式[2].

另外，C23 起还可以通过 `<limits.h>` 中的宏常量 `BOOL_WIDTH` 获取布尔类型的位宽.

### 整数类型

用于存储整数. 最基础的整数类型是 `int`.

**注意**

由于历史原因，C++ 中布尔类型和字符类型会被视作特殊的整型.

在几乎所有的情况下都 **不应该** 将除 `signed char` 和 `unsigned char` 之外的字符类型作为整型使用.

整数类型一般按位宽有 5 个梯度：`char`,`short`,`int`,`long`,`long long`.

C++ 标准保证 `1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)`

由于历史原因，整数类型的位宽有多种流行模型，为解决这一问题，C99/C++11 引入了 定宽整数类型→4@p321.

**`int` 类型的大小**

在 C++ 标准中，规定 `int` 的位数 **至少** 为 16 位.

事实上在现在的绝大多数平台，`int` 的位数均为 32 位.

[2]参见 https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3054.pdf[2]

对于 `int` 关键字，可以使用如下修饰关键字进行修饰：

符号性：

- `signed`：表示带符号整数（默认）；
- `unsigned`：表示无符号整数.

大小：

- `short`：表示 **至少** 16 位整数；
- `long`：表示 **至少** 32 位整数；
- （C++11 起）`long long`：表示 **至少** 64 位整数.

下表给出在 **一般情况下**，各整数类型的位宽和表示范围大小（少数平台上一些类型的表示范围可能与下表不同）：

| 类型名 | 等价类型 | 位宽（C++ 标准） | 位宽（常见） | 位宽（较罕见） |
|---|---|---|---|---|
| `signed char` | `signed char` | 8 | - | - |
| `unsigned char` | `unsigned char` | 8 | - | - |
| `short`,`short int`,`signed short`,`signed short int` | `short int` | $\geq 16$ | 16 | - |
| `unsigned short`,`unsigned short int` | `unsigned short int` | $\geq 16$ | 16 | - |
| `int`,`signed`,`signed int` | `int` | $\geq 16$ | 32 | 16（常见于 Win16 API） |
| `unsigned`,`unsigned int` | `unsigned int` | $\geq 16$ | 32 | 16（常见于 Win16 API） |
| `long`,`long int`,`signed long`,`signed long int` | `long int` | $\geq 32$ | 32 | 64（常见于 64 位 Linux、macOS） |
| `unsigned long`,`unsigned long int` | `unsigned long int` | $\geq 32$ | 32 | 64（常见于 64 位 Linux、macOS） |
| `long long`,`long long int`,`signed long long`,`signed long long int` | `long long int` | $\geq 64$ | 64 | - |
| `unsigned long long`,`unsigned long long int` | `unsigned long long int` | $\geq 64$ | 64 | - |

当位宽为 $x$ 时，有符号类型的表示范围为 $-2^{x-1} \sim 2^{x-1}-1$[3]，无符号类型的表示范围为 $0 \sim 2^{x}-1$. 具体而言，有下表：

| 位宽 | 表示范围 |
|---|---|
| 8 | 有符号：$-2^{7} \sim 2^{7}-1$，无符号：$0 \sim 2^{8}-1$ |
| 16 | 有符号：$-2^{15} \sim 2^{15}-1$，无符号：$0 \sim 2^{16}-1$ |
| 32 | 有符号：$-2^{31} \sim 2^{31}-1$，无符号：$0 \sim 2^{32}-1$ |
| 64 | 有符号：$-2^{63} \sim 2^{63}-1$，无符号：$0 \sim 2^{64}-1$ |

[3] C++20 前规定有符号整数至少要覆盖 反码→9.4@p1099 的表示范围（即 $-2^{x-1}+1 \sim 2^{x-1}-1$），但实际上绝大多数实现中均采用 补码→9.4@p1099 实现；C++20 起进一步规定有符号整数必须使用补码实现. 详见 Range of values - cppreference[3].

**等价的类型表述**

在不引发歧义的情况下，允许省略部分修饰关键字，或调整修饰关键字的顺序. 这意味着同一类型会存在多种等价表述.

例如 `int`，`signed`，`int signed`，`signed int` 表示同一类型，而 `unsigned long` 和 `unsigned long int` 表示同一类型.

另外，一些编译器实现了扩展整数类型，如 GCC 实现了 128 位整数：有符号版的 `__int128_t` 和无符号版的 `__uint128_t`，如果您在比赛时想使用这些类型，**请仔细阅读比赛规则** 以确定是否允许或支持使用扩展整数类型.

**注意**

STL 不一定对扩展整数类型有足够的支持，故使用扩展整数类型时需格外小心.

**示例代码**

```cpp
#include <cmath>
#include <iostream>

int f1(int n) {
  return abs(n);  // Good
}

int f2(int n) {
  return std::abs(n);  // Good
}

__int128_t f3(__int128_t n) {
  return abs(n);  // Bad
}

// Wrong
// __int128_t f4(__int128_t n) {
//   return std::abs(n);
// }

int main() {
  std::cout << "f1: " << f1(-42) << std::endl;
  std::cout << "f2: " << f2(-42) << std::endl;
  // std::cout << "f3: " << f3(-42) << std::endl; // Wrong
  // std::cout << "f4: " << f4(-42) << std::endl; // Wrong
  return 0;
}
```

以上示例代码存在如下问题：

1. `__int128_t f3(__int128_t)` 中使用的是 C 风格的绝对值函数，其签名为 `int abs(int)`，故 `n` 首先会强制转换为 `int`，然后才会调用 `abs` 函数.
2. `__int128_t f4(__int128_t)` 中使用的是 C++ 风格的绝对值函数，其并没有签名为 `__int128_t std::abs(__int128_t)` 的函数重载，所以无法通过编译.
3. C++ 的流式输出不支持 `__int128_t` 与 `__uint128_t`.

以下是一种解决方案：

**修正后的代码**

```cpp
#include <cmath>
#include <iostream>

__int128_t abs(__int128_t n) { return n < 0 ? -n : n; }

std::ostream &operator<<(std::ostream &os, __uint128_t n) {
  if (n > 9) os << n / 10;
  os << (int)(n % 10);
  return os;
}

std::ostream &operator<<(std::ostream &os, __int128_t n) {
  if (n < 0) {
    os << '-';
    n = -n;
  }
  return os << (__uint128_t)n;
}

int f1(int n) { return abs(n); }

int f2(int n) { return std::abs(n); }

__int128_t f3(__int128_t n) { return abs(n); }

int main() {
  std::cout << "f1: " << f1(-42) << std::endl;
  std::cout << "f2: " << f2(-42) << std::endl;
  std::cout << "f3: " << f3(-42) << std::endl;
}
```

### 字符类型

分为「窄字符类型」和「宽字符类型」，由于算法竞赛几乎不会用到宽字符类型，故此处仅介绍窄字符类型.

窄字符型位数一般为 8 位，实际上底层存储方式仍然是整数，一般通过 ASCII 编码[4] 实现字符与整数的一一对应，有如下三种：

- `signed char`：有符号字符表示的类型，表示范围在 $-128 \sim 127$ 之间.
- `unsigned char`：无符号字符表示的类型，表示范围在 $0 \sim 255$ 之间.
- `char` 拥有与 `signed char` 或 `unsigned char` 之一相同的表示和对齐，但始终是独立的类型.

  `char` 的符号性取决于编译器和目标平台：ARM 和 PowerPC 的默认设置通常没有符号，而 x86 与 x64 的默认设置通常有符号.

  GCC 可以在编译参数中添加 `-fsigned-char` 或 `-funsigned-char` 指定将 `char` 视作 `signed char` 或 `unsigned char`，其他编译器请参照文档. 需要注意指定与架构默认值不同的符号有可能会破坏 ABI，造成程序无法正常工作.

> ⚠ **注意**
>
> 与其他整型不同，`char`、`signed char`、`unsigned char` 是 **三种不同的类型**.
>
> 一般来说 `signed char`,`unsigned char` 不应用来存储字符，绝大多数情况下，这两种类型均被视作整数类型.

### 浮点类型

用于存储「实数」(注意并不是严格意义上的实数，而是实数在一定规则下的近似)，包括以下三种:

- `float`: 单精度浮点类型. 如果支持就会匹配 IEEE-754 binary32 格式.
- `double`: 双精度浮点类型. 如果支持就会匹配 IEEE-754 binary64 格式.
- `long double`: 扩展精度浮点类型. 如果支持就会匹配 IEEE-754 binary128 格式，否则如果支持就会匹配 IEEE-754 binary64 扩展格式，否则匹配某种精度优于 binary64 而值域至少和 binary64 一样好的非 IEEE-754 扩展浮点格式，否则匹配 IEEE-754 binary64 格式.

| 浮点格式 | 位宽 | 最大正数 | 精度位数 |
|---|---|---|---|
| IEEE-754 binary32 格式 | 32 | $3.4 \times 10^{38}$ | $6 \sim 9$ |
| IEEE-754 binary64 格式 | 64 | $1.8 \times 10^{308}$ | $15 \sim 17$ |
| IEEE-754 binary64 扩展格式 | $\geq 80$ | $\geq 1.2 \times 10^{4932}$ | $\geq 18 \sim 21$ |
| IEEE-754 binary128 格式 | 128 | $1.2 \times 10^{4932}$ | $33 \sim 36$ |

> IEEE-754 浮点格式的最小负数是最大正数的相反数.

因为 `float` 类型表示范围较小，且精度不高，实际应用中常使用 `double` 类型表示浮点数.

另外，浮点类型可以支持一些特殊值:

- 无穷（正或负）: `INFINITY`.
- 负零: `-0.0`，例如 `1.0 / 0.0 == INFINITY`,`1.0 / -0.0 == -INFINITY`.
- 非数（NaN）: `std::nan`,`NAN`，一般可以由 `0.0 / 0.0` 之类的运算产生. 它与任何值（包括自身）比较都不相等，C++11 后可以 使用 `std::isnan` 判断一个浮点数是不是 NaN.

### 无类型

`void` 类型为无类型，与上面几种类型不同的是，不能将一个变量声明为 `void` 类型. 但是函数的返回值允许为 `void` 类型，表示该函数无返回值.

### 空指针类型

请参阅指针的 对应章节→4.2.6.4@p356

### 定宽整数类型

C++11 起提供了定宽整数的支持，具体如下:

- `<cstdint>`: 提供了若干定宽整数的类型和各定宽整数类型最大值、最小值等的宏常量.
- `<cinttypes>`: 为定宽整数类型提供了用于 `std::fprintf` 系列函数和 `std::fscanf` 系列函数的格式宏常量.

定宽整数有如下几种:

- `intN_t`: 宽度 **恰为** $N$ 位的有符号整数类型，如 `int32_t`.
- `int_fastN_t`: 宽度 **至少** 有 $N$ 位的 **最快的** 有符号整数类型，如 `int_fast32_t`.
- `int_leastN_t`: 宽度 **至少** 有 $N$ 位的 **最小的** 有符号整数类型，如 `int_least32_t`.

无符号版本只需在有符号版本前加一个字母 u 即可，如 `uint32_t`,`uint_least8_t`.

标准规定必须实现如下 16 种类型:

`int_fast8_t`,`int_fast16_t`,`int_fast32_t`,`int_fast64_t`,

`int_least8_t`,`int_least16_t`,`int_least32_t`,`int_least64_t`,

`uint_fast8_t`,`uint_fast16_t`,`uint_fast32_t`,`uint_fast64_t`,

`uint_least8_t`,`uint_least16_t`,`uint_least32_t`,`uint_least64_t`.

绝大多数编译器在此基础上都实现了如下 8 种类型：

`int8_t`,`int16_t`,`int32_t`,`int64_t`,

`uint8_t`,`uint16_t`,`uint32_t`,`uint64_t`.

在实现了对应类型的情况下，C++ 标准规定必须实现表示对应类型的最大值、最小值、位宽的宏常量，格式为将类型名末尾的 `_t` 去掉后转大写并添加后缀：

- `_MAX` 表示最大值，如 `INT32_MAX` 即为 `int32_t` 的最大值.
- `_MIN` 表示最小值，如 `INT32_MIN` 即为 `int32_t` 的最小值.

**⚠ 注意**

定宽整数类型本质上是普通整数类型的类型别名，所以混用定宽整数类型和普通整数类型可能会影响跨平台编译，例如：

**示例代码**

```cpp
#include <algorithm>
#include <cstdint>
#include <iostream>

int main() {
  long long a;
  int64_t b;
  std::cin >> a >> b;
  std::cout << std::max(a, b) << std::endl;
  return 0;
}
```

`int64_t` 在 64 位 Windows 下一般为 `long long int`，而在 64 位 Linux 下一般为 `long int`，所以这段代码在使用 64 位 Linux 下的 GCC 时不能通过编译，而使用 64 位 Windows 下的 MSVC 时可以通过编译，因为 `std::max` 要求输入的两个参数类型必须相同.

此外，C++17 起在 `<limits>` 中提供了 `std::numeric_limits` 类模板，用于查询各种算数类型的属性，如最大值、最小值、是否是整形、是否有符号等.

```cpp
#include <cstdint>
#include <limits>

std::numeric_limits<int32_t>::max();  // int32_t 的最大值, 2'147'483'647
std::numeric_limits<int32_t>::min();  // int32_t 的最小值, -2'147'483'648

std::numeric_limits<double>::min();  // double 的最小值, 约为 2.22507e-308
std::numeric_limits<double>::epsilon();  // 1.0 与 double 的下个可表示值的差,
                                         // 约为 2.22045e-16
```

### 类型转换

在一些时候（比如某个函数接受 `int` 类型的参数，但传入了 `double` 类型的变量），我们需要将某种类型，转换成另外一种类型.

C++ 中类型的转换机制较为复杂，这里主要介绍对于基础数据类型的两种转换：数值提升和数值转换.

#### 数值提升

数值提升过程中，值本身保持不变.

**Note**

C 风格的可变参数域在传值过程中会进行默认参数提升. 如:

**示例代码**

```cpp
#include <stdarg.h>
#include <stdio.h>

void test(int tot, ...) {
  va_list valist;
  int i;

  // 初始化可变参数列表
  va_start(valist, tot);

  for (i = 0; i < tot; ++i) {
    // 获取第 i 个变量的值
    double xx = va_arg(valist, double);  // Correct
    // float xx = va_arg(valist, float); // Wrong

    // 输出第 i 个变量的底层存储内容
    printf("i = %d, value = 0x%016llx\n", i, *(long long *)(&xx));
  }

  // 清理可变参数列表的内存
  va_end(valist);
}

int main() {
  float f;
  double fd, d;
  f = 123.;   // 0x42f60000
  fd = 123.;  // 0x405ec00000000000
  d = 456.;   // 0x407c800000000000
  test(3, f, fd, d);
}
```

在调用 `test` 时，`f` 提升为 `double`，从而底层存储内容和 `fd` 相同，输出为

```
i = 0, value = 0x405ec00000000000
i = 1, value = 0x405ec00000000000
i = 2, value = 0x407c800000000000
```

若将 `double xx = va_arg(valist, double);` 改为 `float xx = va_arg(valist, float);`，GCC 应该给出一条类似下文的警告：

```
In file included from test.c:2:
test.c: In function 'test':
test.c:14:35: warning: 'float' is promoted to 'double' when passed through '...'
  14 |         float xx = va_arg(valist, float);
     |                                   ^
test.c:14:35: note: (so you should pass 'double' not 'float' to 'va_arg')
test.c:14:35: note: if this code is reached, the program will abort
```

此时的程序将会在输出前终止.

这一点也能解释为什么 `printf` 的 `%f` 既能匹配 `float` 也能匹配 `double`.

**整数提升**

小整数类型（如 `char`）的纯右值可转换成较大整数类型（如 `int`）的纯右值.

具体而言，算术运算符不接受小于 `int` 的类型作为它的实参，而在左值到右值转换后，如果适用就会自动实施整数提升.

具体地，有如下规则：

- 源类型为 `signed char`、`signed short / short` 时，可提升为 `int`.
- 源类型为 `unsigned char`、`unsigned short` 时，若 `int` 能保有源类型的值范围，则可提升为 `int`，否则可提升为 `unsigned int`.（C++20 起 `char8_t` 也适用本规则）
- `char` 的提升规则取决于其底层类型是 `signed char` 还是 `unsigned char`.
- `bool` 类型可转换到 `int`：`false` 变为 `0`，`true` 变为 `1`.
- 若目标类型的值范围包含源类型，且源类型的值范围不能被 `int` 和 `unsigned int` 包含，则源类型可提升为目标类型. [4]

**⚠ 注意**

`char->short` 不是数值提升，因为 `char` 要优先提升为 `int / unsigned int`，之后是 `int / unsigned int->short`，不满足数值提升的条件.

如（以下假定 `int` 为 32 位，`unsigned short` 为 16 位，`signed char` 和 `unsigned char` 为 8 位，`bool` 为 1 位）

- `(signed char)'\0' - (signed char)'\xff'` 会先将 `(signed char)'\0'` 提升为 `(int)0`、将 `(signed char)'\xff'` 提升为 `(int)-1`, 再进行 `int` 间的运算，最终结果为 `(int)1`.
- `(unsigned char)'\0' - (unsigned char)'\xff'` 会先将 `(unsigned char)'\0'` 提升为 `(int)0`、将 `(unsigned char)'\xff'` 提升为 `(int)255`, 再进行 `int` 间的运算，最终结果为 `(int)-255`.
- `false - (unsigned short)12` 会先将 `false` 提升为 `(int)0`、将 `(unsigned short)12` 提升为 `(int)12`, 再进行 `int` 间的运算，最终结果为 `(int)-12`.

**浮点提升**

位宽较小的浮点数可以提升为位宽较大的浮点数（例如 `float` 类型的变量和 `double` 类型的变量进行算术运算时，会将 `float` 类型变量提升为 `double` 类型变量），其值不变.

**数值转换**

数值转换过程中，值可能会发生改变.

[4] 不包含宽字符类型、位域和枚举类型，详见 整型转换 - cppreference[5].

> ⚠ **注意**
>
> 数值提升优先于数值转换. 如 `bool->int` 时是数值提升而非数值转换.

**整数转换**

- 如果目标类型为位宽为 $x$ 的无符号整数类型，则转换结果是原值 $\bmod 2^x$ 后的结果.
  - 若目标类型位宽大于源类型位宽:
    - 若源类型为有符号类型，一般情况下需先进行符号位扩展再转换.

      如
      - 将 `(short)-1`（`(short)0b1111'1111'1111'1111`）转换为 `unsigned int` 类型时，先进行符号位扩展，得到 `0b1111'1111'1111'1111'1111'1111'1111'1111`，再进行整数转换，结果为 `(unsigned int)4'294'967'295`（`(unsigned int)0b1111'1111'1111'1111'1111'1111'1111'1111`）.
      - 将 `(short)32'767`（`(short)0b0111'1111'1111'1111`）转换为 `unsigned int` 类型时，先进行符号位扩展，得到 `0b0000'0000'0000'0000'0111'1111'1111'1111`，再进行整数转换，结果为 `(unsigned int)32'767`（`(unsigned int)0b0000'0000'0000'0000'0111'1111'1111'1111`）.
    - 若源类型为无符号类型，则需先进行零扩展再转换.

      如将 `(unsigned short)65'535`（`(unsigned short)0b1111'1111'1111'1111`）转换为 `unsigned int` 类型时，先进行零扩展，得到 `0b0000'0000'0000'0000'1111'1111'1111'1111`，再进行整数转换，结果为 `(unsigned int)65'535`（`(unsigned int)0b0000'0000'0000'0000'1111'1111'1111'1111`）.
  - 若目标类型位宽不大于源类型位宽，则需先截断再转换.

    如将 `(unsigned int)4'294'967'295`（`(unsigned int)0b1111'1111'1111'1111'1111'1111'1111'1111`）转换为 `unsigned short` 类型时，先进行截断，得到 `0b1111'1111'1111'1111`，再进行整数转换，结果为 `(unsigned short)65'535`（`(unsigned short)0b1111'1111'1111'1111`）.
- 如果目标类型为位宽为 $x$ 的带符号整数类型，则 **一般情况下**，转换结果可以认为是原值 $\bmod 2^x$ 后的结果. [5]

  例如将 `(unsigned int)4'294'967'295`（`(unsigned int)0b1111'1111'1111'1111'1111'1111'1111'1111`）转换为 `short` 类型时，结果为 `(short)-1`（`(short)0b1111'1111'1111'1111`）.
- 如果目标类型是 `bool`，则是 布尔转换→4@p321.
- 如果源类型是 `bool`，则 `false` 转为对应类型的 0，`true` 转为对应类型的 1.

**浮点转换**

位宽较大的浮点数转换为位宽较小的浮点数，会将该数舍入到目标类型下最接近的值.

**浮点整数转换**

- 浮点数转换为整数时，会舍弃浮点数的全部小数部分.

  如果目标类型是 `bool`，则是 布尔转换→4@p321.

[5] 自 C++20 起生效. C++20 前结果是实现定义的. 详见 整型转换 - cppreference[6].

- 整数转换为浮点数时，会舍入到目标类型下最接近的值.

  如果该值不能适应到目标类型中，那么行为未定义.

  如果源类型是 `bool`，那么 `false` 转换为零，而 `true` 转换为一.

#### 布尔转换

将其他类型转换为 `bool` 类型时，零值转换为 `false`，非零值转换为 `true`.

## 定义变量

简单地说[6]，定义一个变量，需要包含类型说明符（指明变量的类型），以及要定义的变量名.

例如，下面这几条语句都是变量定义语句.

```
int oi;
double wiki;
char org = 'c';
```

在目前我们所接触到的程序段中，定义在花括号包裹的地方的变量是局部变量，而定义在没有花括号包裹的地方的变量是全局变量. 实际有例外，但是现在不必了解.

定义时没有初始化值的全局变量会被初始化为 0. 而局部变量没有这种特性，需要手动赋初始值，否则可能引起难以发现的 bug.

## 变量作用域

作用域是变量可以发挥作用的代码块.

全局变量的作用域，自其定义之处开始[7]，至文件结束位置为止.

局部变量的作用域，自其定义之处开始，至代码块结束位置为止.

由一对大括号括起来的若干语句构成一个代码块.

```
int g = 20;  // 定义全局变量

int main() {
  int g = 10;          // 定义局部变量
  printf("%d\n", g);  // 输出 g
  return 0;
}
```

如果一个代码块的内嵌块中定义了相同变量名的变量，则内层块中将无法访问外层块中相同变量名的变量.

例如上面的代码中，输出的 $g$ 的值将是 10. 因此为了防止出现意料之外的错误，请尽量避免局部变量与全局变量重名的情况.

## 常量

常量是固定值，在程序执行期间不会改变.

常量的值在定义后不能被修改. 定义时加一个 `const` 关键字即可.

```
const int a = 2;
a = 3;
```

如果修改了常量的值，在编译环节就会报错：`error: assignment of read-only variable 'a'`.

[6]定义一个变量时，除了类型说明符之外，还可以包含其他说明符. 详见 声明 - cppreference[7].
[7]更准确的说法是 声明点[8].

## 参考资料与注释

1. Working Draft, Standard for Programming Language C++[9]
2. 类型 - cppreference.com[10]
3. C 语言的 算术类型 - cppreference.com[11]
4. 基础类型 - cppreference.com[12]
5. 定宽整数类型（C++11 起） - cppreference.com[13]
6. William Kahan (1 October 1997)."Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic"[14].
7. 隐式转换 - cppreference.com[15]
8. 声明 - cppreference[16]
9. 作用域 - cppreference.com[17]

[1] 类型 - cppreference.com

[2] https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3054.pdf

[3] Range of values - cppreference

[4] ASCII 编码

[5] 整型转换 - cppreference

[6] 整型转换 - cppreference

[7] 声明 - cppreference

[8] 声明点

[9] Working Draft, Standard for Programming Language C++

[10] 类型 - cppreference.com

[11] 算术类型 - cppreference.com

[12] 基础类型 - cppreference.com

[13] 定宽整数类型（C++11 起） - cppreference.com

[14] "Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic"

[15] 隐式转换 - cppreference.com

[16] 声明 - cppreference

[17] 作用域 - cppreference.com

## 4.2.4 运算

> **Authors:** aofall, greyqz, Ir1d, Link-cute, Marcythm, ouuan, Shen-Linwood, sshwy, StudyingFather

## 算术运算符

| 运算符 | 功能 |
| --- | --- |
| `+`（单目） | 正 |
| `-`（单目） | 负 |
| `*`（双目） | 乘法 |
| `/` | 除法 |
| `%` | 取模 |
| `+`（双目） | 加法 |
| `-`（双目） | 减法 |

**单目与双目运算符**

单目运算符（又称一元运算符）指被操作对象只有一个的运算符，而双目运算符（又称二元运算符）的被操作对象有两个．例如 `1 + 2` 中加号就是双目运算符，它有 1 和 2 两个被操作数．此外 C++ 中还有唯一的一个三目运算符 `?:`．

算术运算符中有两个单目运算符（正、负）以及五个双目运算符（乘法、除法、取模、加法、减法），其中单目运算符的优先级最高．

其中取模运算符 `%` 意为计算两个整数相除得到的余数，即求余数.

而 `-` 为双目运算符时做减法运算符，如 `2-1`；为单目运算符时做负值运算符，如 `-1`.

使用方法如下

```
op=x-y*z
```

得到的 `op` 的运算值遵循数学中加减乘除的优先规律，首先进行优先级高的运算，同优先级按运算的结合性运算，括号提高优先级.

### 算术运算中的类型转换

对于双目算术运算符，当参与运算的两个变量类型相同时，不发生 类型转换→4.2.3@p328，运算结果将会用参与运算的变量的类型容纳，否则会发生类型转换，以使两个变量的类型一致. 转换的规则参见 类型转换→4.2.3@p328.

例如，对于一个整型（`int`）变量 $x$ 和另一个双精度浮点型（`double`）类型变量 $y$:

- `x/3` 的结果将会是整型;
- `x/3.0` 的结果将会是双精度浮点型;
- `x/y` 的结果将会是双精度浮点型;
- `x*1/3` 的结果将会是整型;
- `x*1.0/3` 的结果将会是双精度浮点型;

## 位操作符

另请参阅：位运算→9.4@p1099.

| 运算符 | 功能 |
| --- | --- |
| `~` | 逐位非 |
| `&`（双目） | 逐位与 |
| `\|` | `\|` |
| `^` | 逐位异或 |
| `<<` | 逐位左移 |
| `>>` | 逐位右移 |

位操作的意义请参考 位操作→9.4@p1099 页面. 需要注意的是，位操作的优先级低于算术运算符（除了取反），而按位与、按位或及异或低于比较运算符（详见 C++ 运算符优先级总表→4@p321），所以使用时需多加注意，在必要时添加括号.

移位运算中如果出现如下情况，则其行为未定义:

1. 右操作数（即移位数）为负值;
2. 右操作数大于等于左操作数的位数;

例如，对于 `int32_t` 类型的变量 `a`，`a<<-1` 和 `a<<32` 都是未定义的.

对于带符号非负数的左移操作，需要确保移位后的结果能被原数的类型容纳，否则行为也是未定义的. [8] 对一个负数执行左移操作也未定义. [9]

---

[8] C++20 前，若原值为带符号类型，且移位后的结果能被原类型的无符号版本容纳，则将该结果 转换→4.2.3@p328 为相应的带符号值，否则行为未定义；无符号数的左移则舍弃移出结果类型的位. C++20 起，规定 `a << b` 为 $a \cdot 2^b$ 在模 $2^N$ 下的值（$N$ 为结果类型的位宽），即无论是带符号数还是无符号数，左移均直接舍弃移出结果类型的位（即 算术左移/逻辑左移→9.4@p1099）.

[9] C++20 前. C++20 起的行为参见[10].

[10] C++20 前，若原值为带符号类型，且移位后的结果能被原类型的无符号版本容纳，则将该结果 转换→4.2.3@p328 为相应的带符号值，否则行为未定义；无符号数的左移则舍弃移出结果类型的位. C++20 起，规定 `a << b` 为 $a \cdot 2^b$ 在模 $2^N$ 下的值（$N$ 为结果类型的位宽），即无论是带符号数还是无符号数，左移均直接舍弃移出结果类型的位（即 算术左移/逻辑左移→9.4@p1099）.

对于右移操作，右侧多余的位将会被舍弃，而左侧较为复杂：对于无符号数，会在左侧补 0[11]；而对于有符号数，则会用最高位的数（其实就是符号位，非负数为 0，负数为 1）补齐[12].

### 自增/自减 运算符

有时我们需要让变量进行增加 1（自增）或者减少 1（自减），这时自增运算符 `++` 和自减运算符 `--` 就派上用场了.

自增/自减运算符可放在变量前或变量后面，在变量前称为前缀，在变量后称为后缀，单独使用时前缀后缀无需特别区别，如果需要用到表达式的值则需注意，具体可看下面的例子. 详细情况可参考 引用 →4.4.5@p424 介绍的例子部分.

```
i = 100;

op1 = i++;  // op1 = 100，先 op1 = i，然后 i = i + 1

i = 100;

op2 = ++i;  // op2 = 101，先 i = i + 1，然后赋值 op2

i = 100;

op3 = i--;  // op3 = 100，先赋值 op3，然后 i = i - 1

i = 100;

op4 = --i;  // op4 = 99，先 i = i - 1，然后赋值 op4
```

### 复合赋值运算符

复合赋值运算符实际上是表达式的缩写形式. 可分为复合算术运算符 `+=`、`-=`、`*=`、`/=`、`%=` 和复合位操作符 `&=`、`|=`、`^=`、`<<=`、`>>=`.

例如，`op = op + 2` 可写为 `op += 2`，`op = op - 2` 可写为 `op -= 2`，`op= op * 2` 可写为 `op *= 2`.

### 条件运算符

条件运算符可以看作 `if` 语句的简写，`a ? b : c` 中如果表达式 `a` 成立，那么这个条件表达式的结果是 `b`，否则条件表达式的结果是 `c`.

### 比较运算符

| 运算符 | 功能 |
| --- | --- |
| `>` | 大于 |
| `>=` | 大于等于 |
| `<` | 小于 |
| `<=` | 小于等于 |
| `==` | 等于 |
| `!=` | 不等于 |

其中特别需要注意的是要将等于运算符 `==` 和赋值运算符 `=` 区分开来，这在判断语句中尤为重要.

[11]即 逻辑右移→9.4@p1099.

[12]即 算术右移→9.4@p1099. C++20 前，带符号的右移是依实现定义的，在大多数实现中，均采用算术右移. C++20 起，规定 `a >> b` 为 $\lfloor a/2^b \rfloor$，所以带符号数右移运算是算术右移.

`if (op=1)` 与 `if (op==1)` 看起来类似，但实际功能却相差甚远．第一条语句是在对 op 进行赋值，若赋值为非 0 时为真值，表达式的条件始终是满足的，无法达到判断的作用；而第二条语句才是对 `op` 的值进行判断．

## 逻辑运算符

| 运算符 | 功能 |
| --- | --- |
| && | 逻辑与 |
| ` | | |
| ! | 逻辑非 |

```
Result = op1 && op2;  // 当 op1 与 op2 都为真时则 Result 为真

Result = op1 || op2;  // 当 op1 或 op2 其中一个为真时则 Result 为真

Result = !op1;  // 当 op1 为假时则 Result 为真
```

**内建的**运算符 && 和 || 进行短路求值（若在求值第一个操作数后结果已知，则不求值第二个），重载的运算符无此特性，并始终对两个操作数都进行求值．

## 逗号运算符

逗号运算符可将多个表达式分隔开来，被分隔开的表达式按从左至右的顺序依次计算，整个表达式的值是最后的表达式的值．逗号表达式的优先级在所有运算符中的优先级是 **最低** 的．

```
exp1, exp2, exp3;  // 最后的值为 exp3 的运算结果．

Result = 1 + 2, 3 + 4, 5 + 6;
//得到 Result 的值为 3 而不是 11，因为赋值运算符 "="
//的优先级比逗号运算符高，先进行了赋值运算才进行逗号运算．

Result = (1 + 2, 3 + 4, 5 + 6);

// 若要让 Result 的值得到逗号运算的结果则应将整个表达式用括号提高优先级，此时
// Result 的值才为 11．
```

## 成员访问运算符

| 运算符 | 功能 |
| --- | --- |
| [] | 数组下标 |
| . | 对象成员 |
| & （单目） | 取地址/获取引用 |
| * （单目） | 间接寻址/解引用 |
| -> | 指针成员 |

这些运算符用来访问对象的成员或者内存，除了最后一个运算符外上述运算符都可被重载．与 `&`，`*` 和 `->` 相关的内容请阅读 指针→4.2.6.4@p356 和 引用→4.4.5@p424 教程．这里还省略了两个很少用到的运算符 `.*` 和 `->*`，其具体用法可以参见 C++ 语言手册[1]．

```
auto result1 = v[1];  // 获取v中下标为2的对象
auto result2 = p.q;   // 获取p对象的q成员
auto result3 = p -> q;  // 获取p指针指向的对象的q成员，等价于 (*p).q
auto result4 = &v;      // 获取指向v的指针
```

```
5 auto result5 = *v;       // 获取v指针指向的对象
```

## C++ 运算符优先级总表

来自 C++ 运算符优先级 - cppreference[2]，有修改.

| 运算符 | 描述 | 例子 | 可重载性 |
|---|---|---|---|
| **第一级别** | | | |
| :: | 作用域解析符 | Class::age = 2; | 不可重载 |
| **第二级别** | | | |
| ++ | 后自增运算符 | for (int i = 0; i < 10; i++) cout << i; | 可重载 |
| -- | 后自减运算符 | for (int i = 10; i > 0; i--) cout << i; | 可重载 |
| type() type{} | 强制类型转换 | unsigned int a = unsigned(3.14); | 可重载 |
| () | 函数调用 | isdigit('1') | 可重载 |
| [] | 数组数据获取 | array[4] = 2; | 可重载 |
| . | 对象型成员调用 | obj.age = 34; | 不可重载 |
| -> | 指针型成员调用 | ptr->age = 34; | 可重载 |
| **第三级别**（从右向左结合） | | | |
| ++ | 前自增运算符 | for (i = 0; i < 10; ++i) cout << i; | 可重载 |
| -- | 前自减运算符 | for (i = 10; i > 0; --i) cout << i; | 可重载 |
| + | 正号 | int i = +1; | 可重载 |
| - | 负号 | int i = -1; | 可重载 |
| ! | 逻辑取反 | if (!done) … | 可重载 |
| ~ | 按位取反 | flags = ~flags; | 可重载 |
| (type) | C 风格强制类型转换 | int i = (int) floatNum; | 可重载 |
| * | 指针取值 | int data = *intPtr; | 可重载 |
| & | 值取指针 | int *intPtr = &data; | 可重载 |
| sizeof | 返回类型内存 | int size = sizeof floatNum; int size = sizeof(float); | 不可重载 |
| new | 动态元素内存分配 | long *pVar = new long; MyClass *ptr = new MyClass(args); | 可重载 |
| new [] | 动态数组内存分配 | long *array = new long[n]; | 可重载 |
| delete | 动态析构元素内存 | delete pVar; | 可重载 |
| delete [] | 动态析构数组内存 | delete [] array; | 可重载 |
| **第四级别** | | | |
| .* | 类对象成员引用 | obj.*var = 24; | 不可重载 |
| ->* | 类指针成员引用 | ptr->*var = 24; | 可重载 |
| **第五级别** | | | |
| * | 乘法 | int i = 2 * 4; | 可重载 |
| / | 除法 | float f = 10.0 / 3.0; | 可重载 |
| % | 取余数（模运算） | int rem = 4 % 3; | 可重载 |
| **第六级别** | | | |
| + | 加法 | int i = 2 + 3; | 可重载 |
| - | 减法 | int i = 5 - 1; | 可重载 |

| **第七级别** | | | |
|---|---|---|---|
| `<<` | 位左移 | `int flags = 33 << 1;` | 可重载 |
| `>>` | 位右移 | `int flags = 33 >> 1;` | 可重载 |
| **第八级别** | | | |
| `<=>` | 三路比较运算符 | `if ((i <=> 42) < 0) ...` | 可重载 |
| **第九级别** | | | |
| `<` | 小于 | `if (i < 42) ...` | 可重载 |
| `<=` | 小于等于 | `if (i <= 42) ...` | 可重载 |
| `>` | 大于 | `if (i > 42) ...` | 可重载 |
| `>=` | 大于等于 | `if (i >= 42) ...` | 可重载 |
| **第十级别** | | | |
| `==` | 等于 | `if (i == 42) ...` | 可重载 |
| `!=` | 不等于 | `if (i != 42) ...` | 可重载 |
| **第十一级别** | | | |
| `&` | 位与运算 | `flags = flags & 42;` | 可重载 |
| **第十二级别** | | | |
| `^` | 位异或运算 | `flags = flags ^ 42;` | 可重载 |
| **第十三级别** | | | |
| `` ` `` | `` ` `` | 位或运算 | `` `flags = flags `` |
| **第十四级别** | | | |
| `&&` | 逻辑与运算 | `if (conditionA && conditionB) ...` | 可重载 |
| **第十五级别** | | | |
| `` ` `` | | `` ` `` | 逻辑或运算 |
| **第十六级别** （从右向左结合） | | | |
| `? :` | 条件运算符 | `int i = a > b ? a : b;` | 不可重载 |
| `throw` | 异常抛出 | `throw EClass("Message");` | 不可重载 |
| `=` | 赋值 | `int a = b;` | 可重载 |
| `+=` | 加赋值运算 | `a += 3;` | 可重载 |
| `-=` | 减赋值运算 | `b -= 4;` | 可重载 |
| `*=` | 乘赋值运算 | `a *= 5;` | 可重载 |
| `/=` | 除赋值运算 | `a /= 2;` | 可重载 |
| `%=` | 模赋值运算 | `a %= 3;` | 可重载 |
| `<<=` | 位左移赋值运算 | `flags <<= 2;` | 可重载 |
| `>>=` | 位右移赋值运算 | `flags >>= 2;` | 可重载 |
| `&=` | 位与赋值运算 | `flags &= new_flags;` | 可重载 |
| `^=` | 位异或赋值运算 | `flags ^= new_flags;` | 可重载 |
| `` ` `` | `=` ` | 位或赋值运算 | `` `flags `` |
| **第十七级别** | | | |
| `,` | 逗号分隔符 | `for (i = 0, j = 0; i < 10; i++, j++) ...` | 可重载 |

需要注意的是，表中并未列出 `const_cast`、`static_cast`、`dynamic_cast`、`reinterpret_cast`、`typeid`、`sizeof...`、`noexcept` 及 `alignof` 等运算符，因为它们的使用形式与函数调用相同，不会出现歧义.

## 参考资料与注释

[1] C++ 语言手册



[2] C++ 运算符优先级 - cppreference



## 4.2.5 流程控制语句

### 4.2.5.1 分支

一个程序默认是按照代码的顺序执行下来的，有时我们需要选择性的执行某些语句，这时候就需要分支的功能来实现．选择合适的分支语句可以提高程序的效率．

#### if 语句

##### 基本 if 语句

以下是基本 if 语句的结构．

```
if (条件) {
  主体;
}
```

if 语句通过对条件进行求值，若结果为真（非 0），执行语句，否则不执行．

如果主体中只有单个语句的话，花括号可以省略．

##### if...else 语句

```
if (条件) {
  主体1;
} else {
  主体2;
}
```

if...else 语句和 if 语句类似，else 不需要再写条件．当 if 语句的条件满足时会执行 if 里的语句，if 语句的条件不满足时会执行 else 里的语句．同样，当主体只有一条语句时，可以省略花括号．

##### else if 语句

```
if (条件1) {
  主体1;
} else if (条件2) {
  主体2;
} else if (条件3) {
  主体3;
} else {
  主体4;
}
```

else if 语句是 if 和 else 的组合，对多个条件进行判断并选择不同的语句分支．在最后一条的 else 语句不需要再写条件．例如，若条件 1 为真，执行主体 1，条件 3 为真而条件 1 和条件 2 都为假，执行主体 3，所有的条件都为假才执行主体 4．

实际上，这一个语句相当于第一个 if 的 else 分句只有一个 if 语句，就将花括号省略之后放在一起了．如果条件相互之间是并列关系，这样写可以让代码的逻辑更清晰．

在逻辑上，大约相当于这一段话：

> 解一元二次方程的时候，方程的根与判别式的关系：
> - 如果 ($\Delta < 0$) 方程无解；

- 否则，如果 ($\Delta = 0$) 方程有两个相同的实数解；
- 否则 方程有两个不相同的实数解；

### switch 语句

```
switch (选择句) {
  case 标签1:
    主体1;
  case 标签2:
    主体2;
  default:
    主体3;
}
```

switch 语句执行时，先求出选择句的值，然后根据选择句的值选择相应的标签，从标签处开始执行. 其中，选择句必须是一个整数类型表达式，而标签都必须是整数类型的常量. 例如：

```
int i = 1;  // 这里的 i 的数据类型是整型 ，满足整数类型的表达式的要求

switch (i) {
  case 1:
    cout << "OI WIKI" << endl;
}
```

```
char i = 'A';

// 这里的 i 的数据类型是字符型 ，但 char
// 也是属于整数的类型，满足整数类型的表达式的要求
switch (i) {
  case 'A':
    cout << "OI WIKI" << endl;
}
```

switch 语句中还要根据需求加入 break 语句进行中断, 否则在对应的 case 被选择之后接下来的所有 case 里的语句和 default 里的语句都会被运行. 具体例子可看下面的示例.

```
char i = 'B';

switch (i) {
  case 'A':
    cout << "OI" << endl;
    break;

  case 'B':
    cout << "WIKI" << endl;

  default:
    cout << "Hello World" << endl;
}
```

以上代码运行后输出的结果为 `WIKI` 和 `Hello World`, 如果不想让下面分支的语句被运行就需要 break 了，具体例子可看下面的示例.

```
char i = 'B';

switch (i) {
```

```
  case 'A':
    cout << "OI" << endl;
    break;

  case 'B':
    cout << "WIKI" << endl;
    break;

  default:
    cout << "Hello World" << endl;
}
```

以上代码运行后输出的结果为 WIKI，因为 break 的存在，接下来的语句就不会继续被执行了．最后一个语句不需要 break，因为下面没有语句了．

处理入口编号不能重复，但可以颠倒．也就是说，入口编号的顺序不重要．各个 case（包括 default）的出现次序可任意．例如：

```
char i = 'B';

switch (i) {
  case 'B':
    cout << "WIKI" << endl;
    break;

  default:
    cout << "Hello World" << endl;
    break;

  case 'A':
    cout << "OI" << endl;
}
```

switch 的 case 分句中也可以选择性的加花括号．不过要注意的是，如果需要在 switch 语句中定义变量，花括号是必须要加的．例如：

```
char i = 'B';

switch (i) {
  case 'A': {
    int i = 1, j = 2;
    cout << "OI" << endl;
    ans = i + j;
    break;
  }

  case 'B': {
    int qwq = 3;
    cout << "WIKI" << endl;
    ans = qwq * qwq;
    break;
  }

  default: {
    cout << "Hello World" << endl;
```

```
20   }
21 }
```

**如何理解 switch**

在上文中，用了大量「case 分句」，「case 子句」等用语，实际上，在底层实现中，switch 相当于一组跳转语句．也因此，有 Duff's Device 这种奇技淫巧，希望了解的人可以自行学习．

#### 4.2.5.2 循环

有时，我们需要做一件事很多遍，为了不写过多重复的代码，我们需要循环．

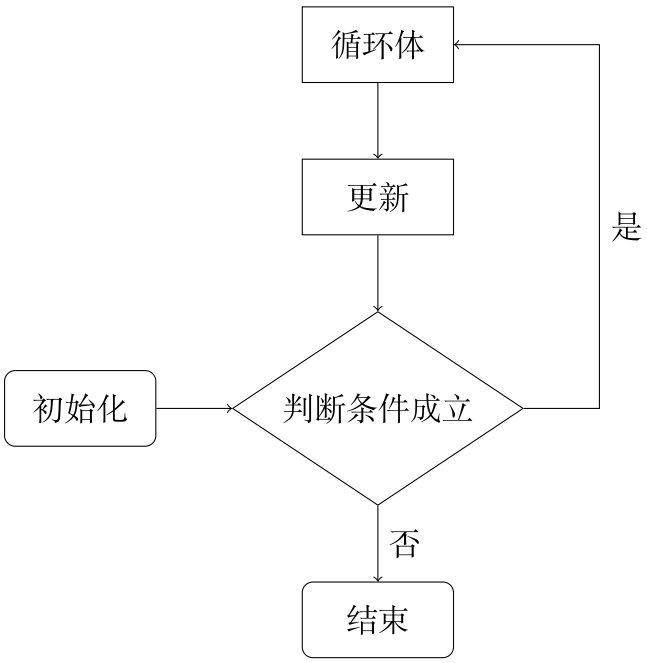
有时，循环的次数不是一个常量，那么我们无法将代码重复多遍，必须使用循环．

**for 语句**

以下是 for 语句的结构：

```
1 for (初始化; 判断条件; 更新) {
2   循环体;
3 }
```

执行顺序：




e.g. 读入 n 个数：

```
1 for (int i = 1; i <= n; ++i) {
2   cin >> a[i];
3 }
```

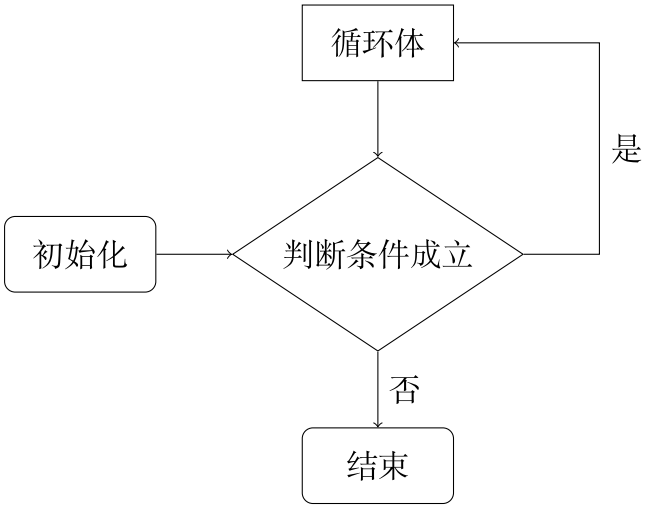
for 语句的三个部分中，任何一个部分都可以省略．其中，若省略了判断条件，相当于判断条件永远为真．

**while 语句**

以下是 while 语句的结构：

```
while (判断条件) {
  循环体;
}
```

执行顺序：




e.g. 验证 3x+1 猜想：

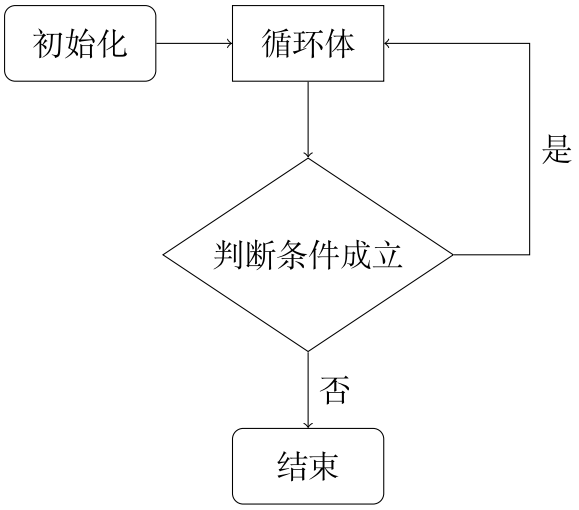
```
while (x > 1) {
  if (x % 2 == 1) {
    x = 3 * x + 1;
  } else {
    x = x / 2;
  }
}
```

### do...while 语句

以下是 do...while 语句的结构：

```
do {
  循环体;
} while (判断条件);
```

执行顺序：




与 while 语句的区别在于，do...while 语句是先执行循环体再进行判断的.

e.g. 枚举排列:

```
do {
  // do someting...
} while (next_permutation(a + 1, a + n + 1));
```

### 三种语句的联系

```
// for 语句

for (statement1; statement2; statement3) {
  statement4;
}

// while 语句

statement1;
while (statement2) {
  statement4;
  statement3;
}
```

在 statement4 中没有 `continue` 语句（见下文）的时候是等价的，但是下面一种方法很少用到.

```
// while 语句

statement1;
while (statement2) {
  statement1;
}

// do...while 语句

do {
  statement1;
} while (statement2);
```

在 statement1 中没有 `continue` 语句的时候这两种方式也是等价的.

```
while (1) {
  // do something...
}

for (;;) {
  // do something...
}
```

这两种方式都是永远循环下去.（可以使用 `break`（见下文）退出.）

可以看出，三种语句可以彼此代替，但一般来说，语句的选用遵守以下原则：

1. 循环过程中有个固定的增加步骤（最常见的是枚举）时，使用 for 语句；
2. 只确定循环的终止条件时，使用 while 语句；
3. 使用 while 语句时，若要先执行循环体再进行判断，使用 do...while 语句. 一般很少用到，常用场景是用户输入.

### break 与 continue 语句

break 语句的作用是退出循环.

continue 语句的作用是跳过循环体的余下部分. 下面以 continue 语句在 do...while 语句中的使用为例：

```
do {
  // do something...
  continue;  // 等价于 goto END;
// do something...
END:;
} while (statement);

```

break 与 continue 语句均可在三种循环语句的循环体中使用.

一般来说，break 与 continue 语句用于让代码的逻辑更加清晰，例如：

```
// 逻辑较为不清晰，大括号层次复杂

for (int i = 1; i <= n; ++i) {
  if (i != x) {
    for (int j = 1; j <= n; ++j) {
      if (j != x) {
        // do something...
      }
    }
  }
}

// 逻辑更加清晰，大括号层次简单明了

for (int i = 1; i <= n; ++i) {
  if (i == x) continue;
  for (int j = 1; j <= n; ++j) {
    if (j == x) continue;
    // do something...
  }
}
```

```
// for 语句判断条件复杂，没有体现「枚举」的本质
```

```
2
3  for (int i = l; i <= r && i % 10 != 0; ++i) {
4    // do something...
5  }
6
7  // for 语句用于枚举，break 用于「到何时为止」
8
9  for (int i = l; i <= r; ++i) {
10   if (i % 10 == 0) break;
11   // do something...
12 }
```

```
1  // 语句重复，顺序不自然
2
3  statement1;
4  while (statement3) {
5    statement2;
6    statement1;
7  }
8
9  // 没有重复语句，顺序自然
10
11 while (1) {
12   statement1;
13   if (!statement3) break;
14   statement2;
15 }
```

## 4.2.6 高级数据类型

### 4.2.6.1 数组

数组是存放相同类型对象的容器，数组中存放的对象没有名字，而是要通过其所在的位置访问．数组的大小是固定的，不能随意改变数组的长度．

#### 定义数组

数组的声明形如 `a[d]`，其中，`a` 是数组的名字，`d` 是数组中元素的个数．在编译时，`d` 应该是已知的，也就是说，`d` 应该是一个整型的常量表达式．

```
1  unsigned int d1 = 42;
2  const int d2 = 42;
3  int arr1[d1];  // 错误：d1 不是常量表达式
4  int arr2[d2];  // 正确：arr2 是一个长度为 42 的数组
```

不能将一个数组直接赋值给另一个数组：

```
1  int arr1[3];
2  int arr2 = arr1;  // 错误
3  arr2 = arr1;      // 错误
```

应该尽量将较大的数组定义为全局变量．因为局部变量会被创建在栈区中，过大（大于栈的大小）的数组会爆栈，进而导致 RE．如果将数组声明在全局作用域中，就会在静态区中创建数组．

### 访问数组元素

可以通过下标运算符 `[]` 来访问数组内元素，数组的索引（即方括号中的值）从 0 开始．以一个包含 10 个元素的数组为例，它的索引为 0 到 9，而非 1 到 10．但在 OI 中，为了使用方便，我们通常会将数组开大一点，不使用数组的第一个元素，从下标 1 开始访问数组元素．

例 1：从标准输入中读取一个整数 $n$，再读取 $n$ 个数，存入数组中．其中，$n \leq 1000$．

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

int arr[1001];  // 数组 arr 的下标范围是 [0, 1001)

int main() {
  int n;
  cin >> n;
  for (int i = 1; i <= n; ++i) {
    cin >> arr[i];
  }
}
```

例 2：（接例 1）求和数组 `arr` 中的元素，并输出和．满足数组中所有元素的和小于等于 $2^{31} - 1$

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

int arr[1001];

int main() {
  int n;
  cin >> n;
  for (int i = 1; i <= n; ++i) {
    cin >> arr[i];
  }

  int sum = 0;
  for (int i = 1; i <= n; ++i) {
    sum += arr[i];
  }

  printf("%d\n", sum);
  return 0;
}
```

#### 越界访问下标

数组的下标 $idx$ 应当满足 $0 \leq idx < size$，如果下标不在这个范围内，则是未定义行为，会产生不可预料的后果，如段错误（Segmentation Fault），或者修改预期以外的变量等等．

### 多维数组

多维数组的实质是「数组的数组」，即外层数组的元素是数组．一个二维数组需要两个维度来定义：数组的长度和数组内元素的长度．访问二维数组时需要写出两个索引：

```cpp
int arr[3][4];  // 一个长度为 3 的数组，它的元素是「元素为 int 的长度为的 4
                // 的数组」
arr[2][1] = 1;  // 访问二维数组
```

我们经常使用嵌套的 for 循环来处理二维数组．

例：从标准输入中读取两个数 $n$ 和 $m$，分别表示黑白图片的高与宽，满足 $n, m \leq 1000$. 对于接下来的 $n$ 行数据，每行有用空格分隔开的 $m$ 个数，代表这一位置的亮度值. 现在我们读取这张图片，并将其存入二维数组中.

```
const int MAXN = 1001;
int pic[MAXN][MAXN];
int n, m;

cin >> n >> m;
for (int i = 1; i <= n; ++i)
  for (int j = 1; j <= m; ++j) cin >> pic[i][j];
```

同样地，你可以定义三维、四维，以及更高维的数组.

#### 4.2.6.2 结构体

**Authors:** Ir1d, cjsoft, Lans1ot

**结构体**（struct），可以看做是一系列称为成员元素的组合体.

可以看做是自定义的数据类型.

> **Note**
>
> 本页描述的 `struct` 不同于 C 中 `struct`，在 C++ 中 `struct` 被扩展为类似 `class`→4.4.1@p407 的类说明符.

##### 定义结构体

```
struct Object {
  int weight;
  int value;
} e[array_length];

const Object a;
Object b, B[array_length], tmp;
Object *c;
```

上例中定义了一个名为 `Object` 的结构体，两个成员元素 `value, weight`，类型都为 `int`.

在 `}` 后，定义了数据类型为 `Object` 的常量 `a`，变量 `b`，变量 `tmp`，数组 `B`，指针 `c`. 对于某种已经存在的类型，都可以使用这里的方法进行定义常量、变量、指针、数组等.

关于指针：不必强求掌握.

##### 定义指针

如果是定义内置类型的指针，则与平常定义指针一样.

如果是定义结构体指针，在定义中使用 `StructName*` 进行定义.

```
struct Edge {
  /*
  ...
  */
  Edge* nxt;
};
```

上例仅作举例，不必纠结实际意义.

### 访问/修改成员元素

可以使用 `变量名.成员元素名` 进行访问. 例如可以使用 `cout << var.v` 来输出 `var` 的 `v` 成员.

也可以使用 `指针名->成员元素名` 或者 使用 `(*指针名).成员元素名` 进行访问. 例如使用 `(*ptr).v = tmp` 或者 `ptr->v = tmp` 可以将结构体指针 `ptr` 指向的结构体的成员元素 `v` 赋值为 `tmp`:.

### 为什么需要结构体?

首先, 条条大路通罗马, 可以不使用结构体达到相同的效果. 但是结构体能够显式地将成员元素 (在算法竞赛中通常是变量) 捆绑在一起, 如本例中的 `Object` 结构体, 便将 `value,weight` 放在了一起 (定义这个结构体的实际意义是表示一件物品的重量与价值). 这样的好处边是限制了成员元素的使用.

想象一下, 如果不使用结构体而且有两个数组 `value[], Value[]`, 很容易写混淆. 但如果使用结构体, 能够减轻出现使用变量错误的几率.

并且不同的结构体 (结构体类型, 如 `Object` 这个结构体) 或者不同的结构体变量 (结构体的实例, 如上方的 `e` 数组) 可以拥有相同名字的成员元素 (如 `tmp.value,b.value`), 同名的成员元素相互独立 (拥有独自的内存, 比如说修改 `tmp.value` 不会影响 `b.value` 的值).

这样的好处是可以使用尽可能相同或者相近的变量去描述一个物品. 比如说 `Object` 里有 `value` 这个成员变量; 我们还可以定义一个 `Car` 结构体, 同时也拥有 `value` 这个成员; 如果不使用结构体, 或许我们就需要定义 `valueOfObject[],valueOfCar[]` 等不同名称的数组来区分.

如果想要更详细的描述一种事物, 还可以定义成员函数. 请参考 类→4.4.1@p407 获取详细内容.

### 更多的操作?

详见 类→4.4.1@p407.

### 注意事项

为了访问内存的效率更高, 编译器在处理结构中成员的实际存储情况时, 可能会将成员对齐在一定的字节位置, 也就意味着结构中有空余的地方. 因此, 该结构所占用的空间可能大于其中所有成员所占空间的总和.

### 参考资料


1. Class - zh.cppreference.com[1]
2. Data structures - cplusplus.com[2]
3. 对齐方式 - Microsoft Docs[3]


### 外部参考资料


[1] Class - zh.cppreference.com 

[2] Data structures - cplusplus.com 

[3] 对齐方式 - Microsoft Docs 


## 4.2.6.3 联合体

**联合体** (union) 是特殊的类类型, 它在一个时刻只能保有其一个非静态数据成员.

联合体在 2023 年正式被加入 NOI 大纲入门级中.

### 定义联合体

联合体声明的类说明符与类或 结构体→4.2.6.2@p354 的声明相似:

```
union MyUnion {
  int x;
  long long y;
} x;
```

联合体的定义与结构体类似．按照上述定义，`MyUnion` 同样可以当作一种自定义类型使用．名称 `MyUnion` 可以省略．

### 访问/修改成员元素

与结构体类似，同样可以使用 `变量名.成员名` 进行访问．

联合体所占用的内存空间大小 **不小于** 其最大的成员的大小，所有成员 **共用内存空间与地址**．当一个成员被赋值，由于内存共享，该联合体中的其他成员都会被覆盖．即同一时刻联合体中只能保存一个成员的值．

联合体的更多用法可以参见 cppreference：联合体声明[1]．

### 外部参考资料

[1] cppreference：联合体声明

## 4.2.6.4 指针

**Authors:** tsagaanbar, Enter-tainer, Xeonacid

### 变量的地址、指针

在程序中，我们的数据都有其存储的地址．在程序每次的实际运行过程中，变量在物理内存中的存储位置不尽相同．不过，我们仍能够在编程时，通过一定的语句，来取得数据在内存中的地址．

地址也是数据．存放地址所用的变量类型有一个特殊的名字，叫做「指针变量」，有时也简称做「指针」．

> **指针变量的大小**
>
> 指针变量的大小在不同环境下有差异．在 32 位机上，地址用 32 位二进制整数表示，因此一个指针的大小为 4 字节．而 64 位机上，地址用 64 位二进制整数表示，因此一个指针的大小就变成了 8 字节．

地址只是一个刻度一般的数据，为了针对不同类型的数据，「指针变量」也有不同的类型，比如，可以有 `int` 类型的指针变量，其中存储的地址（即指针变量存储的数值）对应一块大小为 32 位的空间的起始地址；有 `char` 类型的指针变量，其中存储的地址对应一块 8 位的空间的起始地址．

事实上，用户也可以声明指向指针变量的指针变量．

假如用户自定义了一个结构体：

```
struct ThreeInt {
  int a;
  int b;
  int c;
};
```

则 `ThreeInt` 类型的指针变量，对应着一块 $3 \times 32 = 96$ bit 的空间．

### 指针的声明与使用

C/C++ 中，指针变量的类型为类型名后加上一个星号 `*`．比如，`int` 类型的指针变量的类型名即为 `int*`．

我们可以使用 `&` 符号取得一个变量的地址．

要想访问指针变量地址所对应的空间（又称指针所 **指向** 的空间），需要对指针变量进行 **解引用**（dereference），使用 `*` 符号．

```
int main() {
```

```
  int a = 123;  // a: 123
  int* pa = &a;
  *pa = 321;  // a: 321
}
```

对结构体变量也是类似. 如果要访问指针指向的结构中的成员，需要先对指针进行解引用，再使用 `.` 成员关系运算符. 不过，更推荐使用「箭头」运算符 `->` 这一更简便的写法.

```
struct ThreeInt {
  int a;
  int b;
  int c;
};

int main() {
  ThreeInt x{1, 2, 3}, y{6, 7, 8};
  ThreeInt* px = &x;
  (*px) = y;    // x: {6,7,8}
  (*px).a = 4;  // x: {4,7,8}
  px->b = 5;    // x: {4,5,8}
}
```

### 指针的偏移

指针变量也可以 **和整数** 进行加减操作. 对于 `int` 型指针，每加 1（递增 1），其指向的地址偏移 32 位（即 4 个字节）；若加 2，则指向的地址偏移 $2 \times 32 = 64$ 位. 同理，对于 `char` 型指针，每次递增，其指向的地址偏移 8 位（即 1 个字节）.

#### 使用指针偏移访问数组

我们前面说过，数组是一块连续的存储空间. 而在 C/C++ 中，直接使用数组名，得到的是数组的起始地址.

```
int main() {
  int a[3] = {1, 2, 3};
  int* p = a;  // p 指向 a[0]
  *p = 4;      // a: [4, 2, 3]
  p = p + 1;   // p 指向 a[1]
  *p = 5;      // a: [4, 5, 3]
  p++;         // p 指向 a[2]
  *p = 6;      // a: [4, 5, 6]
}
```

当通过指针访问数组中的元素时，往往需要用到「指针的偏移」，换句话说，即通过一个基地址（数组起始的地址）加上偏移量来访问.

我们常用 `[]` 运算符来访问数组中某一指定偏移量处的元素. 比如 `a[3]` 或者 `p[4]`. 这种写法和对指针进行运算后再引用是等价的，即 `p[4]` 和 `*(p + 4)` 是等价的两种写法.

### 空指针

在 C++11 之前，C++ 和 C 一样使用 `NULL` 宏表示空指针常量，C++ 中 `NULL` 的实现一般如下:

```
// C++11 前
#define NULL 0
```

**C 语言对 NULL 的定义**

C 语言在 C23 前有两个 NULL 的定义，只有类型不同：一个是整型常量表达式，一个是转换为 void * 类型的常量表达式，但其值都为 0，编译器可任选一个实现.

空指针和整数 0 的混用在 C++ 中会导致许多问题，比如：

```
int f(int x);
int f(int* p);
```

在调用 f(NULL) 时，实际调用的函数的类型是 int(int) 而不是 int(int *).

**NULL 在 C 语言中造成的问题**

比起在 C++ 中，因为有两个定义，在 C 语言中 NULL 造成的问题更为严重：如果在一个传递可变参数的函数中，函数编写者想要接受一个指针，但是函数调用者传递了一个定义为整型的 NULL，则会造成未定义行为，因在函数内使用传入的可变参数时，要进行类型转换，而从整型到指针类型的转换是未定义行为. [13]

为了解决这些问题，C++11 引入了 nullptr 关键字作为空指针常量.

C++ 规定 nullptr 可以隐式转换为任何指针类型，这种转换结果是该类型的空指针值.

nullptr 的类型为 std::nullptr_t, 称作空指针类型，可能的实现如下：

```
namespace std {
typedef decltype(nullptr) nullptr_t;
}
```

另外，C++11 起 NULL 宏的实现也被修改为了：

```
// C++11 起
#define NULL nullptr
```

**C 语言对空指针常量的改进**

基于类似的原因，C23 也引入了 nullptr 作为空指针常量，同时引入了 nullptr_t 作为其类型[14].

## 指针的进阶使用

使用指针，使得程序编写者可以操作程序运行时中各处的数据，而不必局限于作用域.

### 指针类型参数的使用

在 C/C++ 中，调用函数（过程）时使用的参数，均以拷贝的形式传入子过程中（引用除外，会在后续介绍）. 默认情况下，函数仅能通过返回值，将结果返回到调用处. 但是，如果某个函数希望修改其外部的数据，或者某个结构体/类的数据量较为庞大、不宜进行拷贝，这时，则可以通过向其传入外部数据的地址，便得以在其中访问甚至修改外部数据.

下面的 my_swap 方法，通过接收两个 int 型的指针，在函数中使用中间变量，完成对两个 int 型变量值的交换.

```
void my_swap(int *a, int *b) {
  int t;
  t = *a;
```

[13] 参见 Introduce the nullptr constant[1]
[14] 参见 Introduce the nullptr constant[2]

```
  *a = *b;
  *b = t;
}

int main() {
  int a = 6, b = 10;
  my_swap(&a, &b);
  // 调用后，main 函数中 a 变量的值变为 10，b 变量的值变为 6
}
```

C++ 中引入了引用的概念，相对于指针来说，更易用，也更安全. 详情可以参见 C++：引用→4.4.5@p424 以及 C 与 C++ 的区别：指针与引用→4.5@p470.

**动态实例化**

除此之外，程序编写时往往会涉及到动态内存分配，即，程序会在运行时，向操作系统动态地申请或归还存放数据所需的内存. 当程序通过调用操作系统接口申请内存时，操作系统将返回程序所申请空间的地址. 要使用这块空间，我们需要将这块空间的地址存储在指针变量中.

在 C++ 中，我们使用 `new` 运算符来获取一块内存，使用 `delete` 运算符释放某指针所指向的空间.

```
int* p = new int(1234);
/* ... */
delete p;
```

上面的语句使用 `new` 运算符向操作系统申请了一块 `int` 大小的空间，将其中的值初始化为 1234，并声明了一个 `int` 型的指针 `p` 指向这块空间.

同理，也可以使用 `new` 开辟新的对象：

```
class A {
  int a;

 public:
  A(int a_) : a(a_) {}
};

int main() {
  A* p = new A(1234);
  /* ... */
  delete p;
}
```

如上，「`new` 表达式」将尝试开辟一块对应大小的空间，并尝试在这块空间上构造这一对象，并返回这一空间的地址.

```
struct ThreeInt {
  int a;
  int b;
  int c;
};

int main() {
  ThreeInt* p = new ThreeInt{1, 2, 3};
  /* ... */
  delete p;
}
```

**列表初始化**

`{}` 运算符可以用来初始化没有构造函数的结构．除此之外，使用 `{}` 运算符可以使得变量的初始化形式变得统一．详见「list initialization (since C++11)[3]」.

需要注意，当使用 `new` 申请的内存不再使用时，需要使用 `delete` 释放这块空间．不能对一块内存释放两次或以上．而对空指针 `nullptr` 使用 `delete` 操作是合法的.

### 动态创建数组

也可以使用 `new[]` 运算符创建数组，这时 `new[]` 运算符会返回数组的首地址，也就是数组第一个元素的地址，我们可以用对应类型的指针存储这个地址．释放时，则需要使用 `delete[]` 运算符.

```
size_t element_cnt = 5;
int *p = new int[element_cnt];
delete[] p;
```

数组中元素的存储是连续的，即 `p + 1` 指向的是 `p` 的后继元素.

### 二维数组

在存放矩阵形式的数据时，可能会用到「二维数组」这样的数据类型．从语义上来讲，二维数组是一个数组的数组．而计算机内存可以视作一个很长的一维数组．要在计算机内存中存放一个二维数组，便有「连续」与否的说法.

所谓「连续」，即二维数组的任意一行（row）的末尾与下一行的起始，在物理地址上是毗邻的，换言之，整个二维数组可以视作一个一维数组；反之，则二者在物理上不一定相邻.

对于「连续」的二维数组，可以仅使用一个循环，借由一个不断递增的指针即可遍历数组中的所有数据．而对于非连续的二维数组，由于每一行不连续，则需要先取得某一行首的地址，再访问这一行中的元素.

**二维数组的存储方式**

这种按照「行（row）」存储数据的方式，称为行优先存储；相对的，也可以按照列（column）存储数据．由于计算机内存访问的特性，一般来说，访问连续的数据会得到更高的效率．因此，需要按照数据可能的使用方式，选择「行优先」或「列优先」的存储方式.

### 动态创建二维数组

在 C/C++ 中，我们可以使用类似下面这样的语句声明一个 N 行（row）M 列（column）的二维数组，其空间在物理上是连续的.

**描述数组的维度**

更通用的方式是使用第 n 维（dimension）的说法．对于「行优先」的存储形式，数组的第一维长度为 N，第二维长度为 M.

```
int a[N][M];
```

这种声明方式要求 N 和 M 为在编译期即可确定的常量表达式.

在 C/C++ 中，数组的第一个元素下标为 0，因此 `a[r][c]` 这样的式子代表二维数组 a 中第 r + 1 行的第 c + 1 个元素，我们也称这个元素的下标为 `(r,c)`.

不过，实际使用中，（二维）数组的大小可能不是固定的，需要动态内存分配.

常见的方式是声明一个长度为 N × M 的 **一维数组**，并通过下标 `r * M + c` 访问二维数组中下标为 `(r, c)` 的元素.

```
int* a = new int[N * M];
```

这种方法可以保证二维数组是 **连续的**.

**数组在物理层面上的线性存储**

实际上，数据在内存中都可以视作线性存放的，因此在一定的规则下，通过动态开辟一维数组的空间，即可在其上存储 n 维的数组.

此外，亦可以根据「数组的数组」这一概念来进行内存的获取与使用. 对于一个存放的若干数组的数组，实际上为一个存放的若干数组的首地址的数组，也就是一个存放若干指针变量的数组.

我们需要一个变量来存放这个「数组的数组」的首地址——也就是一个指针的地址. 这个变量便是一个「指向指针的指针」，有时也称作「二重指针」，如:

```
int** a = new int*[5];
```

接着，我们需要为每一个数组申请空间:

```
for (int i = 0; i < 5; i++) {
  a[i] = new int[5];
}
```

至此，我们便完成了内存的获取. 而对于这样获得的内存的释放，则需要进行一个逆向的操作: 即先释放每一个数组，再释放存储这些数组首地址的数组，如:

```
for (int i = 0; i < 5; i++) {
  delete[] a[i];
}
delete[] a;
```

需要注意，这样获得的二维数组，不能保证其空间是连续的.

还有一种方式，需要使用到「指向数组的指针」.

**数组名和数组首元素地址的区别**

我们之前说到，在 C/C++ 中，直接使用数组名，值等于数组首元素的地址. 但是数组名表示的这一变量的类型实际上是整个数组，而非单个元素.

```
int main() { int a[5] = {1, 2, 3, 4, 5}; }
```

从概念上说，代码中标识符 `a` 的类型是 `int[5]`; 从实际上来说，`a + 1` 所指向的地址相较于 `a` 指向的地址的偏移量为 5 个 `int` 型变量的长度.

```
int main() {
  int(*a)[5] = new int[5][5];
  int* p = a[2];
  a[2][1] = 1;
  delete[] a;
}
```

这种方式获得到的也是连续的内存，但是可以直接使用 `a[n]` 的形式获得到数组的第 n + 1 行（row）的首地址，因此，使用 `a[r][c]` 的形式即可访问到下标为 `(r, c)` 的元素.

由于指向数组的指针也是一种确定的数据类型，因此除数组的第一维外，其他维度的长度均须为一个能在编译器确定的常量．不然，编译器将无法翻译如 `a[n]` 这样的表达式（`a` 为指向数组的指针）．

### 指向函数的指针

关于函数的介绍请参见 C++ 函数→4.2.7@p363 章节．

简单地说，要调用一个函数，需要知晓该函数的参数类型、个数以及返回值类型，这些也统一称作接口类型．

可以通过函数指针调用函数．有时候，若干个函数的接口类型是相同的，使用函数指针可以根据程序的运行 **动态地** 选择需要调用的函数．换句话说，可以在不修改一个函数的情况下，仅通过修改向其传入的参数（函数指针），使得该函数的行为发生变化．

假设我们有若干针对 `int` 类型的二元运算函数，则函数的参数为 2 个 `int`，返回值亦为 `int`．下边是一个使用了函数指针的例子：

```
#include <iostream>

int (*binary_int_op)(int, int);

int foo1(int a, int b) { return a * b + b; }

int foo2(int a, int b) { return (a + b) * b; }

int main() {
  int choice;
  std::cin >> choice;
  if (choice == 1) {
    binary_int_op = foo1;
  } else {
    binary_int_op = foo2;
  }

  int m, n;
  std::cin >> m >> n;
  std::cout << binary_int_op(m, n);
}
```

#### &、* 和函数指针

在 C 语言中，诸如 `void (*p)() = foo;`、`void (*p)() = &foo;`、`void (*p)() = *foo;`、`void (*p)() = ***foo` 等写法的结果是一样的．

因为函数（如 `foo`）是能够被隐式转换为指向函数的指针的，因此 `void (*p)() = foo;` 的写法能够成立．

使用 `&` 运算符可以取得到对象的地址，这对函数也是成立的，因此 `void (*p)() = &foo;` 的写法仍然成立．

对函数指针使用 `*` 运算符可以取得指针指向的函数，而对于 `**foo` 这样的写法来说，`*foo` 得到的是 `foo` 这个函数，紧接着又被隐式转换为指向 `foo` 的指针．如此类推，`**foo` 得到的最终还是指向 `foo` 的函数指针；用户尽可以使用任意多的 `*`，结果也是一样的．

同理，在调用时使用类似 `(*p)()` 和 `p()` 的语句是一样的，可以省去 `*` 运算符．

> 参考资料：Why do function pointer definitions work with any number of ampersands '&' or asterisks '*'? - stackoverflow.com[4]

可以使用 `typedef` 关键字声明函数指针的类型.

```
typedef int (*p_bi_int_op)(int, int);
```

这样我们就可以在之后使用 `p_bi_int_op` 这种类型，即指向「参数为 2 个 `int`，返回值亦为 `int`」的函数的指针.

可以通过使用 `std::function` 来更方便的引用函数.（未完待续）

使用函数指针，可以实现「回调函数」.（未完待续）

### 参考资料与注释

[1] Introduce the nullptr constant 

[2] Introduce the nullptr constant 

[3] list initialization (since C++11) 

[4] Why do function pointer definitions work with any number of ampersands '&' or asterisks '*'? - stackoverflow.com 

## 4.2.7 函数

**Authors:** lr1d, tsagaanbar, yang-lile

### 函数的声明

编程中的函数（function）一般是若干语句的集合. 我们也可以将其称作「**子过程**（subroutine）」. 在编程中，如果有一些重复的过程，我们可以将其提取出来，形成一个函数. 函数可以接收若干值，这叫做函数的参数. 函数也可以返回某个值，这叫做函数的返回值.

声明一个函数，我们需要返回值类型、函数的名称，以及参数列表.

```
// 返回值类型 int
// 函数的名称 some_function
// 参数列表 int, int
int some_function(int, int);
```

如上图，我们声明了一个名为 `some_function` 的函数，它需要接收两个 `int` 类型的参数，返回值类型也为 `int`. 可以认为，这个函数将会对传入的两个整数进行一些操作，并且返回一个同样类型的结果.

### 实现函数：编写函数的定义

只有函数的声明（declaration）还不够，他只能让我们在调用时能够得知函数的 **接口** 类型（即接收什么数据、返回什么数据），但其缺乏具体的内部实现，也就是函数的 **定义**（definition）. 我们可以在 **声明之后的其他地方** 编写代码 **实现**（implement）这个函数（也可以在另外的文件中实现，但是需要将分别编译后的文件在链接时一并给出）.

如果函数有返回值，则需要通过 `return` 语句，将值返回给调用方. 函数一旦执行到 `return` 语句，则直接结束当前函数，不再执行后续的语句.

```
int some_function(int, int);  // 声明

/* some other code here... */

int some_function(int x, int y) {  // 定义
  int result = 2 * x + y;
```

```
  return result;
  result = 3;  // 这条语句不会被执行
}
```

在定义时，我们给函数的参数列表的变量起了名字. 这样，我们便可以在函数定义中使用这些变量了.

如果是同一个文件中，我们也可以直接将 **声明和定义合并在一起**，换句话说，也就是在声明时就完成定义.

```
int some_function(int x, int y) { return 2 * x + y; }
```

如果函数不需要有返回值，则将函数的返回值类型标为 `void`；如果函数不需要参数，则可以将参数列表置空. 同样，无返回值的函数执行到 `return;` 语句也会结束执行.

```
void say_hello() {
  cout << "hello!\n";
  cout << "hello!\n";
  cout << "hello!\n";
  return;
  cout << "hello!\n";  // 这条语句不会被执行
}
```

## 函数的调用

和变量一样，函数需要先被声明，才能使用. 使用函数的行为，叫做「调用（call）」. 我们可以在任何函数内部调用其他函数，包括这个函数自身. 函数调用自身的行为，称为 **递归**（recursion）.

在大多数语言中，调用函数的写法，是 **函数名称加上一对括号** `()`，如 `foo()`. 如果函数需要参数，则我们将其需要的参数按顺序填写在括号中，以逗号间隔，如 `foo(1, 2)`. 函数的调用也是一个表达式，**函数的返回值** 就是 **表达式的值**.

函数声明时候写出的参数，可以理解为在函数 **当前次调用的内部** 可以使用的变量，这些变量的值由调用处传入的值初始化. 看下面这个例子：

```
void foo(int, int);

/* ... */

void foo(int x, int y) {
  x = x * 2;
  y = y + 3;
}

/* ... */

a = 1;
b = 1;
// 调用前：a = 1, b = 1
foo(a, b);  // 调用 foo
            // 调用后：a = 1, b = 1
```

在上面的例子中，`foo(a, b)` 是一次对 `foo` 的调用. 调用时，`foo` 中的 `x` 和 `y` 变量，分别由调用处 `a` 和 `b` 的值初始化. 因此，在 `foo` 中对变量 `x` 和 `y` 的修改，**并不会影响到调用处的变量的值**.

如果我们需要在函数（子过程）中修改变量的值，则需要采用「传引用」的方式.

```
void foo(int& x, int& y) {
```

```
  x = x * 2;
  y = y + 3;
}

/* ... */

a = 1;
b = 1;
// 调用前：a = 1, b = 1
foo(a, b);  // 调用 foo
            // 调用后：a = 2, b = 4
```

上述代码中，我们看到函数参数列表中的「`int`」后面添加了一个「`&`（and 符号）」，这表示对于 `int` 类型的 **引用**（reference）．在调用 `foo` 时，调用处 `a` 和 `b` 变量分别初始化了 `foo` 中两个对 `int` 类型的引用 `x` 和 `y`．在 `foo` 中的 `x` 和 `y`，可以理解为调用处 `a` 和 `b` 变量的「别名」，即 `foo` 中对 `x` 和 `y` 的操作，就是对调用处 `a` 和 `b` 的操作．

### `main` 函数

特别的，每个 C/C++ 程序都需要有一个名为 `main` 的函数．任何程序都将从 `main` 函数开始运行．

> `main` 函数也可以有参数，通过 `main` 函数的参数，我们可以获得外界传给这个程序的指令（也就是「命令行参数」），以便做出不同的反应．

下面是一段调用了函数（子过程）的代码：

```
// hello_subroutine.cpp

#include <iostream>

void say_hello() {
  std::cout << "hello!\n";
  std::cout << "hello!\n";
  std::cout << "hello!\n";
}

int main() {
  say_hello();
  say_hello();
}
```

## 4.2.8 文件操作

**Authors:** Ir1d, cqnuljs, akakw1, MingqiHuang, Chrogeek, henrytbtrue, Planet6174, StudyingFather

### 文件的概念

文件是根据特定的目的而收集在一起的有关数据的集合．C/C++ 把每一个文件都看成是一个有序的字节流，每个文件都是以 **文件结束标志**（EOF）结束，如果要操作某个文件，程序应该首先打开该文件，每当一个文件被打开后（请记得关闭打开的文件），该文件就和一个流关联起来，这里的流实际上是一个字节序列．

C/C++ 将文件分为文本文件和二进制文件．文本文件就是简单的文本文件（重点），另外二进制文件就是特殊格式的文件或者可执行代码文件等．

### 文件的操作步骤

1、打开文件，将文件指针指向文件，决定打开文件类型；

2、对文件进行读、写操作（比赛中主要用到的操作，其他一些操作暂时不写）；

3、在使用完文件后，关闭文件.

### `freopen` 函数

#### 函数简介

函数用于将指定输入输出流以指定方式重定向到文件，包含于头文件 `stdio.h (cstdio)` 中，该函数可以在不改变代码原貌的情况下改变输入输出环境，但使用时应当保证流是可靠的.

函数主要有三种方式：读、写和附加.

#### 命令格式

```
FILE* freopen(const char* filename, const char* mode, FILE* stream);
```

#### 参数说明

- `filename`: 要打开的文件名
- `mode`: 文件打开的模式，表示文件访问的权限
- `stream`: 文件指针，通常使用标准文件流 (`stdin/stdout`) 或标准错误输出流 (`stderr`)
- 返回值：文件指针，指向被打开文件

#### 文件打开格式（选读）

- `r`：以只读方式打开文件，文件必须存在，只允许读入数据 **（常用）**
- `r+`：以读/写方式打开文件，文件必须存在，允许读/写数据
- `rb`：以只读方式打开二进制文件，文件必须存在，只允许读入数据
- `rb+`：以读/写方式打开二进制文件，文件必须存在，允许读/写数据
- `rt+`：以读/写方式打开文本文件，允许读/写数据
- `w`：以只写方式打开文件，文件不存在会新建文件，否则清空内容，只允许写入数据 **（常用）**
- `w+`：以读/写方式打开文件，文件不存在将新建文件，否则清空内容，允许读/写数据
- `wb`：以只写方式打开二进制文件，文件不存在将会新建文件，否则清空内容，只允许写入数据
- `wb+`：以读/写方式打开二进制文件，文件不存在将新建文件，否则清空内容，允许读/写数据
- `a`：以只写方式打开文件，文件不存在将新建文件，写入数据将被附加在文件末尾（保留 EOF 符）
- `a+`: 以读/写方式打开文件, 文件不存在将新建文件, 写入数据将被附加在文件末尾（不保留 EOF 符）
- `at+`：以读/写方式打开文本文件，写入数据将被附加在文件末尾
- `ab+`：以读/写方式打开二进制文件，写入数据将被附加在文件末尾

#### 使用方法

读入文件内容：

```
freopen("data.in", "r", stdin);
// data.in 就是读取的文件名，要和可执行文件放在同一目录下
```

输出到文件：

```
freopen("data.out", "w", stdout);
// data.out 就是输出文件的文件名，和可执行文件在同一目录下
```

关闭标准输入/输出流

```
```

```
fclose(stdin);
fclose(stdout);
```

> **注**
>
> `printf/scanf/cin/cout` 等函数默认使用 `stdin/stdout`，将 `stdin/stdout` 重定向后，这些函数将输入/输出到被定向的文件

### 模板

```
#include <cstdio>
#include <iostream>

int main(void) {
  freopen("data.in", "r", stdin);
  freopen("data.out", "w", stdout);
  /*
  中间的代码不需要改变，直接使用 cin 和 cout 即可
  */
  fclose(stdin);
  fclose(stdout);
  return 0;
}
```

## `fopen` 函数（选读）

函数大致与 `freopen` 相同，函数将打开指定文件并返回打开文件的指针

### 函数原型

```
FILE* fopen(const char* path, const char* mode)
```

各项参数含义同 `freopen`

### 可用读写函数（基本）

- `fread/fwrite`
- `fgetc/fputc`
- `fscanf/fprintf`
- `fgets/fputs`

### 使用方式

```
FILE *in, *out;  // 定义文件指针
in = fopen("data.in", "r");
out = fopen("data.out", "w");
/*
do what you want to do
*/
fclose(in);
fclose(out);
```

### C++ 的 `ifstream/ofstream` 文件输入输出流

#### 使用方法

读入文件内容：

```
ifstream fin("data.in");
// data.in 就是读取文件的相对位置或绝对位置
```

输出到文件：

```
ofstream fout("data.out");
// data.out 就是输出文件的相对位置或绝对位置
```

关闭标准输入/输出流

```
fin.close();
fout.close();
```

#### 模板

```
#include <fstream>
using namespace std;  // 两个类型都在 std 命名空间里

ifstream fin("data.in");
ofstream fout("data.out");

int main(void) {
  /*
  中间的代码改变 cin 为 fin ，cout 为 fout 即可
  */
  fin.close();
  fout.close();
  return 0;
}
```

#### 参考资料

1. 信息学奥赛一本通

# 4.3 C++ 标准库

## 4.3.1 C++ 标准库简介

### C++ 标准

首先需要介绍的是 C++ 本身的版本．由于 C++ 本身只是一门语言，而不同的编译器对 C++ 的实现方法各不一致，因此需要标准化来约束编译器的实现，使得 C++ 代码在不同的编译器下表现一致．C++ 自 1985 年诞生以来，一共由国际标准化组织（ISO）发布了 7 个正式的 C++ 标准，依次为 C++98、C++03、C++11（亦称 C++0x）、C++14（亦称 C++1y）、C++17（亦称 C++1z）、C++20（亦称 C++2a）、C++23（亦称 C++2b）．C++ 标准草案在 open-std[1] 网站上，最新标准的制定进度可以在 Current Status : Standard C++[2] 查看．此外还有一些补充标准，例如 C++ TR1.

每一个版本的 C++ 标准不仅规定了 C++ 的语法、语言特性，还规定了一套 C++ 内置库的实现规范，这个库便是 C++ 标准库．C++ 标准库中包含大量常用代码的实现，如输入输出、基本数据结构、内存管理、多线程支持等．掌握 C++ 标准库是编写更现代的 C++ 代码必要的一步．C++ 标准库的详细文档在 cppreference[3] 网站上，文档对标准库中的类型函数的用法、效率、注意事项等都有介绍，请善用．

需要指出的是，不同的 OJ 平台对 C++ 版本均不相同，例如 最新的 ICPC 比赛规则[4] 支持 C++20 标准. 根据 NOI 科学委员会决议，自 2021 年 9 月 1 日起 NOI Linux 2.0[5] 作为 NOI 系列比赛和 CSP-J/S 等活动的标准环境使用. NOI Linux 2.0 中指定的 g++ 9.3.0 默认支持标准[6] 为 C++14，并支持 C++17 标准，可以满足绝大部分竞赛选手的需求. 因此在学习 C++ 时要注意比赛支持的标准，避免在赛场上时编译报错.

## 标准模板库（STL）

STL 即标准模板库（Standard Template Library），是 C++ 标准库的一部分，里面包含了一些模板化的通用的数据结构和算法. 由于其模板化的特点，它能够兼容自定义的数据类型，避免大量的造轮子工作. NOI 和 ICPC 赛事都支持 STL 库的使用，因此合理利用 STL 可以避免编写无用算法，并且充分利用编译器对模板库优化提高效率. STL 库的详细介绍请参见对应的页面：STL 容器→4.3.2.1@p370 和 STL 算法→4.3.3@p386.

> **什么是造轮子**
>
> 造轮子（Reinventing_the_wheel[7]）指的是重复发明已有的算法，或者重复编写现成优化过的代码. 造轮子通常耗时耗力，同时效果还没有别人好. 但若是为了学习或者练习，造轮子则是必要的.

## Boost 库

Boost[8] 是除了标准库外，另一个久副盛名的开源 C++ 工具库，其代码具有可移植、高质量、高性能、高可靠性等特点. Boost 中的模块数量非常之大，功能全面，并且拥有完备的跨平台支持，因此被看作 C++ 的准标准库. C++ 标准中的不少特性也都来自于 Boost，如智能指针、元编程、日期和时间等. 尽管在 OI 中无法使用 Boost，但是 Boost 中有不少轮子可以用来验证算法或者对拍，如 Boost.Geometry 有 R 树的实现，Boost.Graph 有图的相关算法，Boost.Intrusive 则提供了一套与 STL 容器用法相似的侵入式容器. 有兴趣的读者可以自行在网络搜索教程.

## 参考资料


1. C++ reference[9]
2. C++ 参考手册[10]
3. 维基百科 - C++[11]
4. Boost 官方网站[12]
5. Boost 教程网站[13]


## 外部参考资料


[1] open-std

[2] Current Status : Standard C++

[3] cppreference

[4] 最新的 ICPC 比赛规则

[5] NOI Linux 2.0

[6] 默认支持标准

[7] Reinventing_the_wheel

[8] Boost

[9] C++ reference

[10] C++ 参考手册

[11] 维基百科 - C++

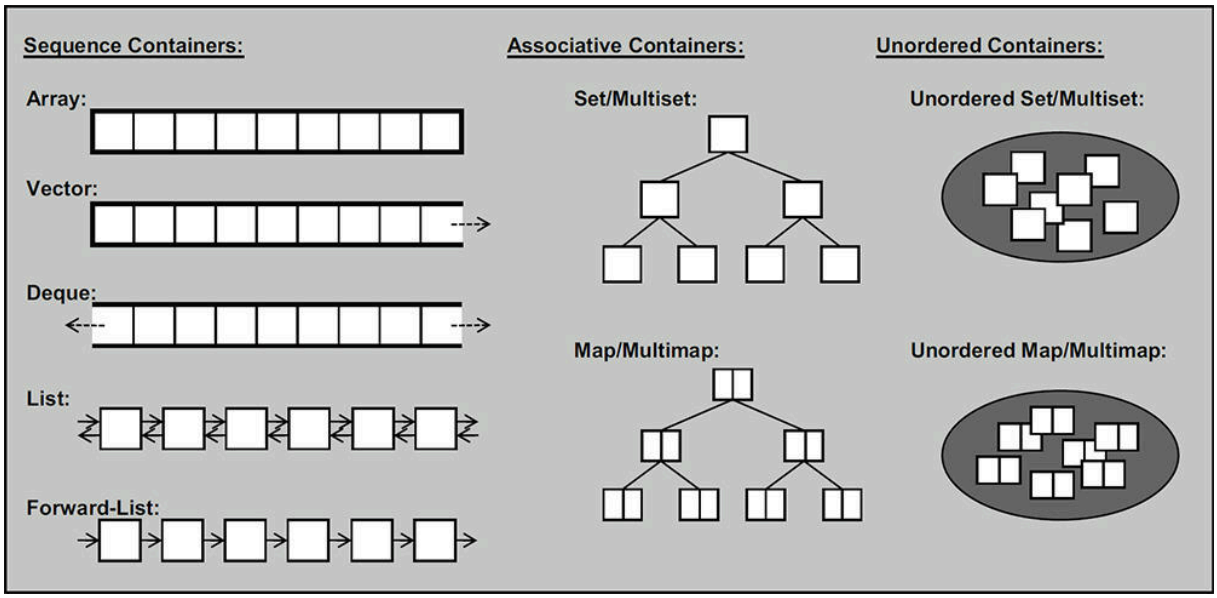
[12] Boost 官方网站

[13] Boost 教程网站

## 4.3.2 STL 容器

### 4.3.2.1 STL 容器简介

#### 分类




**序列式容器**

- **向量**(`vector`) 后端可高效增加元素的顺序表.
- **数组**(`array`)C++11，定长的顺序表，C 风格数组的简单包装.
- **双端队列**(`deque`) 双端都可高效增加元素的顺序表.
- **列表**(`list`) 可以沿双向遍历的链表.
- **单向列表**(`forward_list`) 只能沿一个方向遍历的链表.

**关联式容器**

- **集合**(`set`) 用以有序地存储 **互异** 元素的容器. 其实现是由节点组成的红黑树，每个节点都包含着一个元素，节点之间以某种比较元素大小的谓词进行排列.
- **多重集合**(`multiset`) 用以有序地存储元素的容器. 允许存在相等的元素.
- **映射**(`map`) 由 {键，值} 对组成的集合，以某种比较键大小关系的谓词进行排列.
- **多重映射**(`multimap`) 由 {键，值} 对组成的多重集合，亦即允许键有相等情况的映射.

> **什么是谓词 (Predicate[1])?**
>
> 谓词就是返回值为真或者假的函数. STL 容器中经常会使用到谓词，用于模板参数.

**无序（关联式）容器**

- **无序（多重）集合**(`unordered_set/unordered_multiset`)C++11，与 `set/multiset` 的区别在于元素无序，只关心「元素是否存在」，使用哈希实现.
- **无序（多重）映射**(`unordered_map/unordered_multimap`)C++11，与 `map/multimap` 的区别在于键 (key) 无序，只关心 "键与值的对应关系"，使用哈希实现.

**容器适配器**

容器适配器其实并不是容器. 它们不具有容器的某些特点（如：有迭代器、有 `clear()` 函数……）.

> 「适配器是使一种事物的行为类似于另外一种事物行为的一种机制」，适配器对容器进行包装，使其表现出另外一种行为.

- **栈**(`stack`) 后进先出 (LIFO) 的容器，默认是对双端队列（`deque`）的包装.
- **队列**(`queue`) 先进先出 (FIFO) 的容器，默认是对双端队列（`deque`）的包装.
- **优先队列**(`priority_queue`) 元素的次序是由作用于所存储的值对上的某种谓词决定的一种队列，默认是对向量（`vector`）的包装.

### 共同点

#### 容器声明

都是 `containerName<typeName,...> name` 的形式，但模板参数（`<>` 内的参数）的个数、形式会根据具体容器而变.

本质原因：STL 就是「标准模板库」，所以容器都是模板类.

#### 迭代器

请参考 迭代器→4.3.2.2@p371.

#### 共有函数

`=`：有赋值运算符以及复制构造函数.

`begin()`：返回指向开头元素的迭代器.

`end()`：返回指向末尾的下一个元素的迭代器. `end()` 不指向某个元素，但它是末尾元素的后继.

`size()`：返回容器内的元素个数.

`max_size()`：返回容器 **理论上** 能存储的最大元素个数. 依容器类型和所存储变量的类型而变.

`empty()`：返回容器是否为空.

`swap()`：交换两个容器.

`clear()`：清空容器.

`==`/`!=`/`<`/`>`/`<=`/`>=`：按 **字典序** 比较两个容器的大小.（比较元素大小时 `map` 的每个元素相当于 `set<pair<key, value>>`，无序容器不支持 `<`/`>`/`<=`/`>=`.）

### 外部参考资料

[1] Predicate 

## 4.3.2.2 迭代器

在 STL 中，迭代器（Iterator）用来访问和检查 STL 容器中元素的对象，它的行为模式和指针类似，但是它封装了一些有效性检查，并且提供了统一的访问格式. 类似的概念在其他很多高级语言中都存在，如 Python 的 `__iter__` 函数，C# 的 `IEnumerator`.

### 基础使用

迭代器听起来比较晦涩，其实迭代器本身可以看作一个数据指针. 迭代器主要支持两个运算符：自增 (`++`) 和解引用（单目 `*` 运算符），其中自增用来移动迭代器，解引用可以获取或修改它指向的元素.

指向某个 STL 容器→4.3.2.1@p370 `container` 中元素的迭代器的类型一般为 `container::iterator`.

迭代器可以用来遍历容器，例如，下面两个 for 循环的效果是一样的：

```
1 vector<int> data(10);
2
```

```cpp
for (int i = 0; i < data.size(); i++)
  cout << data[i] << endl;  // 使用下标访问元素

for (vector<int>::iterator iter = data.begin(); iter != data.end(); iter++)
  cout << *iter << endl;  // 使用迭代器访问元素
// 在C++11后可以使用 auto iter = data.begin() 来简化上述代码
```

**auto 在竞赛中的使用**

大部分选手都喜欢使用 `auto` 来代替繁琐的迭代器声明. 根据 2021 年 9 月发布的 关于 NOI 系列活动中编程语言使用限制的补充说明[1]，NOI 系列比赛（包括 CSP J/S）在评测时将使用 C++14，这个版本已经支持了 `auto` 关键字.

### 分类

在 STL 的定义中，迭代器根据其支持的操作依次分为以下几类:

- InputIterator（输入迭代器）：只要求支持拷贝、自增和解引访问.
- OutputIterator（输出迭代器）：只要求支持拷贝、自增和解引赋值.
- ForwardIterator（前向迭代器）：在 InputIterator 的基础上支持多次遍历迭代器解引访问，且保证多次访问的结果一致.
- BidirectionalIterator（双向迭代器）：在 ForwardIterator 的基础上支持自减（即反向访问）.
- RandomAccessIterator（随机访问迭代器）：在 BidirectionalIterator 的基础上支持加减运算和比较运算（即随机访问）.
- ContiguousIterator（连续迭代器）：在 RandomAccessIterator 的基础上要求对可解引用的迭代器 `a + n` 满足 `*(a + n)` 与 `*(std::address_of(*a) + n)` 等价（即连续存储，其中 `a` 为连续迭代器、`n` 为整型值）.

  ContiguousIterator 于 C++17 中正式引入.

**为什么输入迭代器叫输入迭代器?**

「输入」指的是「可以从迭代器中获取输入」，而「输出」指的是「可以输出到迭代器」.

「输入」和「输出」的施动者是程序的其它部分，而不是迭代器自身.

迭代器的这些分类并不互斥. 实际上，除了输出迭代器之外，列表中排在前面的迭代器都包含着排在后面的迭代器. 例如，在要求使用前向迭代器的地方，同样可以使用双向迭代器. 从前向迭代器开始，如果这些迭代器还实现了输出迭代器的功能（即允许写操作），就称它们是可变迭代器. 由此，可以衍生出诸如「可变随机访问迭代器」这样的类别.

不同的 STL 容器→4.3.2.1@p370 支持的迭代器类型不同，在使用时需要留意.

数组指针满足连续迭代器（或随机访问迭代器，对于 C++14 及以前的版本）的所有要求，可以当作连续迭代器使用.

### 相关函数

很多 STL 函数→4.3.3@p386 都使用迭代器作为参数.

可以使用 `std::advance(it, n)` 将迭代器 `it` 向后移动 `n` 步；若 `n` 为负数，则对应向前移动，此时迭代器必须满足双向迭代器，否则行为未定义.

在 C++11 以后可以使用 `std::next(it)` 获得前向迭代器 `it` 的后继（此时迭代器 `it` 不变），`std::next(it, n)` 获得前向迭代器 `it` 的第 `n` 个后继.

在 C++11 以后可以使用 `std::prev(it)` 获得双向迭代器 `it` 的前驱（此时迭代器 `it` 不变），`std::prev(it, n)` 获得双向迭代器 `it` 的第 `n` 个前驱.

STL 容器→4.3.2.1@p370 一般支持从一端或两端开始的访问，以及对 const 修饰符→4.4.6@p427 的支持. 例如容器的 `begin()` 函数可以获得指向容器第一个元素的迭代器，`rbegin()` 函数可以获得指向容器最后一个元素的反向迭代器，`cbegin()` 函数可以获得指向容器第一个元素的 const 迭代器，`end()` 函数可以获得指向容器尾端（「尾端」并不是最后一个元素，可以看作是最后一个元素的后继；「尾端」的前驱是容器里的最后一个元素，其本身不指向任何一个元素）的迭代器.

可在 Iterator library - cppreference.com[2] 查看更多用法.

### 外部参考资料

[1] 关于 NOI 系列活动中编程语言使用限制的补充说明 

[2] Iterator library - cppreference.com 

## 4.3.2.3 序列式容器

Authors: MingqiHuang, Xeonacid, greyqz, i-Yirannn, ChenZ01

#### `vector`

`std::vector` 是 STL 提供的 **内存连续的、可变长度** 的数组（亦称列表）数据结构. 能够提供线性复杂度的插入和删除，以及常数复杂度的随机访问.

#### 为什么要使用 `vector`

作为 OIer，对程序效率的追求远比对工程级别的稳定性要高得多，而 `vector` 由于其对内存的动态处理，时间效率在部分情况下低于静态数组，并且在 OJ 服务器不一定开全优化的情况下更加糟糕. 所以在正常存储数据的时候，通常不选择 `vector`. 下面给出几个 `vector` 优秀的特性，在需要用到这些特性的情况下，`vector` 能给我们带来很大的帮助.

#### `vector` 可以动态分配内存

很多时候我们不能提前开好那么大的空间（eg：预处理 1~n 中所有数的约数）. 尽管我们能知道数据总量在空间允许的级别，但是单份数据还可能非常大，这种时候我们就需要 `vector` 来把内存占用量控制在合适的范围内. `vector` 还支持动态扩容，在内存非常紧张的时候这个特性就能派上用场了.

#### `vector` 重写了比较运算符及赋值运算符

`vector` 重载了六个比较运算符，以字典序实现，这使得我们可以方便的判断两个容器是否相等（复杂度与容器大小成线性关系）. 例如可以利用 `vector<char>` 实现字符串比较（当然，还是用 `std::string` 会更快更方便）. 另外 `vector` 也重载了赋值运算符，使得数组拷贝更加方便.

#### `vector` 便利的初始化

由于 `vector` 重载了 `=` 运算符，所以我们可以方便地进行 `vector` 的整体赋值操作. 此外从 C++11 起 `vector` 还支持 列表初始化[1]，例如 `vector<int> data {1, 2, 3};`.

#### `vector` 的使用方法

以下介绍常用用法，详细内容 请参见 C++ 文档[2].

#### 构造函数

用例参见如下代码（假设你已经 `using` 了 `std` 命名空间相关类型）：

```
// 1. 创建空vector; 常数复杂度
vector<int> v0;
// 1+. 这句代码可以使得向vector中插入前3个元素时，保证常数时间复杂度
v0.reserve(3);
```

```cpp
// 2. 创建一个初始空间为3的vector，其元素的默认值是0; 线性复杂度
vector<int> v1(3);
// 3. 创建一个初始空间为3的vector，其元素的默认值是2; 线性复杂度
vector<int> v2(3, 2);
// 4. 创建一个初始空间为3的vector，其元素的默认值是1,
// 并且使用v2的空间配置器; 线性复杂度
vector<int> v3(3, 1, v2.get_allocator());
// 5. 创建一个v2的拷贝vector v4,  其内容元素和v2一样; 线性复杂度
vector<int> v4(v2);
// 6. 创建一个v4的拷贝vector v5，其内容是{v4[1], v4[2]}; 线性复杂度
vector<int> v5(v4.begin() + 1, v4.begin() + 3);
// 7. 移动v2到新创建的vector v6，不发生拷贝; 常数复杂度; 需要 C++11
vector<int> v6(std::move(v2));  // 或者 v6 = std::move(v2);
```

**测试代码**

```cpp
// 以下是测试代码，有兴趣的同学可以自己编译运行一下本代码.
cout << "v1 = ";
copy(v1.begin(), v1.end(), ostream_iterator<int>(cout, " "));
cout << endl;
cout << "v2 = ";
copy(v2.begin(), v2.end(), ostream_iterator<int>(cout, " "));
cout << endl;
cout << "v3 = ";
copy(v3.begin(), v3.end(), ostream_iterator<int>(cout, " "));
cout << endl;
cout << "v4 = ";
copy(v4.begin(), v4.end(), ostream_iterator<int>(cout, " "));
cout << endl;
cout << "v5 = ";
copy(v5.begin(), v5.end(), ostream_iterator<int>(cout, " "));
cout << endl;
cout << "v6 = ";
copy(v6.begin(), v6.end(), ostream_iterator<int>(cout, " "));
cout << endl;
```

可以利用上述的方法构造一个 `vector`，足够我们使用了.

**元素访问**

`vector` 提供了如下几种方法进行元素访问

1. `at()`

   `v.at(pos)` 返回容器中下标为 `pos` 的引用. 如果数组越界抛出 `std::out_of_range` 类型的异常.

2. `operator[]`

   `v[pos]` 返回容器中下标为 `pos` 的引用. 不执行越界检查.

3. `front()`

   `v.front()` 返回首元素的引用.

4. `back()`

   `v.back()` 返回末尾元素的引用.

5. `data()`

   `v.data()` 返回 `v` 内部存储数据使用的连续内存空间中首元素的指针.

#### 迭代器

vector 提供了如下几种 迭代器→4.3.2.2@p371

1. `begin()/cbegin()`

   返回指向首元素的迭代器，其中 `*begin = front`.

2. `end()/cend()`

   返回指向容器尾端占位符的迭代器，注意是没有元素的.

3. `rbegin()/crbegin()`

   返回指向逆向数组的首元素的逆向迭代器，可以理解为正向容器的末元素.

4. `rend()/crend()`

   返回指向逆向数组末元素后一位置的迭代器，对应容器首的前一个位置，没有元素.

以上列出的迭代器中，含有字符 `c` 的为只读迭代器，你不能通过只读迭代器去修改 `vector` 中的元素的值. 如果一个 `vector` 本身就是只读的，那么它的一般迭代器和只读迭代器完全等价. 只读迭代器自 C++11 开始支持.

#### 长度和容量

`vector` 有以下几个与容器长度和容量相关的函数. 注意，`vector` 的长度（size）指有效元素数量，而容量（capacity）指其实际分配的内存长度，相关细节请参见后文的实现细节介绍.

**与长度相关**:

- `empty()` 返回一个 `bool` 值，即 `v.begin() == v.end()`，`true` 为空，`false` 为非空.
- `size()` 返回容器长度（元素数量），即 `std::distance(v.begin(), v.end())`.
- `resize(n)` 改变 `vector` 的长度为 `n`. 如果 `n` 大于当前长度，则会补充元素，如果参数中提供了要补充的元素，则使用参数，否则使用默认值；如果 `n` 小于当前长度，则保留前 `n` 个元素，后续元素删除.
- `max_size()` 返回容器的最大可能长度.

**与容量相关**:

- `reserve()` 使得 `vector` 预留一定的内存空间，避免不必要的内存分配与拷贝.
- `capacity()` 返回容器的容量，即当前 `vector` 已经为多少个元素分配了空间.
- `shrink_to_fit()` 使得 `vector` 的容量与长度一致，去除该 `vector` 没有用到的容量.

#### 元素增删及修改

- `clear()` 清除所有元素
- `insert()` 支持在某个迭代器位置插入元素、可以插入多个. **复杂度与 `pos` 距离末尾长度成线性而非常数的**
- `erase()` 删除某个迭代器或者区间的元素，返回最后被删除的迭代器. 复杂度与 `insert` 一致.
- `push_back()` 在末尾插入一个元素，均摊复杂度为 **常数**，最坏为线性复杂度.
- `pop_back()` 删除末尾元素，常数复杂度.
- `swap()` 与另一个容器进行交换，此操作是 **常数复杂度** 而非线性的.

#### `vector` 的实现细节

`vector` 的底层其实仍然是定长数组，它能够实现动态扩容的原因是增加了避免数量溢出的操作. 首先需要指明的是 `vector` 中元素的数量（长度）$n$ 与它已分配内存最多能包含元素的数量（容量）$N$ 是不一致的，`vector` 会分开存储这两个量. 当向 `vector` 中添加元素时，如发现 $n > N$，那么容器会分配一个尺寸为 $2N$ 的数组，然后将旧数据从原本的位置拷贝到新的数组中，再将原来的内存释放. 尽管这个操作的渐近复杂度是 $O(n)$，但是可以证明其均摊复杂度为 $O(1)$. 而在末尾删除元素和访问元素则都仍然是 $O(1)$ 的开销. 因此，只要对 `vector` 的尺寸估计得当并善用 `resize()` 和 `reserve()`，就能使得 `vector` 的效率与定长数组不会有太大差距.

#### vector<bool>

标准库特别提供了对 bool 的 vector 特化，每个「bool」只占 1 bit，且支持动态增长．但是其 operator[] 的返回值的类型不是 bool& 而是 vector<bool>::reference．因此，使用 vector<bool> 使需谨慎，可以考虑使用 deque<bool> 或 vector<char> 替代．而如果你需要节省空间，请直接使用 bitset→4.3.4@p389.

### array(C++11)

std::array 是 STL 提供的 **内存连续的、固定长度** 的数组数据结构. 其本质是对原生数组的直接封装.

#### 为什么要用 array

array 实际上是 STL 对数组的封装．它相比 vector 牺牲了动态扩容的特性，但是换来了与原生数组几乎一致的性能（在开满优化的前提下）．因此如果能使用 C++11 特性的情况下，能够使用原生数组的地方几乎都可以直接把定长数组都换成 array，而动态分配的数组可以替换为 vector.

#### 成员函数

#### 隐式定义的成员函数

| 函数 | 作用 |
|---|---|
| operator= | 以来自另一 array 的每个元素重写 array 的对应元素 |

#### 元素访问

| 函数 | 作用 |
|---|---|
| at | 访问指定的元素，同时进行越界检查 |
| operator[] | 访问指定的元素，**不** 进行越界检查 |
| front | 访问第一个元素 |
| back | 访问最后一个元素 |
| data | 返回指向内存中数组第一个元素的指针 |

at 若遇 pos >= size() 的情况会抛出 std::out_of_range.

#### 容量

| 函数 | 作用 |
|---|---|
| empty | 检查容器是否为空 |
| size | 返回容纳的元素数 |
| max_size | 返回可容纳的最大元素数 |

由于每个 array 都是固定大小容器，size() 返回的值等于 max_size() 返回的值.

#### 操作

| 函数 | 作用 |
|---|---|
| fill | 以指定值填充容器 |
| swap | 交换内容 |

**注意，交换两个 array 是 $\Theta(\text{size})$ 的，而非与常规 STL 容器一样为 $O(1)$.**

#### 非成员函数

| 函数 | 作用 |
|---|---|
| operator== 等 | 按照字典序比较 array 中的值 |
| std::get | 访问 array 的一个元素 |
| std::swap | 特化的 std::swap 算法 |

下面是一个 `array` 的使用示例：

```cpp
// 1. 创建空array，长度为3; 常数复杂度
std::array<int, 3> v0;
// 2. 用指定常数创建array; 常数复杂度
std::array<int, 3> v1{1, 2, 3};

v0.fill(1);  // 填充数组

// 访问数组
for (int i = 0; i != arr.size(); ++i) cout << arr[i] << " ";
```

#### deque

`std::deque` 是 STL 提供的 双端队列→10.3@p1980 数据结构. 能够提供线性复杂度的插入和删除，以及常数复杂度的随机访问.

#### deque 的使用方法

以下介绍常用用法，详细内容 请参见 C++ 文档[3]. `deque` 的迭代器函数与 `vector` 相同，因此不作详细介绍.

#### 构造函数

参见如下代码（假设你已经 `using` 了 `std` 命名空间相关类型）：

```cpp
// 1. 定义一个int类型的空双端队列 v0
deque<int> v0;
// 2. 定义一个int类型的双端队列 v1，并设置初始大小为10; 线性复杂度
deque<int> v1(10);
// 3. 定义一个int类型的双端队列 v2，并初始化为10个1; 线性复杂度
deque<int> v2(10, 1);
// 4. 复制已有的双端队列 v1; 线性复杂度
deque<int> v3(v1);
// 5. 创建一个v2的拷贝deque v4，其内容是v4[0]至v4[2]; 线性复杂度
deque<int> v4(v2.begin(), v2.begin() + 3);
// 6. 移动v2到新创建的deque v5，不发生拷贝; 常数复杂度; 需要 C++11
deque<int> v5(std::move(v2));
```

#### 元素访问

与 `vector` 一致，但无法访问底层内存. 其高效的元素访问速度可参考实现细节部分.

- `at()` 返回容器中指定位置元素的引用，执行越界检查，**常数复杂度**.
- `operator[]` 返回容器中指定位置元素的引用. 不执行越界检查，**常数复杂度**.
- `front()` 返回首元素的引用.
- `back()` 返回末尾元素的引用.

#### 迭代器

与 `vector` 一致.

#### 长度

与 `vector` 一致，但是没有 `reserve()` 和 `capacity()` 函数.（仍然有 `shrink_to_fit()` 函数）

#### 元素增删及修改

与 `vector` 一致，并额外有向队列头部增加元素的函数.

- `clear()` 清除所有元素
- `insert()` 支持在某个迭代器位置插入元素、可以插入多个. **复杂度与 `pos` 与两端距离较小者成线性**.

- `erase()` 删除某个迭代器或者区间的元素，返回最后被删除的迭代器．复杂度与 `insert` 一致．
- `push_front()` 在头部插入一个元素，**常数复杂度**．
- `pop_front()` 删除头部元素，**常数复杂度**．
- `push_back()` 在末尾插入一个元素，**常数复杂度**．
- `pop_back()` 删除末尾元素，**常数复杂度**．
- `swap()` 与另一个容器进行交换，此操作是 **常数复杂度** 而非线性的．

#### deque 的实现细节

deque 通常的底层实现是多个不连续的缓冲区，而缓冲区中的内存是连续的．而每个缓冲区还会记录首指针和尾指针，用来标记有效数据的区间．当一个缓冲区填满之后便会在之前或者之后分配新的缓冲区来存储更多的数据．更详细的说明可以参考《STL 源码剖析》deque 实现原理[4]．

### `list`

`std::list` 是 STL 提供的 双向链表→10.4@p1988 数据结构．能够提供线性复杂度的随机访问，以及常数复杂度的插入和删除．

#### `list` 的使用方法

`list` 的使用方法与 `deque` 基本相同，但是增删操作和访问的复杂度不同．详细内容 请参见 C++ 文档[5]．`list` 的迭代器、长度、元素增删及修改相关的函数与 `deque` 相同，因此不作详细介绍．

#### 元素访问

由于 `list` 的实现是链表，因此它不提供随机访问的接口．若需要访问中间元素，则需要使用迭代器．

- `front()` 返回首元素的引用．
- `back()` 返回末尾元素的引用．

#### 操作

`list` 类型还提供了一些针对其特性实现的 STL 算法函数．由于这些算法需要 随机访问迭代器→4.3.2.2@p371，因此 `list` 提供了特别的实现以便于使用．这些算法有 `splice()`、`remove()`、`sort()`、`unique()`、`merge()` 等．

### `forward_list`（C++11）

`std::forward_list` 是 STL 提供的 单向链表→10.4@p1988 数据结构，相比于 `std::list` 减小了空间开销．

#### `forward_list` 的使用方法

`forward_list` 的使用方法与 `list` 几乎一致，但是迭代器只有单向的，因此其具体用法不作详细介绍．详细内容 请参见 C++ 文档[6]

### 外部参考资料

[1] 列表初始化 

[2] 请参见 C++ 文档 

[3] 请参见 C++ 文档 

[4] 《STL 源码剖析》deque 实现原理 

[5] 请参见 C++ 文档 

[6] 请参见 C++ 文档 

## 4.3.2.4 关联式容器

### `set`

`set` 是关联容器，含有键值类型对象的已排序集，搜索、移除和插入拥有对数复杂度．`set` 内部通常采用 红黑树→10.15.9@p2362 实现．平衡二叉树→10.15.1@p2213 的特性使得 `set` 非常适合处理需要同时兼顾查找、插入与删除的情况．

和数学中的集合相似，`set` 中不会出现值相同的元素．如果需要有相同元素的集合，需要使用 `multiset`．`multiset` 的使用方法与 `set` 的使用方法基本相同．

**插入与删除操作**

- `insert(x)` 当容器中没有等价元素的时候，将元素 x 插入到 `set` 中．
- `erase(x)` 删除值为 x 的 **所有** 元素，返回删除元素的个数．
- `erase(pos)` 删除迭代器为 pos 的元素，要求迭代器必须合法．
- `erase(first,last)` 删除迭代器在 $[first, last)$ 范围内的所有元素．
- `clear()` 清空 `set`．

> **insert 函数的返回值**
>
> insert 函数的返回值类型为 `pair<iterator, bool>`，其中 iterator 是一个指向所插入元素（或者是指向等于所插入值的原本就在容器中的元素）的迭代器，而 bool 则代表元素是否插入成功，由于 `set` 中的元素具有唯一性质，所以如果在 `set` 中已有等值元素，则插入会失败，返回 false，否则插入成功，返回 true；`map` 中的 insert 也是如此．

**迭代器**

`set` 提供了以下几种迭代器：

1. `begin()/cbegin()`

   返回指向首元素的迭代器，其中 `*begin = front`．

2. `end()/cend()`

   返回指向数组尾端占位符的迭代器，注意是没有元素的．

3. `rbegin()/crbegin()`

   返回指向逆向数组的首元素的逆向迭代器，可以理解为正向容器的末元素．

4. `rend()/crend()`

   返回指向逆向数组末元素后一位置的迭代器，对应容器首的前一个位置，没有元素．

以上列出的迭代器中，含有字符 `c` 的为只读迭代器，你不能通过只读迭代器去修改 `set` 中的元素的值．如果一个 `set` 本身就是只读的，那么它的一般迭代器和只读迭代器完全等价．只读迭代器自 C++11 开始支持．

**查找操作**

- `count(x)` 返回 `set` 内键为 x 的元素数量．
- `find(x)` 在 `set` 内存在键为 x 的元素时会返回该元素的迭代器，否则返回 `end()`．
- `lower_bound(x)` 返回指向首个不小于给定键的元素的迭代器．如果不存在这样的元素，返回 `end()`．
- `upper_bound(x)` 返回指向首个大于给定键的元素的迭代器．如果不存在这样的元素，返回 `end()`．
- `empty()` 返回容器是否为空．
- `size()` 返回容器内元素个数．

> **`lower_bound` 和 `upper_bound` 的时间复杂度**
>
> `set` 自带的 `lower_bound` 和 `upper_bound` 的时间复杂度为 $O(\log n)$．
>
> 但使用 `algorithm` 库中的 `lower_bound` 和 `upper_bound` 函数对 `set` 中的元素进行查询，时间复杂度为 $O(n)$．

> ⚠ **`nth_element` 的时间复杂度**
>
> `set` 没有提供自带的 `nth_element`. 使用 `algorithm` 库中的 `nth_element` 查找第 $k$ 大的元素时间复杂度为 $O(n)$.
>
> 如果需要实现平衡二叉树所具备的 $O(\log n)$ 查找第 $k$ 大元素的功能，需要自己手写平衡二叉树或权值线段树，或者选择使用 pb_ds 库中的平衡二叉树.

### 使用样例

#### `set` 在贪心中的使用

在贪心算法中经常会需要出现类似 **找出并删除最小的大于等于某个值的元素**. 这种操作能轻松地通过 `set` 来完成.

```
// 现存可用的元素
set<int> available;
// 需要大于等于的值
int x;

// 查找最小的大于等于x的元素
set<int>::iterator it = available.lower_bound(x);
if (it == available.end()) {
  // 不存在这样的元素，则进行相应操作……
} else {
  // 找到了这样的元素，将其从现存可用元素中移除
  available.erase(it);
  // 进行相应操作……
}
```

#### `map`

`map` 是有序键值对容器，它的元素的键是唯一的. 搜索、移除和插入操作拥有对数复杂度. `map` 通常实现为 红黑树→10.15.9@p2362.

设想如下场景：现在需要存储一些键值对，例如存储学生姓名对应的分数：`Tom 0`，`Bob 100`，`Alan 100`. 但是由于数组下标只能为非负整数，所以无法用姓名作为下标来存储，这个时候最简单的办法就是使用 STL 中的 `map`.

`map` 重载了 `operator[]`，可以用任意定义了 `operator <` 的类型作为下标（在 `map` 中叫做 `key`，也就是索引）：

```
map<Key, T> yourMap;
```

其中，`Key` 是键的类型，`T` 是值的类型，下面是使用 `map` 的实例：

```
map<string, int> mp;
```

`map` 中不会存在键相同的元素，`multimap` 中允许多个元素拥有同一键. `multimap` 的使用方法与 `map` 的使用方法基本相同.

> ⚠ **Warning**
>
> 正是因为 `multimap` 允许多个元素拥有同一键的特点，`multimap` 并没有提供给出键访问其对应值的方法.

#### 插入与删除操作

- 可以直接通过下标访问来进行查询或插入操作. 例如 `mp["Alan"]=100`.
- 通过向 `map` 中插入一个类型为 `pair<Key, T>` 的值可以达到插入元素的目的, 例如 `mp.insert(pair<string,int>("Alan",100));;`
- `erase(key)` 函数会删除键为 `key` 的 **所有** 元素. 返回值为删除元素的数量.
- `erase(pos)`: 删除迭代器为 pos 的元素, 要求迭代器必须合法.
- `erase(first,last)`: 删除迭代器在 $[first, last)$ 范围内的所有元素.
- `clear()` 函数会清空整个容器.

> **下标访问中的注意事项**
>
> 在利用下标访问 `map` 中的某个元素时, 如果 `map` 中不存在相应键的元素, 会自动在 `map` 中插入一个新元素, 并将其值设置为默认值 (对于整数, 值为零; 对于有默认构造函数的类型, 会调用默认构造函数进行初始化).
>
> 当下标访问操作过于频繁时, 容器中会出现大量无意义元素, 影响 `map` 的效率. 因此一般情况下推荐使用 `find()` 函数来寻找特定键的元素.

#### 查询操作

- `count(x)`: 返回容器内键为 x 的元素数量. 复杂度为 $O(\log(size) + ans)$ (关于容器大小对数复杂度, 加上匹配个数).
- `find(x)`: 若容器内存在键为 x 的元素, 会返回该元素的迭代器; 否则返回 `end()`.
- `lower_bound(x)`: 返回指向首个不小于给定键的元素的迭代器.
- `upper_bound(x)`: 返回指向首个大于给定键的元素的迭代器. 若容器内所有元素均小于或等于给定键, 返回 `end()`.
- `empty()`: 返回容器是否为空.
- `size()`: 返回容器内元素个数.

#### 使用样例

##### `map` 用于存储复杂状态

在搜索中, 我们有时需要存储一些较为复杂的状态 (如坐标, 无法离散化的数值, 字符串等) 以及与之有关的答案 (如到达此状态的最小步数). `map` 可以用来实现此功能. 其中的键是状态, 而值是与之相关的答案. 下面的示例展示了如何使用 `map` 存储以 `string` 表示的状态.

```cpp
// 存储状态与对应的答案
map<string, int> record;

// 新搜索到的状态与对应答案
string status;
int ans;
// 查找对应的状态是否出现过
map<string, int>::iterator it = record.find(status);
if (it == record.end()) {
  // 尚未搜索过该状态，将其加入状态记录中
  record[status] = ans;
  // 进行相应操作……
} else {
  // 已经搜索过该状态，进行相应操作……
}
```

#### 遍历容器

可以利用迭代器来遍历关联式容器的所有元素.

```
set<int> s;
using si = set<int>::iterator;
for (si it = s.begin(); it != s.end(); it++) cout << *it << endl;
```

需要注意的是，对 `map` 的迭代器解引用后，得到的是类型为 `pair<Key, T>` 的键值对.

在 C++11 中，使用范围 for 循环会让代码简洁很多:

```
set<int> s;
for (auto x : s) cout << x << endl;
```

对于任意关联式容器，使用迭代器遍历容器的时间复杂度均为 $O(n)$.

#### 自定义比较方式

`set` 在默认情况下的比较函数为 `<`（如果是非内置类型需要 重载 `<` 运算符→4.4.4@p420）. 然而在某些特殊情况下，我们希望能自定义 `set` 内部的比较方式.

这时候可以通过传入自定义比较器来解决问题.

具体来说，我们需要定义一个类，并在这个类中 重载 `()` 运算符→4.4.4@p420.

例如，我们想要维护一个存储整数，且较大值靠前的 `set`，可以这样实现:

```
struct cmp {
  bool operator()(int a, int b) const { return a > b; }
};

set<int, cmp> s;
```

对于其他关联式容器，可以用类似的方式实现自定义比较，这里不再赘述.

### 4.3.2.5 无序关联式容器

#### 概述

自 C++11 标准起, 四种基于 哈希→10.5@p1995 实现的无序关联式容器正式纳入了 C++ 的标准模板库中, 分别是: `unordered_set`, `unordered_multiset`, `unordered_map`, `unordered_multimap`.

> **编译器不支持 C++11 的使用方法**
>
> 在 C++11 之前，无序关联式容器属于 C++ 的 TR1 扩展. 所以，如果编译器不支持 C++11，在使用时需要在头文件的名称中加入 `tr1/` 前缀，并且使用 `std::tr1` 命名空间. 如 `#include <unordered_map>` 需要改成 `#include <tr1/unordered_map>`; `std::unordered_map` 需要改为 `std::tr1::unordered_map`（如果使用 `using namespace std;`，则为 `tr1::unordered_map`）.

它们与相应的关联式容器在功能, 函数等方面有诸多共同点, 而最大的不同点则体现在普通的关联式容器一般采用红黑树实现, 内部元素按特定顺序进行排序; 而这几种无序关联式容器则采用哈希方式存储元素, 内部元素不以任何特定顺序进行排序, 所以访问无序关联式容器中的元素时，访问顺序也没有任何保证.

采用哈希存储的特点使得无序关联式容器 **在平均情况下** 大多数操作（包括查找，插入，删除）都能在常数时间复杂度内完成，相较于关联式容器与容器大小成对数的时间复杂度更加优秀.

> ⚠ **Warning**
>
> 在最坏情况下，对无序关联式容器进行插入、删除、查找等操作的时间复杂度会 **与容器大小成线性关系**！这一情况往往在容器内出现大量哈希冲突时产生.
>
> 同时，由于无序关联式容器的操作时通常存在较大的常数，其效率有时并不比普通的关联式容器好太多.
>
> 因此应谨慎使用无序关联式容器，尽量避免滥用（例如懒得离散化，直接将 `unordered_map<int, int>` 当作空间无限的普通数组使用）.

由于无序关联式容器与相应的关联式容器在用途和操作中有很多共同点，这里不再介绍无序关联式容器的各种操作，这些内容读者可以参考 关联式容器→4.3.2.4@p378.

### 制造哈希冲突

上文中提到了，在最坏情况下，对无序关联式容器进行一些操作的时间复杂度会与容器大小成线性关系.

在哈希函数确定的情况下，可以构造出数据使得容器内产生大量哈希冲突，导致复杂度达到上界.

在标准库实现里，每个元素的散列值是将值对一个质数取模得到的，更具体地说，是 这个列表[1] 中的质数（g++ 6 及以前版本的编译器，这个质数一般是 126271，g++ 7 及之后版本的编译器，这个质数一般是 107897）.

因此可以通过向容器中插入这些模数的倍数来达到制造大量哈希冲突的目的.

### 自定义哈希函数

使用自定义哈希函数可以有效避免构造数据产生的大量哈希冲突.

要想使用自定义哈希函数，需要定义一个结构体，并在结构体中重载 `()` 运算符，像这样：

```
struct my_hash {
  size_t operator()(int x) const { return x; }
};
```

当然，为了确保哈希函数不会被迅速破解（例如 Codeforces 中对使用无序关联式容器的提交进行 hack），可以试着在哈希函数中加入一些随机化函数（如时间）来增加破解的难度.

例如，在 这篇博客[2] 中给出了如下哈希函数：

```
struct my_hash {
  static uint64_t splitmix64(uint64_t x) {
    x += 0x9e3779b97f4a7c15;
    x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
    x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
    return x ^ (x >> 31);
  }

  size_t operator()(uint64_t x) const {
    static const uint64_t FIXED_RANDOM =
        chrono::steady_clock::now().time_since_epoch().count();
    return splitmix64(x + FIXED_RANDOM);
  }

  // 针对 std::pair<int, int> 作为主键类型的哈希函数
  size_t operator()(pair<uint64_t, uint64_t> x) const {
    static const uint64_t FIXED_RANDOM =
```

```
        chrono::steady_clock::now().time_since_epoch().count();
    return splitmix64(x.first + FIXED_RANDOM) ^
           (splitmix64(x.second + FIXED_RANDOM) >> 1);
  }
};
```

写完自定义的哈希函数后，就可以通过 `unordered_map<int, int, my_hash> my_map;` 或者 `unordered_map<pair<int, int>, int, my_hash> my_pair_map;` 的定义方式将自定义的哈希函数传入容器了.

### 外部参考资料

[1] 这个列表  [2] 这篇博客 

## 4.3.2.6 容器适配器

> **Authors:** Xeonacid, ksyx, Early0v0

### 栈

STL 栈→10.2@p1978(`std::stack`) 是一种后进先出 (Last In, First Out) 的容器适配器，仅支持查询或删除最后一个加入的元素（栈顶元素），不支持随机访问，且为了保证数据的严格有序性，不支持迭代器.

#### 头文件

```
#include <stack>
```

#### 定义

```
std::stack<TypeName> s;  // 使用默认底层容器 deque，数据类型为 TypeName
std::stack<TypeName, Container> s;  // 使用 Container 作为底层容器
std::stack<TypeName> s2(s1);        // 将 s1 复制一份用于构造 s2
```

#### 成员函数

**以下所有函数均为常数复杂度**

- `top()` 访问栈顶元素（如果栈为空，此处会出错）
- `push(x)` 向栈中插入元素 x
- `pop()` 删除栈顶元素
- `size()` 查询容器中的元素数量
- `empty()` 询问容器是否为空

#### 简单示例

```
std::stack<int> s1;
s1.push(2);
s1.push(1);
std::stack<int> s2(s1);
s1.pop();
std::cout << s1.size() << " " << s2.size() << std::endl;  // 1 2
std::cout << s1.top() << " " << s2.top() << std::endl;    // 2 1
s1.pop();
std::cout << s1.empty() << " " << s2.empty() << std::endl;  // 1 0
```

## 队列

STL 队列→10.3@p1980(`std::queue`) 是一种先进先出 (First In, First Out) 的容器适配器，仅支持查询或删除第一个加入的元素（队首元素），不支持随机访问，且为了保证数据的严格有序性，不支持迭代器.

### 头文件

```cpp
#include <queue>
```

### 定义

```cpp
std::queue<TypeName> q;  // 使用默认底层容器 deque，数据类型为 TypeName
std::queue<TypeName, Container> q;  // 使用 Container 作为底层容器

std::queue<TypeName> q2(q1);  // 将 q1 复制一份用于构造 q2
```

### 成员函数

**以下所有函数均为常数复杂度**

- `front()` 访问队首元素（如果队列为空，此处会出错）
- `push(x)` 向队列中插入元素 x
- `pop()` 删除队首元素
- `size()` 查询容器中的元素数量
- `empty()` 询问容器是否为空

### 简单示例

```cpp
std::queue<int> q1;
q1.push(2);
q1.push(1);
std::queue<int> q2(q1);
q1.pop();
std::cout << q1.size() << " " << q2.size() << std::endl;    // 1 2
std::cout << q1.front() << " " << q2.front() << std::endl;  // 1 2
q1.pop();
std::cout << q1.empty() << " " << q2.empty() << std::endl;  // 1 0
```

## 优先队列

优先队列 `std::priority_queue` 是一种 堆→10.7.1@p2025，一般为 二叉堆→10.7.2@p2026.

### 头文件

```cpp
#include <queue>
```

### 定义

```cpp
std::priority_queue<TypeName> q;              // 数据类型为 TypeName
std::priority_queue<TypeName, Container> q;  // 使用 Container 作为底层容器
std::priority_queue<TypeName, Container, Compare> q;
// 使用 Container 作为底层容器，使用 Compare 作为比较类型

// 默认使用底层容器 vector
// 比较类型 less<TypeName>（此时为它的 top() 返回为最大值）
// 若希望 top() 返回最小值，可令比较类型为 greater<TypeName>
// 注意：不可跳过 Container 直接传入 Compare

```

```cpp
// 从 C++11 开始，如果使用 lambda 函数自定义 Compare
// 则需要将其作为构造函数的参数代入，如：
auto cmp = [](const std::pair<int, int> &l, const std::pair<int, int> &r) {
  return l.second < r.second;
};
std::priority_queue<std::pair<int, int>, std::vector<std::pair<int, int>>,
                    decltype(cmp)>
    pq(cmp);
```

**成员函数**

**以下所有函数均为常数复杂度**

- `top()` 访问堆顶元素（此时优先队列不能为空）
- `empty()` 询问容器是否为空
- `size()` 查询容器中的元素数量

**以下所有函数均为对数复杂度**

- `push(x)` 插入元素，并对底层容器排序
- `pop()` 删除堆顶元素（此时优先队列不能为空）

**简单示例**

```cpp
std::priority_queue<int> q1;
std::priority_queue<int, std::vector<int>> q2;
// C++11 后空格可省略
std::priority_queue<int, std::deque<int>, std::greater<int>> q3;
// q3 为小根堆
for (int i = 1; i <= 5; i++) q1.push(i);
// q1 中元素 :  [1, 2, 3, 4, 5]
std::cout << q1.top() << std::endl;
// 输出结果 : 5
q1.pop();
// 堆中元素 : [1, 2, 3, 4]
std::cout << q1.size() << std::endl;
// 输出结果 : 4
for (int i = 1; i <= 5; i++) q3.push(i);
// q3 中元素 :  [1, 2, 3, 4, 5]
std::cout << q3.top() << std::endl;
// 输出结果 : 1
```

### 4.3.3 STL 算法

STL 提供了大约 100 个实现算法的模版函数，基本都包含在 `<algorithm>` 之中，还有一部分包含在 `<numeric>` 和 `<functional>`. 完备的函数列表请 参见参考手册[1]，排序相关的可以参考 排序内容的对应页面→5.7.14@p613.

- `find`: 顺序查找. `find(v.begin(), v.end(), value)`，其中 `value` 为需要查找的值.
- `reverse`: 翻转数组、字符串. `reverse(v.begin(), v.end())` 或 `reverse(a + begin, a + end)`.
- `unique`: 去除容器中相邻的重复元素. `unique(ForwardIterator first, ForwardIterator last)`，返回值为指向 **去重后** 容器结尾的迭代器，原容器大小不变. 与 `sort` 结合使用可以实现完整容器去重.
- `random_shuffle`: 随机地打乱数组. `random_shuffle(v.begin(), v.end())` 或 `random_shuffle(v + begin, v + end)`.

> ⚠ **`random_shuffle` 函数在最新 C++ 标准中已被移除**
>
> `random_shuffle` 自 C++14 起被弃用，C++17 起被移除.
>
> 在 C++11 以及更新的标准中，您可以使用 `shuffle` 函数代替原来的 `random_shuffle`. 使用方法为 `shuffle(v.begin(), v.end(), rng)`（最后一个参数传入的是使用的随机数生成器，一般情况使用以真随机数生成器 `random_device`[2] 播种的梅森旋转伪随机数生成器 `mt19937`[3]）.
>
> ```
> // #include <random>
> std::mt19937 rng(std::random_device{}());
> std::shuffle(v.begin(), v.end(), rng);
> ```

- `sort`：排序. `sort(v.begin(), v.end(), cmp)` 或 `sort(a + begin, a + end, cmp)`，其中 `end` 是排序的数组最后一个元素的后一位，`cmp` 为自定义的比较函数.
- `stable_sort`：稳定排序，用法同 `sort()`.
- `nth_element`：按指定范围进行分类，即找出序列中第 $n$ 大的元素，使其左边均为小于它的数，右边均为大于它的数. `nth_element(v.begin(), v.begin() + n, v.end(), cmp)` 或 `nth_element(a + begin, a + begin + n, a + end, cmp)`.
- `binary_search`：二分查找. `binary_search(v.begin(), v.end(), value)`，其中 `value` 为需要查找的值.
- `merge`：将两个（已排序的）序列 **有序合并** 到第三个序列的 **插入迭代器** 上. `merge(v1.begin(), v1.end(), v2.begin(), v2.end() ,back_inserter(v3))`.
- `inplace_merge`：将两个（已按小于运算符排序的）：`[first,middle)`, `[middle,last)` 范围 **原地合并为一个有序序列**. `inplace_merge(v.begin(), v.begin() + middle, v.end())`.
- `lower_bound`：在一个有序序列中进行二分查找，返回指向第一个 **大于等于** $x$ 的元素的位置的迭代器. 如果不存在这样的元素，则返回尾迭代器. `lower_bound(v.begin(),v.end(),x)`.
- `upper_bound`：在一个有序序列中进行二分查找，返回指向第一个 **大于** $x$ 的元素的位置的迭代器. 如果不存在这样的元素，则返回尾迭代器. `upper_bound(v.begin(),v.end(),x)`.

> ⚠ **`lower_bound` 和 `upper_bound` 的时间复杂度**
>
> 在一般的数组里，这两个函数的时间复杂度均为 $O(\log n)$，但在 `set` 等关联式容器中，直接调用 `lower_bound(s.begin(),s.end(),val)` 的时间复杂度是 $O(n)$ 的.
>
> `set` 等关联式容器中已经封装了 `lower_bound` 等函数（像 `s.lower_bound(val)` 这样），这样调用的时间复杂度是 $O(\log n)$ 的.

- `next_permutation`：将当前排列更改为 **全排列中的下一个排列**. 如果当前排列已经是 **全排列中的最后一个排列**（元素完全从大到小排列），函数返回 `false` 并将排列更改为 **全排列中的第一个排列**（元素完全从小到大排列）；否则，函数返回 `true`. `next_permutation(v.begin(), v.end())` 或 `next_permutation(v + begin, v + end)`.
- `prev_permutation`：将当前排列更改为 **全排列中的上一个排列**. 用法同 `next_permutation`.
- `partial_sum`：求前缀和. 设源容器为 $x$，目标容器为 $y$，则令 $y[i] = x[0] + x[1] + ... + x[i]$. `partial_sum(src.begin(), src.end(), back_inserter(dst))`.

### 使用样例

- 使用 `next_permutation` 生成 1 到 9 的全排列. 例题：Luogu P1706 全排列问题[4]

**实现**

```
int N = 9, a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
do {
  for (int i = 0; i < N; i++) cout << a[i] << " ";
  cout << endl;
} while (next_permutation(a, a + N));
```

- 使用 `lower_bound` 与 `upper_bound` 查找有序数组 $a$ 中小于 $x$，等于 $x$，大于 $x$ 元素的分界线.

**实现**

```
int N = 10, a[] = {1, 1, 2, 4, 5, 5, 7, 7, 9, 9}, x = 5;
int i = lower_bound(a, a + N, x) - a, j = upper_bound(a, a + N, x) - a;
// a[0] ~ a[i - 1] 为小于x的元素， a[i] ~ a[j - 1] 为等于x的元素，
// a[j] ~ a[N - 1] 为大于x的元素
cout << i << " " << j << endl;
```

- 使用 `partial_sum` 求解 $src$ 中元素的前缀和，并存储于 $dst$ 中.

**实现**

```
vector<int> src = {1, 2, 3, 4, 5}, dst;
// 求解src中元素的前缀和，dst[i] = src[0] + ... + src[i]
// back_inserter 函数作用在 dst 容器上，提供一个迭代器
partial_sum(src.begin(), src.end(), back_inserter(dst));
for (unsigned int i = 0; i < dst.size(); i++) cout << dst[i] << " ";
```

- 使用 `lower_bound` 查找有序数组 $a$ 中最接近 $x$ 的元素. 例题：UVa10487 Closest Sums[5]

**实现**

```
int N = 10, a[] = {1, 1, 2, 4, 5, 5, 8, 8, 9, 9}, x = 6;
// lower_bound将返回a中第一个大于等于x的元素的地址，计算出的i为其下标
int i = lower_bound(a, a + N, x) - a;
// 在以下两种情况下，a[i] (a中第一个大于等于x的元素) 即为答案：
// 1. a中最小的元素都大于等于x;
// 2. a中存在大于等于x的元素，且第一个大于等于x的元素 (a[i])
// 相比于第一个小于x的元素 (a[i - 1]) 更接近x;
// 否则，a[i - 1] (a中第一个小于x的元素) 即为答案
if (i == 0 || (i < N && a[i] - x < x - a[i - 1]))
  cout << a[i];
else
  cout << a[i - 1];
```

- 使用 `sort` 与 `unique` 查找数组 $a$ 中 **第 $k$ 小的值**（注意：重复出现的值仅算一次，因此本题不是求解第 $k$ 小的元素）. 例题：Luogu P1138 第 k 小整数[6]

**实现**

```
```

```cpp
int N = 10, a[] = {1, 3, 3, 7, 2, 5, 1, 2, 4, 6}, k = 3;
sort(a, a + N);
// unique将返回去重之后数组最后一个元素之后的地址，计算出的cnt为去重后数组的长度
int cnt = unique(a, a + N) - a;
cout << a[k - 1];
```

### 外部参考资料

[1] 参见参考手册 

[2] `random_device` 

[3] `mt19937` 

[4] Luogu P1706 全排列问题 

[5] UVa10487 Closest Sums 

[6] Luogu P1138 第 k 小整数 

## 4.3.4 bitset

**Authors:** i-Yirannn, Xeonacid, ouuan

### 介绍

`std::bitset` 是标准库中的一个存储 `0/1` 的大小不可变容器．严格来讲，它并不属于 STL.

**bitset 与 STL**

> The C++ standard library provides some special container classes, the so-called container adapters (stack, queue, priority queue). In addition, a few classes provide a container-like interface (for example, strings, bitsets, and valarrays). All these classes are covered separately.1 Container adapters and bitsets are covered in Chapter 12.
>
> The C++ standard library provides not only the containers for the STL framework but also some containers that fit some special needs and provide simple, almost self-explanatory, interfaces. You can group these containers into either the so-called container adapters, which adapt standard STL containers to fit special needs, or a bitset, which is a containers for bits or Boolean values. There are three standard container adapters: stacks, queues, and priority queues. In priority queues, the elements are sorted automatically according to a sorting criterion. Thus, the "next" element of a priority queue is the element with the "highest" value. A bitset is a bitfield with an arbitrary but fixed number of bits. Note that the C++ standard library also provides a special container with a variable size for Boolean values: vector.

——摘自《The C++ Standard Library 2nd Edition》

由此看来，`bitset` 并不属于 STL，而是一种标准库中的 "Special Container"．事实上，它作为一种容器，也并不满足 STL 容器的要求．说它是适配器，它也并不依赖于其它 STL 容器作为底层实现．

由于内存地址是按字节即 `byte` 寻址，而非比特 `bit`，一个 `bool` 类型的变量，虽然只能表示 `0/1`, 但是也占了 1 byte 的内存．

`bitset` 就是通过固定的优化，使得一个字节的八个比特能分别储存 8 位的 `0/1`.

对于一个 4 字节的 `int` 变量，在只存 `0/1` 的意义下，`bitset` 占用空间只是其 $\frac{1}{32}$，计算一些信息时，所需时间也是其 $\frac{1}{32}$.

在某些情况下通过 `bitset` 可以优化程序的运行效率．至于其优化的是复杂度还是常数，要看计算复杂度的角度．一般 `bitset` 的复杂度有以下几种记法：（设原复杂度为 $O(n)$）

1. $O(n)$，这种记法认为 `bitset` 完全没有优化复杂度.
2. $O(\frac{n}{32})$，这种记法不太严谨（复杂度中不应出现常数），但体现了 `bitset` 能将所需时间优化至 $\frac{1}{32}$.

3. $O(\frac{n}{w})$，其中 $w = 32$（计算机的位数），这种记法较为普遍接受.
4. $O(\frac{n}{\log w})$，其中 $w$ 为计算机一个整型变量的大小.

另外，`vector` 的一个特化 `vector<bool>` 的储存方式同 `bitset` 一样，区别在于其支持动态开空间，`bitset` 则和我们一般的静态数组一样，是在编译时就开好了的. 然而，`bitset` 有一些好用的库函数，不仅方便，而且有时可以实现 SIMD 进而减小常数. 另外，`vector<bool>` 的部分表现和 `vector` 不一致（如对 `std::vector<bool> vec` 来说，`&vec[0] + i` 不等于 `&vec[i]`）. 因此，一般不使用 `vector<bool>`.

## 使用

参见 std::bitset - cppreference.com[1].

### 头文件

```
#include <bitset>
```

### 指定大小

```
std::bitset<1000> bs;  // a bitset with 1000 bits
```

### 构造函数

- `bitset()`: 每一位都是 `false`.
- `bitset(unsigned long val)`: 设为 `val` 的二进制形式.
- `bitset(const string& str)`: 设为 01 串 `str`.

### 运算符

- `operator []`: 访问其特定的一位.
- `operator ==/operator !=`: 比较两个 `bitset` 内容是否完全一样.
- `operator &/operator &=/operator |/operator |=/operator ^/operator ^=/operator ~`: 进行按位与/或/异或/取反操作.

  注意：**`bitset` 只能与 `bitset` 进行位运算**，若要和整型进行位运算，要先将整型转换为 `bitset`.
- `operator <</operator >>/operator <<=/operator >>=`: 进行二进制左移/右移.

此外，`bitset` 还提供了 C++ 流式 IO 的支持，这意味着你可以通过 `cin/cout` 进行输入输出.

### 成员函数

- `count()`: 返回 `true` 的数量.
- `size()`: 返回 `bitset` 的大小.
- `test(pos)`: 它和 `vector` 中的 `at()` 的作用是一样的，和 `[]` 运算符的区别就是越界检查.
- `any()`: 若存在某一位是 `true` 则返回 `true`，否则返回 `false`.
- `none()`: 若所有位都是 `false` 则返回 `true`，否则返回 `false`.
- `all()`: 若所有位都是 `true` 则返回 `true`，否则返回 `false`.
- 1. `set()`: 将整个 `bitset` 设置成 `true`.
  2. `set(pos, val = true)`: 将某一位设置成 `true/false`.
- 1. `reset()`: 将整个 `bitset` 设置成 `false`.
  2. `reset(pos)`: 将某一位设置成 `false`. 相当于 `set(pos, false)`.
- 1. `flip()`: 翻转每一位. ($0 \leftrightarrow 1$，相当于异或一个全是 1 的 `bitset`)
  2. `flip(pos)`: 翻转某一位.
- `to_string()`: 返回转换成的字符串表达.
- `to_ulong()`: 返回转换成的 `unsigned long` 表达（`long` 在 NT 及 32 位 POSIX 系统下与 `int` 一样，在 64 位 POSIX 下与 `long long` 一样）.

- `to_ullong()`：（C++11 起）返回转换成的 `unsigned long long` 表达.

另外，libstdc++ 中有一些较为实用的内部成员函数[1]：

- `_Find_first()`：返回 `bitset` 第一个 `true` 的下标，若没有 `true` 则返回 `bitset` 的大小.
- `_Find_next(pos)`：返回 `pos` 后面（下标严格大于 `pos` 的位置）第一个 `true` 的下标，若 `pos` 后面没有 `true` 则返回 `bitset` 的大小.

## 应用

### 「LibreOJ β Round #2」贪心只能过样例[3]

这题可以用 dp 做，转移方程很简单：

$f(i,j)$ 表示前 $i$ 个数的平方和能否为 $j$，那么 $f(i,j)=\bigvee_{k=a}^{b} f(i-1,j-k^2)$（或起来）.

但如果直接做的话是 $O(n^5)$ 的，（看起来）过不了.

发现可以用 `bitset` 优化，左移再或起来就好了：

**提交记录：std::bitset[4]**

```cpp
#include <bitset>
#include <cstdio>
#include <iostream>

using namespace std;

constexpr int N = 101;

int n, a[N], b[N];
bitset<N * N * N> f[N];

int main() {
  int i, j;

  cin >> n;

  for (i = 1; i <= n; ++i) cin >> a[i] >> b[i];

  f[0][0] = 1;

  for (i = 1; i <= n; ++i) {
    for (j = a[i]; j <= b[i]; ++j) {
      f[i] |= (f[i - 1] << (j * j));
    }
  }

  cout << f[n].count();

  return 0;
}
```

[1] libstdc++: SGI STL extensions[2]

由于 libstdc++ 的实现为压 `__CHAR_BIT__ * sizeof(unsigned long)` 位的[2], 在一些平台中其为 32. 所以, 可以手写 `bitset`（只需要支持左移后或起来这一种操作）压 64 位（`__CHAR_BIT__ * sizeof(unsigned long long)`）来进一步优化：

**提交记录：手写 bitset**[6]

```cpp
#include <cstdio>
#include <iostream>

using namespace std;

constexpr int N = 101;
constexpr int W = 64;

struct Bitset {
  unsigned long long a[N * N * N >> 6];

  void shiftor(const Bitset &y, int p, int l, int r) {
    int t = p - p / W * W;
    int tt = (t == 0 ? 0 : W - t);
    int to = (r + p) / W;
    int qaq = (p + W - 1) / W;

    for (int i = (l + p) / W; i <= to; ++i) {
      if (i - qaq >= 0) a[i] |= y.a[i - qaq] >> tt;

      a[i] |= ((y.a[i - qaq + 1] & ((1ull << tt) - 1)) << t);
    }
  }
} f[N];

int main() {
  int n, a, b, l = 0, r = 0, ans = 0;

  scanf("%d", &n);

  f[0].a[0] = 1;

  for (int i = 1; i <= n; ++i) {
    scanf("%d%d", &a, &b);

    for (int j = a; j <= b; ++j) f[i].shiftor(f[i - 1], j * j, l, r);

    l += a * a;
    r += b * b;
  }

  for (int i = l / W; i <= r / W; ++i)
    ans += __builtin_popcount(f[n].a[i] & 0xffffffffu) +
           __builtin_popcount(f[n].a[i] >> 32);

  printf("%d", ans);
```

[2]libstdc++: std::bitset<_Nb> Class Template Reference[5]

```
47
48   return 0;
49 }
```

另外，加了几个剪枝的暴力也能过：

**提交记录：加了几个剪枝的暴力**[7]

```
1  #include <cstdio>
2  #include <iostream>
3
4  using namespace std;
5
6  constexpr int N = 101;
7  constexpr int W = 64;
8
9  bool f[N * N * N];
10
11 int main() {
12   int n, i, j, k, a, b, l = 0, r = 0, ans = 0;
13
14   scanf("%d", &n);
15
16   f[0] = true;
17
18   for (i = 1; i <= n; ++i) {
19     scanf("%d%d", &a, &b);
20     l += a * a;
21     r += b * b;
22
23     for (j = r; j >= l; --j) {
24       f[j] = false;
25
26       for (k = a; k <= b; ++k) {
27         if (j - k * k < l - a * a) break;
28
29         if (f[j - k * k]) {
30           f[j] = true;
31           break;
32         }
33       }
34     }
35   }
36
37   for (i = l; i <= r; ++i) ans += f[i];
38
39   printf("%d", ans);
40
41   return 0;
42 }
```

### CF1097F Alex and a TV Show[8]

**题意**

给你 $n$ 个可重集，四种操作：

1. 把某个可重集设为一个数.
2. 把某个可重集设为另外两个可重集加起来.
3. 把某个可重集设为从另外两个可重集中各选一个数的 gcd. 即：$A = \{\gcd(x, y) | \ x \in B, y \in C\}$.
4. 询问某个可重集中某个数的个数，**在模 2 意义下**.

可重集个数 $10^5$，操作个数 $10^6$，值域 7000.

**做法**

看到「在模 2 意义下」，可以想到用 `bitset` 维护每个可重集.

这样的话，操作 1 直接设，操作 2 就是异或（因为模 2），操作 4 就是直接查，但 .. 操作 3 怎么办?

我们可以尝试维护每个可重集的所有约数构成的可重集，这样的话，操作 3 就是直接按位与.

我们可以把值域内每个数的约数构成的 `bitset` 预处理出来，这样操作 1 就解决了. 操作 2 仍然是异或.

现在的问题是，如何通过一个可重集的约数构成的可重集得到该可重集中某个数的个数.

令原可重集为 $A$，其约数构成的可重集为 $A'$，我们要求 $A$ 中 $x$ 的个数，用 莫比乌斯反演→9.11.23@p1331 推一推：

$$\begin{aligned}
&\sum_{i \in A} [\frac{i}{x} = 1] \\
=&\sum_{i \in A} \sum_{d \mid \frac{i}{x}} \mu(d) \\
=&\sum_{d \in A', x \mid d} \mu(\frac{d}{x})
\end{aligned}$$

由于是模 2 意义下，$-1$ 和 1 是一样的，只用看 $\frac{d}{x}$ 有没有平方因子即可. 所以，可以对值域内每个数预处理出其倍数中除以它不含平方因子的位置构成的 `bitset`，求答案的时候先按位与再 `count()` 就好了.

这样的话，单次询问复杂度就是 $O(\frac{v}{w})$（$v = 7000, w = 32$）.

至于预处理的部分，$O(v\sqrt{v})$ 或者 $O(v^2)$ 预处理比较简单，log 预处理就如下面代码所示，复杂度为调和级数，所以是 $O(v \log v)$.

**参考代码**

```cpp
#include <bitset>
#include <cctype>
#include <cmath>
#include <cstdio>
#include <iostream>

using namespace std;

int read() {
  int out = 0;
  char c;
  while (!isdigit(c = getchar()));
  for (; isdigit(c); c = getchar()) out = out * 10 + c - '0';
  return out;
```

```
}

constexpr int N = 100005;
constexpr int M = 1000005;
constexpr int V = 7005;

bitset<V> pre[V], pre2[V], a[N], mu;
int n, m, tot;
char ans[M];

int main() {
  int i, j, x, y, z;

  n = read();
  m = read();

  mu.set();
  for (i = 2; i * i < V; ++i) {
    for (j = 1; i * i * j < V; ++j) {
      mu[i * i * j] = 0;
    }
  }
  for (i = 1; i < V; ++i) {
    for (j = 1; i * j < V; ++j) {
      pre[i * j][i] = 1;
      pre2[i][i * j] = mu[j];
    }
  }

  while (m--) {
    switch (read()) {
      case 1:
        x = read();
        y = read();
        a[x] = pre[y];
        break;
      case 2:
        x = read();
        y = read();
        z = read();
        a[x] = a[y] ^ a[z];
        break;
      case 3:
        x = read();
        y = read();
        z = read();
        a[x] = a[y] & a[z];
        break;
      case 4:
        x = read();
        y = read();
        ans[tot++] = ((a[x] & pre2[y]).count() & 1) + '0';
```

```
67          break;
68      }
69    }
70
71    printf("%s", ans);
72
73    return 0;
74  }
```

### 与埃氏筛结合

由于 `bitset` 快速的连续读写效率，使得它非常适合用于与 埃氏筛→9.11.6@p1199 结合打质数表.

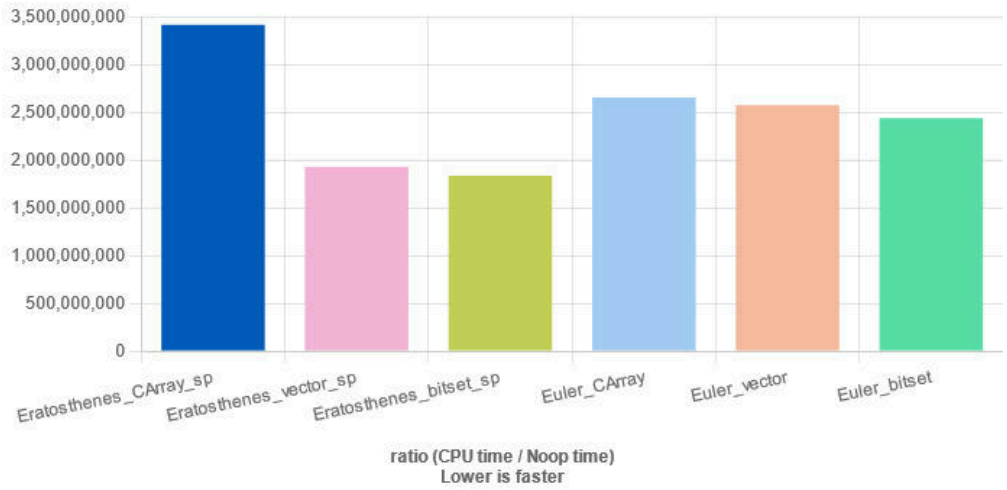
使用的方式也很简单，只需要将埃氏筛中的布尔数组替换成 `bitset` 即可.

**速度测试**

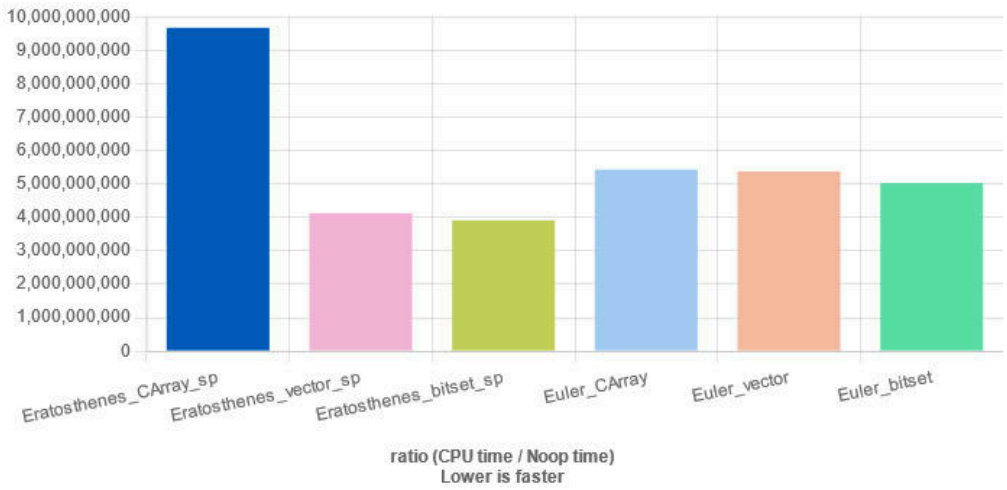
使用 Quick C++ Benchmarks[9] 进行测试，编译器采用 `GCC 13.2`，编译参数为 `-std=c++20 -O2`.

| 算法 | 函数名 |
| --- | --- |
| 埃氏筛 + C 风格布尔数组，不存储筛出来的素数 | `Eratosthenes_CArray` |
| 埃氏筛 +`vector<bool>`，不存储筛出来的素数 | `Eratosthenes_vector` |
| 埃氏筛 +`bitset`，不存储筛出来的素数 | `Eratosthenes_bitset` |
| 埃氏筛 + C 风格布尔数组，存储筛出来的素数 | `Eratosthenes_CArray_sp` |
| 埃氏筛 +`vector<bool>`，存储筛出来的素数 | `Eratosthenes_vector_sp` |
| 埃氏筛 +`bitset`，存储筛出来的素数 | `Eratosthenes_bitset_sp` |
| 欧拉筛 + C 风格布尔数组 | `Euler_CArray` |
| 欧拉筛 +`vector<bool>` | `Euler_vector` |
| 欧拉筛 +`bitset` | `Euler_bitset` |

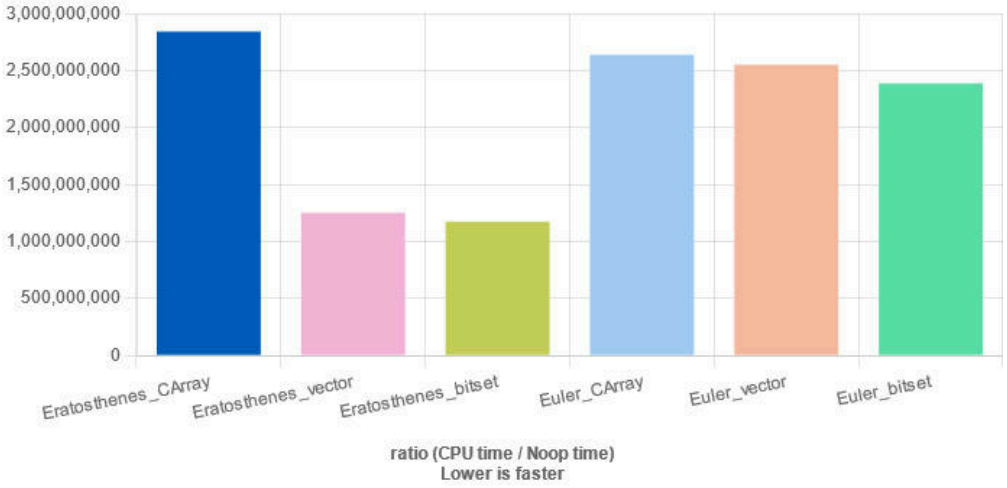
- 当埃氏筛 **存储** 筛出来的素数时:
  - $N = 5 \times 10^7 + 1$ 时的 测试结果[10]:



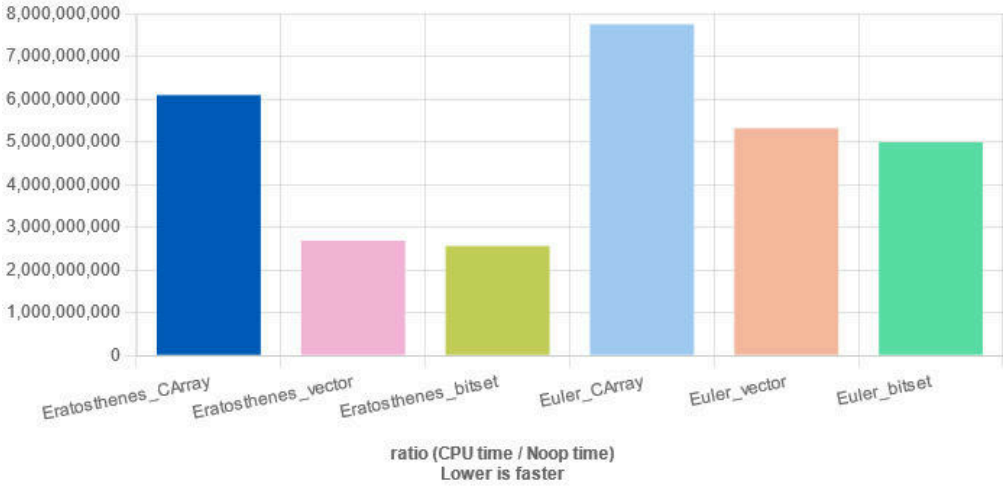

  - $N = 10^8 + 1$ 时的 测试结果[11]:




- 当埃氏筛 **不存储** 筛出来的素数时：
  - $N = 5 \times 10^7 + 1$ 时的 测试结果[12]：




  - $N = 10^8 + 1$ 时的 测试结果[13]：




从测试结果中可知：

1. 时间复杂度 $O(n \log \log n)$ 的埃氏筛在使用 `bitset` 或 `vector<bool>` 优化后，性能甚至超过时间复杂度 $O(n)$ 的欧拉筛；

2. 欧拉筛使用 `bitset` 或 `vector<bool>` 后的优化效果在大多数情况下均不明显；
3. `bitset` 的优化效果略强于 `vector<bool>`.

**参考代码**

需安装 google/benchmark[14].

```cpp
#include <benchmark/benchmark.h>
#include <bits/stdc++.h>
using namespace std;
using u32 = uint32_t;
using u64 = uint64_t;

#define ERATOSTHENES_STORAGE_PRIME
#define ENABLE_EULER
constexpr u32 N = 5e7 + 1;

#ifndef ERATOSTHENES_STORAGE_PRIME

void Eratosthenes_CArray(benchmark::State &state) {
  static bool is_prime[N];
  for (auto _ : state) {
    fill(is_prime, is_prime + N, true);
    is_prime[0] = is_prime[1] = false;
    for (u32 i = 2; (u64)i * i < N; ++i)
      if (is_prime[i])
        for (u32 j = i * i; j < N; j += i) is_prime[j] = false;
    benchmark::DoNotOptimize(0);
  }
}

BENCHMARK(Eratosthenes_CArray);

void Eratosthenes_vector(benchmark::State &state) {
  static vector<bool> is_prime(N);
  for (auto _ : state) {
    fill(is_prime.begin(), is_prime.end(), true);
    is_prime[0] = is_prime[1] = false;
    for (u32 i = 2; (u64)i * i < N; ++i)
      if (is_prime[i])
        for (u32 j = i * i; j < N; j += i) is_prime[j] = false;
    benchmark::DoNotOptimize(0);
  }
}

BENCHMARK(Eratosthenes_vector);

void Eratosthenes_bitset(benchmark::State &state) {
  static bitset<N> is_prime;
  for (auto _ : state) {
    is_prime.set();
    is_prime.reset(0);
    is_prime.reset(1);
```

```
    for (u32 i = 2; (u64)i * i < N; ++i)
      if (is_prime[i])
        for (u32 j = i * i; j < N; j += i) is_prime.reset(j);
    benchmark::DoNotOptimize(0);
  }
}

BENCHMARK(Eratosthenes_bitset);

#else

void Eratosthenes_CArray_sp(benchmark::State &state) {
  static bool is_prime[N];
  for (auto _ : state) {
    vector<u32> prime;
    fill(is_prime, is_prime + N, true);
    is_prime[0] = is_prime[1] = false;
    for (u32 i = 2; (u64)i * i < N; ++i)
      if (is_prime[i])
        for (u32 j = i * i; j < N; j += i) is_prime[j] = false;
    for (u32 i = 2; i < N; ++i)
      if (is_prime[i]) prime.push_back(i);
    benchmark::DoNotOptimize(prime);
  }
}

BENCHMARK(Eratosthenes_CArray_sp);

void Eratosthenes_vector_sp(benchmark::State &state) {
  static vector<bool> is_prime(N);
  for (auto _ : state) {
    vector<u32> prime;
    fill(is_prime.begin(), is_prime.end(), true);
    is_prime[0] = is_prime[1] = false;
    for (u32 i = 2; (u64)i * i < N; ++i)
      if (is_prime[i])
        for (u32 j = i * i; j < N; j += i) is_prime[j] = false;
    for (u32 i = 2; i < N; ++i)
      if (is_prime[i]) prime.push_back(i);
    benchmark::DoNotOptimize(prime);
  }
}

BENCHMARK(Eratosthenes_vector_sp);

void Eratosthenes_bitset_sp(benchmark::State &state) {
  static bitset<N> is_prime;
  for (auto _ : state) {
    vector<u32> prime;
    is_prime.set();
    is_prime.reset(0);
    is_prime.reset(1);
```

```
 99     for (u32 i = 2; (u64)i * i < N; ++i)
100       if (is_prime[i])
101         for (u32 j = i * i; j < N; j += i) is_prime.reset(j);
102     for (u32 i = 2; i < N; ++i)
103       if (is_prime[i]) prime.push_back(i);
104     benchmark::DoNotOptimize(prime);
105   }
106 }
107
108 BENCHMARK(Eratosthenes_bitset_sp);
109
110 #endif
111
112 #ifdef ENABLE_EULER
113
114 void Euler_CArray(benchmark::State &state) {
115   static bool not_prime[N];
116   for (auto _ : state) {
117     vector<u32> prime;
118     fill(not_prime, not_prime + N, false);
119     not_prime[0] = not_prime[1] = true;
120     for (u32 i = 2; i < N; ++i) {
121       if (!not_prime[i]) prime.push_back(i);
122       for (u32 pri_j : prime) {
123         if (i * pri_j >= N) break;
124         not_prime[i * pri_j] = true;
125         if (i % pri_j == 0) break;
126       }
127     }
128     benchmark::DoNotOptimize(prime);
129   }
130 }
131
132 BENCHMARK(Euler_CArray);
133
134 void Euler_vector(benchmark::State &state) {
135   static vector<bool> not_prime(N);
136   for (auto _ : state) {
137     vector<u32> prime;
138     fill(not_prime.begin(), not_prime.end(), false);
139     not_prime[0] = not_prime[1] = true;
140     for (u32 i = 2; i < N; ++i) {
141       if (!not_prime[i]) prime.push_back(i);
142       for (u32 pri_j : prime) {
143         if (i * pri_j >= N) break;
144         not_prime[i * pri_j] = true;
145         if (i % pri_j == 0) break;
146       }
147     }
148     benchmark::DoNotOptimize(prime);
149   }
150 }
```

```
151
152 BENCHMARK(Euler_vector);
153
154 void Euler_bitset(benchmark::State &state) {
155   static bitset<N> not_prime;
156   for (auto _ : state) {
157     vector<u32> prime;
158     not_prime.reset();
159     not_prime.set(0);
160     not_prime.set(1);
161     for (u32 i = 2; i < N; ++i) {
162       if (!not_prime[i]) prime.push_back(i);
163       for (u32 pri_j : prime) {
164         if (i * pri_j >= N) break;
165         not_prime.set(i * pri_j);
166         if (i % pri_j == 0) break;
167       }
168     }
169     benchmark::DoNotOptimize(prime);
170   }
171 }
172
173 BENCHMARK(Euler_bitset);
174
175 #endif
176
177 static void Noop(benchmark::State &state) {
178   for (auto _ : state) benchmark::DoNotOptimize(0);
179 }
180
181 BENCHMARK(Noop);
182 BENCHMARK_MAIN();
```

### 与树分块结合

`bitset` 与树分块结合可以解决一类求树上多条路径信息并的问题，详见 数据结构/树分块→10.8.4@p2057.

### 与莫队结合

详见 杂项/莫队配合 bitset→13.4.4.8@p3243.

### 计算高维偏序

详见 FHR 课件[15].

## 参考资料与注释


[1] std::bitset - cppreference.com

[2] libstdc++: SGI STL extensions

[3] 「LibreOJ β Round #2」贪心只能过样例

[4] std::bitset

[5] libstdc++: std::bitset<_Nb> Class Template Reference

[6] 手写 bitset

[7] 加了几个剪枝的暴力

[8] CF1097F Alex and a TV Show

[9] Quick C++ Benchmarks

[10] 测试结果  [11] 测试结果  [12] 测试结果 

[13] 测试结果  [14] google/benchmark  [15] FHR 课件 

## 4.3.5 string

Authors: johnvp22, Ir1d

### `string` 是什么

`std::string` 是在标准库 `<string>`（注意不是 C 语言中的 `<string.h>` 库）中提供的一个类，本质上是 `std::basic_string<char>` 的别称.

### 为什么要使用 `string`

在 C 语言中，提供了字符串的操作，但只能通过字符数组的方式来实现字符串．而 `string` 则是一个简单的类，使用简单，在 OI 竞赛中被广泛使用．并且相较于其他 STL 容器，`string` 的常数可以算是非常优秀的，基本与字符数组不相上下.

#### `string` 可以动态分配空间

和许多 STL 容器相同，`string` 能动态分配空间，这使得我们可以直接使用 `std::cin` 来输入，但其速度则同样较慢．这一点也同样让我们不必为内存而烦恼.

#### `string` 重载了加法运算符和比较运算符

`string` 的加法运算符可以直接拼接两个字符串或一个字符串和一个字符．和 `std::vector` 类似，`string` 重载了比较运算符，同样是按字典序比较的，所以我们可以直接调用 `std::sort` 对若干字符串进行排序.

### 使用方法

下面介绍 `string` 的基本操作，具体可看 C++ 文档[1].

#### 声明

```
std::string s;
```

#### 转 char 数组

在 C 语言里，也有很多字符串的函数，但是它们的参数都是 char 指针类型的，为了方便使用，`string` 有两个成员函数能够将自己转换为 char 指针——`data()`/`c_str()`（在 C++11 之前，`c_str()` 保证末尾有空字符，而 `data()` 不保证；C++11 后两者行为一致[3]），如：

```
printf("%s", s);          // 不能保证编译通过，行为未定义
printf("%s", s.data());   // C++11 前行为未定义，C++11 后正确输出
printf("%s", s.c_str());  // 一定能够正确输出
```

#### 获取长度

很多函数都可以返回 string 的长度：

```
printf("s 的长度为 %zu", s.size());
printf("s 的长度为 %zu", s.length());
printf("s 的长度为 %zu", strlen(s.c_str()));
```

[3] C++ 标准草案 [basic.string][2]

**这些函数的复杂度**

`strlen()` 的复杂度一定是与字符串长度线性相关的.

`size()` 和 `length()` 的复杂度在 C++98 中没有指定，在 C++11 中被指定为常数复杂度. 但在常见的编译器上，即便是 C++98，这两个函数的复杂度也是常数.

**Warning**

这三个函数（以及下面将要提到的 `find` 函数）的返回值类型都是 `size_t`（`unsigned long`）. 因此，这些返回值不支持直接与负数比较或运算，建议在需要时进行强制转换.

### 寻找某字符（串）第一次出现的位置

`find(str,pos)` 函数可以用来查找字符串中一个字符/字符串在 `pos`（含）之后第一次出现的位置（若不传参给 `pos` 则默认为 `0`）. 如果没有出现，则返回 `string::npos`（被定义为 `-1`，但类型仍为 `size_t`/`unsigned long`）.

示例:

```
string s = "OI Wiki", t = "OI", u = "i";
int pos = 5;
printf("字符 I 在 s 的 %lu 位置第一次出现\n", s.find('I'));
printf("字符 a 在 s 的 %lu 位置第一次出现\n", s.find('a'));
printf("字符 a 在 s 的 %d 位置第一次出现\n", s.find('a'));
printf("字符串 t 在 s 的 %lu 位置第一次出现\n", s.find(t));
printf("在 s 中自 pos 位置起字符串 u 第一次出现在 %lu 位置", s.find(u, pos));
```

输出:

```
字符 I 在 s 的 1 位置第一次出现
字符 a 在 s 的 18446744073709551615 位置第一次出现 // 即为 size_t(-1)，具体数值与平台有关.
字符 a 在 s 的 -1 位置第一次出现 // 强制转换为 int 类型则正常输出 -1
字符串 t 在 s 的 0 位置第一次出现
在 s 中自 pos 位置起字符串 u 第一次出现在 6 位置
```

### 截取子串

`substr(pos, len)` 函数返回从 `pos` 位置开始截取最多 `len` 个字符组成的字符串（如果从 `pos` 开始的后缀长度不足 `len` 则截取这个后缀）.

示例:

```
string s = "OI Wiki", t = "OI";
printf("从字符串 s 的第四位开始的最多三个字符构成的子串是 %s\n",
       s.substr(3, 3).c_str());
printf("从字符串 t 的第二位开始的最多三个字符构成的子串是 %s",
       t.substr(1, 3).c_str());
```

输出:

```
从字符串 s 的第四位开始的最多三个字符构成的子串是 Wik
从字符串 t 的第二位开始的最多三个字符构成的子串是 I
```

### 插入/删除字符（串）

`insert(index,count,ch)` 和 `insert(index,str)` 是比较常见的插入函数．它们分别表示在 `index` 处连续插入 `count` 次字符串 `ch` 和插入字符串 `str`．

`erase(index,count)` 函数将字符串 `index` 位置开始（含）的 `count` 个字符删除（若不传参给 `count` 则表示删去 `index` 位置及以后的所有字符）．

示例：

```
string s = "OI Wiki", t = " Wiki";
char u = '!';
s.erase(2);
printf("从字符串 s 的第三位开始删去所有字符后得到的字符串是 %s\n", s.c_str());
s.insert(2, t);
printf("在字符串 s 的第三位处插入字符串 t 后得到的字符串是 %s\n", s.c_str());
s.insert(7, 3, u);
printf("在字符串 s 的第八位处连续插入 3 次字符串 u 后得到的字符串是 %s",
       s.c_str());
```

输出：

```
从字符串 s 的第三位开始删去所有字符后得到的字符串是 OI
在字符串 s 的第三位处插入字符串 t 后得到的字符串是 OI Wiki
在字符串 s 的第八位处连续插入 3 次字符串 u 后得到的字符串是 OI Wiki!!!
```

### 替换字符（串）

`replace(pos,count,str)` 和 `replace(first,last,str)` 是比较常见的替换函数．它们分别表示将从 `pos` 位置开始 `count` 个字符的子串替换为 `str` 以及将以 `first` 开始（含）、`last` 结束（不含）的子串替换为 `str`，其中 `first` 和 `last` 均为迭代器．

示例：

```
string s = "OI Wiki";
s.replace(2, 5, "");
printf("将字符串 s 的第 3~7 位替换为空串后得到的字符串是 %s\n", s.c_str());
s.replace(s.begin(), s.begin() + 2, "NOI");
printf("将字符串 s 的前两位替换为 NOI 后得到的字符串是 %s", s.c_str());
```

输出：

```
将字符串 s 的第 3~7 位替换为空串后得到的字符串是 OI
将字符串 s 的前两位替换为 NOI 后得到的字符串是 NOI
```

## 参考资料与注释

[1] C++ 文档 [2] C++ 标准草案 [basic.string]

## 4.3.6 pair

**Authors:** sbofgayschool

`std::pair` 是标准库中定义的一个类模板．用于将两个变量关联在一起，组成一个「对」，而且两个变量的数据类型可以是不同的．

**类模板**

类模板（class template）本身不是一个类，而是可以根据 **不同数据类型** 产生 **不同类** 的「模板」.

在使用时，编译器会根据传入的数据类型产生对应的类，再创建对应实例.

模板属于 C++ 较为高级的语言特性，在信息学竞赛中几乎不会出现. 如果对此感兴趣，可以进一步阅读《C++ Primer》以学习更深层次的 C++ 知识.

通过灵活使用 `pair`，可以轻松应对 **需要将关联数据捆绑存储、处理** 的场景.

**Struct**

与自定义的 `struct` 相比，`pair` 不需要额外定义结构与重载运算符，因此使用起来更加简便.

然而，自定义 `struct` 的变量命名往往更加清晰（`pair` 只能使用 `first` 与 `second` 访问包含的两个变量）. 同时，如果需要将两个以上的变量进行关联，自定义 `struct` 会更加合适.

## 使用

### 初始化

可以在定义时直接完成 `pair` 的初始化.

```
pair<int, double> p0(1, 2.0);
```

也可以使用先定义，后赋值的方法完成 `pair` 的初始化.

```
pair<int, double> p1;
p1.first = 1;
p1.second = 2.0;
```

还可以使用 `std::make_pair` 函数. 该函数接受两个变量，并返回由这两个变量组成的 `pair`.

```
pair<int, double> p2 = make_pair(1, 2.0);
```

一种常用的方法是使用宏定义 `#define mp make_pair`，将有些冗长的 `make_pair` 化简为 `mp`.

在 C++11 以及之后的版本中，`make_pair` 可以配合 `auto` 使用，以避免显式声明数据类型.

```
auto p3 = make_pair(1, 2.0);
```

关于 `auto` 的在信息学竞赛中的使用，参见 迭代器→4.3.2.2@p371 部分的说明.

### 访问

通过成员函数 `first` 与 `second`，可以访问 `pair` 中包含的两个变量.

```
int i = p0.first;
double d = p0.second;
```

也可以对其进行修改.

```
p1.first++;
```

### 比较

`pair` 已经预先定义了所有的比较运算符，包括 `<`、`>`、`<=`、`>=`、`==`、`!=`. 当然，这需要组成 `pair` 的两个变量所属的数据类型定义了 `==` 和/或 `<` 运算符.

其中，`<`、`>`、`<=`、`>=` 四个运算符会先比较两个 `pair` 中的第一个变量，在第一个变量相等的情况下再比较第二个变量.

```cpp
if (p2 >= p3) {
  cout << "do something here" << endl;
}
```

由于 `pair` 定义了 STL 中常用的 `<` 与 `==`，使得其能够很好的与其他 STL 函数或数据结构配合．比如，`pair` 可以作为 `priority_queue` 的数据类型．

```cpp
priority_queue<pair<int, double>> q;
```

### 赋值与交换

可以将 `pair` 的值赋给另一个类型一致的 `pair`．

```cpp
p0 = p1;
```

也可以使用 `swap` 函数交换 `pair` 的值．

```cpp
swap(p0, p1);
p2.swap(p3);
```

## 应用举例

### 离散化

`pair` 可以轻松实现离散化．

我们可以创建一个 `pair` 数组，将原始数据的值作为每个 `pair` 第一个变量，将原始数据的位置作为第二个变量．在排序后，将原始数据值的排名（该值排序后所在的位置）赋给该值原本所在的位置即可．

```cpp
// a为原始数据
pair<int, int> a[MAXN];
// ai为离散化后的数据
int ai[MAXN];
for (int i = 0; i < n; i++) {
  // first为原始数据的值，second为原始数据的位置
  scanf("%d", &a[i].first);
  a[i].second = i;
}
// 排序
sort(a, a + n);
for (int i = 0; i < n; i++) {
  // 将该值的排名赋给该值原本所在的位置
  ai[a[i].second] = i;
}
```

### Dijkstra

如前所述，`pair` 可以作为 `priority_queue` 的数据类型．

那么，在 Dijkstra 算法的堆优化中，可以使用 `pair` 与 `priority_queue` 维护节点，将节点当前到起点的距离作为第一个变量，将节点编号作为第二个变量．

```cpp
priority_queue<pair<int, int>, std::vector<pair<int, int>>,
               std::greater<pair<int, int>>>
    q;
... while (!q.empty()) {
  // dis为入堆时节点到起点的距离，i为节点编号
  int dis = q.top().first, i = q.top().second;
```

```
  q.pop();
  ...
}
```

### pair 与 map

`map` 的是 C++ 中存储键值对的数据结构．很多情况下，`map` 中存储的键值对通过 `pair` 向外暴露．

```
map<int, double> m;
m.insert(make_pair(1, 2.0));
```

关于 `map` 更多的内容，请见 关联式容器→4.3.2.4@p378 与 无序关联式容器→4.3.2.5@p382 中相关部分．

# 4.4 C++ 进阶

## 4.4.1 类

**Authors:** Ir1d, cjsoft, Lans1ot, JasonkayZK

类（class）是结构体的拓展，不仅能够拥有成员元素，还拥有成员函数．

在面向对象编程（OOP）中，对象就是类的实例，也就是变量．

C++ 中 `struct` 关键字定义的也是类，上文中的 **结构体** 的定义来自 C．因为某些历史原因，C++ 保留并拓展了 `struct`．

### 定义类

类使用关键字 `class` 或者 `struct` 定义，下文以 `class` 举例．

```
class ClassName {
  ...
};

// Example:
class Object {
 public:
  int weight;
  int value;
} e[array_length];

const Object a;
Object b, B[array_length];
Object *c;
```

与使用 `struct` 大同小异．该例定义了一个名为 `Object` 的类．该类拥有两个成员元素，分别为 `weight,value`；并在 `}` 后使用该类型定义了一个数组 `e`．

定义类的指针形同 `struct`→4.2.6.2@p354．

### 访问说明符

不同于 `struct`→4.2.6.2@p354 中的举例，本例中出现了 `public`，这属于访问说明符．

- `public`：该访问说明符之后的各个成员都可以被公开访问，简单来说就是无论 **类内** 还是 **类外** 都可以访问．
- `protected`：该访问说明符之后的各个成员可以被 **类内**、派生类或者友元的成员访问，但类外 **不能访问**．

- `private`：该访问说明符之后的各个成员 **只能** 被 **类内** 成员或者友元的成员访问，**不能** 被从类外或者派生类中访问.

对于 `struct`，它的所有成员都是默认 `public`. 对于 `class`，它的所有成员都是默认 `private`.

**关于友元以及派生类的基本概念**

友元（`friend`）：使用 `friend` 关键字修饰某个函数或者类. 可以使得在 **被修饰者** 在不成为成员函数或者成员类的情况下，访问该类的私有（`private`）或者受保护（`protected`）成员. 简单来说就是只要带有这个类的 `friend` 标记，就可以访问私有或受保护的成员元素.

派生类（`derived class`）：C++ 允许使用一个类作为 **基类**，并通过基类 **派生** 出 **派生类**. 其中派生类（根据特定规则）继承基类中的成员变量和成员函数. 可以提高代码的复用率.

派生类似 "is" 的关系. 如猫（派生类）"is" 哺乳动物（基类）.

对于上面 `private` 和 `protected` 的区别，可以看做派生类可以访问基类的 `protected` 的元素（`public` 同），但不能访问 `private` 元素.

## 访问与修改成员元素的值

方法形同 `struct`→4.2.6.2@p354

- 对于变量，使用 `.` 符号.
- 对于指针，使用 `->` 符号.

## 成员函数

成员函数，顾名思义. 就是类中所包含的函数.

**常见成员函数举例**

```
vector.push_back();
set.insert();
queue.empty();
```

```
class Class_Name {
  ... type Function_Name(...) { ... }
};

// Example:
class Object {
 public:
  int weight;
  int value;

  void print() {
    cout << weight << endl;
    return;
  }

  void change_w(int);
};

void Object::change_w(int _weight) { weight = _weight; }

```

```
21 Object var;
```

该类有一个打印 `Object` 成员元素的函数，以及更改成员元素 `weight` 的函数.

和函数类似，对于成员函数，也可以先声明，在定义，如第十四行（声明处）以及十七行后（定义处）.

如果想要调用 `var` 的 `print` 成员函数，可以使用 `var.print()` 进行调用.

### 重载运算符

> **何为重载**
>
> C++ 允许编写者为名称相同的函数或者运算符指定不同的定义. 这称为 **重载**（overload）.
>
> 如果同名函数的参数种类、数量中的一者或多者两两不相同，则这些同名函数被看做是不同的.
>
> 需要注意的是: 如果两个同名函数的区别仅仅是返回值的类型不同则无法进行重载, 此时编译器会拒绝编译!
>
> 如果在调用时不会出现混淆（指调用某些同名函数时, 无法根据所填参数种类和数量唯一地判断出被调用函数. 常发生在具有默认参数的函数中）, 则编译器会根据调用时所填参数判断应调用函数.
>
> 而上述过程被称作重载解析.

重载运算符，可以部分程度上代替函数，简化代码.

下面给出重载运算符的例子.

```
1  class Vector {
2   public:
3    int x, y;
4
5    Vector() : x(0), y(0) {}
6
7    Vector(int _x, int _y) : x(_x), y(_y) {}
8
9    int operator*(const Vector& other) const { return x * other.x + y * other.y; }
10
11   Vector operator+(const Vector&) const;
12   Vector operator-(const Vector&) const;
13 };
14
15 Vector Vector::operator+(const Vector& other) const {
16   return Vector(x + other.x, y + other.y);
17 }
18
19 Vector Vector::operator-(const Vector& other) const {
20   return Vector(x - other.x, y - other.y);
21 }
22
23 // 关于4,5行表示为x,y赋值，具体实现参见后文.
```

该例定义了一个向量类，并重载了 `* + -` 运算符，并分别代表向量内积，向量加，向量减.

重载运算符的模板大致可分为下面几部分.

```
1 /*类定义内重载*/ 返回类型 operator符号(参数){...}
2
3 /*类定义内声明，在外部定义*/ 返回类型 类名称::operator符号(参数){...}
```

对于自定义的类，如果重载了某些运算符（一般来说只需要重载 `<` 这个比较运算符），便可以使用相应的 STL 容器或算法，如 `sort`→5.7.14@p613.

如要了解更多，可参见「参考资料」第四条.

**可以被重载的运算符**

```
+       -       *       /       %       ^       &
|       ~       !       =       <       >       +=
-=      *=      /=      %=      ^=      &=      |=
<<      >>      >>=     <<=     ==      !=      <=
>=      &&      ||      ++      --      ,       ->*
->      ()      []      new     new []  delete  delete []
```

### 在实例化变量时设定初始值

为完成这种操作，需要定义 **默认构造函数**(Default constructor).

```
class ClassName {
  ... ClassName(...)... { ... }
};

// Example:
class Object {
 public:
  int weight;
  int value;

  Object() {
    weight = 0;
    value = 0;
  }
};
```

该例定义了 `Object` 的默认构造函数，该函数能够在我们实例化 `Object` 类型变量时，将所有的成员元素初始化为 `0`.

若无显式的构造函数，则编译器认为该类有隐式的默认构造函数. 换言之，若无定义任何构造函数，则编译器会自动生成一个默认构造函数，并会根据成员元素的类型进行初始化（与定义 内置类型 变量相同）.

在这种情况下，成员元素都是未初始化的，访问未初始化的变量的结果是未定义的（也就是说并不知道会返回何值）.

如果需要自定义初始化的值，可以再定义（或重载）构造函数.

**关于定义（或重载）构造函数**

一般来说，默认构造函数是不带参数的，这区别于构造函数. 构造函数和默认构造函数的定义大同小异，只是参数数量上的不同.

构造函数可以被重载（当然首次被叫做定义）. 需要注意的是，如果已经定义了构造函数，那么编译器便不会再生成无参数的默认构造函数. 这会可能会使试图以默认方法构造变量的行为编译失败（指不填入初始化参数）.

使用 C++11 或以上时，可以使用 `{}` 进行变量的初始化.

### 关于 {}

使用 {} 进行初始化，会用到 std::initializer_list 这一个轻量代理对象进行初始化.

初始化步骤大概如下

1. 尝试寻找参数中有 std::initializer_list 的默认构造函数，如果有则调用（调用完后不再进行下面的查找，下同）.
2. 尝试将 {} 中的元素填入其他构造参数，如果能将参数按照顺序填满（默认参数也算在内），则调用该默认构造函数.
3. 若无 private 成员元素，则尝试在 **类外** 按照元素定义顺序或者下标顺序依次赋值.

上述过程只是完整过程的简化版本，详细内容参见 "参考资料九"

```cpp
class Object {
 public:
  int weight;
  int value;

  Object() {
    weight = 0;
    value = 0;
  }

  Object(int _weight = 0, int _value = 0) {
    weight = _weight;
    value = _value;
  }

  // the same as
  // Object(int _weight,int _value):weight(_weight),value(_value) {}
};

// the same as
// Object::Object(int _weight,int _value){
//   weight = _weight;
//   value = _value;
// }
//}

Object A;         // ok
Object B(1, 2);   // ok
Object C{1, 2};   // ok,(C++11)
```

### 关于隐式类型转换

有时候会写出如下的代码

```cpp
class Node {
 public:
  int var;

  Node(int _var) : var(_var) {}
};
```

```
7
8 Node a = 1;
```

看上去十分不符合逻辑，一个 `int` 类型不可能转化为 `node` 类型．但是编译器不会进行 `error` 提示．

原因是在进行赋值时，首先会将 `1` 作为参数调用 `node::node(int)`，然后调用默认的复制函数进行赋值．

但大多数情况下，编写者会希望编译器进行报错．这时便可以在构造函数前追加 `explicit` 关键字．这会告诉编译器必须显式进行调用．

```
1 class Node {
2  public:
3   int var;
4
5   explicit Node(int _var) : var(_var) {}
6 };
```

也就是说 `node a=1` 将会报错，但 `node a=node(1)` 不会．因为后者显式调用了构造函数．当然大多数人不会写出后者的代码，但此例足以说明 explicit 的作用．

不过在算法竞赛中，为了避免此类情况常用的是 "加强对代码的规范程度"，从源头上避免

### 销毁

这是不可避免的问题．每一个变量都将在作用范围结束走向销毁．

但对于已经指向了动态申请的内存的指针来说，该指针在销毁时不会自动释放所指向的内存，需要手动释放动态内存．

如果结构体的成员元素包含指针，同样会遇到这种问题．需要用到析构函数来手动释放动态内存．

**析构** 函数（Destructor）将会在该变量被销毁时被调用．重载的方法形同构造函数，但需要在前加 `~`

默认定义的析构函数通常对于算法竞赛已经足够使用，通常我们只有在成员元素包含指针时才会重载析构函数．

```
1 class Object {
2  public:
3   int weight;
4   int value;
5   int* ned;
6
7   Object() {
8     weight = 0;
9     value = 0;
10  }
11
12  ~Object() { delete ned; }
13 };
```

### 为类变量赋值

默认情况下，赋值时会按照对应成员元素赋值的规则进行．也可以使用 `类名称()` 或 `类名称{}` 作为临时变量来进行赋值．

前者只是调用了复制构造函数（copy constructor），而后者在调用复制构造函数前会调用默认构造函数.

另外默认情况下, 进行的赋值都是对应元素间进行 **浅拷贝**, 如果成员元素中有指针, 则在赋值完成后, 两个变量的成员指针具有相同的地址.

```
// A,tmp1,tmp2,tmp3类型为Object
tmp1 = A;
tmp2 = Object(...);
tmp3 = {...};
```

如需解决指针问题或更多操作，需要重载相应的构造函数.

更多 构造函数（*constructor*）内容，参见「参考资料」第六条.

### 参考资料

1. cppreference class[1]
2. cppreference access[2]
3. cppreference default_constructor[3]
4. cppreference operator[4]
5. cplusplus Data structures[5]
6. cplusplus Special members[6]
7. C++11 FAQ[7]
8. cppreference Friendship and inheritance[8]
9. cppreference value initialization[9]

### 外部参考资料

[1] cppreference class

[2] cppreference access

[3] cppreference default_constructor

[4] cppreference operator

[5] cplusplus Data structures

[6] cplusplus Special members

[7] C++11 FAQ

[8] cppreference Friendship and inheritance

[9] cppreference value initialization

## 4.4.2 命名空间

### 概述

C++ 的 **命名空间** 机制可以用来解决复杂项目中名字冲突的问题.

举个例子：C++ 标准库的所有内容均定义在 `std` 命名空间中，如果你定义了一个叫 `cin` 的变量，则可以通过 `cin` 来访问你定义的 `cin` 变量，通过 `std::cin` 访问标准库的 `cin` 对象，而不用担心产生冲突.

### 声明

下面的代码声明了一个名字叫 `A` 的命名空间：

```
namespace A {
int cnt;

void f(int x) { cnt = x; }
}  // namespace A
```

声明之后，在这个命名空间外部，你可以通过 `A::f(x)` 来访问命名空间 `A` 内部的 `f` 函数，也可以通过 `A::cnt` 来访问命名空间 `A` 内部的 `cnt` 变量.

命名空间的声明是可以嵌套的，因此下面这段代码也是允许的：

```
namespace A {
namespace B {
void f() { ... }
}  // namespace B

void f() {
  B::f();  // 实际访问的是 A::B::f()，由于当前位于命名空间 A
           // 内，所以可以省略前面的 A::
}
}  // namespace A

void f()  // 这里定义的是全局命名空间的 f 函数，与 A::f 和 A::B::f
          // 都不会产生冲突
{
  A::f();
  A::B::f();
}
```

### using 指令

声明了命名空间之后，如果在命名空间外部访问命名空间内部的成员，需要在成员名前面加上 `命名空间::`.

有没有什么比较方便的方法能让我们直接通过成员名访问命名空间内的成员呢? 答案是肯定的. 我们可以使用 `using` 指令.

`using` 指令有如下两种形式：

1. `using 命名空间::成员名;`：这条指令可以让我们省略某个成员名前的命名空间，直接通过成员名访问成员，相当于将这个成员导入了当前的作用域.
2. `using namespace 命名空间;`：这条指令可以直接通过成员名访问命名空间中的 **任何** 成员，相当于将这个命名空间的所有成员导入了当前的作用域.

因此，如果执行了 `using namespace std;`，就会在当前作用域将 `std` 中的所有名字引入到全局命名空间当中. 这样，我们就可以用 `cin` 代替 `std::cin`，用 `cout` 代替 `std::cout`.

> ⚠ **`using` 指令可能会导致命名冲突!**
>
> 由于 `using namespace std;` 会将 `std` 中的 **所有名字** 引入，因此如果声明了与 `std` 重名的变量或函数，就可能会因为命名冲突而导致编译错误.
>
> 因此在工程中，并不推荐使用 `using namespace 命名空间;` 的指令.

有了 `using` 指令，C++ 语法基础→4.2.2@p323 中的代码可以有这两种等价写法：

```
#include <iostream>

using std::cin;
using std::cout;
using std::endl;

int main() {
  int x, y;
  cin >> x >> y;
  cout << y << endl << x;
```

```
  return 0;
}
```

```
#include <iostream>

using namespace std;

int main() {
  int x, y;
  cin >> x >> y;
  cout << y << endl << x;
  return 0;
}
```

## 无名命名空间

当我们在一个作用域里只定义了一个用于防止名字冲突的命名空间时，其定义和使用将可以变得非常简洁．我们可以使用无名命名空间．

形如 `namespace { /* something ... */ }`（省略命名空间的名字）定义的命名空间被称为无名命名空间．一个文件里的无名命名空间会被视为拥有独有的名字，和其他命名空间都不同，但同一个作用域内多个无名命名空间被视为相同的命名空间．在无名命名空间定义后，其中的名字在其外的作用域内可以在使用时被查找到，如同在无名命名空间定义后加入了一条 `using namespace` 指令．

## 应用

### 防止子任务间名字冲突

在一些具有多个子任务的问题中，我们可以对每个子任务各定义一个命名空间，在其中定义我们解决该子任务所需要的变量与函数，这样即使两个子任务的实现中即使声明了相同名字也不会冲突，从而使各个子任务间互不干扰，会在一定程度上方便调试，也会改善程序的可读性．

### 防止与标准库以及环境引入的名字冲突

同时，使用命名空间也可以防止一些算法竞赛中常用的名字与标准冲突，如下例：

```
#include <math.h>

#include <vector>

using namespace std;

namespace Sol {
int end;  // std::end 被 using namespace std; 引入

int y1;  // y1 是 POSIX 定义的第二类 Bessel 函数

// 因此通常情况下，在 Linux 下会有冲突而在 Windows 下没有

void solve() {
  // 在 Sol::solve() 里无限定（不用 ::）地使用我们声明的 end 以及 y1
  // 并不会导致名字冲突； 而若以上代码在全局命名空间中，将会导致冲突： 其中 end
  // 只会在名字查找（即编译使用它的代码）时与 std::end 冲突，而 y1
  // 在声明时就会冲突； 并且 y1 的冲突因为与环境有关甚至在 Windows
  // 下不会被发现，却会在 Linux 的评测环境下造成编译错误．
```

```
}
}  // namespace Sol

int main() { Sol::solve(); }
```

### 参考

- Namespaces - cppreference.com[1]

### 外部参考资料

[1] Namespaces - cppreference.com

## 4.4.3 值类别

值类别是 C++ 中一个非常重要的概念，虽然在算法竞赛中可能用处不大，但了解它可以帮助我们发现并避免不必要的复制，从而提高代码的效率和性能.

值类别的概念在 C 语言、C++98、C++11 和 C++17 中经历了多次发展，逐渐成为一个较为复杂的概念.

### 不必要的复制

我们考虑将字符串塞入 vector 这一过程:

```
int main() {
  std::vector<std::string> vec;
  vec.reserve(3);
  for (int i = 0; i < 3; ++i) {
    std::string str;
    std::cin >> str;
    vec.push_back(str);
  }
  return 0;
}
```

可以发现字符串在转移的过程中，在 `str` 和 `vec` 中各保存了一份，内存占用加倍.

如果非要省下这一部分的内存，我们可以实现一个简陋的移动操作：自定义 `MyString` 结构体，内有一指针指向我们的字符串, 即我们只需要把指针复制过去, 并小心地清理原对象的指针, 防止被错误析构.

```
struct MyString {
  char *beg, *end;
  // ...
};

void move_to(MyString &src, MyString &dst) {
  dst.beg = src.beg;
  dst.end = src.end;
  src.beg = src.end = nullptr;
}
```

由于这种高效转移对象的需求较为常见，且与 C++ 的构造、析构等操作交互困难，C++11 将移动语义引入了语言核心.

### C 语言中的值类别

在 C 语言标准中，对象是一个比变量更为一般化的概念，它指代一块内存区域，具有内存地址．对象的主要属性包括：大小、有效类型、值和标识符．标识符即变量名，值是该内存以其类型解释时的含义．例如，`int` 和 `float` 类型虽然都占用 4 字节，但对于同一块内存，我们会解释出不同的含义．

C 语言中每个表达式都具有类型和值类别．值类别主要分为三类：

- 左值（lvalue）：隐含指代一个对象的表达式．即我们可以对该表达式取地址．
- 右值（rvalue）：不指代对象的表达式，即指代没有存储位置的值，我们无法取该值的地址．
- 函数指代符：函数类型的表达式．

因此，只有可修改的左值（没有 `const` 修饰且非数组的左值）可以位于赋值表达式左侧．

对于某个要求右值作为它的操作数的运算符，每当左值被用作操作数，都会对该表达式应用左值到右值，数组到指针，或者函数到指针标准转换以将它转换成右值．

常见误区：

- 右值表达式继续运算可能是左值．例如 `int *a`，表达式 `a + 1` 是右值，但 `*(a + 1)` 是左值．
- 表达式才有值类别，变量没有．例如 `int *a`，不能说变量 `a` 是左值，可以说其在表达式 `a` 中做左值．

### C++98 中的值类别

C++98 在值类别方面与 C 语言几乎一致，但增加了一些新的规则：

- 函数为左值，因为可以取地址．
- 左值引用（T&）是左值，因为可以取地址．
- 仅有 `const T&` 可绑定到右值．

#### 复制消除

C++ 允许编译器执行复制消除（Copy Elision），可以减少临时对象的创建和销毁．

例如下面的代码，就触发了复制消除中的返回值优化（Return Value Optimization，RVO），你只会看到一次构造和一次复制构造，即便构造与析构有副作用．

```
1  struct X {
2    X() { std::puts("X::X()"); }
3
4    X(const X &) { std::puts("X::X(const X &)"); }
5
6    ~X() { std::puts("X::~X()"); }
7  };
8
9  X get() {
10   X x;
11   return x;
12 }
13
14 int main() {
15   X x = get();
16   X y = X(X(X(X(x))));
17   return 0;
18 }
```

### C++11 中的值类别

C++11 引入了移动语义和右值引用（`T&&`），包括移动构造、移动赋值函数．这给了我们利用临时对象的方法．

我们上面的 `move_to` 可以改写如下：

```
struct MyString {
  // ...
  MyString(MyString&& other) {
    beg = other.beg;
    end = other.end;
    other.beg = other.end = nullptr;
  }
};
```

我们现在关注的表达式特性增加了一点：

- 是否具有身份：是否指代一个对象，即是否有地址.
- 是否可被移动：是否具有移动构造、移动赋值等函数，让我们有办法利用这些临时对象.

因此我们有三种值类别：

- 有身份，不可移动：左值（lvalue）.
- 有身份，可被移动：亡值（xvalue）.
- 无身份，可被移动：纯右值（prvalue）.
- 无身份，不可移动：此类表达式无法使用.

另外 C++11 还引入了两个复合类别：

- 具有身份：泛左值（glvalue），即左值和亡值.
- 可被移动：右值（rvalue），即纯右值和亡值.

**std::move**

为了配合移动语义，C++11 还引入了一个工具函数 `std::move`，其作用是将左值强制转换为右值，以便触发移动语义.

```
int main() {
  std::vector<int> a = {1, 2, 3};
  std::cout << "a: " << a.data() << std::endl;
  std::vector<int> b = a;
  std::cout << "b: " << b.data() << std::endl;
  std::vector<int> c = std::move(b);
  std::cout << "c: " << c.data() << std::endl;
}
```

因此我们只需将 `push_back(str)` 改为 `push_back(std::move(str))` 即可避免复制.

```
int main() {
  std::vector<std::string> vec;
  vec.reserve(3);
  for (int i = 0; i < 3; ++i) {
    std::string str;
    std::cin >> str;
    vec.push_back(std::move(str));
    // 另一种巧妙的写法，需要 C++17
    // std::cin >> vec.emplace_back();
  }
  return 0;
}
```

> 由于 `std::string` 有小对象优化（Small String Optimization，SSO），短字符串直接存储于结构体内，你可能得输入较长的字符串才能观察到 `data` 指针的不变性.

### C++17 中的值类别

C++17 进一步简化了值类别：

- 左值（lvalue）：有身份，不可移动.
- 亡值（xvalue）：有身份，可以移动.
- 纯右值（prvalue）：对象的初始化.

C++11 将复制消除扩展到了移动上，下面的代码中 `urvo` 在编译器启用 RVO 的情况下是没有移动的.

C++17 要求纯右值非必须不实质化，直接构造到其最终目标的存储中，在构造之前对象尚不存在. 因此在 C++17 中我们就没有返回这一步，也就不必依赖 RVO. 也可以理解为强制了 URVO（Unnamed RVO），但对于 NRVO（Named RVO）还是非强制的.

```cpp
std::string urvo() { return std::string("123"); }

std::string nrvo() {
  std::string s;
  s = "123";
  std::cout << s;
  return s;
}

int main() {
  std::string str = urvo();  // 直接构造
  std::string str = nrvo();  // 不一定直接构造，依赖于优化
}
```

同时 C++17 引入了临时量实质化的机制，当我们需要访问成员变量、调用成员函数等需要泛左值的情形时，可以隐式转换为亡值.

### 常见误区

下面的例子中：

- 在 `f1` 中返回 `std::move(x)` 是多余的，并不会带来性能上的提升，反而会干扰编译器进行 NRVO 优化.
- 在 `f2` 中返回 `std::move(x)` 是危险的，函数返回右值引用指向了已被销毁的局部变量 `s`，出现了悬空引用问题.

```cpp
std::string f1() {
  std::string s = "123";
  // 等价于 return std::string(std::move(s))
  return std::move(s);
}

std::string&& f2() {
  std::string s = "123";
  return std::move(s);
}
```

### 参考文献与推荐阅读


1. Value categories[1]
2. Wording for guaranteed copy elision through simplified value categories[2]
3. C++ 中的值类别[3]
4. C++ 的右值引用、移动和值类别系统，你所需要的一切[4]
5. Copy elision[5]

### 外部参考资料

[1] Value categories 

[2] Wording for guaranteed copy elision through simplified value categories 

[3] C++ 中的值类别 

[4] C++ 的右值引用、移动和值类别系统, 你所需要的一切 

[5] Copy elision 

## 4.4.4 重载运算符

重载运算符是通过对运算符的重新定义, 使得其支持特定数据类型的运算操作. 重载运算符是重载函数的特殊情况.

> 当一个运算符出现在一个表达式中, 并且运算符的至少一个操作数具有一个类或枚举的类型时, 则使用重载决议（overload resolution）确定应该调用哪个满足相应声明的用户定义函数. [1]

通俗的讲, 如果把使用「运算符」看作一个调用特殊的函数（如将 `1+2` 视作调用 `add(1, 2)`）, 并且这个函数的参数（操作数）至少有一个是 `class`、`struct` 或 `enum` 的类型, 编译器就需要根据操作数的类型决定应当调用哪个自定义函数.

在 C++ 中, 我们可以重载几乎所有可用的运算符.

**一些可重载运算符的列举**

一元运算: `+`（正号）; `-`（负号）; `~`（按位取反）; `++`; `--`; `!`（逻辑非）; `*`（取指针对应值）; `&`（取地址）; `->`（类成员访问运算符）等.

二元运算: `+`; `-`; `&`（按位与）; `[]`（取下标）; `==`; `=`（赋值）等.

其它: `()`（函数调用）; `""`（后缀标识符[2], C++11 起）; `new`（内存分配）; `,`（逗号运算符）; `<=>`（三路比较[3], C++20 起）等.

### 限制

重载运算符存在如下限制:

- 只能对现有的运算符进行重载, 不能自行定义新的运算符.
- 以下运算符不能被重载: `::`（作用域解析）, `.`（成员访问）, `.*`（通过成员指针的成员访问）, `?:`（三目运算符）.
- 重载后的运算符, 其运算优先级, 运算操作数, 结合方向不得改变.
- 对 `&&`（逻辑与）和 `||`（逻辑或）的重载失去短路求值.

### 实现

重载运算符分为两种情况, 重载为成员函数或非成员函数.

当重载为成员函数时, 因为隐含一个指向当前成员的 `this` 指针作为参数, 此时函数的参数个数与运算操作数相比少一个.

而当重载为非成员函数时, 函数的参数个数与运算操作数相同.

其基本格式为（假设需要被重载的运算符为 `@`）:

```
1 class Example {
2   // 成员函数的例子
```

[1]运算符重载 - cppreference[1]
[2]运算符重载 - cppreference[2]
[3]用户定义字面量 - cppreference[3]

```
  返回值 operator@(除本身外的参数) { /* ... */ }
};

// 非成员函数的例子
返回值 operator@(所有参与运算的参数) { /* ... */ }
```

下面将给出几个重载运算符的示例.

### 基本算数运算符

下面定义了一个二维向量结构体 `Vector2D` 并实现了相应的加法和内积的重载.

**重载算数运算符的例子**

```
struct Vector2D {
  double x, y;

  Vector2D(double a = 0, double b = 0) : x(a), y(b) {}

  Vector2D operator+(Vector2D v) const { return Vector2D(x + v.x, y + v.y); }

  // 注意返回值的类型可以不是这个类
  double operator*(Vector2D v) const { return x * v.x + y * v.y; }
};
```

### 自增自减运算符

自增自减运算符分为两类，前置（`++a`）和后置（`a++`）. 为了区分前后置运算符，重载后置运算时需要添加一个类型为 `int` 的空置形参.

可以将前置自增理解为调用 `operator++(a)` 或 `a.operator++()`，后置自增理解为调用 `operator++(a, 0)` 或 `a.operator++(0)`.

**分别重载前后置自增运算符的例子**

```
struct MyInt {
  int x;

  // 前置，对应 ++a
  MyInt &operator++() {
    x++;
    return *this;
  }

  // 后置，对应 a++
  MyInt operator++(int) {
    MyInt tmp;
    tmp.x = x;
    x++;
    return tmp;
  }
};
```

另外一点是，内置的自增自减运算符中，前置的运算符返回的是引用，而后置的运算符返回的是值. 虽然重载后的运算符不必遵循这一限制，不过在语义上，仍然期望重载的运算符与内置的运算符在返回值的类型上保持一致.

对于类型 T，典型的重载自增运算符的定义如下：

| 重载定义（以 ++ 为例） | 成员函数 | 非成员函数 |
| --- | --- | --- |
| 前置 | `T& T::operator++();` | `T& operator++(T& a);` |
| 后置 | `T T::operator++(int);` | `T operator++(T& a, int);` |

### 函数调用运算符

函数调用运算符 `()` 只能重载为成员函数. 通过对一个类重载 `()` 运算符，可以使该类的对象能像函数一样调用.

重载 `()` 运算符的一个常见应用是，将重载了 `()` 运算符的结构体作为自定义比较函数传入优先队列等 STL 容器中.

下面就是一个例子：给出 $n$ 个学生的姓名和分数，按分数降序排序，分数相同者按姓名字典序升序排序，输出排名最靠前的人的姓名和分数.

下面定义了一个比较结构体，实现自定义优先队列的排序方式.

**重载函数调用运算符的例子**

```cpp
struct student {
  string name;
  int score;
};

struct cmp {
  bool operator()(const student& a, const student& b) const {
    return a.score < b.score || (a.score == b.score && a.name > b.name);
  }
};

// 注意传入的模板参数为结构体名称而非实例
priority_queue<student, vector<student>, cmp> pq;
```

### 比较运算符

在 `std::sort` 和一些 STL 容器中，需要用到 `<` 运算符. 在使用自定义类型时，我们需要手动重载.

下面是一个例子，实现了和上一节相同的功能

**重载比较运算符的例子**

```cpp
struct student {
  string name;
  int score;

  // 重载 < 号运算符
  bool operator<(const student& a) const {
    return score < a.score || (score == a.score && name > a.name);
    // 上面省略了 this 指针，完整表达式如下：
```

```
    // this->score<a.score||(this->score==a.score&&this->name>a.name);
  }
};

priority_queue<student> pq;
```

上面的代码将小于号重载为了成员函数，当然重载为非成员函数也是可以的.

**重载为非成员函数**

```
struct student {
  string name;
  int score;
};

bool operator<(const student& a, const student& b) {
  return a.score < b.score || (a.score == b.score && a.name > b.name);
}

priority_queue<student> pq;
```

事实上，只要有了 `<` 运算符，则其他五个比较运算符的重载也可以很容易实现.

```
/* clang-format off */

// 下面的几种实现均将小于号重载为非成员函数

bool operator<(const T& lhs, const T& rhs) { /* 这里重载小于运算符 */ }
bool operator>(const T& lhs, const T& rhs) { return rhs < lhs; }
bool operator<=(const T& lhs, const T& rhs) { return !(lhs > rhs); }
bool operator>=(const T& lhs, const T& rhs) { return !(lhs < rhs); }
bool operator==(const T& lhs, const T& rhs) { return !(lhs < rhs) && !(lhs > rhs); }
bool operator!=(const T& lhs, const T& rhs) { return !(lhs == rhs); }
```

**关于 C++20 下的三路比较运算符**

如果使用 C++20 或更高版本，我们可以直接使用默认三路比较运算符简化代码.[4]

```
auto operator<=>(const T &lhs, const T &rhs) = default;
```

默认比较的顺序按照成员变量声明的顺序逐个比较.[5]

也可以使用自定义三路比较. 此时要求选择比较内含的序关系（`std::strong_ordering`、`std::weak_ordering` 或 `std::partial_ordering`），或者返回一个对象，使得:

- 若 `a < b`，则 `(a <=> b) < 0`;
- 若 `a > b`，则 `(a <=> b) > 0`;
- 若 `a` 和 `b` 相等或等价，则 `(a <=> b) == 0`.

具体实现细节请参考 比较运算符 # 三路比较 - cppreference[6].

[4] 比较运算符 # 三路比较 - cppreference[4]
[5] 默认比较 - cppreference[5]

参考资料与注释：

[1] 运算符重载 - cppreference 

[2] 运算符重载 - cppreference 

[3] 用户定义字面量 - cppreference 

[4] 比较运算符 # 三路比较 - cppreference 

[5] 默认比较 - cppreference 

[6] 比较运算符 # 三路比较 - cppreference 

## 4.4.5 引用

> 声明具名变量为引用，即既存对象或函数的别名.

引用可以看成是 C++ 封装的非空指针，可以用来传递它所指向的对象，在声明时必须指向对象.

引用不是对象，因此不存在引用的数组、无法获取引用的指针，也不存在引用的引用.

**引用类型不属于对象类型**

如果想让引用能完成一般的复制、赋值等操作，比如作为容器元素，则需要 `reference_wrapper` [1]，通常维护一个非空指针实现.

引用主要分为两种，左值引用和右值引用.

**左值和右值**

对左值和右值的讲解，请参考 值类别→4.4.3@p416 页面.

### 左值引用 T&

通常我们会接触到的引用为左值引用，即绑定到左值的引用，同时 `const` 限定的左值引用可以绑定右值. 以下是来自 参考手册[2] 的一段示例代码.

```cpp
#include <iostream>
#include <string>

int main() {
  std::string s = "Ex";
  std::string& r1 = s;
  const std::string& r2 = s;

  r1 += "ample";  // 修改 r1，即修改了 s
  // r2 += "!"; // 错误：不能通过到 const 的引用修改
  std::cout << r2 << '\n';  // 打印 r2，访问了s，输出 "Example"
}
```

左值引用最常用的地方是函数参数，用于避免不需要的拷贝.

```cpp
#include <iostream>
#include <string>

// 参数中的 s 是引用，在调用函数时不会发生拷贝
char& char_number(std::string& s, std::size_t n) {
  s += s;  // 's' 与 main() 的 'str'
           // 是同一对象，此处还说明左值也是可以放在等号右侧的
  return s.at(n);  // string::at() 返回 char 的引用
}

int main() {
```

```
  std::string str = "Test";
  char_number(str, 1) = 'a';  // 函数返回是左值，可被赋值
  std::cout << str << '\n';   // 此处输出 "TastTest"
}
```

## 右值引用 T&&（C++ 11）

右值引用是绑定到右值的引用，用于移动对象，也可以用于 **延长临时对象生存期**.

```
#include <iostream>
#include <string>

using namespace std;

int main() {
  string s1 = "Test";
  // string&& r1 = s1; // 错误：不能绑定到左值，需要 std::move 或者 static_cast

  const string& r2 = s1 + s1;  // 可行：到常量的左值引用延长生存期
  // r2 += "Test"; // 错误：不能通过到常量的引用修改
  cout << r2 << '\n';

  string&& r3 = s1 + s1;  // 可行：右值引用延长生存期
  r3 += "Test";
  cout << r3 << '\n';

  const string& r4 = r3;  // 右值引用可以转换到 const 限定的左值
  cout << r4 << '\n';

  string& r5 = r3;  // 右值引用可以转换到左值
  cout << r5 << '\n';
}
```

## 悬垂引用

当引用指代的对象已经销毁，引用就会变成悬垂引用，访问悬垂引用这是一种未定义行为，可能会导致程序崩溃.

以下为常见的悬垂引用的例子：

- 引用局部变量

```
#include <iostream>

int& foo() {
  int a = 1;
  return a;
}

int main() {
  int& b = foo();
  std::cout << b << std::endl;  // 未定义行为
}
```

- 解分配导致的悬垂引用

```
#include <iostream>
```

```
2
3 int main() {
4   int* ptr = new int(10);
5   int& ref = *ptr;
6   delete ptr;
7
8   std::cout << ref << std::endl;  // 未定义行为
9 }
```

- 内存重分配导致的悬垂引用

```
1 #include <iostream>
2
3 int main() {
4   std::string str = "hello";
5
6   const char& ref = str.front();
7
8   str.append("world");  // 可能会重新分配内存，导致 ref 指向的内存被释放
9
10  std::cout << ref << std::endl;  // 未定义行为
11 }
```

类似 `std::vector`，`std::unordered_map` 等容器的插入操作，均有可能导致内存重新分配. 使用引用时，应时刻关注引用指向的对象的生命周期，避免造成悬垂引用.

通常静态检查工具和良好的代码习惯能让我们避免悬垂引用的问题.

## 引用相关的优化技巧

### 消除非轻量对象入参的拷贝开销

常见的 **非轻量对象** 有:

- 容器 `vector`，`array`，`map` 等
- `string`
- 其他实现了或继承了自定义拷贝构造、移动构造等特殊函数的类型

而对 **轻量对象** 使用引用不能带来任何好处，引用类型作为参数的空间占用大小，甚至可能会比类型本身还大.

这可能会带来些的性能负担，同时可能会阻止编译器优化.

以下属于 **轻量对象**

- 基本类型 `int`，`float` 等
- 较小的 聚合体类型[3]
- 标准库容器的迭代器

### 将左值转换为右值

使用 `std::move` 转移→4.4.3@p416 对象的所有权. 这通常见于局部变量之间，或参数与局部变量之间:

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4
5 using namespace std;
6
7 string world(string str) { return std::move(str) += " world!"; }
```

```cpp

int main() {
  // 1
  cout << world("hello") << '\n';

  vector<string> vec0;

  // 2
  {
    string&& size = to_string(vec0.size());

    size += ", " + to_string(size.size());

    vec0.emplace_back(std::move(size));
  }

  cout << vec0.front();
}
```

但不是所有时候都需要这么做，比如 函数返回值优化→4.4.3@p416.

### 右值延长临时量生命期

从语义上，临时量可能会带来的额外的复制或移动，尽管多数情况下编译器能通过 复制消除→4.4.3@p416 进行优化，但引用能强制编译器不进行这些多余操作，避免不确定性.

### 参考内容

1. C++ 语言文档——引用声明[4]
2. C++ 语言文档——值类别[5]
3. Does const ref lvalue to non-const func return value specifically reduce copies?[6]

### 外部参考资料

[1] `reference_wrapper` [2] 参考手册 [3] 聚合体类型

[4] C++ 语言文档——引用声明 [5] C++ 语言文档——值类别 [6] Does const ref lvalue to non-const func return value specifically reduce copies?

## 4.4.6 常量

C++ 定义了一套完整的只读量定义方法，被 `const` 修饰的变量都是只读量，编译器会在编译期进行冲突检查，避免对只读量的修改，同时可能会执行一些优化.

在通常情况下，应该尽可能使用 `const` 修饰变量、参数，提高代码健壮性.

### `const` 类型限定符

#### 常量

const 修饰的变量在初始化后不可改变值

```cpp
const int a = 0;  // a 的类型为 const int

// a = 1; // 不能修改常量
```

### 常量引用、常量指针

常量引用和常量指针均限制了对指向的值的修改

```cpp
int a = 0;
const int b = 0;

int *p1 = &a;
*p1 = 1;
const int *p2 = &a;
// *p2 = 2; // 不能通过常量指针修改变量
// int *p3 = &b; // 不能用 int* 指向 const int 变量
const int *p4 = &b;

int &r1 = a;
r1 = 1;
const int &r2 = a;
// r2 = 2; // 不能通过常量引用修改变量
// int &p3 = b; // 不能用 int& 引用 const int变量
const int &r4 = b;
```

另外需要区分开的是常量指针（`const t*`）和指针常量（`t* const`），例如下列声明

```cpp
int* const p1;  // 指针常量，初始化后指向地址不可改，可更改指向的值
const int* p2;  // 常量指针，解引用的值不可改，可指向其他 int 变量
const int* const p3;  // 常量指针常量，值不可改，指向地址不可改

// 使用别名能更好提高可读性
using const_int = const int;
using ptr_to_const_int = const_int*;
using const_ptr_to_const_int = const ptr_to_const_int;
```

在函数参数里使用 `const` 限定参数类型，可以避免变量被错误地修改，同时增加代码可读性

```cpp
void sum(const std::vector<int> &data, int &total) {
  for (auto iter = data.begin(); iter != data.end(); ++iter)
    total += *iter;  // iter 是迭代器，解引用后的类型是 const int
}
```

### `const` 成员函数

类型中 `const` 限定的成员函数，可以用来限制对成员的修改.

```cpp
#include <iostream>

struct ConstMember {
  int s = 0;

  void func() { std::cout << "General Function" << std::endl; }

  void constFunc1() const { std::cout << "Const Function 1" << std::endl; }

  void constFunc2(int ss) const {
    // func(); // const 成员函数不能调用非 const 成员函数
    constFunc1();

    // s = ss; // const 成员函数不能修改成员变量
```

```
  }
};

int main() {
  int b = 1;
  ConstMember c{};
  const ConstMember d = c;
  // d.func(); // 常量不能调用非 const 成员函数
  d.constFunc2(b);
  return 0;
}
```

## 常量表达式 constexpr（C++11）

常量表达式是指编译时能计算出结果的表达式，`constexpr` 则要求编译器能在编译时求得函数或变量的值.

编译时计算能允许更好的优化，比如将结果硬编码到汇编中，消除运行时计算开销. 与 `const` 的带来的优化不同，当 `constexpr` 修饰的变量满足常量表达式的条件，就强制要求编译器在编译时计算出结果而非运行时.

**更直观的理解是把 const 理解成「只读」，constexpr 理解成「不可变」**

```
constexpr int a = 10;  // 直接定义常量

constexpr int FivePlus(int x) { return 5 + x; }

void test(const int x) {
  std::array<int, x> c1;            // 错误，x在编译时不可知
  std::array<int, FivePlus(6)> c2;  // 可行，FivePlus编译时可知
}
```

以下例子很好说明了 `const` 和 `constexpr` 的区别，代码使用递归实现计算斐波那契数列，并用控制流输出.

**实现**

```
#include <iostream>

using namespace std;

constexpr unsigned fib0(unsigned n) {
  return n <= 1 ? 1 : (fib0(n - 1) + fib0(n - 2));
}

unsigned fib1(unsigned n) { return n <= 1 ? 1 : (fib1(n - 1) + fib1(n - 2)); }

int main() {
  constexpr auto v0 = fib0(9);
  const auto v1 = fib1(9);

```

```
  cout << v0;
  cout << ' ';
  cout << v1;
}
```

**编译后的可能的汇编代码（使用 Compiler Explorer，Clang 19）**

```
fib1(unsigned int):
        push    r14
        push    rbx
        push    rax
        mov     ebx, 1
        cmp     edi, 2
        jb      .LBB0_4
        mov     r14d, edi
        xor     ebx, ebx
.LBB0_2:
        lea     edi, [r14 - 1]
        call    fib1(unsigned int)
        add     r14d, -2
        add     ebx, eax
        cmp     r14d, 1
        ja      .LBB0_2
        inc     ebx
.LBB0_4:
        mov     eax, ebx
        add     rsp, 8
        pop     rbx
        pop     r14
        ret

main:
        push    r14
        push    rbx
        push    rax
        mov     edi, 9
        call    fib1(unsigned int) # `v1` 的初始化进行了函数调用
        mov     ebx, eax
        mov     r14, qword ptr [rip + std::__1::cout@GOTPCREL]
        mov     rdi, r14
        mov     esi, 55 # `v0` 被最终计算结果替代
        call    std::__1::basic_ostream<char,
std::__1::char_traits<char>>::operator<<(unsigned int)@PLT
        mov     byte ptr [rsp + 7], 32
        lea     rsi, [rsp + 7]
        mov     edx, 1
        mov     rdi, r14
        call    std::__1::basic_ostream<char, std::__1::char_traits<char>>&
std::__1::__put_character_sequence[abi:ne200000]<char,
std::__1::char_traits<char>>(std::__1::basic_ostream<char,
std::__1::char_traits<char>>&, char const*, unsigned long)
```

```
41          mov     rdi, r14
42          mov     esi, ebx # 读取了变量值
43          call    std::__1::basic_ostream<char,
std::__1::char_traits<char>>::operator<<(unsigned int)@PLT
44          xor     eax, eax
45          add     rsp, 8
46          pop     rbx
47          pop     r14
48          ret
```

`constexpr` 修饰的 `fib0` 函数在唯一的调用处用了常量参数，使得整个函数仅在编译期运行．由于函数没有运行时执行，编译器也就判断不需要生成汇编代码．

在同时注意到汇编中，`v0` 没有初始化代码，在调用 `cout` 输出 `v0` 的代码中，`v0` 已被最终结算结果替代，说明变量值已在编译时求出，优化掉了运行时运算． 而 `v1` 的初始化还是普通的 `fib1` 递归调用．

所以 `constexpr` 可以用来替换宏定义的常量，规避 宏定义的风险→4.2.2@p323．

算法题中可以使用 `constexpr` 存储数据规模较小的变量，以消除对应的运行时计算开销．尤为常见在「打表→2.6.2@p110」技巧中，使用 `constexpr` 修饰的数组等容器存储答案．

**编译时计算量过大会导致编译错误**

编译器会限制编译时计算的开销，如果计算量过大会导致无法通过编译，应该考虑使用 `const`．

```cpp
#include <iostream>

using namespace std;

constexpr unsigned long long fib(unsigned long long i) {
  return i <= 2 ? i : fib(i - 2) + fib(i - 1);
}

int main() {
  // constexpr auto v = fib(32); evaluation exceeded maximum depth
  const auto v = fib(32);
  cout << v;
  return 0;
}
```

**使用 constexpr 时 Clang 给出的编译错误**

```
<source>:10:20: error: constexpr variable 'v' must be initialized by a
constant expression
    10 |     constexpr auto v = fib(32);
       |                    ^   ~~~~~~~~~~~~
<source>:6:25: note: constexpr evaluation exceeded maximum depth of 512
calls
    6 |     return i <= 2 ? i : fib(i - 2) + fib(i - 1);
      |                         ^
<source>:6:25: note: in call to 'fib(32)'
    6 |     return i <= 2 ? i : fib(i - 2) + fib(i - 1);
```

```
 9      |                          ^~~~~~~~~~
10 <source>:6:25: note: in call to ...
```

### 参考资料

- C++ 关键字——const[1]
- C++ 关键字——constexpr[2]

### 外部参考资料

[1] C++ 关键字——const  [2] C++ 关键字——constexpr 

## 4.4.7 新版 C++ 特性

**注意**：考虑到算法竞赛的实际情况，本文将不会全面研究语法，只会讲述在算法竞赛中可能会应用到的部分.

本文语法参照 C++11 标准. 语义不同的将以 C++11 作为标准，C++14、C++17 等语法视情况提及并会特别标注.

### `auto` 类型说明符

`auto` 类型说明符用于自动推导变量等的类型. 例如:

```
auto a = 1;         // a 是 int 类型
auto b = a + 0.1;   // b 是 double 类型
```

注意 `auto` 会去除引用，如果不希望出现拷贝开销，需要手动指定:

```
int a = 1;
int& b = a;
auto c = b;   // c 是 int 类型，有拷贝开销
auto& e = a;  // e 是 int& 类型，没有拷贝开销
```

### decltype 说明符

`decltype` 可以根据 **实体** 或 **表达式** 推断类型，注意二者推导类型的方式不同，错误使用可能造成悬垂引用. 竞赛中不常用，此处仅粗略介绍.

```
#include <iostream>
#include <vector>

int main() {
  int a = 1926;
  decltype(a) b;                  // 根据实体推断， b 是 int 类型
  decltype(1 + 1) c;              // 根据表达式推断，c 是 int 类型
  decltype((a)) d = a;            // 根据表达式推断，d 是 int& 类型!
  std::vector<decltype(b)> vec;   // 根据实体推断，vec 是 std::vector <int> 类型
  return 0;
}
```

### constexpr

另请参阅 常量表达式 constexpr（C++11）→4.4.6@p427

### 基于范围的 `for` 循环

使用范围 for 遍历可迭代对象，与使用迭代器遍历的效率相同. 上述二者的效率一般优于索引遍历，因为不需要根据索引寻址.

下面是一种简单的基于范围的 `for` 循环的语法：

```
for (item_declaration : range_initializer) statement
```

比如：

```
std::array<int, 4> arr = {1, 2, 3, 4};
for (int x : arr) {
  std::cout << x << std::endl;
}
```

上述语法产生的代码效果等价于下列代码：

```
std::array<int, 4> arr = {1, 2, 3, 4};
for (auto px = arr.begin(), ed = arr.end(); px != ed; ++px) {
  std::cout << *px << std::endl;
}
```

#### item-declaration 项声明

声明一个变量用于接受右侧容器中的元素，变量类型要与容器内子元素类型一致. 可以用 `auto` 自动推导类型，复杂类型常用 `auto&` 防止拷贝开销.

#### range-initializer 范围初始化器

范围初始化器可以是任何一种可迭代的对象（比如数组，或定义了 `begin` 和 `end` 成员函数的类对象）. 如果放入表达式，表达式也只会计算一次.

例子：

```
int a[] = {1, 1, 4, 5, 1, 4};
std::vector<int> b{1, 1, 4, 5, 1, 4};
std::map<std::string, int> c{{"114", 114}, {"514", 514}};
for (int i : a) std::cout << i;
for (auto i : b) std::cout << i;
// 下方 i 的类型是 std::pair<const std::string, int>&
for (auto& i : c) std::cout << i.first << i.second;
for (auto i : {1, 1, 4, 5, 1, 4}) std::cout << i;
```

#### 自定义类型支持范围 for

只需提供 `begin` 和 `end` 成员函数，返回类型需要支持比较、自增和解引用（`*` 运算符）.

这里有一个例子：

```
#include <iostream>

struct C {
  int a[4];

  int* begin() { return a; }

  int* end() { return a + 4; }
};

```

```cpp
int main() {
  C c = {1, 9, 2, 6};
  for (auto i : c) std::cout << i << " ";
  std::cout << std::endl;
  // output: 1 9 2 6
  return 0;
}
```

### 初始化语句（C++20）

在 C++20 中还可以使用初始化语句实现一些功能，例如循环计数器：

```cpp
#include <iostream>
#include <vector>

int main() {
  std::vector<int> v = {0, 1, 2, 3, 4, 5};

  for (int counter = 0; auto i : v)  // the init-statement (C++20)
    std::cout << counter++ << ' ' << i << std::endl;
}
```

### 结构化绑定（C++17）

结构化绑定（Structured binding）是 C++17 提供的一种语法糖，可以方便的提取子元素或子元素的引用，像这样：

```cpp
struct C {
  int x{1}, y{2};
};

int arr[]{4, 5, 6};

auto [c1, c2] = C{};       // c1=1,c2=2; int 类型
auto& [a1, a2, a3] = arr;  // a1=arr[0],a2=arr[1],a3=arr[2]; int& 类型
```

注意以下几点：

- 左侧声明的变量数和右侧对象的子元素数必须一致
- 类型声明需要使用 `auto`
- 可以使用 `&` 修饰获取引用

你可以在遍历 `map` 容器时这样写：

```cpp
std::map<std::string, int> m = {{"k1", 1}, {"k2", 2}};

// 使用 "auto&" ，没有拷贝开销
for (auto& [k, v] : m) {
  // k 的类型是 const std::string& ，因为键自带 const 修饰
  // v 的类型是 int&
  std::cout << k << ' ' << v << std::endl;
}
```

### std::tuple 元组

元组[1] 定义于头文件 `<tuple>`，是 `std::pair` 的推广，可以存储多个不同类型的值．下面来看一个例子：

```cpp
#include <iostream>
#include <tuple>
#include <vector>

constexpr auto expr = 4 - 1;  // expr = 3

int main() {
  std::vector<int> vec = {1, 9, 2, 6, 0};
  std::tuple<int, int, std::string, std::vector<int>> tup =
      std::make_tuple(817, 114, "514", vec);

  // 使用 get<> 获取子元素，尖括号内必须是整型常量表达式
  for (auto i : std::get<expr>(tup)) std::cout << i << " ";
  // 首元素编号为 0，故我们 std::get<3> 得到了一个 std::vector<int>
  return 0;
}
```

在 C++17 之后可以使用结构化绑定提取值，像这样：

```cpp
std::vector<int> vec = {1, 9, 2, 6, 0};
std::tuple<int, int, std::string, std::vector<int>> tup =
    std::make_tuple(817, 114, "514", vec);

auto& [a, b, c, d] = tup;  // C++17 Structured binding
std::cout << a << ' ' << b << c << std::endl;
std::cout << d.size() << ' ' << d[2] << std::endl;
```

### 成员函数

| 函数 | 作用 |
| --- | --- |
| `operator=` | 赋值一个 `tuple` 的内容给另一个 |
| `swap` | 交换两个 `tuple` 的内容 |

例子：

```cpp
constexpr std::tuple<int, int> tup = {1, 2};
std::tuple<int, int> tupA = {2, 3}, tupB;
tupB = tup;
tupB.swap(tupA);
```

### 非成员函数

| 函数 | 作用 |
| --- | --- |
| `make_tuple` | 创建一个 `tuple` 对象，其类型根据各实参类型定义 |
| `std::get` | 元组式访问指定的元素 |
| `std::tie` | 将元组中的值赋值到已有变量 |
| `operator==` 等 | 按字典顺序比较 `tuple` 中的值 |
| `std::swap` | 特化的 `std::swap` 算法 |

例子：

```cpp
std::tuple<int, int> tupA = {2, 3}, tupB;
tupB = std::make_tuple(1, 2);
std::swap(tupA, tupB);
```

```
std::cout << std::get<1>(tupA) << std::endl;
int x;
std::tie(x, std::ignore) = tupB;
std::cout << x << std::endl;
```

`std::tie` 将元组元素赋值给已有变量，可以使用 `std::ignore` 跳过不需要的元素. 结构化绑定直接声明新变量（支持值/引用绑定），必须接受所有元素.

## 函数对象

可以使用函数调用运算符 `operator()` 的对象，称为函数对象（FunctionObject）.

它不是一种语言特性，而是一种 概念或者要求[2]，在标准库中广泛应用.

函数对象大致可以分成两类:

1. 函数指针
2. 重载了 `operator()` 运算符的类对象

lambda→4.4.8@p442 就是典型的第二类函数对象，它将捕获的内容存放在成员变量中，并重载了函数调用运算符.

## Lambda 表达式

> 请参考 Lambda 表达式→4.4.8@p442 页面.

### std::function

> ⚠ **请注意性能开销**
>
> `std::function` 会引入一定的性能开销，经 Benchmark→4.4.8@p442 测试，通常会造成 2 到 3 倍以上的性能损失.
>
> 因为它使用了类型擦除的技术，而这通常借由虚函数机制实现，调用虚函数会引入额外的 开销[3].
>
> 请考虑使用 **Lambda 表达式**→4.4.8@p442 或者 **函数对象**→4@p321 代替.

`std::function` 是通用函数封装器，定义于头文件 `<functional>`.

`std::function` 的实例能存储、复制及调用任何 **可调用**[4] 对象，这包括 **Lambda 表达式**→4.4.8@p442、成员函数指针或其他 **函数对象**→4@p321.

若 `std::function` 不含任何可调用对象（比如默认构造），调用时将抛出 `std::bad_function_call`[5] 异常.

```
#include <functional>
#include <iostream>

struct Foo {
  Foo(int num) : num_(num) {}

  void print_add(int i) const { std::cout << num_ + i << '\n'; }

  int num_;
};

void print_num(int i) { std::cout << i << '\n'; }

```

```cpp
struct PrintNum {
  void operator()(int i) const { std::cout << i << '\n'; }
};

int main() {
  // 存储自由函数
  std::function<void(int)> f_display = print_num;
  f_display(-9);

  // 存储 Lambda
  std::function<void()> f_display_42 = []() { print_num(42); };
  f_display_42();

  // 存储到成员函数的调用
  std::function<void(const Foo&, int)> f_add_display = &Foo::print_add;
  const Foo foo(314159);
  f_add_display(foo, 1);
  f_add_display(314159, 1);

  // 存储到数据成员访问器的调用
  std::function<int(Foo const&)> f_num = &Foo::num_;
  std::cout << "num_: " << f_num(foo) << '\n';

  // 存储到函数对象的调用
  std::function<void(int)> f_display_obj = PrintNum();
  f_display_obj(18);
}
```

## 可变参数函数模板

在 C++11 之前，类模板和函数模板都只能接受固定数目的模板参数. C++11 允许 **任意个数、任意类型** 的模板参数.

这里仅简要介绍可变参数 **函数** 模板.

下列代码声明的函数模板 `fun` 可以接受任意个数、任意类型的模板参数作为它的模板形参.

```cpp
template <typename... Clazz>
void fun(Clazz... paras) {}
```

`paras` 是一个函数参数包（function parameter pack），接受 0 个或多个函数实参. `Clazz` 是一个模板参数包（template parameter pack），接受 0 个或多个模板实参（非类型、类型或模板），以 `typename` 标记时只接受类型.

可以简单理解如下:

- 模板参数包通常是一些类型名（但也可以使用编译期常量或模板名）
- 函数参数包通常是一些变量名

现在可以这么调用 `fun` 函数:

```cpp
fun();
fun(1);
fun(1, 2, 3);
fun(1, 0.0, "abc");
```

## 参数包展开

### 参数包展开语法

参数包展开非常简单，使用 `...` 即可，将自动使用 `,` 分隔．比如：

```
template <class A, class... C>
void func(A arg1, C... arg2) {
  // C 是 模板参数包
  tuple<A, C...>();  // 展开成 tuple<int, int, double, bool>();

  // arg2 是函数参数包
  func(arg2...);  // 展开成 func( 2, 1.1, true );
}

func(1, 2, 1.1, true);
```

参数包展开时还可以附带需要的运算，比如：

```
template <class A, class... C>
void func(A arg1, C... arg2) {
  func((arg2 + 1)...);
  // 展开成 func( (2+1) , (1.1+1), (2.1f+1) );
}

func(1, 2, 1.1, 2.1f);
```

### 终止函数

上面的函数无法运行，因为参数数量不断减少，最后变为空参并报错．

我们需要指定终止条件，可以提供一个普通函数，像这样：

```
void func() {}

template <class A, class... C>
void func(A arg1, C... arg2) {
  std::cout << arg1 << std::endl;
  func((arg2 + 1)...);
}

func(1, 2, 1.1, 2.1f);
```

这样，参数数量不为 0 时会调用模板，空参时会调用普通函数，就能正常运行了．

## 折叠表达式（C++17）

C++17 提供了一种简便的语法处理 **函数参数包**，他的语法是这样的（必须用小括号包裹）：

1. `( pack op ... )`，会变成 `(E1 op (... op (EN-1 op EN)))`
2. `( ... op pack )`，会变成 `(((E1 op E2) op ...) op EN)`
3. `( pack op ... op init )`，会变成 `(E1 op (... op (EN−1 op (EN op I))))`
4. `( init op ... op pack )`，会变成 `((((I op E1) op E2) op ...) op EN)`

简单演示一下就好理解了：

```
template <class... C>
void func(C... args) {
  (std::cout << ... << args) << std::endl;
  // 语法 4，等价于 ↓
  // ( ( ( std::cout << 1 ) << 2.1 ) << true ) << std::endl;
```

```
  // 输出: 12.11  注意true输出成了1，因为这里没有指定boolalpha

  std::cout << (args && ...) << std::endl;
  // 语法 1, 等价于 ↓
  // std::cout << ( 1 && ( 2.1 && true ) ) ) << std::endl;
  // 输出: 1
}

func(1, 2.1, true);
```

### 缩写函数模板（C++20）

C++20 起可以直接使用 `auto ...` 作为参数类型，实现函数模板的缩写：

```
void func(auto... args) { (std::cout << ... << args) << std::endl; }
```

注意它本质上仍然是函数模板，与下面的写法等价：

```
template <class... T>
void func(T... args) {
  (std::cout << ... << args) << std::endl;
}
```

## 范围库（C++20）

> 范围库是对迭代器和泛型算法库的一个扩展，使得迭代器和算法可以通过组合变得更强大，并且减少错误.

范围即可遍历的序列，包括数组、容器、视图等.

在需要对容器等范围进行复杂操作时，范围库[6] 可以使得算法编写更加容易和清晰.

### View 视图

视图是一种轻量对象，通过特定机制（如自定义迭代器）来实现一些算法，给范围提供了更多的遍历方式以满足需求.

范围库中已实现了一些常用的视图，大致分为两种：

1. **范围工厂**，用于构造一些特殊的范围，使用这类工厂可以省去手动构造容器的步骤，降低开销，直接生成一个范围.
2. **范围适配器**，提供多种多样的遍历支持，既能像函数一样调用，也可以通过管道运算符 `|` 连接，实现链式调用.

**范围适配器** 作为 **范围适配器闭包对象**[7]，也属于 **函数对象**→4@p321，它们重载了 `operator|`，使得它们能够像管道一样拼装起来.

> **管道运算符**
>
> 此处的 `|` 应该理解成管道运算符，而非按位或运算符，这个用法来自于 Linux 中的 管道[8].

在复杂操作下，也能保持良好可读性，有以下特性：

若 A、B、C 为一些范围适配器闭包对象，R 为某个范围，其他字母为可能的有效参数，表达式

```
R | A(a) | B(b) | C(c, d)
```

等价于

```
C(B(A(R, a), b), c, d)
```

下面以 `ranges::take_view` 与 `ranges::iota_view` 为例:

```cpp
#include <iostream>
#include <ranges>

int main() {
  const auto even = [](int i) { return 0 == i % 2; };

  for (int i : std::views::iota(0, 6) | std::views::filter(even))
    std::cout << i << ' ';
}
```

1. 范围工厂 `std::views::iota(0, 6)` 生成了从 0 到 5 的整数序列的范围
2. 范围适配器 `std::views::filter(even)` 过滤前一个范围，生成了一个只剩下偶数的范围
3. 两个操作使用管道运算符链接

上述代码不需要额外分配堆空间存储每步生成的范围, 实际的生成和过滤运算发生在遍历操作中（更具体而言，内部的迭代器构造、自增和解引用），也就是零开销（Zero Overhead）.

同时，外部输入的范围生命周期，等同于 **范围适配器** 的内部元素的生命周期. 如果外部范围（比如容器、范围工厂）已经销毁，那么再对这些的视图遍历，其效果与解引用悬垂指针一致，属于未定义行为.

为了避免上述情况，应该严格要求适配器的生命周期位于其使用的任何范围的生命周期内.

**范围被销毁时，视图内元素均悬垂**

```cpp
#include <iostream>
#include <ranges>
#include <vector>

using namespace std;

int main() {
  auto view = [] {
    vector<int> vec{1, 2, 3, 4, 5};
    return vec | std::views::filter([](int i) { return 0 == i % 2; });
  }();

  for (int i : view) cout << i << ' ';  // runtime undefined behavior

  return 0;
}
```

### Constrained Algorithm 受约束的算法

> C++20 在命名空间 std::ranges 中提供大多数算法的受约束版本，可以用迭代器 - 哨位对或单个 range 作为实参来指定范围，并且支持投影和指向成员指针可调用对象. 另外还更改了大多数算法的返回类型，以返回算法执行过程中计算的所有潜在有用信息.

这些算法可以理解成旧标准库算法的改良版本，均为函数对象，提供更友好的重载和入参类型检查（基于 `concept`[9]），让我们先以 `std::sort` 和 `ranges::sort` 的对比作为例子

```cpp
#include <algorithm>
```

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main() {
  vector<int> vec{4, 2, 5, 3, 1};

  sort(vec.begin(), vec.end());  // {1, 2, 3, 4, 5}

  for (const int i : vec) cout << i << ", ";
  cout << '\n';

  ranges::sort(vec, ranges::greater{});  // {5, 4, 3, 2, 1}

  for (const int i : vec) cout << i << ", ";

  return 0;
}
```

`ranges::sort` 和 `sort` 的算法实现相同，但提供了基于范围的重载，使得传参更为简洁. 其他的 `std` 命名空间下的算法，多数也有对应的范围重载版本位于 `ranges` 命名空间中.

使用这些范围入参，再结合使用上节视图，能允许我们在进行复杂操作的同时，保持代码可读性，让我们看一个例子:

```cpp
#include <algorithm>
#include <array>
#include <iostream>
#include <ranges>

using namespace std;

int main() {
  const auto& inputs = views::iota(0u, 9u);  // 生产 0 到 8 的整数序列
  const auto& chunks = inputs | views::chunk(3);  // 将序列分块，每块 3 个元素
  const auto& cartesian_product =
      views::cartesian_product(chunks, chunks);  // 计算对块自身进行笛卡尔积

  for (const auto [l_chunk, r_chunk] : cartesian_product)
    // 计算笛卡尔积下的两个块整数的和
    cout << ranges::fold_left(l_chunk, 0u, plus{}) +
                ranges::fold_left(r_chunk, 0u, plus{})
         << ' ';
}
```

**输出:**

```
6 15 24 15 24 33 24 33 42
```

## 参考

1. C++ 参考手册[10]

### 外部参考资料

[1] 元组 [2] 概念或者要求 [3] 开销

[4] **可调用** [5] `std::bad_function_call` [6] 范围库

[7] **范围适配器闭包对象** [8] 管道 [9] `concept`

[10] C++ 参考手册

## 4.4.8 Lambda 表达式

**注意**：考虑到算法竞赛的实际情况，本文将不会全面研究语法，只会讲述在算法竞赛中可能会应用到的部分.

本文语法参照 C++11 标准，其他高版本的标准语法视情况提及并会特别标注.

### Lambda 表达式

Lambda 表达式因数学中的 $\lambda$ 演算得名, 直接对应于其中的 lambda 抽象. 编译器在编译时会根据语法生成一个匿名的 **函数对象**→4.4.7@p432, 以捕获的变量作为其成员, 参数和函数体用于实现 `operator()` 重载.

> **函数对象（Function Object）**
>
> 函数对象是一种类对象，一般通过重载 `operator()` 实现，所以能像函数一样调用. 相较于使用普通的函数，函数对象有很多优点，例如可以保存状态，可以作为参数传递给其他函数等.

以下是 lambda 的一种语法:

```
[capture] (parameters) mutable -> return-type {statement}
```

Lambda 表达式本身是一个类，展开后如以下形式:

```
class Lambda_1 {
 private:
  Lambda_1() : capture-list(init-value) { }

 public:
  return-type operator()(parameters) const { statement }

 private:
  mutable capture-list
};
```

空的 capture 可以隐式转换为函数指针，例如:

```
void (*f)(int, int) = [](int, int) -> void {};
```

下面我们分别对语法中的各部分进行介绍.

#### statement 函数体

Lambda 表达式的函数体与普通函数的函数体类似，除了能访问参数和全局变量等，还可访问 捕获→4@p321 的变量.

### capture 捕获子句

lambda 以 capture 子句开头，它指定哪些变量被捕获，捕获列表可为空，或指定捕获方式：有 `&` 符号前缀的变量通过 引用→4.4.5@p424 访问，没有该前缀的变量通过值访问.

我们也可以使用默认捕获模式，捕获 Lambda 中提及的所有变量：`&` 表示捕获到的所有变量都通过引用访问，`=` 表示捕获到的所有变量都通过值访问.

在默认捕获之后，仍然可以为特定的变量 **显式** 指定捕获模式.

如果需要引用访问外部变量 `a`，并通过值访问外部变量 `b`，那么以下捕获子句都可以做到：

- `[&a, b]`
- `[b, &a]`
- `[&, b]`
- `[b, &]`
- `[=, &a]`

同时捕获列表也可以用于声明变量，类型由初始化器推导，类似于使用 `auto` 声明变量.

以下是一些常见的例子：

```
int a = 0;
auto f0 = []() { return a * 9; };   // Error, 无法访问 'a'
auto f1 = [a]() { return a * 9; };  // OK, 'a' 被值「捕获」
auto f2 = [&a]() { return a++; };   // OK, 'a' 被引用「捕获」
auto f3 = [v = a + 1]() {
  return v + 1;
};  // OK, 使用初始化器声明变量 v，类型与 a 相同

// 注意，使用引用捕获时，请保证被调用时 a 没有被销毁
auto b = f2();  // f2 从捕获列表里获得 a 的值，无需通过参数传入 a
```

### generalized capture 带初始化的捕获（C++14）

自 C++14 起，capture 不仅可以用来捕获外部变量，还可用于声明新的变量并初始化，例如：

```
auto f1 = [val = 520]() {
  return val;
};  // OK, 定义 val 类型为 int，初始值为 520，返回值类型 int

auto f2 = [val = 520LL]() {
  return val;
};  // OK, 定义 val 类型为 long long，初始值为 520，返回值类型 long long

auto f3 = [val = "520"]() {
  return val;
};  // OK, 定义 val 类型为 const char*，初始值为 "520"，返回值类型 const char*

auto f4 = [val = "520"s]() {
  return val;
};  // OK, C++14 起，需要 using namespace std; 或 using namespace std::literals;
    // 定义 val 类型为 std::string，初始值为 std::string("520")，返回值类型
    // std::string

auto f5 = [val = std::string("520")]() {
  return val;
};  // OK, 定义 val 类型为 std::string，初始值为 std::string("520")，返回值类型
    // std::string
```

```
23
24 auto f6 = [val = std::vector<int>(3, 6)]() {
25   return val;
26 };  // OK, 定义 val 类型为 std::vector<int>，大小为 3，元素填充 6，返回值类型
27     // std::vector<int>
28
29 auto f7 = [val = 520]() -> int {
30   return val;
31 };  // OK, 定义 val 类型为 int，初始值为 520，返回值类型 int
32
33 auto f8 = [val = 520]() -> long long {
34   return val;
35 };  // OK, 定义 val 类型为 int，初始值为 520，返回值类型 long long
```

定义新的变量不可以省略初始值，变量的类型由初始值的类型决定，相当于：

```
1 auto val = init-value;
```

以下是错误的写法：

```
1 auto f = [val]() { return val; };  // Error: 'val' was not declared in this
2                                    // scope, identifier "val" is undefined
```

初始化值也可以是外部变量，例如：

```
1 int value = 520;
2 auto f = [val = value]() { return val; };
3 std::cout << f();  // Output: 520
```

`val` 也可以是一个引用类型，可以引用一个外部变量，通过这种方式可以为通过引用捕获的外部变量取个别名，例如：

```
1 int value = 520;
2
3 auto f = [&val = value]() {
4   return val;
5 };  // OK, 定义 val 类型为 int&，返回值类型 int，相当于 int& val = value;
6
7 std::cout << f() << '\n';  // Output: 520
8
9 value = 1314;
10
11 std::cout << f() << '\n';  // Output: 1314
```

捕获外部变量和定义新变量可以同时使用.

如果你想在 Lambda 表达式内修改 capture 中定义的新变量，需要使用 `mutable` 关键字，如果是引用则不需要，例如：

```
1 int value = 520;
2
3 {
4   auto f = [val = value]() mutable -> int {
5     return val = 1314;
6   };  // 需要 mutable
7   auto val_f = f();
8   std::cout << value << ' ' << val_f << std::endl;  // Output: 520 1314
9 }
```

```cpp

{
  auto f = [&val = value]() -> int { return val = 1314; };  // 不需要 mutable
  auto val_f = f();
  std::cout << value << ' ' << val_f << std::endl;  // Output: 1314 1314
}
```

详见 mutable 可变规范→4@p321.

在 capture 中定义的变量的生命周期跟随 Lambda 表达式的接收方，在以上几个示例中为变量 $f$，因为 Lambda 本身其实是一个类，capture 中的所有内容都是这个类的 `private` 成员变量，例如：

```cpp
int main() {
  auto f = [val = 0]() mutable -> int { return ++val; };  // val 被构造和初始化

  std::cout << f() << '\n';  // Output: 1
  std::cout << f() << '\n';  // Output: 2
  std::cout << f() << '\n';  // Output: 3
}  // val 跟随 f 被销毁
```

### parameters 参数列表

大多数情况下类似于函数的参数列表，例如：

```cpp
int x[] = {5, 1, 7, 6, 1, 4, 2};
std::sort(x, x + 7, [](int a, int b) { return (a > b); });
for (auto i : x) std::cout << i << " ";
```

这将打印出 `x` 数组从大到小排序后的结果.

由于 **parameters 参数列表** 是可选的, 如果不将参数传递给 lambda, 并且其声明不包含 mutable→4@p321, 且没有后置返回值类型，则可以省略空括号.

> **使用 auto 声明的参数**
>
> C++14 后，若参数使用 `auto` 声明类型，那么会构造一个 泛型 Lambda 表达式→4@p321.

### 显式对象形参（C++23）

C++23 起，显式对象形参[1] 可以在 lambda 的参数中使用.

```cpp
auto nth_fibonacci = [](this auto self, unsigned n) -> unsigned {
  return n < 2 ? n : self(n - 1) + self(n - 2);
};

cout << nth_fibonacci(10u);
```

### mutable 可变规范

使得函数体可以修改通过值捕获的变量.

```cpp
int a = 0;
auto by_value = [a]() mutable { ++a; };
auto by_ref = [&a] { ++a; };

by_value();
by_ref();
```

在执行完 `by_value()` 后，`by_value` 的捕获成员 `a` 为 1，但外部的变量 `a` 依然为 0. 而在执行完 `by_ref()` 后，外部 `a` 的值变为 1.

### return-type 返回类型

用于指定 lambda 表达式的返回类型. 如果省略，则返回类型将被自动推断（行为与用 `auto` 声明返回值的函数一致）.

多个 `return` 语句且推导类型不一致时，将产生编译错误.

```
auto lam = [](int a, int b) -> int { return 0; };

auto x1 = [](int i) { return i; };

auto x2 = [](bool condition) {
  if (condition) return 1;
  return 1.0;
};  // Error, 推导类型不一致
```

### 泛型 Lambda（C++14）

使用 `auto` 作为参数类型，可以构造泛型 lambda.

```
auto add = [](auto a, auto b) { return a + b; };
```

在 cpp insights[2] 中可以观察到编译器生成的 `lambda` 类定义：

```
class add_lambda {
 public:
  template <class T, class U>
  auto operator()(T a, U b) const {
    return a + b;
  }
};

add_lambda add{};
```

`add` 两个参数声明均使用了 `auto`，对应为 `add_lambda` 类的 `operator()` 函数模板的两个模板参数 `T` 和 `U`.

### Lambda 中的递归

先来看一个编译失败的例子：

```
int n = 10;

auto dfs = [&](int i) -> void {
  if (i == n)
    return;
  else
    dfs(i + 1);  // Error: a variable declared with an auto type specifier
                 // cannot appear in its own initializer
};
```

我们这里尝试在捕获列表中捕获 $dfs$，但是有一个问题，$dfs$ 的类型为 `auto`，要等待等号右边的类型推导完成后才会推导出 $dfs$ 的类型，而 Lambda 要捕获 $dfs$ 就必须要确定 $dfs$ 的类型后才能创建它的引用变量，好，这会陷入了一个套娃过程.

怎么解决这个问题呢?

1. 显式指定 $dfs$ 的类型，可以使用 `std::function` 替代.

**修改如上代码为：**

```cpp
int n = 10;

std::function<void(int)> dfs = [&](int i) -> void {
  if (i == n)
    return;
  else
    dfs(i + 1);  // OK
};

dfs(1);
```

**不建议使用 `std::function`→4.4.7@p432 实现的递归**

`std::function` 的类型擦除通常需要分配额外内存，同时间接调用带来的寻址操作会进一步降低性能.

在 Benchmark[3] 测试中，使用 Clang 17 编译器，libc++ 作为标准库，`std::function` 实现比 lambda 实现的递归慢了约 2.5 倍.

**测试代码**

```cpp
#include <algorithm>
#include <functional>
#include <numeric>
#include <random>

using namespace std;

const auto& nums = [] {
  random_device rd;
  mt19937 gen{rd()};
  array<unsigned, 32> arr{};

  std::iota(arr.begin(), arr.end(), 0u);
  ranges::shuffle(arr, gen);

  return arr;
}();

static void std_function_fib(benchmark::State& state) {
  std::function<int(int)> fib;

  fib = [&](int n) { return n <= 2 ? 1 : fib(n - 1) + fib(n - 2); };

  unsigned i = 0;

  for (auto _ : state) {
```

```
    auto res = fib(nums[i]);
    benchmark::DoNotOptimize(res);

    ++i;

    if (i == nums.size()) i = 0;
  }
}

BENCHMARK(std_function_fib);

static void template_lambda_fib(benchmark::State& state) {
  auto n_fibonacci = [](const auto& self, int n) -> int {
    return n <= 2 ? 1 : self(self, n - 1) + self(self, n - 2);
  };

  unsigned i = 0;

  for (auto _ : state) {
    auto res = n_fibonacci(n_fibonacci, nums[i]);
    benchmark::DoNotOptimize(res);

    ++i;

    if (i == nums.size()) i = 0;
  }
}

BENCHMARK(template_lambda_fib);
```

2. 不通过捕获的方式获取 $dfs$，而是通过函数传参的方式.

**修改如上代码为：**

```
int n = 10;

// 参数列表中有参数类型为 auto，则这个 Lambda 类中的 operator()
// 函数将被定义为模板函数，模板函数可以在稍后被调用时再进行实例化
auto dfs = [&](auto& self,
               int i) -> void  // [&] 只会捕获用到的变量，所以不会捕获 auto dfs
{
  if (i == n)
    return;
  else
    self(self, i + 1);  // OK
};

dfs(dfs, 1);
```

**`auto self`、`auto& self` 和 `auto&& self` 的区别：**

`auto& self` 和 `auto&& self` 理论上都只会使用 8 个字节（指针的大小）用作传参，不会发生其他的拷贝．具体要看编译器对 Lambda 的实现方式和对应的优化． 而使用 `auto self` 会发生对象拷贝，拷贝的大小取决于捕获列表中的元素，因为它们都是这个 Lambda 类中的私有成员变量．

3. 可以通过手动展开 Lambda 类，或使用类似写法，这样可以直接声明 $dfs$ 的类型．

**修改如上代码为：**

```cpp
int n = 10;

class Lambda_1 {
 public:
  auto operator()(int i) const -> void {
    if (i == n)
      return;
    else
      (*this)(i + 1);  // OK
  }

  explicit Lambda_1(int& __n) : n(__n) {}

 private:
  int& n;
} dfs(n);

dfs(1);
```

4. 如果 lambda 没有捕获任何变量，我们也可以利用函数指针．

   如果 lambda 没有捕获任何变量，那么它可以隐式转换为函数指针．同时 lambda 此时也可以声明为 `static`，函数指针类型也可以声明为 `static`．如此依赖，lambda 可以不需要捕获就能访问函数指针，从而实现递归．

**示例**

```cpp
static unsigned (*fptr)(unsigned);

static const auto lambda = [](const unsigned a) {
  return a < 2 ? a : (*fptr)(a - 2) + (*fptr)(a - 1);
};

static auto init = [] {
  fptr = +lambda;
  // Or
  // fptr = static_cast<unsigned (*)(unsigned)>(lambda);
  return 0;
}();

cout << lambda(10);
```

### Lambda 表达式的应用

#### 作为标准库算法的 Predicate（谓词）

从大到小排序：

```
std::vector<int> v = {1, 2, 3, 4, 5};
std::sort(v.begin(), v.end(), [](int a, int b) { return a > b; });
```

使用 std::find_if[4] 查找第一个大于 3 的元素：

```
std::vector<int> v = {1, 2, 3, 4, 5};
auto it = std::find_if(v.begin(), v.end(), [](int a) { return a > 3; });
```

#### 控制中间变量的生命周期

在算法竞赛中，我们会遇到这样的场景：一个变量的初始化需要使用之前声明的变量，其初始化过程又生成占用空间较大的中间变量.

我们希望能尽快析构这些中间变量，以降低内存消耗. 此时，我们可以使用 lambda 来控制这些中间变量的生命周期.

```
void solution(const vector<int>& input) {
  int b = [&] {
    vector<int> large_objects(input.size());
    int c = 0;

    for (int i = 0; i < large_objects.size(); ++i)
      large_objects[i] = i + input[i];

    for (int i = 0; i < input.size(); ++i) c += large_objects[input[i]];

    return c;
  }();

  // ...
}
```

相较于使用块作用域，lambda 可以允许我们使用返回值，使得代码更加简洁；相较于函数，我们不需要额外起名和声明被捕获的各种参数，使得代码更加紧凑.

## 参考文献


- cppreference-lambda[5]
- Stackoverflow: Overhead with std::function[6]


## 外部参考资料


[1] 显式对象形参 

[2] cpp insights 

[3] Benchmark 

[4] std::find_if 

[5] cppreference-lambda 

[6] Stackoverflow: Overhead with std::function 


# 4.4.9 pb_ds

## 4.4.9.1 pb_ds 简介

**Authors:** HeRaNO, Xeonacid, saffahyjp

pb_ds 库全称 Policy-Based Data Structures.

pb_ds 库封装了很多数据结构，比如哈希（Hash）表，平衡二叉树，字典树（Trie 树），堆（优先队列）等.

就像 `vector`、`set`、`map` 一样，其组件均符合 STL 的相关接口规范. 部分（如优先队列）包含 STL 内对应组件的所有功能，但比 STL 功能更多.

pb_ds 只在使用 libstdc++ 为标准库的编译器下可以用.

可以使用 `begin()` 和 `end()` 来获取 `iterator` 从而遍历

可以 `increase_key`,`decrease_key` 以及删除单个元素

由于 pb_ds 库的主要内容在以下划线开头的 `__gnu_pbds` 命名空间中，在 NOI 系列活动中的合规性一直没有确定. 2021 年 9 月 1 日，根据《关于 NOI 系列活动中编程语言使用限制的补充说明》[1]，允许使用以下划线开头的库函数或宏（但具有明确禁止操作的库函数和宏除外），在 NOI 系列活动中使用 pb_ds 库的合规性有了文件上的依据.

**参考资料：《C++ 的 pb_ds 库在 OI 中的应用》[2]**

### 外部参考资料

[1]《关于 NOI 系列活动中编程语言使用限制的补充说明》

[2]《C++ 的 pb_ds 库在 OI 中的应用》

### 4.4.9.2 堆

> **Authors:** Xeonacid, ouuan, Ir1d, WAAutoMaton, Chrogeek, abc1763613206, Planet6174, i-Yirannn, opsiff, GoodCoder666

#### `__gnu_pbds::priority_queue`

附：官方文档地址——复杂度及常数测试[1]

```
#include <ext/pb_ds/priority_queue.hpp>
using namespace __gnu_pbds;
__gnu_pbds::priority_queue<T, Compare, Tag, Allocator>
```

### 模板形参

- `T`: 储存的元素类型
- `Compare`: 提供严格的弱序比较类型
- `Tag`: 是 `__gnu_pbds` 提供的不同的五种堆，Tag 参数默认是 `pairing_heap_tag` 五种分别是:
  - `pairing_heap_tag`：配对堆 官方文档认为在非原生元素（如自定义结构体/`std::string`/`pair`）中，配对堆表现最好
  - `binary_heap_tag`：二叉堆 官方文档认为在原生元素中二叉堆表现最好，不过笔者测试的表现并没有那么好
  - `binomial_heap_tag`：二项堆 二项堆在合并操作的表现要优于二叉堆，但是其取堆顶元素操作的复杂度比二叉堆高
  - `rc_binomial_heap_tag`：冗余计数二项堆
  - `thin_heap_tag`：除了合并的复杂度都和 Fibonacci 堆一样的一个 tag
- `Allocator`：空间配置器，由于 OI 中很少出现，故这里不做讲解

由于本篇文章只是提供给学习算法竞赛的同学们，故对于后四个 tag 只会简单的介绍复杂度，第一个会介绍成员函数和使用方法.

经作者本机 Core i5 @3.1 GHz On macOS 测试堆的基础操作，结合 GNU 官方的复杂度测试，Dijkstra 测试，都表明： 至少对于 OIer 来讲，除了配对堆的其他四个 tag 都是鸡肋，要么没用，要么常数大到不

如 `std` 的，且有可能造成 MLE，故这里只推荐用默认的配对堆．同样，配对堆也优于 `algorithm` 库中的 `make_heap()`.

### 构造方式

要注明命名空间因为和 `std` 的类名称重复.

```
// __gnu_pbds::priority_queue<int>;
// __gnu_pbds::priority_queue<int, greater<int>>;
// __gnu_pbds::priority_queue<int, greater<int>, pairing_heap_tag>;
__gnu_pbds::priority_queue<int>::point_iterator id;  // 点类型迭代器
// 在 modify 和 push 的时候都会返回一个 point_iterator，下文会详细的讲使用方法
id = q.push(1);
```

### 成员函数

- `push()`: 向堆中压入一个元素，返回该元素位置的迭代器.
- `pop()`: 将堆顶元素弹出.
- `top()`: 返回堆顶元素.
- `size()` 返回元素个数.
- `empty()` 返回是否非空.
- `modify(point_iterator, const key)`: 把迭代器位置的 `key` 修改为传入的 `key`，并对底层储存结构进行排序.
- `erase(point_iterator)`: 把迭代器位置的键值从堆中擦除.
- `join(__gnu_pbds::priority_queue &other)`: 把 `other` 合并到 `*this` 并把 `other` 清空.

使用的 tag 决定了每个操作的时间复杂度:

| | push | pop | modify | erase | Join |
|---|---|---|---|---|---|
| `pairing_heap_tag` | $O(1)$ | 最坏 $\Theta(n)$ 均摊 $\Theta(\log(n))$ | 最坏 $\Theta(n)$ 均摊 $\Theta(\log(n))$ | 最坏 $\Theta(n)$ 均摊 $\Theta(\log(n))$ | $O(1)$ |
| `binary_heap_tag` | 最坏 $\Theta(n)$ 均摊 $\Theta(\log(n))$ | 最坏 $\Theta(n)$ 均摊 $\Theta(\log(n))$ | $\Theta(n)$ | $\Theta(n)$ | $\Theta(n)$ |
| `binomial_heap_tag` | 最坏 $\Theta(\log(n))$ 均摊 $O(1)$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ |
| `rc_binomial_heap_tag` | $O(1)$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ |
| `thin_heap_tag` | $O(1)$ | 最坏 $\Theta(n)$ 均摊 $\Theta(\log(n))$ | 最坏 $\Theta(\log(n))$ 均摊 $O(1)$ | 最坏 $\Theta(n)$ 均摊 $\Theta(\log(n))$ | $\Theta(n)$ |

### 示例

```
#include <algorithm>
#include <cstdio>
#include <ext/pb_ds/priority_queue.hpp>
#include <iostream>
using namespace __gnu_pbds;
// 由于面向OIer，本文以常用堆 : pairing_heap_tag作为范例
// 为了更好的阅读体验，定义宏如下 :
using pair_heap = __gnu_pbds::priority_queue<int>;
pair_heap q1;  // 大根堆，配对堆
pair_heap q2;
pair_heap::point_iterator id;  // 一个迭代器

int main() {
```

```
  id = q1.push(1);
  // 堆中元素 : [1];
  for (int i = 2; i <= 5; i++) q1.push(i);
  // 堆中元素 : [1, 2, 3, 4, 5];
  std::cout << q1.top() << std::endl;
  // 输出结果 : 5;
  q1.pop();
  // 堆中元素 : [1, 2, 3, 4];
  id = q1.push(10);
  // 堆中元素 : [1, 2, 3, 4, 10];
  q1.modify(id, 1);
  // 堆中元素 : [1, 1, 2, 3, 4];
  std::cout << q1.top() << std::endl;
  // 输出结果 : 4;
  q1.pop();
  // 堆中元素 : [1, 1, 2, 3];
  id = q1.push(7);
  // 堆中元素 : [1, 1, 2, 3, 7];
  q1.erase(id);
  // 堆中元素 : [1, 1, 2, 3];
  q2.push(1), q2.push(3), q2.push(5);
  // q1中元素 : [1, 1, 2, 3], q2中元素 : [1, 3, 5];
  q2.join(q1);
  // q1中无元素, q2中元素 : [1, 1, 1, 2, 3, 3, 5];
}
```

### __gnu_pbds 迭代器的失效保证（invalidation_guarantee）

在上述示例以及一些实践中（如使用本章的 pb-ds 堆来编写单源最短路等算法），常常需要保存并使用堆的迭代器（如 `__gnu_pbds::priority_queue<int>::point_iterator` 等）.

可是例如对于 `__gnu_pbds::priority_queue` 中不同的 Tag 参数，其底层实现并不相同，迭代器的失效条件也不一样，根据__gnu_pbds 库的设计，以下三种由上至下派生的情况：

1. 基本失效保证（basic_invalidation_guarantee）：即不修改容器时，点类型迭代器（point_iterator）、指针和引用（key/value）**保持** 有效.
2. 点失效保证（point_invalidation_guarantee）：即 **修改** 容器后，点类型迭代器（point_iterator）、指针和引用（key/value）只要对应在容器中没被删除 **保持** 有效.
3. 范围失效保证（range_invalidation_guarantee）：即 **修改** 容器后，除（2）的特性以外，任何范围类型的迭代器（包括 `begin()` 和 `end()` 的返回值）是正确的，具有范围失效保证的 Tag 有 `rb_tree_tag` 和 适用于 `__gnu_pbds::tree` 的 `splay_tree_tag`，以及 适用于 `__gnu_pbds::trie` 的 `pat_trie_tag`.

从运行下述代码中看出，除了 `binary_heap_tag` 为 `basic_invalidation_guarantee` 在修改后迭代器会失效，其余的均为 `point_invalidation_guarantee` 可以实现修改后点类型迭代器（point_iterator）不失效的需求.

```
#include <iostream>
using namespace std;
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/priority_queue.hpp>
using namespace __gnu_pbds;
#include <cxxabi.h>

template <typename T>
void print_invalidation_guarantee() {
```

```
  using gute = __gnu_pbds::container_traits<T>::invalidation_guarantee;
  cout << abi::__cxa_demangle(typeid(gute).name(), 0, 0, 0) << endl;
}

int main() {
  using pairing =
      __gnu_pbds::priority_queue<int, greater<int>, pairing_heap_tag>;
  using binary = __gnu_pbds::priority_queue<int, greater<int>, binary_heap_tag>;
  using binomial =
      __gnu_pbds::priority_queue<int, greater<int>, binomial_heap_tag>;
  using rc_binomial =
      __gnu_pbds::priority_queue<int, greater<int>, rc_binomial_heap_tag>;
  using thin = __gnu_pbds::priority_queue<int, greater<int>, thin_heap_tag>;
  print_invalidation_guarantee<pairing>();
  print_invalidation_guarantee<binary>();
  print_invalidation_guarantee<binomial>();
  print_invalidation_guarantee<rc_binomial>();
  print_invalidation_guarantee<thin>();
  return 0;
}
```

### 外部参考资料

[1] 官方文档地址——复杂度及常数测试



### 4.4.9.3 平衡树

**`__gnu_pbds::tree`**

附：官方文档地址[1]

```
#include <ext/pb_ds/assoc_container.hpp>  // 因为 tree 定义在这里 所以需要包含这个头文件
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
__gnu_pbds::tree<Key, Mapped, Cmp_Fn = std::less<Key>, Tag = rb_tree_tag,
                 Node_Update = null_tree_node_update,
                 Allocator = std::allocator<char>>
```

### 模板形参

- `Key`: 储存的元素类型，如果想要存储多个相同的 `Key` 元素，则需要使用类似于 `std::pair` 和 `struct` 的方法，并配合使用 `lower_bound` 和 `upper_bound` 成员函数进行查找
- `Mapped`: 映射规则（Mapped-Policy）类型，如果要指示关联容器是 **集合**，类似于存储元素在 `std::set` 中，此处填入 `null_type`，低版本 `g++` 此处为 `null_mapped_type`；如果要指示关联容器是 **带值的集合**，类似于存储元素在 `std::map` 中，此处填入类似于 `std::map<Key, Value>` 的 `Value` 类型
- `Cmp_Fn`: 关键字比较函子，例如 `std::less<Key>`
- `Tag`: 选择使用何种底层数据结构类型，默认是 `rb_tree_tag`. `__gnu_pbds` 提供不同的三种平衡树，分别是：
  - `rb_tree_tag`：红黑树，一般使用这个，后两者的性能一般不如红黑树
  - `splay_tree_tag`：splay 树
  - `ov_tree_tag`：有序向量树，只是一个由 `vector` 实现的有序结构，类似于排序的 `vector` 来实现平衡树，性能取决于数据想不想卡你

- `Node_Update`：用于更新节点的策略，默认使用 `null_node_update`，若要使用 `order_of_key` 和 `find_by_order` 方法，需要使用 `tree_order_statistics_node_update`
- `Allocator`：空间分配器类型

### 构造方式

```
__gnu_pbds::tree<std::pair<int, int>, __gnu_pbds::null_type,
                 std::less<std::pair<int, int>>, __gnu_pbds::rb_tree_tag,
                 __gnu_pbds::tree_order_statistics_node_update>
    trr;
```

### 成员函数

- `insert(x)`：向树中插入一个元素 `x`，返回 `std::pair<point_iterator, bool>`，其中第一个元素代表插入位置的迭代器，第二个元素代表是否插入成功.
- `erase(x)`：从树中删除一个元素/迭代器 `x`. 如果 `x` 是迭代器，则返回指向 `x` 下一个的迭代器（如果 `x` 是 `end()` 则返回 `end()`）；如果 `x` 是 `Key`，则返回是否删除成功（如果不存在则删除失败）.
- `order_of_key(x)`：返回严格小于 `x` 的元素个数（以 `Cmp_Fn` 作为比较逻辑），即从 0 开始的排名.
- `find_by_order(x)`：返回 `Cmp_Fn` 比较的排名所对应元素的迭代器.
- `lower_bound(x)`：返回第一个不小于 `x` 的元素所对应的迭代器（以 `Cmp_Fn` 作为比较逻辑）.
- `upper_bound(x)`：返回第一个严格大于 `x` 的元素所对应的迭代器（以 `Cmp_Fn` 作为比较逻辑）.
- `join(x)`：将 `x` 树并入当前树，`x` 树被清空（必须确保两树的 **比较函数** 和 **元素类型** 相同）.
- `split(x,b)`：以 `Cmp_Fn` 比较，小于等于 `x` 的属于当前树，其余的属于 `b` 树.
- `empty()`：返回是否为空.
- `size()`：返回大小.

> ⚠ **注意**
>
> `join(x)` 函数需要保证并入树的键的值域与被并入树的键的值域 **不相交**（也就是说并入树内所有值必须全部大于/小于当前树内的所有值），否则会抛出 `join_error` 异常.
>
> 如果要合并两棵值域有交集的树，需要将一棵树的元素一一插入到另一棵树中.

### 示例

```
// Common Header Simple over C++11
#include <iostream>
using namespace std;
using ll = long long;
using ull = unsigned long long;
using ld = long double;
using pii = pair<int, int>;
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
__gnu_pbds::tree<pair<int, int>, __gnu_pbds::null_type, less<pair<int, int>>,
                 __gnu_pbds::rb_tree_tag,
                 __gnu_pbds::tree_order_statistics_node_update>
    trr;

int main() {
  int cnt = 0;
  trr.insert(make_pair(1, cnt++));
  trr.insert(make_pair(5, cnt++));
```

```
  trr.insert(make_pair(4, cnt++));
  trr.insert(make_pair(3, cnt++));
  trr.insert(make_pair(2, cnt++));
  // 树上元素 {(1,0), (2,4), (3,3), (4,2), (5,1)}

  auto it = trr.lower_bound(make_pair(2, 0));
  trr.erase(it);
  // 树上元素 {(1,0), (3,3), (4,2), (5,1)}

  // 输出排名 0 1 2 3 中的排名 1 的元素的 first
  auto it2 = trr.find_by_order(1);
  cout << (*it2).first << endl;  // 输出: 3

  // 输出其排名
  int pos = trr.order_of_key(*it2);
  cout << pos << endl;  // 输出: 1

  // 按照 it2 分裂 trr
  decltype(trr) newtr;
  trr.split(*it2, newtr);
  for (auto i = newtr.begin(); i != newtr.end(); ++i) {
    cout << (*i).first << ' ';  // 输出: 4 5
  }
  cout << endl;

  // 将 newtr 树并入 trr 树，newtr 树被清空.
  trr.join(newtr);
  for (auto i = trr.begin(); i != trr.end(); ++i) {
    cout << (*i).first << ' ';  // 输出: 1 3 4 5
  }
  cout << endl;
  cout << newtr.size() << endl;  // 输出: 0

  return 0;
}
```

### 参考资料

- Tree-Based Containers[2]
- `join` 函数在 GCC 14.1.0 中的实现[3]
- `erase` 函数在 GCC 14.1.0 中的实现[4]

### 外部参考资料

[1] 官方文档地址

[2] Tree-Based Containers

[3] `join` 函数在 GCC 14.1.0 中的实现

[4] `erase` 函数在 GCC 14.1.0 中的实现

## 4.4.10 编译优化

**Authors:** inclyc

OI 界的常用编程语言是 C++. 既然使用了这门语言，就注定要和编译器、语言标准打交道了. 众所周知，C++ 非常混乱邪恶，本文旨在给出实用的编译器相关知识，足够竞赛使用.

## 编译器优化简介

### 什么是优化 (Optimization)

根据 如同规则[1]（The as-if Rule），在保持语义不变的情况下，对程序运行速度、程序可执行文件大小作出改进.

## 常见的编译器优化

### 常量折叠 (Constant Folding)

常量折叠，又称常量传播 (Constant Propagation)，如果一个表达式可以确定为常量，在他的下一个定义 (Definition) 前，可以进行常量传播.

```
int x = 1;
int y = x;  // x = 1, => y = 1
x = 3;
int z = 2 * y;   // z => 2 * y = 2 * 1 = 2
int y2 = x * 2;  // x = 3, => y2 = 6
```

这段代码在编译期间即可被转换为:

```
int x = 1;
int y = 1;
x = 3;
int z = 2;
int y2 = 6;
```

实例：https://godbolt.org/z/oEfY35TTd[2]

### 死代码消除 (Deadcode Elimination)

故名思义，就是一段代码没用上就会被删去.

```
int test() {
  int a = 233;
  int b = a * 2;
  int c = 234;
  return c;
}
```

将被转换为

```
int test() { return 234; }
```

注意，这个代码首先进行了常量折叠，使得返回值可以确定为 234，a, b 为不活跃变量，因此删除.

### 循环旋转 (Loop Rotate)

将循环从 "for" 形式，转换为 "do-while" 形式，前面再多加一个条件判断. 这个变换主要为其他变换做准备.

```
for (int i = 0; i < n; ++i) {
  auto v = *p;
  use(v);
}
```

变换为

```
if (0 < n) {
  do {
```

```
    auto v = *p;
    use(v);
    ++i;
  } while (i < n);
}
```

### 循环不变量外提 (Loop Invariant Code Motion)

基于别名分析 (Alias Analysis)，将循环中被证明是不变量（可能包含内存访问，load/store，因此依赖别名分析）的代码外提出循环体，这样可以让循环体内部少一些代码.

```
for (int i = 0; i < n; ++i) {
  auto v = *p;
  use(v);
}
```

这个代码直观来看可以外提为:

```
auto v = *p;
for (int i = 0; i < n; ++i) {
  use(v);
}
```

但实际上, 如果`n <= 0`, 这个循环永远不会被进入, 但我们又执行了一条多的指令 (可能有副作用!). 因此，循环通常被 Rotate 为 do-while 形式，这样可以方便插入一个 "loop guard". 之后再进行循环不变量外提.

```
if (0 < n) {  // loop guard
  auto v = *p;
  do {
    use(v);
    ++i;
  } while (i < n);
}
```

### 循环展开 (Loop Unroll)

循环包含循环体和各类分支语句，需要现代 CPU 进行一定的分支预测. 直接把循环展开，用一定的代码大小来换取运行时间.

```
for (int i = 0; i < 3; i++) {
  a[i] = i;
}
```

变换为:

```
a[0] = 0;
a[1] = 1;
a[2] = 2;
```

### 循环判断外提 (Loop Unswitching)

循环判断外提将循环中的条件式移到循环之外, 然后在外部的两个条件各放置两个循环, 这样可以增加循环向量化、并行化的可能性（通常简单循环更容易被向量化）.

```
// clang-format off
void before(int x) {
```

```
  for(;/* i in some range */;) {
    /* A */;
    if (/* condition */ x % 2) {
      /* B */;
    }
    /* C */;
  }
}

void after(int x) {
  if (/* condition */ x % 2) {
    for(;/* i in some range */;) {
      /* A */;
      /* B */; // 直接执行 B ，不进行循环判断
      /* C */;
    }
  } else {
    for(;/* i in some range */;) {
      /* A */;
                // 不执行 B
      /* C */;
    }
  }
}
```

### 代码布局优化 (Code Layout Optimizations)

程序在执行时，可以将执行的路径分为冷热路径 (cold/hot path)．CPU 跳转执行，绝大多数情况下没有直接顺序执行快，后者通常被编译器作者称为 "fallthrough"．与之对应的，经常被执行到的代码成为热代码，与之相对的成为冷代码．OI 代码中，如果有一段是循环中的特判边界条件，或者异常处理，类似的逻辑，则此部分代码为冷代码．

基本块 (Basic Block)，是控制流的基本结构，一个过程 (Procedure) 由若干个基本块组成，形成一个有向图．生成可执行文件的过程中，编译器需要安排一个放置基本块的布局 (Layout)，而如何编排布局，是此优化的重点．

原则上，应该更偏好与将热代码放在一起，而将冷代码隔开．原因是这样能够更好地利用指令缓存，热代码的局部性会更好．

```
// clang-format off
int hotpath; // <-- 热!
if (/* 边界条件 */ false) {
    // <-- 冷!
}
int hotpath_again;  // <-- 热!
```

#### 基本块放置 (Basic Block Placement)

我们用 label 来表达一种「伪机器码」，这个 C++ 程序有两种翻译方法：

**布局 1**

```
// clang-format off
hotblock1:
```

```
    Stmts; // <-- 热!
    if (/* 边界条件不成立 */ true)
        goto hotblock2; // 经常发生!  ------+
coldblock:                              /*   |   */
    Stmt; // <- 冷                            |
    Stmt; // <- 冷                            |
    Stmt; // <- 冷                            |  跨越了大量指令，代价高昂!
    Stmt; // <- 冷                            |
    Stmt; // <- 冷                            |
    Stmt; // <- 冷                            |
    Stmt; // <- 冷                            |
hotblock2:                              /*    |   */
    Stmts; // <- 热!            <----------+
```

另一种布局为:

**布局 2**

```
// clang-format off
hotblock1:
    Stmts; // <-- 热!
    if (/* 边界条件 */ false)
        goto coldblock; // 很少发生
hotblock2:                              /*   |  低代价!   */
    Stmts; // <- 热!   <-----------------+
coldblock:
    Stmt; // <- 冷
    Stmt; // <- 冷
    Stmt; // <- 冷
    Stmt; // <- 冷
    Stmt; // <- 冷
```

我们看到后一种布局中，两个热代码块被放到了一起，执行效率更优秀.

为了告诉编译器分支是否容易被执行，可以使用 C++20 `[[likely]]` 和 `[[unlikely]]`:https://en.cppreference.com/w/cpp/language/attributes/likely[3]

如果比赛没有采用 C++20 以上标准，则可以利用 `__builtin_expect`(GNU Extension).

```
#define likely(x) __builtin_expect(!!(x), 1)
#define unlikely(x) __builtin_expect(!!(x), 0)

if (unlikely(/* 一些边界条件检查 */ false)) {
  // 冷代码
}
```

**冷热代码分离 (Hot Cold Splitting)**

一个过程 (Procedure) 包含同时包含冷热路径，而冷代码较长，更好的做法是让冷代码作为函数调用，而不是阻断热路径. 这同时也提示我们不要自作聪明的让所有函数 `inline`. 冷代码对执行速度的阻碍比函数调用要多得多.

**不好的代码布局**

```
// clang-format off
void foo() {
      // clang-format off
hotblock1:
    Stmts; // <-- 热!
    if (/* 边界条件不成立 */ true)
        goto hotblock2; // 经常发生!  ------+
coldblock:                              /*   |   */
    Stmt; // <- 冷                           |
    Stmt; // <- 冷                           |
    Stmt; // <- 冷                           |  跨越了大量指令，代价高昂!
    Stmt; // <- 冷                           |
    Stmt; // <- 冷                           |
    Stmt; // <- 冷                           |
    Stmt; // <- 冷                           |
hotblock2:                              /*   |   */
    Stmts; // <- 热!            <----------+
}
```

**好的代码布局**

```
// clang-format off
void foo() {
hotblock1:
  Stmts;  // <-- 热!
  if (/* 边界条件 */ false)
    coldBlock();  // 将冷代码分离出，使得热路径对 cache 更友好
hotblock2:
  Stmts;  // <- 热!
}

void coldBlock() {
  Stmt;  // <- 冷
  Stmt;  // <- 冷
  Stmt;  // <- 冷
  Stmt;  // <- 冷
  Stmt;  // <- 冷
  Stmt;  // <- 冷
  Stmt;  // <- 冷
}
```

冷热代码分离，其实就是函数内联 (Function Inlining) 的反向操作，这一优化的存在启示我们，函数内联不一定会让程序跑的更快. 甚至如果内联代码是冷代码，反而会让程序跑的更慢! 一些编译器存在强制内联的编译选项，但不推荐使用. 编译器内部有一个静态分析过程，计算每个基本块、分支的概率，以及一个函数调用相关的代价模型，以此决定是否内联，自己决定是否内联不一定比编译器的决策好.

事实上，在没有额外信息的情况下，编译器通常会假设分支跳转与不跳转的概率一致，以此为依据传播各个控制流路径的冷热程度. PGO (Profile Guided Optimization) 的一部分便是通过若干次性能测试与实验得出真正环境下的程序分支概率，这些信息可以让代码布局更加优秀.

### 函数内联 (Function Inlining)

函数调用通常需要寄存器和栈传递参数，调用者 (caller) 和被调用者 (callee) 都需要保存一定的寄存器状态，这个过程通常被叫做调用约定 (calling convention)．一个函数调用因此会引起一些时间损耗，而内联函数就是指将函数直接写在调用方过程中，不进行真正的函数调用．

```
int add(int x) { return x + 1; }

int foo() {
  int a = 1;
  a = add(a);
}
```

`add()` 可以被内联到 `foo()` 当中：

```
int foo() {
  int a = 1;
  a = a + 1;  // <-- add() 的函数体，未经过传参
}
```

#### `always_inline`,`__force_inline`

https://clang.llvm.org/docs/AttributeReference.html#always-inline-force-inline[4]

一些编译器提供了手动内联函数调用的方法，在函数前加 `__attribute__((always_inline))`．这样使用不一定会比函数调用快，编译器在这个时候相信程序员有足够好的判断能力．

### 尾调用优化 (Tail Call Optimization)

当一个函数调用位于函数体尾部的位置时，这种函数调用被成为尾调用 (Tail Call)．对于这种特殊形式的调用，可以进行一些特别的优化．绝大多数体系结构拥有 Frame Pointer (a.k.a FP) 和 Stack Pointer (a.k.a SP)，维护者函数的调用帧 (Frame)，而如果调用位于函数尾部，则我们可以不保留外层函数的调用记录，直接用内层函数取代．

#### 用跳转指令代替函数调用

函数调用在绝大多数体系结构下，需要保存当前程序计数器 `$pc` 的位置，保存若干 caller saved register，以便回到现场．而尾调用不需要此过程，将被直接翻译为跳转指令，因为尾递归永远不会返回到函数运行的位置．

一个简单的例子：https://godbolt.org/z/e7b1safaW[5]

```
int test(int a);

int tailCall(int x) { return test(x); }
```

```
tailCall(int):                          ; @tailCall(int)
        jmp     test(int)@PLT                   ; TAILCALL
```

#### 自动尾递归改写

如果一个函数的尾调用是自身，则此函数是尾递归的．广义来讲，间接递归（由两个函数 以上共同形成递归）形成递归，且都是尾调用的，也属于尾递归的范畴．尾递归可以被编译器优化为非递归的形式，减小额外的栈开销和函数调用代价．许多算法竞赛选手热衷于写非递归的代码，在不开优化下这样可以极大优化代码的常数，然而如果开优化，递归代码生成的二进制质量和手写的代码没有什么区别．

```
int fac(int n) {
  if (n < 2) return 1;
  return /* 使用 */ n * fac(n - 1); /* 使用了变量 n ，无法直接做尾递归优化！*/
```

```
}
```

注意到这个函数并不是尾递归的，但可以改写为：

```
int fac(int acc, int n) {
  if (n < 2) return acc;
  return fac(acc * n, n - 1);
}
```

新的代码即是尾递归的.

现代编译器可以自动帮你完成这个过程，如果你的代码有机会被改写为尾递归，则编译器可以识别出这种形式，然后完成改写.

**尾递归消除 -Rpass=tailcallelim**

既然函数已经尾递归，那就可以直接删除递归语句，通过一定的静态分析，将函数直接转换为非递归的形式. 我们此处并不去深究编译器作者如何做到这一点，从实际体验来看，绝大多数 OI 代码，如果存在递归版本和非递归版本，则此代码一般可自动优化为非递归版本. 这里给读者一些具体的例子：

**GCD**[6]

```
int gcd(int a, int b) { return b ? gcd(b, a % b) : a; }
```

**斐波那契数列**[7]

```
// 展开 fib(n - 2) 这一项
// fib(n - 1) 不能变换为非递归，优化后的代码依然是指数级别的
int fib(int n) {
  if (n < 2) return 1;
  return fib(n - 1) + fib(n - 2);
}
```

**阶乘**[8]

```
// 展开成标量循环，然后执行自动向量化，生成的代码是 SIMD 的
unsigned fac(unsigned n) {
  if (n < 2) return 1;
  return n * fac(n - 1);
}
```

这些函数被优化后的汇编和非递归版完全相同，递归将被直接消除. 对于 OI 选手而言，可以在开 O2 的情况下放心写递归版本的各种算法，和非递归版不会有什么区别. 如果你写的函数本身无法被改写成非递归的形式，那么编译器也无能为力.

**强度削减 (Strength Reduction)**

常见的编译优化. 最简单的例子是 `x * 2` 变为 `x << 1`，第二种写法在 OI 中相当常见. 编译器会自动做类似的优化，在打开优化开关的情况下，`x * 2` 和 `x << 1` 是完全等价的. 强度削减 (Strength Reduction) 将高开销的指令转换为低开销的指令.

### 标量运算符变换

#### 移位代替乘法

```
int a;
a = x * 2;   // bad!
a = x << 1;  // good!
```

需要注意的是有符号数和无符号数在移位 (shifting) 和类型提升 (promotion) 层面有明显的差异. 符号位在移位时有着特别的处理, 包括算术移位和逻辑移位两种类型. 这在编写二分查找/线段树等含有大量除二操作的时候表现突出, 有符号整数除法不能直接优化为一步右移位运算.

```
int l, r;
/* codes */
int mid = (l + r) / 2; /* 如果编译器不能假定 l, r 非负, 则会生成较差的代码 */
                       // 不能优化为
                       // mid = (l + r) >> 1
                       // 反例:
                       // mid = -127
                       // mid / 2 = -63
                       // mid >> 1 = -64
```

```
int mid = (l + r);
int sign = mid >> 31; /* 逻辑右移, 得到符号位 */
mid += sign;
mid >>= 1; /* 算术右移 */
```

可行的解决方案:

- 用 `unsigned l, r;`, 下标本来就应该是无符号的
- 在源代码中使用移位

#### 乘法代替除法

```
int x = a / 3;
```

此过程可以被变换为 `x = a * 0x55555556 >> 32`, 具体可以看 这篇知乎回答[9] 或者 原始论文[10].

#### 索引变量强度削减 (IndVars)

编译器自动识别出循环中的索引变量, 并将相关的高开销过程转换为低开销

```
int a = 0;
for (int i = 1; i < 10; i++) {
  a = 3 * i;  // bad!
  a = a + 3;  // good!
}
```

此处如果直接使用 `a = 3 * i` 在 OI 中很常见, 而编译器可以自动分析出, 等价的变换为 `a = a + 3`, 用代价更低的加法代替乘法. 分析循环变量的迭代过程, 被称为 SCEV (Scalar Evolution).

SCEV 还可以做到优化一些循环:

```
int test(int n) {
  int ans = 1;
  for (int i = 0; i < n; i++) {
    ans += i * (i + 1);
  }
  return ans;
}
```

此函数会被优化为 $O(1)$ 公式求和，参考 https://godbolt.org/z/ET8d89vvK[11]. 这个行为目前仅有基于 LLVM 的编译器会出现，GCC 编译器更加保守.

```
test(int):                              # @test(int)
        test    edi, edi
        jle     .LBB0_1
        lea     eax, [rdi - 1]
        lea     ecx, [rdi - 2]
        imul    rcx, rax
        lea     eax, [rdi - 3]
        imul    rax, rcx
        shr     rax
        imul    eax, eax, 1431655766
        and     ecx, -2
        lea     eax, [rax + 2*rcx]
        lea     eax, [rax + 2*rdi]
        dec     eax
        ret
.LBB0_1:
        mov     eax, 1
        ret
```

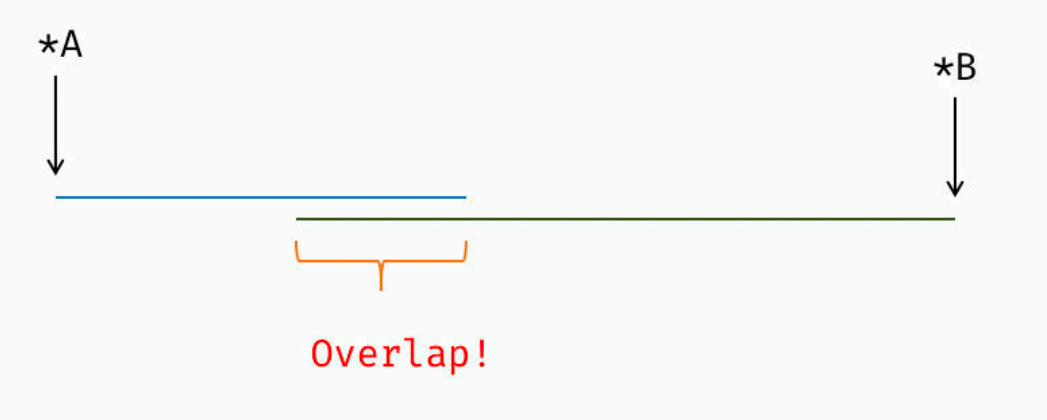
### 自动向量化 (Auto-Vectorization)

单指令流多数据流是很好的提供单核并行的方法. 使用这种指令，可以利用 CPU 的 SIMD 寄存器，比通用寄存器更宽，例如一次放 4 个整数然后计算. OI 选手不需要了解自动向量化的细节，通常而言，Clang 编译器会做比 GCC 更激进的自动向量化：

```
// https://godbolt.org/z/h1hx5sWoE
void test(int *a, int *b, int n) {
  for (int i = 0; i < n; i++) {
    a[i] += b[i];
  }
}
```

#### `__restrict` type specifier (GNU, MSVC)

两个任意指针对应的区域可能出现重叠 (overlap)，此时需要特判是否可以使用向量代码. 下图展示了一个指针重叠的例子：




`__restrict` 作为一种约定使编译器假定两个指针所指向的内存区域永远不会重叠.

```
void test(int* __restrict a, int* __restrict b, int n) {
```

```
  for (int i = 0; i < n; i++) {
    a[i] += b[i];
  }
}
```

`__restrict` 并非 C++ 标准的一部分，但各大编译器都可以使用．此关键字影响自动向量化的代码生成质量，极端卡常的情况下可以使用．

## 和编译优化相关的常见语言误用

### inline - 内联

函数内联在开 O2 的情况下通常由编译器自动完成．结构体定义中的 `inline` 完全是多余的，如果准备的比赛开 O2 优化，则完全不必声明为内联．如果不开 O2 则使用 `inline` 也不会让编译器真正内联．

`inline` 关键字在现代 C++ 被当作是一种链接、与导出符号的语义行为，而不是做函数内联．

### register - 虚假的寄存器建议

现代编译器会直接忽略你的 `register` 关键字，你自己认为的寄存器分配一般没有编译器直接跑寄存器分配算法来的聪明．此关键字于 C++11 被弃用，于 C++17 被删除 [6]．

https://en.cppreference.com/w/cpp/keyword/register[13]

## 未定义行为（Undefined Behavior）与编译优化

编译器可以认为 C++ 程序不存在 未定义行为[14]（undefined behavior，UB），因此在编译存在 UB 的程序时，编译器可能会产生意想不到的结果．同时，编译器也可以在假定不存在 UB 的情况下进行更加激进而自由的优化．

常见的 UB 有：

1. 有符号溢出[15]；
2. 使用未初始化的变量；
3. 访问越界；
4. 空指针解引用；
5. 无副作用的无限循环．

其他 UB 和示例等可通过扩展阅读详细了解．

### 有符号溢出

```
int f(int x) { return x * 2 / 2; }
```

编译器可以假定程序不存在有符号溢出的行为，进而此函数可能被优化为

```
int f(int x) { return x; }
```

示例：https://godbolt.org/z/WKv3W5hvM[16]、https://godbolt.org/z/qqE9nxP1j[17]．

可通过 `-fwrapv`[18] 选项禁用该假设．示例：https://godbolt.org/z/5x3K5KGnr[19]、https://godbolt.org/z/4r4a4EzMW[20]．

### 使用未初始化的变量

```
int f(int x) {
  int a;
  if (x)  // either x nonzero or UB
```

[6] Remove Deprecated Use of the register Keyword (open-std.org)[12]

```
    a = 42;
  return a;
}
```

编译器可以假定程序不存在使用未初始化变量的行为，所以 `a` 一定会被初始化，进而此函数可能被优化为

```
int f(int) { return 42; }
```

示例：https://godbolt.org/z/8WYMYYjdG[21]、https://godbolt.org/z/qvGd1nvv9[22].

### 访问越界

```
int table[4] = {};

bool exists_in_table(int v) {
  // return true in one of the first 4 iterations or UB due to out-of-bounds
  // access
  for (int i = 0; i <= 4; i++)
    if (table[i] == v) return true;
  return false;
}
```

编译器可以假定程序不存在访问越界的行为，所以该函数一定会在发生访问越界之前返回，进而此函数可能被优化为

```
bool exists_in_table(int) { return true; }
```

示例：https://godbolt.org/z/xfePeYsE3[23].

### 空指针解引用

```
int f(int* p) {
  int x = *p;
  if (!p)
    return x;  // Either UB above or this branch is never taken
  else
    return 0;
}
```

编译器可以假定程序不存在空指针解引用的行为，从而 `!p` 恒为 `false`，进而此函数可能被优化为

```
int f(int*) { return 0; }
```

示例：https://godbolt.org/z/GY1jvsrb5[24]、https://godbolt.org/z/4ronPsnxf[25].

### 无副作用的无限循环

**验证 Fermat 大定理**

由 Fermat 大定理[26] 可知，不定方程 $a^3 = b^3 + c^3$ 没有正整数解. 下面的程序试图枚举 $[1, 1000]$ 内的整数验证该方程是否成立，若返回 `true` 则说明在 $[1, 1000]$ 范围内找到了一组整数解，从而 Fermat 大定理不成立.

```
#include <iostream>

bool fermat() {
```

```
  const int max_value = 1000;

  // Endless loop with no side effects is UB
  for (int a = 1, b = 1, c = 1; true;) {
    if (((a * a * a) == ((b * b * b) + (c * c * c))))
      return true;  // disproved :()
    a++;
    if (a > max_value) {
      a = 1;
      b++;
    }
    if (b > max_value) {
      b = 1;
      c++;
    }
    if (c > max_value) c = 1;
  }

  return false;  // not disproved
}

int main() {
  std::cout << "Fermat's Last Theorem ";
  fermat() ? std::cout << "has been disproved!\n"
           : std::cout << "has not been disproved.\n";
}
```

编译器可以假定程序不存在无副作用的无限循环，从而认为 `fermat()` 函数中的 for 循环一定会在某一时刻终止并返回 `true`，最终程序可能输出：

```
Fermat's Last Theorem has been disproved!
```

示例：https://godbolt.org/z/d834MK7bz[27]、https://godbolt.org/z/Eov9nsKqf[28].

## Sanitizer

理智保证器. 在运行时检查你的程序是否有未定义行为、数组越界、空指针，等等功能. 在本地调试模式下，建议开启一些 sanitizer，可以极大缩短你的 Debug 时间. 这些 sanitizer 由 Google 开发，绝大多数可以在 GCC 和 Clang 中使用. sanitizer 在 LLVM 中更加成熟，因此推荐选手本地使用 Clang 编译器进行相关除错.

### Address Sanitizer -fsanitize=address

https://clang.llvm.org/docs/AddressSanitizer.html[29]

GCC 和 Clang 都支持这个 Sanitizer. 包括如下检查项：

- 越界
- 释放后使用 (use-after-free)
- 返回后使用 (use-after-return)
- 重复释放 (double-free)
- 内存泄漏 (memory-leaks)
- 离开作用域后使用 (use-after-scope)

应用这项检查会让你的程序慢 2x 左右.

### Undefined Behavior Sanitizer -fsanitize=undefined

https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html[30]

Undefined Behavior Sanitizer (a.k.a UBSan) 用于检查代码中的未定义行为. GCC 和 Clang 都支持这个 Sanitizer. 自动检查你的程序有无未定义行为. UBSan 的检查项目包括:

- 移位溢出, 例如 32 位整数左移 72 位
- 有符号整数溢出
- 浮点数转换到整数数据溢出

UBSan 的检查项可选, 对程序的影响参考提供的网页地址.

## 杂项

### Compiler Explorer

在这里观察各个编译器的行为和汇编代码: https://godbolt.org[31]

## 扩展阅读

1. The LLVM Project Blog: What Every C Programmer Should Know About Undefined Behavior #1/3[32]
2. The LLVM Project Blog: What Every C Programmer Should Know About Undefined Behavior #2/3[33]
3. The LLVM Project Blog: What Every C Programmer Should Know About Undefined Behavior #3/3[34]

## 参考资料与注释

[1] 如同规则

[2] https://godbolt.org/z/oEfY35TTd

[3] https://en.cppreference.com/w/cpp/language/attributes/likely

[4] https://clang.llvm.org/docs/AttributeReference.html#always-inline-force-inline

[5] https://godbolt.org/z/e7b1safaW

[6] GCD

[7] 斐波那契数列

[8] 阶乘

[9] 这篇知乎回答

[10] 原始论文

[11] https://godbolt.org/z/ET8d89vvK

[12] Remove Deprecated Use of the register Keyword (open-std.org)

[13] https://en.cppreference.com/w/cpp/keyword/register

[14] 未定义行为

[15] 有符号溢出

[16] https://godbolt.org/z/WKv3W5hvM

[17] https://godbolt.org/z/qqE9nxP1j

[18] `-fwrapv`

[19] https://godbolt.org/z/5x3K5KGnr

[20] https://godbolt.org/z/4r4a4EzMW

[21] https://godbolt.org/z/8WYMYYjdG

[22] https://godbolt.org/z/qvGd1nvv9

[23] https://godbolt.org/z/xfePeYsE3

[24] https://godbolt.org/z/GY1jvsrb5

[25] https://godbolt.org/z/4ronPsnxf

[26] Fermat 大定理

[27] https://godbolt.org/z/d834MK7bz

[28] https://godbolt.org/z/Eov9nsKqf

[29] https://clang.llvm.org/docs/AddressSanitizer.html

[30] https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[31] https://godbolt.org

[32] The LLVM Project Blog: What Every C Programmer Should Know About Undefined Behavior #1/3

[33] The LLVM Project Blog: What Every C Programmer Should Know About Undefined Behavior #2/3

[34] The LLVM Project Blog: What Every C Programmer Should Know About Undefined Behavior #3/3

# 4.5 C++ 与其他常用语言的区别

本文介绍 C++ 与其他常用语言的区别，重点介绍 C 与 C++ 之间重要的或者容易忽略的区别．尽管 C++ 几乎是 C 的超集，C/C++ 代码混用一般也没什么问题，但是了解 C/C++ 间比较重要的区别可以避免碰到一些奇怪的 bug．如果你是以 C 为主力语言的 OIer，那么本文也能让你更顺利地上手 C++．C++ 相比 C 增加的独特特性可以阅读 C++ 进阶→4.4.1@p407 部分的教程．此外，本文也简要介绍了 Python, Java 和 C++ 的区别．

## C 与 C++ 的区别

### 宏与模板

C++ 的模板在设计之初的一个用途就是用来替换宏定义．学会模板编程是从 C 迈向 C++ 的重要一步．模板不同于宏的文字替换，在编译时会得到更全面的编译器检查，便于编写更健全的代码．模板特性在 C++11 后支持了可变长度的模板参数表，可以用来替代 C 中的可变长度函数并保证类型安全．

### 指针与引用

C++ 中你仍然可以使用 C 风格的指针，但是对于变量传递而言，更推荐使用 C++ 的 引用→4.4.5@p424 特性来实现类似的功能．由于引用指向的对象不能为空，因此可以避免一些空地址访问的问题．不过指针由于其灵活性，也仍然有其用武之地．值得一提的是，C 中的 `NULL` 空指针在 C++11 起有类型安全的替代品 `nullptr`．引用和指针之间可以通过 `*` 和 `&` 运算符→4.2.4@p339 相互转换．

### bool

另请参阅 布尔类型→4.2.3@p328．

与 C++ 不同的是，C 语言最初并没有布尔类型．

C99 标准加入了 `_Bool` 关键字（以及等效的 `bool` 宏）以及 `true` 和 `false` 两个宏．如果需要使用 `bool`, `true`, `false` 这三个宏，需要在程序中引入 `stdbool.h` 头文件．而使用 `_Bool` 则不需要引入任何额外头文件．

```c
bool x = true;  // 需要引入 stdbool.h
_Bool x = 1;    // 不需要引入 stdbool.h
```

C23 起，`true`,`false` 和 `bool` 成为 C 语言中的关键字，使用它们不需要再引入 `stdbool.h` 头文件，同时保留 `_Bool` 作为 `bool` 的替代拼写形式[1]．

下表展示了 C 语言不同标准下，bool 类型支持的变化情况（作为对照，加入了 C++ 的支持情况）：

| 语言标准 | `bool` | `true`/`false` | `_Bool` |
|---|---|---|---|
| C89 | / | / | 保留[2] |
| C99 起，C23 以前 | 宏，与 `_Bool` 等价，需要 `stdbool.h` 头文件 | 宏，`true` 与 `1` 等价，`false` 与 `0` 等价，需要 `stdbool.h` 头文件 | 关键字 |
| C23 起 | 关键字 | 关键字 | 关键字 `bool` 的替代拼写形式 |
| C++ | 关键字 | 关键字 | 保留[3] |

[1] https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3054.pdf[1].

[2] C 和 C++ 均规定，以一个下划线跟着一个大写字母开头的标识符是被保留的，详见 https://en.cppreference.com/w/c/language/identifier[2].

[3] C 和 C++ 均规定，以一个下划线跟着一个大写字母开头的标识符是被保留的，详见 https://en.cppreference.com/w/c/language/identifier[3].

### struct

尽管在 C 和 C++ 中都有 struct 的概念，但是他们对应的东西是不能混用的！C 中的 struct 用来描述一种固定的内存组织结构，而 C++ 中的 struct 就是一种类，**它与类唯一的区别就是它的成员和继承行为默认是 public 的**，而一般类的默认成员是 private 的．这一点在写 C/C++ 混合代码时尤其致命．

另外，声明 struct 时 C++ 也不需要像 C 那么繁琐，C 版本：

```
typedef struct Node_t {
  struct Node_t *next;
  int key;
} Node;
```

C++ 版本

```
struct Node {
  Node *next;
  int key;
};
```

### const

const 在 C 中只有限定变量不能修改的功能，而在 C++ 中，由于大量新特性的出现，const 也被赋予的更多用法．C 中的 const 在 C++ 中的继任者是 constexpr，而 C++ 中的 const 的用法请参见 常值→4.4.6@p427 页面的说明．

## 内存分配

C++ 中新增了 `new` 和 `delete` 关键字用来在「自由存储区」上分配空间，这个自由存储区可以是堆也可以是静态存储区，他们是为了配合「类」而出现的．其中 `delete[]` 还能够直接释放动态数组的内存，非常方便．`new` 和 `delete` 关键字会调用类型的构造函数和析构函数，相比 C 中的 `malloc()`、`realloc()`、`free()` 函数，他们对类型有更完善的支持，但是效率不如 C 中的这些函数．

简而言之，如果你需要动态分配内存的对象是基础类型或他们的数组，那么你可以使用 `malloc()` 进行更高效的内存分配；但如果你新建的对象是非基础的类型，那么建议使用 `new` 以获得安全性检查．值得注意的是尽管 `new` 和 `malloc()` 都是返回指针，但是 `new` 出来的指针 **只能** 用 `delete` 回收，而 `malloc()` 出来的指针也只能用 `free()` 回收，否则会有内存泄漏的风险．

## 变量声明

C99 前，C 的变量声明必须位于语句块开头，C++ 和 C99 后无此限制．

## 可变长数组

C99 后 C 语言支持 VLA（可变长数组），C++ 始终不支持．

## 结构体初始化

C99 后 C 语言支持结构体的 指派符初始化[4]（但是在 C11 中为可选特性），C++ 直到 C++20 才支持有顺序要求的指派符初始化，且 C 语言支持的乱序、嵌套、与普通初始化器混用、数组的指派符初始化特性 C++ 都不支持[4]．

## 注释语法

C++ 风格单行注释 `//`，C 于 C99 前不支持．

[4] https://en.cppreference.com/w/cpp/language/aggregate_initialization[5]

## Python 与 C++ 的区别

Python 是目前机器学习界最常用的语言. 相比于 C++, Python 的优势在于易于学习, 易于实践. Python 有着更加简单直接的语法, 比如在定义变量时, 不需要提前声明变量类型. 但是, 这样的简单也是有代价的. Python 相比于 C++ 牺牲了性能. C++ 几乎适用于包括嵌入式系统的所有平台, 并且有着更快的执行速度, 但是 Python 只可以在某些支持高级语言的平台上使用. C++ 更接近底层, 所以可以用来进行编写操作系统.

## Java 与 C++ 的区别

Java 与 C++ 都是面向对象的语言, 都使用了面向对象的思想 (封装、继承、多态), 由于面向对象有许多非常好的特性 (继承、组合等), 因此二者有很好的可重用性. 所以相比于 Python, Java 和 C++ 更加类似.

二者最大的区别在于 Java 有 JVM 的机制. JVM 全称是 Java Virtual Machine, 中文意为 Java 虚拟机. Java 语言的一个非常重要的特点就是与平台的无关性. 而使用 Java 虚拟机是实现这一特点的关键. 一般的高级语言如果要在不同的平台上运行, 至少需要编译成不同的目标代码. 而引入 Java 语言虚拟机后, Java 语言在不同平台上运行时不需要重新编译. Java 语言使用 Java 虚拟机屏蔽了与具体平台相关的信息, 使得 Java 语言编译程序只需生成在 Java 虚拟机上运行的目标代码 (字节码), 就可以在多种平台上不加修改地运行.

因为这个特点, Java 经常被用于需要移植到不同平台程序的开发. 但是也由于编译 Java 程序时需要从字节码开始, 所以 Java 的性能没有 C++ 好.

## 参考资料

[1] https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3054.pdf

[2] https://en.cppreference.com/w/c/language/identifier

[3] https://en.cppreference.com/w/c/language/identifier

[4] 指派符初始化

[5] https://en.cppreference.com/w/cpp/language/aggregate_initialization

# 4.6 Pascal 转 C++ 急救

**Authors:** kexplorning, Ir1d, lvneg1

## C++ 快速安装与环境配置

以下过程均在 Windows 系统中操作.

### 使用 IDE

可以参考如下页面中内容:

- Dev-C++→3.2.8@p171
- Code::Blocks→3.2.15@p219

### 使用 代码编辑器 + 编译器 + 调试器

可以参考 VS Code→3.2.3@p147 页面中内容. Visual Studio Code 官方网站上有文档解释如何进行 C++ 的配置. 一般而言 VS Code 搭配插件使用更方便, 见 VS Code 的官方网站[1].

## C++ 语法快速提要

C++ 程序都是从 `main` 这个部分开始运行的.

大括号表示块语句的开始与结束: `{` 就相当于 Pascal 里面的 `begin`, 而 `}` 就相当于 `end`.

注意，和 Pascal 一样，C++ 每句话结束要加分号 `;`，不过大括号结尾不需要有分号，而且程序结束末尾不用打句号 `.`.

对于注释，`//` 表示行内注释，`/* */` 表示块注释.

按照惯例，看看 Hello World 吧.

## Hello World：第一个 C++ 程序

```cpp
#include <iostream>  // 导入 iostream 库

int main()  // main 部分
{
  std::cout << "Hello World!" << std::endl;

  return 0;
}
```

然后编译运行一下，看看结果.

### 简要解释

第一行，`#include <iostream>` 的意思是，导入 `iostream` 这个库.

> **Pascal 的库文件**
>
> Pascal 其实是有库文件的，只不过，很多同学从来都没有用过……

看到第三行的 `main` 吗？程序从 `main` 开始执行.

接下来最重要的一句话是

```cpp
std::cout << "Hello World!" << std::endl;
```

`std::cout` 是输出命令. 你可能看过有些 C++ 程序中直接写的是 `cout`.

> **有关 std:: 前缀**
>
> 有关 `std::` 这个前缀的问题，请见 这节→4.2.2@p323 底下的注释「什么是 std?」.

中间的 `<<` 很形象地表示流动，其实它就是表示输出怎么「流动」的. 这句代码的意思就是，`"Hello World!"` 会先被推到输出流，之后 `std::endl` 再被推到输出流.

而 `std::endl` 是 **输出** 换行命令，这与 Pascal 的 `writeln` 类似，不过 C++ 里面可没有 `coutln`. Pascal 与 C++ 的区别在于，`write('Hello World!')` 等价于 `std::cout << "Hello World!"`，而 `writeln('Hello World!')` 等价于 `std::cout << "Hello World!" << std::endl`.

此处 `"Hello World!"` 是字符串，Pascal 中字符串都是用单引号 `'` 不能用双引号，而 C++ 的字符串必须用双引号. C++ 中单引号包围的字符会有别的含义，后面会再提及的.

好了，到这里 Hello World 应该解释的差不多了.

可能有同学会问，后面那个 `return 0` 是什么意思？那个 `int main()` 是啥意思？**先别管它**，一开始写程序的时候先把它当作模板来写吧（这里也是用模板写的）. 因为入门时并不会用到 `main` 中参数，所以不需要写成 `int main(int argc, char const *argv[])`.

### 简单练习

1. 试着换个字符串输出.
2. 试着了解转义字符.

### A+B Problem：第二个 C++ 程序

经典的 A+B Problem.

```cpp
#include <iostream>

int main() {
  int a, b, c;

  std::cin >> a >> b;

  c = a + b;

  std::cout << c << std::endl;

  return 0;
}
```

注：代码空行较多，若不习惯可去掉空行.

#### 简要解释

`std::cin` 是读入（`cin` 即 C-in），`>>` 也与输出语法的类似.

这里多出来的语句中最重要的是两个，一个是变量声明语句.

```cpp
int a, b, c;
```

你可能习惯于 Pascal 里面的声明变量：

```pascal
var
a, b, c: integer;
```

C++ 的声明是直接以数据类型名开头的，在这里，`int`（整型）开头表示接下来要声明变量.

接着一个最重要的语句就是赋值语句.

```cpp
c = a + b;
```

这是 Pascal 与 C++ 语法较大的不同：Pascal 的赋值是 `:=`，C++ 是 `=`；而 C++ 判断相等是 `==`.

C++ 也可直接在声明时进行变量初始化赋值.

```cpp
int a = 0, b = 0, c = 0;
```

#### 简单练习

1. 重写一遍代码，提交到 OJ 上，并且 AC.
2. 更多的输入输出语法参考 这节内容→4.2.2@p323，并试着了解 C++ 的格式化输出.

### 结束语与下一步

好了，到现在为止，你已经掌握了一些最基本的东西了，剩下就是找 Pascal 和 C++ 里面对应的语法和不同的特征.

不过在此之前，强烈建议先看 变量作用域：全局变量与局部变量→4@p321.

请善用 `Alt` + `←` 与 `Alt` + `→` 返回跳转.

# 语法

## 变量

### 基本数据类型

C++ 与 Pascal 基本上差不多，常见的有

- `bool`：布尔类型
- `int`：整型
- `float`：单精度浮点型
- `double`：双精度浮点型
- `char`：字符型
- `void`：无类型

C++ 的单引号是专门用于表示单个字符的（字符型），比如 `'a'`，而字符串（字符型数组）必须要用双引号.

C++ 还要很多额外的数据类型，请参考 基础类型 - cppreference.com[2].

### 常量声明

```
1 const double PI = 3.1415926;
```

若不清楚有关宏展开的问题，建议使用常量，而不用宏定义.

## 运算符

请直接参考 运算→4.2.4@p339 一文中内容. 附录中也提供了运算符与数学函数语法对比表.

## 条件

### `if` 语句

```
1 if (a = b) and (a > 0) and (b > 0) then
2     begin
3         b := a;
4     end
5 else
6     begin
7         a := b;
8     end;
```

```
1 if (a == b && a > 0 && b > 0) {
2   b = a;
3 } else {
4   a = b;
5 }
```

布尔运算与比较

- `and -> &&`
- `or -> ||`
- `not -> !`
- `= -> ==`
- `<> -> !=`

注释：

1. Pascal 中 `and` 与 C++ 中 `&&` 优先级不同，C++ 不需要给判断条件加括号.

2. Pascal 中判断相等是 `=`，赋值是 `:=`；C++ 中判断相等是 `==`，赋值是 `=`.
3. 如果在 `if` 语句的括号内写了 `a = b` 而不是 `a == b`，程序不会报错，而是会把 `b` 赋值给 `a`，并使赋值表达式 `a = b` 整体具有 `a` 在赋值操作完成后的值.
4. C++ 不需要思考到底要不要在 `end` 后面加分号.
5. C++ 布尔运算中，非布尔值可以自动转化为布尔值.

> ⚠ **易错提醒**
>
> 特别注意：**不要把 `==` 写成 `=`!**
>
> 由于 C/C++ 比 Pascal 语法灵活，如果在判断语句中写了 `if (a=b)`，那么程序会顺利运行下去，因为 C++ 中 `a=b` 是有返回值的.

### `case` 与 `switch`

用到得不多，此处不详细展开.

需要注意：C++ 没有 `1..n`，也没有连续不等式（比如 `1 < x < 2`）.

## 循环

以下三种循环、六份代码实现的功能是一样的.

### `while` 循环

`while` 很相似.（C++ 此处并非完整程序，省略一些框架模板，后同）

```
var i: integer;

begin
    i := 1;
    while i <= 10 do
        begin
            write(i,' ');
            inc(i); // 或者 i := i + 1;
        end;
end.
```

```
int i = 1;
while (i <= 10) {
  std::cout << i << " ";
  i++;
}
```

### `for` 循环

C++ 的 `for` 语句非常不同.

```
var i: integer;

begin
    for i:= 1 to 10 do
        begin
            write(i, ' ');
        end;
end.
```

```cpp
for (int i = 1; i <= 10; i++) {
  std::cout << i << " ";
}
```

注释：

1. `for (int i = 1; i <= 10; i++){` 这一行语句很多，`for` 中有三个语句.
2. 第一个语句 `int i = 1;` 此时声明一局部变量 `i` 并初始化.（这个设计比 Pascal 要合理得多.）
3. 第二个语句 `i <= 10;` 作为判断循环是否继续的标准.
4. 第三个语句 `i++`，在每次循环结尾执行，意思大约就是 Pascal 中的 `inc(i)`，此处写成 `++i` 也是一样的. `i++` 与 `++i` 的区别请参考其他资料.

### `repeat until` 与 `do while` 循环

注意，`repeat until` 与 `do while` 是不同的，请对比以下代码

```pascal
var i: integer;

begin
    i := 1;
    repeat
        write(i, ' ');
        inc(i);
    until i = 11;
end.
```

```cpp
int i = 1;
do {
  std::cout << i << " ";
  i++;
} while (i <= 10);
```

### 循环控制

C++ 中 `break` 的作用与 Pascal 是一样的，退出循环.

而 `continue` 也是一样的，跳过当前循环，进入下一次循环（回到开头）.

## 数组与字符串

### 不定长数组：标准库类型 Vector

请参考 序列式容器→4.3.2.3@p373 页面中内容.

C++ 标准库中提供了 `vector`，相当于不定长数组，调用前需导入库文件.

```cpp
#include <iostream>
#include <vector>  // 导入 vector 库

int main() {
  std::vector<int> a;  // 声明 vector a 并定义 a 为空 vector 对象
  int n;

  std::cin >> n;
  // 读取 a
  for (int i = 0; i < n; i++) {
    int t;
    std::cin >> t;
```

```
    a.push_back(t);  // 将读入的数字 t，放到 vector a 的末尾；该操作复杂度 O(1)
    /* 这里不能使用下标访问来赋值，因为声明时，a 大小依然为空，
    此处使用 `a[i] = t;` 是错误做法.
    */
  }

  // 将读入到 a 中的所有数打印出
  for (int i = 0; i < n; i++) {
    std::cout << a[i] << ", ";  // !注意，a 中第一个数是 a[0];
    // 如果下标越界，它会返回一个未知的值（溢出），而不会报错
  }
  std::cout << std::endl;

  return 0;
}
```

C++ 访问数组成员，与 Pascal 类似，不过有很重要的区别：数组的第一项是 `a[0]`，而 Pascal 中是可以自行指定的.

### 字符串：标准库类型 String

请参考 string→4.3.5@p402 页面中内容.

C++ 标准库中提供了 `string`，与 `vector` 可以进行的操作有些相同，同样需要导入库文件.

```
#include <iostream>
#include <string>

int main() {
  std::string s;  // 声明 string s

  std::cin >> s;  // 读入 s;
  // 读入时会忽略开头所有空格符（空格、换行符、制表符），读入的字串直到下一个空格符为止.

  std::cout << s << std::endl;

  return 0;
}
```

### C 风格数组

请参考 数组→4.2.6.1@p352 页面中内容.

如果要用不定长的数组请用 `vector`，不要用 C 风格的数组.

C 风格的数组与指针有密切关系，所以此处不多展开.

## 重要不同之处

### 变量作用域：全局变量与局部变量

C++ 几乎可以在 **任何地方** 声明变量. 请参考 变量作用域→4.2.3@p328.

在写 Pascal 过程/函数时，容易忘记声明局部变量 `i` 或者 `j`，而一般主程序里会有循环，于是大部分情况下 `i` 与 `j` 都是全局变量，于是，在这种情况下，过程/函数中对 `i` 操作极易出错. 更要命的是，如果忘记声明这种局部变量，编译器编译不报错，程序可以运行.（有很多难找的 bug 就是这么来的.）

所以，在使用 C++ 时，声明变量，比如循环中使用的 `i`，**不要用全局变量，能用局部变量就用局部变量**. 如果这么做，不用担心函数中变量名（比如 `i`）冲突.

**额外注**

Pascal 可在某种程度上避免这个问题，仿照 C++ 的方法，主程序只有调用过程/函数，不声明 `i`, `j` 这类极易名称冲突的全局变量，如果需要循环，另写一个过程进行调用.

## C++ 可以自动转换类型

```cpp
int i = 2;
if (i) {  // i = 0 会返回 false，其余返回 true
  std::cout << "true";
} else {
  std::cout << "false";
}
```

不光是 `int` 转成 `bool`，还有 `int` 与 `float` 相互转换. 在 Pascal 中可以把整型赋给浮点型，但不能反过来. C++ 没有这个问题.

```cpp
int a;
a = 3.2;      // 此时 a = 3
float b = a;  // 此时 b = 3.0
```

区分 `/` 是整除还是浮点除法，是通过除数与被除数的类型判断的

```cpp
float a = 32 / 10;    // 32/10 的结果是 3（整除）；a = 3.0
float b = 32.0 / 10;  // 32.0/10 的结果是 3.2；b = 3.2
```

`pow(a, b)` 计算 $a^b$，该函数返回的是浮点型，如果直接用来计算整数的幂，由于有自动转换，不需要担心它会报错

```cpp
int a = pow(2, 3);  // 计算 2^3
```

还有 `char` 与 `int` 之间相互转换.

```cpp
char a = 48;              // ASCII 48 是 '0'
int b = a + 1;            // b = 49
std::cout << (a == '0');  // true 输出 1
```

其实 C++ 中的 `char` 与 `bool` 本质上是整型.

详细内容请参考 隐式转换 - cppreference.com[3] 一文.

## C++ 很多语句有返回值：以如何实现读取数量不定数据为例

有些时候需要读取到数据结束，比如，求一组不定数量的数之和（数据可以多行），直到文件末尾，实现方式是

**文件末尾 EOF**

EOF，文件末尾标识符，在命令行中 Windows 上以 `Ctrl` + `Z` 输入（还需按 `Enter`），Unix-like 系统以 `Ctrl` + `D` 输入.

```cpp
#include <iostream>

int main() {
  int sum = 0, a = 0;

  while (std::cin >> a) {
```

```
    sum += a;
  }
  std::cout << sum << std::endl;

  return 0;
}
```

实现原理：`while (std::cin >> a)` 中 `std::cin >> a` 若在输入有问题或遇到文件结尾时，会返回 `false`，使得循环中断.

## 函数

C++ 只有函数没有过程但有 `void`，没有函数值变量但有 `return`.

Pascal 函数与 C++ 函数对比示例：

```
function abs(x:integer):integer;
begin
    if x < 0 then
        begin
            abs := -x;
        end
    else
        begin
            abs := x;
        end;
end;
```

```
int abs(int x) {
  if (x < 0) {
    return -x;
  } else {
    return x;
  }
}
```

C++ 中函数声明 `int abs`, 就定义了 `abs()` 函数且返回值为 `int` 型（整型）, 函数的返回值就是 `return` 语句给出的值.

如果不想有返回值（即 Pascal 的「过程」），就用 `void`. `void` 即「空」，什么都不返回.

```
var ans: integer;

procedure printAns(ans:integer);
begin
    writeln(ans);
end;

begin
    ans := 10;
    printAns(ans);
end.
```

```
#include <iostream>

void printAns(int ans) {
```

```
  std::cout << ans << std::endl;

  return;
}

int main() {
  int ans = 10;
  printAns(ans);

  return 0;
}
```

C++ 的 `return` 与 Pascal 中给函数变量赋值有一点非常大的不同．C++ 的 `return` 即返回一个值，执行完这个语句，函数就执行结束了；而 Pascal 中给函数变量赋值并不会跳出函数本身，而是继续执行．于是，如果 Pascal 需要某处中断函数/过程，就需要一个额外的命令，即 `exit`．而 C++ 则不需要，如果需要在某处中断，可以直接使用 `return`．比如（由于实在想不出来简短且实用的代码，所以就先这样）

```
#include <iostream>

void printWarning(int x) {
  if (x >= 0) {
    return;  // 该语句在此处相当于 Pascal 中的 `exit;`
  }
  std::cout << "Warning: input a negative number.";
}

int main() {
  int a;

  std::cin >> a;
  printWarning(a);

  return 0;
}
```

而在某种意义上，前面的 `abs` 函数，这样才是严格等效的

```
function abs(x:integer):integer;
begin
    if x < 0 then
        begin
            abs := -x; exit; // !注意此处
        end
    else
        begin
            abs := x;  exit; // !注意此处
        end;
end;
```

```
int abs(int x) {
  if (x < 0) {
    return -x;
  } else {
    return x;
  }
```

```
}
```

> **特别提醒**
>
> C++ 中 `exit` 是退出程序；不要顺手把 `exit` 打上去，要用 `return`！

C++ 把函数和过程统统视作函数，连 `main` 都不放过，比如写 `int main`，C++ 视 `main` 为一个整型的函数，这里返回值是 `0`. 它是一种习惯约定，返回 `0` 代表程序正常退出.

也许你已经猜到了，`main(int argc, char const *argv[])` 中的参数就是 `int argc` 与 `char const *argv[]`，不过意义请参考其他资料.

## 在函数中传递参数

C++ 中没有 Pascal 的 `var` 关键字可以改变传递的参数，但是 C++ 可以使用引用和指针达到同样的效果.

```
var a, b: integer;

procedure swap(var x,y:integer);
var temp:integer;
begin
    temp := x;
    x := y;
    y := temp;
end;

begin
    a := 10; b:= 20;
    swap(a, b);
    writeln(a, ' ', b);
end.
```

```
// 使用指针的代码
#include <iostream>

void swap(int* x, int* y) {
  int temp;
  temp = *x;
  *x = *y;
  *y = temp;
}

int main() {
  int a = 10, b = 20;
  swap(&a, &b);
  std::cout << a << " " << b;

  return 0;
}
```

注意，此处 C++ 代码 **涉及指针问题**. 指针问题还是很麻烦的，建议去阅读相关资料.

```
// 使用引用的代码
#include <iostream>
```

```
3
4  void swap(int& x, int& y) {
5    int temp;
6    temp = x;
7    x = y;
8    y = temp;
9  }
10
11 int main(int argc, char const* argv[]) {
12   int a = 10, b = 20;
13   swap(a, b);
14   std::cout << a << " " << b;
15
16   return 0;
17 }
```

注意，此处 C++ 代码涉及 **引用相关类型问题**. 在用引用调用一些 STL 库、模板库的时候可能会遇到一些问题，这时候需要手动声明别类型. 具体资料可以在《C++ Primer》第五版或者网络资料中自行查阅.

C++ 中函数传递参数还有其他方法，其中一种是 **直接使用全局变量传递参数**，如果不会用指针，可以先用这种方法. 但是这种方法的缺陷是没有栈保存数据，**没有办法在递归函数中传参**.（除非手写栈，注意，手写栈也是一种突破系统栈限制的方法.）

## C++ 标准库与参考资料

千万不要重复造轮子（除非为了练习），想要自己动手写一个功能出来之前，先去看看有没有这个函数或者数据结构.

### C++ 标准库

C++ 标准库中 `<algorithm>` 有很多有用的函数比如快排、二分查找等，可以直接调用. 请参考 STL 算法→4.3.3@p386 页面.

还有 STL 容器，比如数组、向量（可变大小的数组）、队列、栈等，附带很多函数. 请参考 STL 容器简介→4.3.2.1@p370 页面.

如果要找关于字符串操作的函数见

- std::basic_string - cppreference.com[4]
- `<string>`- C++ Reference[5]

C/C++ 的指针是很灵活的东西，可以参考 指针→4.2.6.4@p356 页面. 如果想要彻底理解指针，建议找本书或者参考手册仔细阅读.

### 错误排查与技巧

- 常见错误→2.6.3@p110
- 常见技巧→2.6.4@p120

### C++ 语言资料

- 学习资源→2.5@p96
- cppreference.com[6]：最重要的 C/C++ 参考资料
- C++ 教程 - 菜鸟教程[7]
- C++ Language - C++ Tutorials[8]
- Reference - C++ Reference[9]
- C++ Standard Library - Wikipedia[10]
- The Ultimate Question of Programming, Refactoring, and Everything[11]

- Google C++ Style Guide[12]

# 后记

写到这里，很多同学会觉得这一点都不急救啊，有很多东西没有提到啊．那也是没办法的事情．

虽然是为了急救，但很多东西像怎么把字符串转化为数字，怎么搜索字符串中的字符，这些东西也不适合一篇精悍短小的急救帖，如果把这些都写出来，那就是 C++ 入门教程，所以请充分利用本 Wiki、参考手册与搜索引擎．

需要指出的一点是，上面说 C++ 的语法，其实有很多语法是从 C 语言来的，标题这么写比较好——《Pascal 转 C/C++ 急救帖》．

Pascal 在上个世纪后半叶是门很流行的语言，它早于 C 语言，不过随着 UNIX 系统的普及，微软使用 C 语言，现在 Pascal 已经成为历史了．Pascal 后期发展也是有的，比如 Free Pascal 这个开源编译器项目，增加面向对象的特性（Delphi 语言）．Pascal 目前的用处除了在信息竞赛外，有一个特点是其他语言没有的——编译支持非常非常多老旧机器，比如 Gameboy 这种上个世纪的任天堂游戏机，还有一个用处就是以伪代码的形式（Pascal 风格的伪代码）出现在各种教科书中．

最后，Pascal 的圈子其实很小，C/C++ 的圈子很大，帮助手册与教程很多很全，一定要掌握好英语．世界上还有很多很多编程语言，而计算机这门学科与技术不光是信息竞赛和编程语言．

## 本文 Pascal 语言的参考文献

- Lazarus wiki[13]
- Free Pascal Reference guide[14]

# 附录：Pascal 与 C++ 运算符与数学函数语法对比表

仅包括最常用的运算符与函数．

## 基本算术

| | Pascal | C++ |
|---|---|---|
| 加法 | `a + b` | `a + b` |
| 减法 | `a - b` | `a - b` |
| 乘法 | `a * b` | `a * b` |
| 整除 | `a div b` | `a / b` |
| 浮点除法 | `a / b` | `a / b` |
| 取模 | `a mod b` | `a % b` |

## 逻辑

| | Pascal | C++ |
|---|---|---|
| 非 | `not(a)` | `!a` |
| 且 | `a and b` | `a && b` |
| 或 | `a or b` | a \|\| b |

## 比较

| | Pascal | C++ |
|---|---|---|
| 相等 | `a = b` | `a == b` |
| 不等 | `a <> b` | `a != b` |

| | | |
|---|---|---|
| 大于 | `a > b` | `a > b` |
| 小于 | `a < b` | `a < b` |
| 大于等于 | `a >= b` | `a >= b` |
| 小于等于 | `a <= b` | `a <= b` |

## 赋值

| Pascal | C++ |
|---|---|
| `a := b` | `a = b` |
| `a := a + b` | `a += b` |
| `a := a - b` | `a -= b` |
| `a := a * b` | `a *= b` |
| `a := a div b` 或 `a := a / b` | `a /= b` |
| `a := a mod b` | `a %= b` |

## 自增/自减

| | Pascal | C++ |
|---|---|---|
| 自增 | `inc(a)` | `a++` |
| 自增 | `inc(a)` | `++a` |
| 自减 | `dec(a)` | `a--` |
| 自减 | `dec(a)` | `--a` |

## 数学函数

使用需要导入 `<cmath>` 库.

| | Pascal | C++ |
|---|---|---|
| 绝对值 | `abs(a)` | `abs(a)`（整数） |
| 绝对值 | `abs(a)` | `fabs(a)`（浮点数） |
| $a^b$ | N/A[1] | `pow(a, b)` |
| 截断取整 | `trunc(a)` | `trunc(a)` |
| 近似取整 | `round(a)` | `round(a)` |

其他函数请参考:

- 常用数学函数 - cppreference.com[15]

[1] VS Code 的官方网站

[2] 基础类型 - cppreference.com

[3] 隐式转换 - cppreference.com

[4] std::basic_string - cppreference.com

[5] `<string>`- C++ Reference

[6] cppreference.com

[7] C++ 教程 - 菜鸟教程

[8] C++ Language - C++ Tutorials

[9] Reference - C++ Reference

[10] C++ Standard Library - Wikipedia

[11] The Ultimate Question of Programming, Refactoring, and Everything

[12] Google C++ Style Guide

[1] Extended Pascal 中有 `a**b`，不过需要导入 `Math` 库.

[13] Lazarus wiki  [14] Free Pascal Reference guide  [15] 常用数学函数 - cppreference.com 

# 4.7 Python 速成

**Authors:** cmpute, Henry-ZHR, ranwen, abc1763613206, billchenchina, chinggg, ChungZH, CoelacanthusHex, countercurrent-time, Dong Tsing-hsuen, Early0v0, Enter-tainer, F1shAndCat, Great-designer, hensier, HeRaNO, Hszzzx, imba-tjd, Ir1d, ksyx, lingxier, LovelyBuggies, Marcythm, mgt, Mooos-MoSheng, NachtgeistW, ouuan, Rottenwooood, shawlleyw, shuzhouliu, sshwy, SukkaW, Suyun514, Tiphereth-A, tLLWtG, wineee, wxh06, Xeonacid, yusancky, zyouxam, zzjjbb, jiangmuran, CuriosityQiu

## 关于 Python

Python 是一门已在世界上广泛使用的解释型语言. 它提供了高效的高级数据结构, 还能简单有效地面向对象编程，也可以在算法竞赛.

### Python 的优点

- Python 是一门 **解释型** 语言：Python 不需要编译和链接，可以在一定程度上减少操作步骤.
- Python 是一门 **交互式** 语言：Python 解释器实现了交互式操作，可以直接在终端输入并执行指令.
- Python **易学易用**：Python 提供了大量的数据结构，也支持开发大型程序.
- Python **兼容性强**：Python 同时支持 Windows、macOS 和 Unix 操作系统.
- Python **实用性强**：从简单的输入输出到科学计算甚至于大型 WEB 应用，都可以写出适合的 Python 程序.
- Python **程序简洁、易读**：Python 代码通常比实现同种功能的其他语言的代码短.
- Python **支持拓展**：Python 会开发 C 语言程序（即 CPython），支持把 Python 解释器和用 C 语言开发的应用链接，用 Python 扩展和控制该应用.

### 学习 Python 的注意事项

- 目前主要使用的 Python 版本是 Python 3.7 及以上的版本，Python 2 和 Python 3.6 及以前的 Python 3 已经 不被支持[1]，但仍被一些老旧系统与代码所使用. 本文将 **介绍较新版本的 Python**. 如果遇到 Python 2 代码，可以尝试 `2to3`[2] 程序将 Python 2 代码转换为 Python 3 代码.
- Python 的设计理念和语法结构 **与一些其他语言的差异较大**, 隐藏了许多底层细节, 所以呈现出实用而优雅的风格.
- Python 是高度动态的解释型语言，因此其 **程序运行速度相对较慢**，尤其在使用其内置的 `for` 循环语句时. 在使用 Python 时，应尽量使用 `filter`、`map` 等内置函数，或使用 列表生成[3] 语法的手段来提高程序性能.

## 环境搭建

参见 Python 3→3.6@p241. 或者:

- Windows：也可以在 Microsoft Store 中免费而快捷地获取 Python.
- macOS/Linux：通常情况下，大部分的 Linux 发行版中已经自带了 Python. 如果只打算学习 Python 语法，并无其它开发需求，不必另外安装 Python.

> **⚠ 注意**
>
> 在一些默认安装（指使用软件包管理器安装）Python 的系统（如 Unix 系统）中，应在终端中运行 `python3` 打开 Python 3 解释器. [1]

[1] 2. Python 解释器—Python 3 文档[4]

此外，也可以通过 venv、conda、Nix 等工具管理 Python 工具链和 Python 软件包，创建隔离的虚拟环境，避免出现依赖问题.

作为一种解释型语言，Python 的执行方式和 C++ 有所不同，这种差异在使用 IDE 编程时往往得不到体现，因此这里需要强调一下运行程序的不同方式.

当在命令行中键入 `python3` 或刚刚打开 IDLE 时，你实际进入了一种交互式的编程环境，也称「REPL」(「读取 - 求值 - 输出」循环)，初学者可以在这里输入语句并立即看到结果，这让验证一些语法变得极为容易，我们也将在后文中大量使用这种形式.

但若要编写完整的程序，你最好还是新建一个文本文件（通常后缀为 `.py`），然后在命令行中执行 `python3 filename.py`，就能够运行代码看到结果了.

### 一些平台提供的 Python 版本

| 系统名/版本 | python 版本 |
| --- | --- |
| Noi Linux 2.0 | 3.8.0, Include requests |
| Luogu 评测机 | 3.11.5, NumPy 1.25.2 |
| 基于 Hydro 的 OJ | 3.8.0+ Include NumPy |
| Ubuntu 22.04（内置） | 3.10.4 |
| 微软商店 | 最新正式版 |

> ⚠ **注意**
>
> 本表格在本文撰写时（2025/01/15）时有效，建议前往相关平台重新查证.

目前国内关于 **源码** 的镜像缓存主要是 北京交通大学自由与开源软件镜像站[5] 和 华为开源镜像站[6]，可以到那里尝试下载 Python 安装文件.

## 使用 `pip` 安装第三方库

Python 的生命力很大程度上来自于丰富的第三方库，编写一些实用程序时「调库」是常规操作，`pip` 是首选的安装第三方库的程序. 自 Python 3.4 版本起，它被默认包含在 Python 二进制安装程序中.

`pip` 中的第三方库主要存储在 Python 包索引（PyPI）[7] 上，用户也可以指定其它第三方库的托管平台. 使用方法可参照 pypi 镜像使用帮助 - 清华大学开源软件镜像站[8] 等使用帮助. 你可以在 MirrorZ[9] 上获取更多 PyPI 镜像源.

> ℹ **使用清华大学开源镜像站安装一个包**
>
> ```
> pip install -i https://mirrors.tuna.tsinghua.edu.cn/pypi/web/simple <some-package>
> ```

## 基本语法

Python 的语法简洁而易懂，也有许多官方和第三方文档与教程. 这里仅介绍一些对 OIer 比较实用的语言特性，你可以在 Python 文档[10] 和 Python Wiki[11] 等网页上了解更多关于 Python 的教程.

### 注释

加入注释并不会对代码的运行产生影响，但加入注释可以使代码更加易懂易用.

```
# 用 # 字符开头的是单行注释

```

```
"""
跨多行字符串会用三引号
（即三个单引号或三个双引号）
包裹，但也通常被用于注释
"""
```

加入注释代码并不会对代码产生影响．我们鼓励加入注释来使代码更加易懂易用．

## 基本数据类型

### 一切皆对象

在 Python 中，你无需事先声明变量名及其类型，直接赋值即可创建各种类型的变量：

```
>>> x = -3  # 语句结尾不用加分号
>>> x
-3
>>> f = 3.1415926535897932384626; f  # 实在想加分号也可以，这里节省了一行
3.141592653589793
>>> s1 = "O"
>>> s1  # 在 Python 中双引号和单引号的作用相同
'O'
>>> b = 'A' == 65  # 'A' 和 65 不是一个数据类型，所以不相等
>>> b  # True, False 首字母均大写
False
>>> True + 1 == 2 and not False != 0  # Python 中的表达式中大多使用单词，但是也支持符号
True
```

但这不代表 Python 没有类型的概念，实际上解释器会根据赋值或运算自动推断变量类型，你可以使用内置函数 `type()` 查看这些变量的类型：

```
>>> type(x)
<class 'int'>
>>> type(f)
<class 'float'>
>>> type(s1)  # 请注意，不要给字符串起名为 str，否则 str 对象会被篡改
<class 'str'>
>>> type(b)
<class 'bool'>
```

**内置函数[12]是什么?**

在 C/C++ 中，很多常用函数都分散在不同的头文件中，但 Python 的解释器内置了许多实用且通用的函数，你可以直接使用而无需注意它们的存在，但这也带来了小问题，这些内置函数的名称多为常见单词，你需要注意避免给自己的变量起相同的名字，否则可能会产生奇怪的结果．

正如我们所看到的，Python 内置有整数、浮点数、字符串和布尔类型，可以类比为 C++ 中的 `int`，`float`，`string` 和 `bool`．但有一些明显的不同之处，比如没有 `char` 字符类型，也没有 `double` 类型（但 `float` 其实对应 C 中的双精度），如果需要更精确的浮点运算，可以使用标准库中的 decimal[13] 模块，如果需要用到复数，Python 还内置了 `complex` 类型（而这也意味着最好不要给变量起名为 `complex`）．可以看到这些类型都以 `class` 开头，而这正是 Python 不同于 C++ 的关键之处，Python 程序中的所有数据都是由对象或对象间关系来表示的，函数是对象，类型本身也是对象：

```
>>> type(int)
<class 'type'>
```

```
>>> type(pow)  # 求幂次的内置函数，后文会介绍
<class 'builtin_function_or_method'>
>>> type(type)  # type() 也是内置函数，但有些特殊，感兴趣可自行查阅
<class 'type'>
```

你或许会觉得这些概念一时难以理解且没有用处，所以我们暂时不再深入，在后文的示例中你或许能慢慢体会到，Python 的对象提供了强大的方法，我们在编程时应当优先考虑围绕对象而不是过程进行操作，这会让我们的代码显得更加紧凑明晰.

### 数字运算

有人说，你可以把你系统里装的 Python 当作一个多用计算器，这是事实.

在交互模式下，你可以在提示符 `>>>` 后面输入一个表达式，就像其他大部分语言（如 C++）一样使用运算符 `+`、`-`、`*`、`/`、`%` 来对数字进行运算，也可以使用 `()` 来进行符合结合律的分组，读者可以自行试验，在这里我们仅展示与 C++ 差异较大的部分：

```
>>> 5.0 * 6  # 浮点数的运算结果是浮点数
30.0
>>> 15 / 3  # 与 C/C++ 不同，除法永远返回浮点 float 类型
5.0
>>> 5 / 100000  # 位数太多，结果显示成科学计数法形式
5e-05
>>> 5 // 3 # 使用整数除法（地板除）则会向下取整，输出整数类型
1
>>> -5 // 3 # 符合向下取整原则，注意这与 C/C++ 不同
-2
>>> 5 % 3 # 取模
2
>>> -5 % 3 # 负数取模结果一定是非负数，这点也与 C/C++ 不同，不过都满足 (a//b)*b+(a%b)==a
1
>>> x = abs(-1e4)  # 求绝对值的内置函数
>>> x += 1  # 没有自增/自减运算符
>>> x  # 科学计数法默认为 float
10001.0
```

在上面的实践中可以发现，除法运算（`/`）永远返回浮点类型（在 Python 2 中返回整数）. 如果你想要整数或向下取整的结果的话，可以使用整数除法（`//`）. 同样的，你也可以像 C++ 中一样，使用模（`%`）来计算余数，科学计数法的形式也相同.

特别地，Python 用 `**` 即可进行幂运算，还通过内置的 `pow(a, b, mod)` 提供了 快速幂→9.7@p1131 的高效实现.

```
>>> 3 ** 4 # 幂运算
81
>>> 2 ** 512
13407807929942597099574024998205846127479365820592393377723561443721764030073546976801874298166
>>> pow(2, 512, int(1e4)) # 即 2**512 % 10000 的快速实现，1e4 是 float 所以要转 int
4096
>>> 2048 ** 2048 # 在IDLE里试试大整数?
>>> 0.1 + 0.1 + 0.1 - 0.3 == 0.  # 和 C/C++ 一样需要注意浮点数不能直接判相等
False
```

### 数据类型判断

对于一个变量，可以使用 `type(object)` 返回变量的类型，例如 `type(8)` 和 `type('a')` 的值分别为 `<class 'int'>` 和 `<class 'str'>`.

### 基本输入输出[14]

Python 中的输入输出主要通过内置函数 `input()` 和 `print()` 完成，`print()` 的用法十分符合直觉：

```
>>> a = [1,2,3]; print(a[-1])  # 打印时默认末尾换行
3
>>> print(ans[0], ans[1])  # 可以输出任意多个变量，默认以空格间隔
1 2
>>> print(a[0], a[1], end='')  # 令 end='', 使末尾不换行
1 2>>>
>>> print(a[0], a[1], sep=', ')  # 令 sep=', ', 改变间隔样式
1, 2
>>> print(str(a[0]) + ', ' + str(a[1]))  # 输出同上，但是手动拼接成一整个字符串
```

`input()` 函数的行为接近 C++ 中的 `getline()`，即将一整行作为字符串读入，且末尾没有换行符.

```
>>> s = input('请输入一串数字: '); s  # 自己调试时可以向 input() 传入字符串作为提示
请输入一串数字: 1 2 3 4 5 6
'1 2 3 4 5 6'
```

### 字符串

Python 3 提供了强大的基于 Unicode[15] 的字符串类型，使用起来和 C++ 中的 `string` 类似，一些概念如转义字符也都相通，除了加号拼接和索引访问，还额外支持数乘 `*` 重复字符串，和 `in` 操作符.

```
>>> s1 = "O"  # 单引号和双引号都能包起字符串，有时可节省转义字符
>>> s1 += 'I-Wiki'  # 为和 C++ 同步建议使用双引号
>>> 'OI' in s1  # 检测子串很方便
True
>>> len(s1)  # 类似 C++ 的 s.length()，但更通用
7
>>> s2 = """ 感谢你的阅读
... 欢迎参与贡献!
"""   # 使用三重引号的字符串可以跨越多行
>>> s1 + s2
'OI-Wiki 感谢你的阅读\n欢迎参与贡献!'
>>> print(s1 + s2)  # 这里使用了 print() 函数打印字符串
OI-Wiki 感谢你的阅读
欢迎参与贡献!
>>> s2[2] * 2 + s2[3] + s2[-1]  # 负数索引从右开始计数，加上 len(s)，相当于模 n 的剩余类环
'谢谢你!'
>>> s1[0] = 'o'  # str 是不可变类型，不能原地修改，其实 += 也是创建了新的对象
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Python 支持多种复合数据类型，可将不同值组合在一起. 最常用的 `list`，类型是用方括号标注、逗号分隔的一组值. 例如，`[1, 2, 3]` 和 `['a','b','c']` 都是列表.

除了索引，字符串还支持切片，它的设计非常精妙，格式为 `s[左闭索引:右开索引:步长]`：

```
>>> s = 'OI-Wiki 感谢你的阅读\n欢迎参与贡献!'
>>> s[:8]  # 省略左闭索引则从头开始
'OI-Wiki '
>>> s[8:14]  # 左闭右开设计的妙处，长度为 14-8=6，还和上一个字符串无缝衔接
'感谢你的阅读'
>>> s[-4:]  # 省略右开索引则直到结尾
```

```
'与贡献!'
>>> s[8:14:2]  # 步长为2
'感你阅'
>>> s[::-1]  # 步长为 -1 时，获得了反转的字符串
'!献贡与参迎欢\n读阅的你谢感 ikiW-IO'
>>> s  # 但原来的字符串并未改变
'OI-Wiki 感谢你的阅读\n欢迎参与贡献!'
```

在最新的 Python 3 版本中，字符串是以 Unicode 编码的，也就是说，Python 的字符串支持多语言．[2]在 Python 中，可以对一个 Unicode 字符使用内置函数 `ord()` 将其转换为对应的 Unicode 编码，逆向的转换使用内置函数 `chr()`．C/C++ 中 `char` 类型也可以和 对应的 ASCII 码互转．

如果想把数字转换成对应的字符串，可以使用内置函数 `str()`，反之可以使用 `int()` 和 `float()`，你可以类比为 C/C++ 中的强制类型转换，但括号不是加在类型上而是作为函数的一部分括住参数．

Python 的字符串类型提供了许多强大的方法，包括计算某字符的索引与出现次数，转换大小写等等，这里就不一一列举，强烈建议查看 官方文档[17] 熟悉常用方法，遇到字符串操作应当首先考虑使用这些方法而非自力更生．

## 创建数组

从 C++ 转过来的同学可能很迷惑怎么在 Python 中创建数组，这里就介绍在 Python 开「数组」的语法，需要强调我们介绍的其实是几种 序列类型[18]，和 C 的数组有着本质区别，而更接近 C++ 中的 `vector`．

### 使用 `list`

列表(`list`)大概是 Python 中最常用也最强大的序列类型，列表中可以存放任意类型的元素，包括嵌套的列表，这符合数据结构中「广义表」的定义．请注意不要将其与 C++ STL 中的双向链表 `list`→4.3.2.3@p373 混淆，故本文将使用「列表」而非 `list` 以免造成误解．

```
>>> []  # 创建空列表，注意列表使用方括号
[]
>>> nums = [0, 1, 2, 3, 5, 8, 13]; nums  # 初始化列表，注意整个列表可以直接打印
[0, 1, 2, 3, 5, 8, 13]
>>> nums[0] = 1; nums  # 支持索引访问，支持修改元素
[1, 1, 2, 3, 5, 8, 13]
>>> nums.append(nums[-2]+nums[-1]); nums  # append() 同 vector 的 push_back()，也都没有返回值
[1, 1, 2, 3, 5, 8, 13, 21]
>>> nums.pop()  # 弹出并返回末尾元素，可以当栈使用；其实还可指定位置，默认是末尾
21
>>> nums.insert(0, 1); nums  # 同 vector 的 insert(position, val)
[1, 1, 1, 2, 3, 5, 8, 13]
>>> nums.remove(1); nums  # 按值移除元素（只删第一个出现的），若不存在则抛出错误
[1, 1, 2, 3, 5, 8, 13]
>>> len(nums)  # 求列表长度，类似 vector 的 size()，但 len() 是内置函数
7
>>> nums.reverse(); nums  # 原地逆置
[13, 8, 5, 3, 2, 1, 1]
>>> sorted(nums)  # 获得排序后的列表
[1, 1, 2, 3, 5, 8, 13]
>>> nums  # 但原来的列表并未排序
[13, 8, 5, 3, 2, 1, 1]
>>> nums.sort(); nums  # 原地排序，可以指定参数 key 作为排序标准
```

[2]Unicode 指南—Python 3 文档[16]

```
[1, 1, 2, 3, 5, 8, 13]
>>> nums.count(1)  # 类似 std::count()
2
>>> nums.index(1)  # 返回值首次出现项的索引号，若不存在则抛出错误
0
>>> nums.clear(); nums  # 同 vector 的 clear()
```

以上示例展现了列表与 `vector` 的相似之处，`vector` 中常用的操作一般也都能在列表中找到对应方法，不过某些方法如 `len()`,`sorted()` 会以内置函数的面目出现，而 STL 算法中的函数如 `find()`,`count()`,`max_element()`,`sort()`,`reverse()` 在 Python 中又成了对象的方法，使用时需要注意区分，更多方法请参见官方文档的 列表详解[19]. 下面将展示列表作为 Python 的基本序列类型的一些强大功能：

Python 支持多种复合数据类型，可将不同值组合在一起. 最常用的 `list`，类型是用方括号标注、逗号分隔的一组值. 例如，`[1, 2, 3]` 和 `['a','b','c']` 都是列表.

```
>>> lst = [1, '1'] + ["2", 3.0]  # 列表直接相加生成一个新列表
>>> lst  # 这里存放不同的类型只是想说明可以这么做，但这不是好的做法
[1, '1', '2', 3.0]
>>> 3 in lst  # 实用的成员检测操作，字符串也有该操作且还支持子串检测
True
>>> [1, '1'] in lst  # 仅支持单个成员检测，不会发现「子序列」
False
>>> lst[1:3] = [2, 3]; lst  # 切片并赋值，原列表被修改
[1, 2, 3, 3.0]
>>> lst[::-1]  # 获得反转后的新列表
[3.0, 3, 2, 1]
>>> lst *= 2; lst  # 数乘拼接
[1, 2, 3, 3.0, 1, 2, 3, 3.0]
>>> del lst[4:]; lst  # 也可写 lst[4:] = []，del 语句不止可以用于删除序列中元素
[1, 2, 3, 3.0]
```

以上示例展现了列表作为序列的一些常用操作，可以看出许多操作如切片是与字符串相通的，但字符串是「不可变序列」而列表是「可变序列」，故可以通过切片灵活地修改列表. 在 C/C++ 中我们往往会通过循环处理字符数组，下面将展示如何使用 「列表推导式」[20] 在字符串和列表之间转换：

```
>>> # 建立一个 [65, 70) 区间上的整数数组，range 也是一种类型，可看作左闭右开区间，第三个参数为步长可省略
>>> nums = list(range(65,70))  # 记得 range 外面还要套一层 list()
[65, 66, 67, 68, 69]
>>> lst = [chr(x) for x in nums]  # 列表推导式的典型结构，[exp for var in iterable if cond]
>>> lst  # 上两句可以合并成 [chr(x) for x in range(65,70)]
['A', 'B', 'C', 'D', 'E']
>>> s = ''.join(lst); s # 用空字符串 '' 拼接列表中的元素生成新字符串
'ABCDE'
>> list(s)  # 字符串生成字符列表
['A', 'B', 'C', 'D', 'E']
>>> # 如果你不知道有 s.lower() 方法就可能写出下面这样新瓶装旧酒的表达式
>>> ''.join([chr(ord(ch) - 65 + 97) for ch in s if ch >= 'A' and ch <= 'Z'])
'abcde'
```

下面演示一些在 OI 中更常见的场景，比如二维「数组」：

```
>>> vis = [[0] * 3] * 3  # 开一个 3*3 的全 0 数组
```

```
>>> vis
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> vis[0][0] = 1; vis  # 怎么会把其他行也修改了?
[[1, 0, 0], [1, 0, 0], [1, 0, 0]]
>>> # 先来看下一维列表的赋值
>>> a1 = [0, 0, 0]; a2 = a1; a3 = a1[:]  # 列表也可以直接被赋给新的变量
>>> a1[0] = 1; a1  # 修改列表 a1，似乎正常
[1, 0, 0]
>>> a2  # 怎么 a2 也被改变了
[1, 0, 0]
>>> a3  # a3 没有变化
[0, 0, 0]
>>> id(a1) == id(a2) and id(a1) != id(a3)  # 内置函数 id() 给出对象的「标识值」，可类比为地址，地址相同说明是一个对象
True
>>> vis2 = vis[:]  # 拷贝一份二维列表
>>> vis[0][1] = 2; vis  # vis 会被批量修改
>>> [[1, 2, 0], [1, 2, 0], [1, 2, 0]]
>>> vis2  # 但 vis2 是切片拷贝还是被改了
>>> [[1, 2, 0], [1, 2, 0], [1, 2, 0]]
>>> id(vis) != id(vis2)  # vis 和 vis2 不是一个对象
True
>>> # vis2 虽然不是 vis 的引用，但其中对应行都指向相同的对象
>>> [id(vis[i]) == id(vis2[i]) for i in range(3)]
[True, True, True]
>>> # 回看二维列表自身
>>> [id(x) for x in vis]  # 具体数字和这里不一样但三个值一定相同，说明是三个相同对象
[139760373248192, 139760373248192, 139760373248192]
```

其实有一个重要的事实，Python 中赋值只传递了引用而非创建新值，你可以创建不同类型的变量并赋给新变量，验证发现二者的标识值是相同的，只不过直到现在我们才介绍了列表这一种可变类型，而给数字、字符串这样的不可变类型赋新值时实际上创建了新的对象，故而前后两个变量互不干扰．但列表是可变类型，所以我们修改一个列表的元素时，另一个列表由于指向同一个对象所以也被修改了．创建二维数组也是类似的情况，示例中用乘法创建二维列表相当于把 `[0]*3` 这个一维列表重复了 3 遍，所以涉及其中一个列表的操作会同时影响其他两个列表．更不幸的是，在将二维列表赋给其他变量的时候，就算用切片来拷贝，也只是「浅拷贝」，其中的元素仍然指向相同的对象，解决这个问题需要使用标准库中的 `deepcopy`[21]，或者尽量避免整个赋值二维列表．不过还好，创建二维列表时避免创建重复的列表还是比较简单，只需使用「列表推导式」：

```
>>> vis1 = [[0] * 3 for _ in range(3)]  # 把用不到的循环计数变量设为下划线 _ 是一种惯例
>>> # 但在 REPL 中 _ 默认指代上一个表达式输出的结果，故也可使用双下划线
>>> vis1
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> [id(x) for x in vis1]  # 具体数字和这里不一样但三个值一定不同，说明是三个不同对象
[139685508981248, 139685508981568, 139685508981184]
>>> vis1[0][0] = 1
[[1, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> a2[0][0] = 10  # 访问和赋值二维数组
```

我们未讲循环的用法就先介绍了列表推导式，这是由于 Python 是高度动态的解释型语言，因此其程序运行有大量的额外开销．尤其是 **for 循环在 Python 中运行的奇慢无比**．因此在使用 Python 时若想获得高性能，尽量使用列表推导式，或者 `filter`,`map` 等内置函数直接操作整个序列来避免循环，当然这还是要根据具体问题而定．

### 使用 NumPy

**什么是 NumPy**

NumPy[22] 是著名的 Python 科学计算库，提供高性能的数值及矩阵运算．在测试算法原型时可以利用 NumPy 避免手写排序、求最值等算法．NumPy 的核心数据结构是 `ndarray`，即 n 维数组，它在内存中连续存储，是定长的．此外 NumPy 核心是用 C 编写的，运算效率很高．不过需要注意，它不是标准库的一部分，可以使用 `pip install numpy` 安装，但不保证 OI 考场环境中可用（参见文首 Python 版本→4@p321）．

下面的代码将介绍如何利用 NumPy 建立多维数组并进行访问．

```
>>> import numpy as np  # 请自行搜索 import 的意义和用法
>>> np.empty(3) # 开容量为 3 的空数组，注意没有初始化为 0
array([0.00000000e+000, 0.00000000e+000, 2.01191014e+180])
>>> np.zeros((3, 3)) # 开 3*3 的数组，并初始化为 0
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
>>> a1 = np.zeros((3, 3), dtype=int) # 开 3×3 的整数数组
>>> a1[0][0] = 1 # 访问和赋值
>>> a1[0, 0] = 1 # 更友好的语法
>>> a1.shape # 数组的形状
(3, 3)

>>> a1[:2, :2] # 取前两行、前两列构成的子阵，无拷贝
array([[1, 0],
       [0, 0]])

>>> a1[:, [0, 2]] # 获取第 1、3 列，无拷贝
array([[1, 0],
       [0, 0],
       [0, 0]])
>>> np.max(a1) # 获取数组最大值
1
>>> a1.flatten() # 将数组展平
array([1, 0, 0, 0, 0, 0, 0, 0, 0])

>>> np.sort(a1, axis = 1) # 沿行方向对数组进行排序，返回排序结果
array([[0, 0, 1],
       [0, 0, 0],
       [0, 0, 0]])
>>> a1.sort(axis = 1) # 沿行方向对数组进行原地排序
```

### 使用 `array`

`array`[23] 是 Python 标准库提供的一种高效数值数组，可以紧凑地表示基本类型值的数组，但不支持数组嵌套，也很少见到有人使用它，这里只是顺便提一下．

若无特殊说明，后文出现「数组」一般指「列表」．

### 输入输出[24]

Python 中的输入输出主要通过内置函数 `input()` 和 `print()` 完成．前文已经介绍过，下面介绍进阶用法．

### 格式化输出

算法竞赛中通常只涉及到基本的数值和字符串输出，`print()` 已基本足够，只有当涉及到浮点数位数时需要用到格式化字符串输出．格式化有三种方法，第一种也是最老旧的方法是使用 `printf()` 风格的 `%` 操作符；另一种是利用 `format` 函数[25]；第三种是 Python 3.6 新增的 f-string[26]，最为简洁，但不保证考场中的 Python 版本足够新．详细丰富的说明可以参考 这个网页[27]，尽管更推荐使用 `format()` 方法，但为了获得与 C 接近的体验，下面仅演示与 `printf()` 类似的老式方法：

```
>>> pi = 3.1415926; print('%.4f' % pi)   # 格式为 %[flags][width][.precision]type
3.1416
>>> '%.4f - %8f = %d' % (pi, 0.1416, 3)  # 右边多个参数用 () 括住，后面会看到其实是「元组」
'3.1416 - 0.141600 = 3'
```

### `split()` 函数

`input()` 函数的行为接近 C++ 中的 `getline()`，即将一整行作为字符串读入，且末尾没有换行符，但在算法竞赛中，常见的输入形式是一行输入多个数值，因此就需要使用字符串的 `split()` 方法并搭配列表推导式得到存放数值类型的列表，下面以输入 n 个数求平均值为例演示输入 n 个数得到「数组」的方法：

```
>>> s = input('请输入一串数字: '); s  # 自己调试时可以向 input() 传入字符串作为提示
请输入一串数字: 1 2 3 4 5 6
'1 2 3 4 5 6'
>>> a = s.split(); a
['1', '2', '3', '4', '5', '6']
>>> a = [int(x) for x in a]; a
[1, 2, 3, 4, 5, 6]
>>> # 以上输入过程可写成一行 a = [int(x) for x in input().split()]
>>> sum(a) / len(a)  # sum() 是内置函数
3.5
```

有时题目会在每行输入固定几个数，比如边的起点、终点、权重，如果只用上面提到的方法就只能每次读入数组然后根据下标赋值，这时可以使用 Python 的「拆包」特性一次赋值多个变量：

```
>>> u, v, w = [int(x) for x in input().split()]
1 2 4
>>> print(u,v,w)
1 2 4
```

题目中经常遇到输入 N 行的情况，可我们还没有讲最基本的循环语句，但 Python 强大的序列操作能在不使用循环的情况下应对多行输入，下面假设将各条边的起点、终点、权值分别读入三个数组：

```
>>> N = 4; mat = [[int(x) for x in input().split()] for i in range(N)]
1 3 3
1 4 1
2 3 4
3 4 1
>>> mat  # 先按行读入二维数组
[[1, 3, 3], [1, 4, 1], [2, 3, 4], [3, 4, 1]]
>>> u, v, w = map(list, zip(*mat))
# *将 mat 解包得到里层的多个列表
# zip() 将多个列表中对应元素聚合成元组，得到一个迭代器
# map(list, iterable) 将序列中的元素（这里为元组）转成列表
>>> print(u, v, w)  # 直接将 map() 得到的迭代器拆包，分别赋值给 u, v, w
[1, 1, 2, 3] [3, 4, 3, 4] [3, 1, 4, 1]
```

上述程序实际上相当于先读入一个 N 行 3 列的矩阵，然后将其转置成 3 行 N 列的矩阵，也就是外层列表中嵌套了 3 个列表，最后将代表这起点、终点、权值的 3 个列表分别赋值给 u, v, w. 内置函数 `zip()`[28] 可以将多个等长序列中的对应元素拼接在「元组」内，得到新序列. 而 `map()` 其实是函数式编程的一种操作，它将一个给定函数作用于 `zip()` 所产生序列的元素，这里就是用 `list()` 将元组变成列表. 你可以自行练习使用 `*` 和 `zip()`[29]，`map()`[30] 以理解其含义. 需要注意的是 Python 3 中 `zip()` 和 `map()` 创建的不再返回列表而是返回迭代器，这里暂不解释它们之间的异同，你可以认为迭代器可以产生列表中的各个元素，用 `list()` 套住迭代器就能生成列表.

### 文件读写[31]

Python 内置函数 `open()`[32] 用于文件读写，为了防止读写过程中出错导致文件未被正常关闭，这里只介绍使用 `with`[33] 语句的安全读写方法：

```
a = []
with open("in.txt") as f:
    N = int(f.readline())  # 读入第一行的 N
    a[len(a) :] = [[int(x) for x in f.readline().split()] for i in range(N)]

with open("out.txt", "w") as f:
    f.write("1\n")
```

关于文件读写的函数有很多，分别适用于不同的场景，由于 OI 赛事尚不支持使用 Python，这里从略.

### 控制流程[34]

尽管我们已经学习了 Python 的许多特性，但到目前为止我们展示的 Python 代码都是单行语句，这掩盖了 Python 和 C 在代码风格上的重大差异：首先，Python 中不用 `{}` 而是用缩进表示块结构，如果缩进没有对齐会直接报错，如果 tab 和 空格混用也会报错；其次，块结构开始的地方比如 `if` 和 `for` 语句的行末要有冒号 `:`. 这有助于代码的可读性，但你也可能怀念 C 那种自由的体验，毕竟如果复制粘贴时因为丢失缩进而不得不手动对齐是很恼人的.

### 循环结构

列表推导式能在一行内高效地完成批量操作，但有时为了压行我们已经显得过分刻意，许多场景下还是只能使用循环结构，所以我们再以读入多行数据为例展示 Python 中的循环是如何编写的：

```
# 请注意从现在开始我们不再使用 REPL，请自行复制多行数据
u, v, w = ([] for i in range(3))  # 多变量赋值
for i in range(4):  # 这里假设输入 4 行数据
    _u, _v, _w = [int(x) for x in input().split()]
    u.append(_u), v.append(_v), w.append(_w)
    # 不可进行类似 cin >> u[i] >> v[i] >> w[i] 的操作，因为必定超出列表当前的长度
    # 当然你可以选择初始化长度为 MAXN 的全 0 列表，不过需要记住真实长度并删掉多余元素
print(u, v, w)
```

需要注意，Python 中的 for 循环和 C/C++ 有较大的差别，其作用类似 C++ 11 引入的 「基于范围的循环」→4.4.7@p432，实质是迭代序列中的元素，比如编写循环遍历数组下标需要迭代 `range(len(lst))`，而非真正定义起始和终止条件，所以使用起来并没有 C/C++ 灵活.

下面再用 while 循环展示行数不定的情况下如何输入：

```
u, v, w = [], [], []  # 多变量赋值，其实同上
s = input()  # 注意 Python 中赋值语句不能放在条件表达式中
while s:  # 不能像 C 那样 while(!scanf())
    # 用切片拼接避免了 append()，注意列表推导式中又嵌套了列表
    u[len(u) :], v[len(v) :], w[len(w) :] = [[int(x)] for x in s.split()]
```

```
    s = input()
# Python 3.8 引入了 walrus operator 海象运算符后，你可以节省两行，但考场环境很可能不支持
while s := input():
    u[len(u) :], v[len(v) :], w[len(w) :] = [[int(x)] for x in s.split()]
print(u, v, w)
```

### 选择结构

和 C/C++ 大同小异，一些形式上的差别都在下面的示例中有所展示，此外还需注意条件表达式中不允许使用赋值运算符（Python 3.8 以上可用 `:=`[35]），以及 没有 switch 语句[36].

```
# 条件表达式两侧无括号
if 4 >= 3 > 2 and 3 != 5 == 5 != 7:
    print("关系运算符可以连续使用")
    x = None or [] or -2
    print("&&  ||  !", "与  或  非", "and or not", sep="\n")
    print("善用 and/or 可节省行数")
    if not x:
        print("负数也是 True，不执行本句")
    elif x & 1:
        print("用 elif 而不是 else if\n" "位运算符与 C 相近，偶数&1 得 0，不执行本句")
    else:
        print("也有三目运算符") if x else print("注意结构")
```

### 异常处理

尽管 C++ 中有 try 块[37] 用于异常处理，但竞赛中一般从不使用，而 Python 中常见的是 EAFP[38] 风格，故而代码中可能大量使用 `try-except`[39] 语句，在后文介绍 `dict` 这一结构时还会用到，这里展示：

```
s = "OI-wiki"
pat = "NOIP"
x = s.find(pat)  # find() 找不到返回 -1
try:
    y = s.index(pat)  # index() 找不到则抛出错误
    print(y)  # 这句被跳过
except ValueError:
    print("没找到")
    try:
        print(y)  # 此时 y 并没有定义，故又会抛出错误
    except NameError as e:
        print("无法输出 y")
        print("原因:", e)
```

## 内置容器

Python 内置了许多强大的容器类型，只有熟练使用并了解其特点才能真正让 Python 在算法竞赛中有用武之地，除了上面详细介绍的 `list`（列表），还有 `tuple`（元组）、`dict`[40]（字典）和 `set`（集合）这几种类型.

元组可以简单理解成不可变的列表，不过还需注意「不可变」的内涵，如果元组中的某元素是可变类型比如列表，那么仍可以修改该列表的值，元组中存放的是对列表的引用所以元组本身并没有改变. 元组的优点是开销较小且「可哈希[41]」，后者在创建字典和集合时非常有用.

```
tup = tuple([[1, 2], 4])  # 由列表得到元组
# 等同于 tup = ([1,2], 4)
```

```
tup[0].append(3)
print(tup)
a, b = 0, "I-Wiki"  # 多变量赋值其实是元组拆包
print(id(a), id(b))
b, a = a, b
print(id(a), id(b))  # 你应该会看到 a, b 的 id 值现在互换了
# 这更说明 Python 中，变量更像是名字，赋值只是让其指代对象
```

字典就像 C++ STL 中的 map→4.3.2.4@p378（请注意和 Python 中内置函数 map()[42] 区分）用于存储键值对，形式类似 JSON[43]，但 JSON 中键必须是字符串且以双引号括住，字典则更加灵活强大，可哈希的对象都可作为字典的键. 需要注意 Python 几次版本更新后字典的特性有了较多变化，包括其中元素的顺序等，请自行探索.

```
dic = {"key": "value"}  # 基本形式
dic = {chr(i): i for i in range(65, 91)}  # 大写字母到对应 ASCII 码的映射，注意断句
dic = dict(zip([chr(i) for i in range(65, 91)], range(65, 91)))  # 效果同上
dic = {dic[k]: k for k in dic}  # 将键值对逆转，for k in dic 迭代其键
dic = {v: k for k, v in dic.items()}  # 和上行作用相同，dic.items() 以元组存放单个键值对
dic = {
    k: v for k, v in sorted(dic.items(), key=lambda x: -x[1])
}  # 字典按值逆排序，用到了 lambda 表达式

print(dic["A"])  # 返回 dic 中 以 'A' 为键的项，这里值为65
dic["a"] = 97  # 将 d[key] 设为 value，字典中原无 key 就是直接插入
if "b" in dic:  # LBYL(Look Before You Leap) 风格
    print(dic["b"])  # 若字典中无该键则会出错，故先检查
else:
    dic["b"] = 98

# 经典场景 统计出现次数
# 新键不存在于原字典，需要额外处理
try:  # EAFP (Easier to Ask for Forgiveness than Permission) 风格
    cnter[key] += 1
except KeyError:
    cnter[key] = 1
```

集合就像 C++ STL 中的 set→4.3.2.4@p378，不会保存重复的元素，可以看成只保存键的字典. 需要注意集合和字典都用 {} 括住，不过单用 {} 会创建空字典而不是空集合，这里就不再给出示例.

## 编写函数

Python 中定义函数无需指定参数类型和返回值类型，无形中为 OI 选手减少了代码量

```
def add(a, b):
    return a + b  # 动态类型的优势，a 和 b 也可以是字符串


def add_no_swap(a, b):
    print("in func #1:", id(a), id(b))
    a += b
    b, a = a, b
    print("in func #2:", id(a), id(b))  # a, b 已交换
    return a, b  # 返回多个值，其实就是返回元组，可以拆包接收

```

```
12
13 lst1 = [1, 2]
14 lst2 = [3, 4]
15 print("outside func #1:", id(lst1), id(lst2))
16 add_no_swap(lst1, lst2)
17 # 函数外 lst1, lst2 并未交换
18 print("outside func #2:", id(lst1), id(lst2))
19 # 不过值确实已经改变
20 print(lst1, lst2)
```

### 默认参数

Python 中函数的参数非常灵活，有关键字参数、可变参数等，但在算法竞赛中这些特性的用处并不是很大，这里只介绍一下默认参数，因为 C++ 中也有默认参数，且在 Python 中使用默认参数很有可能遇到坑．例如如下代码．

```
1 def append_to(element, to=[]):
2     to.append(element)
3     return to
4
5
6 lst1 = append_to(12)
7 lst2 = append_to(42)
8 print(lst1, lst2)
9
10 # 你可能以为输出是 [12] [42]
11 # 但运行结果其实是 [12, 42] [12, 42]
```

之所以出现以上的运行结果，是因为默认参数的值仅仅在函数定义的时候赋值一次，对于可变对象（如列表、字典、集合），所有调用会共享同一个对象，`lst1` 和 `lst2` 实际上都指向内存中同一个默认列表对象．因此，第二次调用后，这个共享列表的内容被修改为 `[12, 42]`．所以函数的默认参数的值应该设为不可变对象，使用 `None` 占位是一种最佳实践：

```
1 def append_to(element, to=None):
2     if to is None:
3         to = []
4     to.append(element)
5     return to
6
7
8 lst1 = append_to(12)
9 lst2 = append_to(42)
10 print(lst1, lst2)
11
12 # 运行结果为 [12] [42]
```

### 类型标注

Python 是一个动态类型检查的语言，以灵活但隐式的方式处理类型，Python 解释器仅仅在运行时检查类型是否正确，并且允许在运行时改变变量类型，俗话说「动态类型一时爽，代码重构火葬场」，程序中的一些错误可能在运行时才会暴露：

```
1 >>> if False:
2 ...     1 + "two"  # This line never runs, so no TypeError is raised
3 ... else:
```

```
...     1 + 2
...
3

>>> 1 + "two"  # Now this is type checked, and a TypeError is raised
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Python 3.5 后引入了类型标注，允许设置函数参数和返回值的类型，但只是作为提示，并没有实际的限制作用，需要静态检查工具才能排除这类错误（例如 PyCharm[44] 和 Mypy[45]），所以显得有些鸡肋，对于 OIer 来说更是只需了解，可按如下方式对函数的参数和返回值设置类型标注：

```
def headline(
    text,  # type: str
    width=80,  # type: int
    fill_char="-",  # type: str
):  # type: (...) -> str
    return f"{text.title()}".center(width, fill_char)


print(headline("type comments work", width=40))
```

除了函数参数，变量也是可以类型标注的，你可以通过调用 `__annotations__` 来查看函数中所有的类型标注．变量类型标注赋予了 Python 静态语言的性质，即声明与赋值分离：

```
>>> nothing: str
>>> nothing
NameError: name 'nothing' is not defined

>>> __annotations__
{'nothing': <class 'str'>}
```

## 装饰器

装饰器是一个函数，接受一个函数或方法作为其唯一的参数，并返回一个新函数或方法，其中整合了修饰后的函数或方法，并附带了一些额外的功能．简而言之，可以在不修改函数代码的情况下，增加函数的功能．相关知识可以参考 官方文档[46]．

部分装饰器在竞赛中非常实用，比如 `lru_cache`[47]，可以为函数自动增加记忆化的能力，在递归算法中非常实用：

`@lru_cache(maxsize=128,typed=False)`

- 传入的参数有 2 个：`maxsize` 和 `typed`，如果不传则 `maxsize` 的默认值为 128，`typed` 的默认值为 `False`．
- 其中 `maxsize` 参数表示的是 LRU 缓存的容量，即被装饰的方法的最大可缓存结果的数量．如果该参数值为 128，则表示被装饰方法最多可缓存 128 个返回结果；如果 `maxsize` 传入为 `None` 则表示可以缓存无限个结果．
- 如果 `typed` 设置为 `True`，不同类型的函数参数将被分别缓存，例如，`f(3)` 和 `f(3.0)` 会缓存两次．

以下是使用 `lru_cache` 优化计算斐波那契数列的例子：

```
@lru_cache(maxsize=None)
def fib(n):
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)
```

## 常用内置库

在这里介绍一些写算法可能用得到的内置库，具体用法可以自行搜索或者阅读 官方文档[48].

| 库名 | 用途 |
| --- | --- |
| `array`[49] | 定长数组 |
| `argparse`[50] | 命令行参数处理 |
| `bisect`[51] | 二分查找 |
| `collections`[52] | 有序字典、双端队列等数据结构 |
| `fractions`[53] | 有理数 |
| `heapq`[54] | 基于堆的优先级队列 |
| `io`[55] | 文件流、内存流 |
| `itertools`[56] | 迭代器 |
| `math`[57] | 数学函数 |
| `os.path`[58] | 系统路径等 |
| `random`[59] | 随机数 |
| `re`[60] | 正则表达式 |
| `struct`[61] | 转换结构体和二进制数据 |
| `sys`[62] | 系统信息 |

## 从例题对比 C++ 与 Python

**例题 洛谷 P4779【模板】单源最短路径（标准版）**[63]

给定一个 $n(1 \le n \le 10^5)$ 个点、$m(1 \le m \le 2 \times 10^5)$ 条有向边的带非负权图，请你计算从 $s$ 出发，到每个点的距离．数据保证能从 $s$ 出发到任意点．

### 声明常量

- C++

```cpp
#include <cstdio>
#include <cstring>
#include <queue>
#include <vector>
using namespace std;
constexpr int N = 1e5 + 5, M = 2e5 + 5;
```

- Python

```python
try:  # 引入优先队列模块
    import Queue as pq  # python version < 3.0
except ImportError:
    import queue as pq  # python3.*

N = int(1e5 + 5)
M = int(2e5 + 5)
INF = 0x3F3F3F3F
```

### 声明前向星结构体和其它变量

- C++

```cpp
struct qxx {
  int nex, t, v;
};

qxx e[M];
int h[N], cnt;

void add_path(int f, int t, int v) { e[++cnt] = qxx{h[f], t, v}, h[f] = cnt; }

using pii = pair<int, int>;
priority_queue<pii, vector<pii>, greater<pii>> q;
int dist[N];
```

- Python

```python
class qxx:  # 前向星类（结构体）
    def __init__(self):
        self.nex = 0
        self.t = 0
        self.v = 0


e = [qxx() for i in range(M)]  # 链表
h = [0 for i in range(N)]
cnt = 0

dist = [INF for i in range(N)]
q = pq.PriorityQueue()  # 定义优先队列，默认第一元小根堆


def add_path(f, t, v):  # 在前向星中加边
    # 如果要修改全局变量，要使用 global 来声明
    global cnt, e, h
    # 调试时的输出语句，多个变量使用元组
    # print("add_path(%d,%d,%d)" % (f,t,v))
    cnt += 1
    e[cnt].nex = h[f]
    e[cnt].t = t
    e[cnt].v = v
    h[f] = cnt
```

## Dijkstra 算法

- C++

```cpp
void dijkstra(int s) {
  memset(dist, 0x3f, sizeof(dist));
  dist[s] = 0, q.push(make_pair(0, s));
  while (q.size()) {
    pii u = q.top();
    q.pop();
    if (dist[u.second] < u.first) continue;
    for (int i = h[u.second]; i; i = e[i].nex) {
      const int &v = e[i].t, &w = e[i].v;
```

```
      if (dist[v] <= dist[u.second] + w) continue;
      dist[v] = dist[u.second] + w;
      q.push(make_pair(dist[v], v));
    }
  }
}
```

- Python

```
def nextedgeid(u):  # 生成器，可以用在 for 循环里
    i = h[u]
    while i:
        yield i
        i = e[i].nex


def dijkstra(s):
    dist[s] = 0
    q.put((0, s))
    while not q.empty():
        u = q.get()  # get 函数会顺便删除堆中对应的元素
        if dist[u[1]] < u[0]:
            continue
        for i in nextedgeid(u[1]):
            v = e[i].t
            w = e[i].v
            if dist[v] <= dist[u[1]] + w:
                continue
            dist[v] = dist[u[1]] + w
            q.put((dist[v], v))
```

## 主函数

- C++

```
int n, m, s;

int main() {
  scanf("%d%d%d", &n, &m, &s);
  for (int i = 1; i <= m; i++) {
    int u, v, w;
    scanf("%d%d%d", &u, &v, &w);
    add_path(u, v, w);
  }
  dijkstra(s);
  for (int i = 1; i <= n; i++) printf("%d ", dist[i]);
  return 0;
}
```

- Python

```
if __name__ == "__main__":
    # 一行读入多个整数．注意它会把整行都读进来
    n, m, s = map(int, input().split())
    for i in range(m):
        u, v, w = map(int, input().split())
```

```
        add_path(u, v, w)

    dijkstra(s)

    for i in range(1, n + 1):
        print(dist[i], end=" ")

    print()
```

## 完整代码

- C++

```
#include <cstdio>
#include <cstring>
#include <queue>
#include <vector>
using namespace std;
constexpr int N = 1e5 + 5, M = 2e5 + 5;

struct qxx {
  int nex, t, v;
};

qxx e[M];
int h[N], cnt;

void add_path(int f, int t, int v) { e[++cnt] = qxx{h[f], t, v}, h[f]
= cnt; }

using pii = pair<int, int>;
priority_queue<pii, vector<pii>, greater<pii>> q;
int dist[N];

void dijkstra(int s) {
  memset(dist, 0x3f, sizeof(dist));
  dist[s] = 0, q.push(make_pair(0, s));
  while (q.size()) {
    pii u = q.top();
    q.pop();
    if (dist[u.second] < u.first) continue;
    for (int i = h[u.second]; i; i = e[i].nex) {
      const int &v = e[i].t, &w = e[i].v;
      if (dist[v] <= dist[u.second] + w) continue;
      dist[v] = dist[u.second] + w;
      q.push(make_pair(dist[v], v));
    }
  }
}

int n, m, s;

int main() {
  scanf("%d%d%d", &n, &m, &s);
  for (int i = 1; i <= m; i++) {
```

```
    int u, v, w;
    scanf("%d%d%d", &u, &v, &w);
    add_path(u, v, w);
  }
  dijkstra(s);
  for (int i = 1; i <= n; i++) printf("%d ", dist[i]);
  return 0;
}
```

- Python

```
try:  # 引入优先队列模块
    import Queue as pq  # python version < 3.0
except ImportError:
    import queue as pq  # python3.*

N = int(1e5 + 5)
M = int(2e5 + 5)
INF = 0x3F3F3F3F


class qxx:  # 前向星类（结构体）
    def __init__(self):
        self.nex = 0
        self.t = 0
        self.v = 0


e = [qxx() for i in range(M)]  # 链表
h = [0 for i in range(N)]
cnt = 0

dist = [INF for i in range(N)]
q = pq.PriorityQueue()  # 定义优先队列，默认第一元小根堆


def add_path(f, t, v):  # 在前向星中加边
    # 如果要修改全局变量，要使用 global 来声名
    global cnt, e, h
    # 调试时的输出语句，多个变量使用元组
    # print("add_path(%d,%d,%d)" % (f,t,v))
    cnt += 1
    e[cnt].nex = h[f]
    e[cnt].t = t
    e[cnt].v = v
    h[f] = cnt


def nextedgeid(u):  # 生成器，可以用在 for 循环里
    i = h[u]
    while i:
        yield i
        i = e[i].nex


```

```
def dijkstra(s):
    dist[s] = 0
    q.put((0, s))
    while not q.empty():
        u = q.get()
        if dist[u[1]] < u[0]:
            continue
        for i in nextedgeid(u[1]):
            v = e[i].t
            w = e[i].v
            if dist[v] <= dist[u[1]] + w:
                continue
            dist[v] = dist[u[1]] + w
            q.put((dist[v], v))


# 如果你直接运行这个 Python 代码（不是模块调用什么的）就执行命令
if __name__ == "__main__":
    # 一行读入多个整数. 注意它会把整行都读进来
    n, m, s = map(int, input().split())
    for i in range(m):
        u, v, w = map(int, input().split())
        add_path(u, v, w)

    dijkstra(s)

    for i in range(1, n + 1):
        # 两种输出语法都是可以用的
        print("{}".format(dist[i]), end=" ")
        # print("%d" % dist[i],end=' ')

    print()  # 结尾换行
```

## 参考文档

1. Python Documentation[64]
2. Python 官方中文教程[65]
3. Learn Python3 In Y Minutes[66]
4. Real Python Tutorials[67]
5. 廖雪峰的 Python 教程[68]
6. GeeksforGeeks: Python Tutorials[69]

## 参考资料和注释

[1] 不被支持

[2] 2to3

[3] 列表生成

[4] 2. Python 解释器—Python 3 文档

[5] 北京交通大学自由与开源软件镜像站

[6] 华为开源镜像站

[7] Python 包索引（PyPI）

[8] pypi 镜像使用帮助 - 清华大学开源软件镜像站

[9] MirrorZ

[10] Python 文档 

[11] Python Wiki 

[12] **内置函数** 

[13] decimal 

[14] 基本输入输出 

[15] Unicode 

[16] Unicode 指南—Python 3 文档 

[17] 官方文档 

[18] 序列类型 

[19] 列表详解 

[20] 「列表推导式」 

[21] `deepcopy` 

[22] NumPy 

[23] `array` 

[24] 输入输出 

[25] `format` 函数 

[26] f-string 

[27] 这个网页 

[28] `zip()` 

[29] `zip()` 

[30] `map()` 

[31] 文件读写 

[32] `open()` 

[33] `with` 

[34] 控制流程 

[35] `:=` 

[36] 没有 switch 语句 

[37] try 块 

[38] EAFP 

[39] `try-except` 

[40] `dict` 

[41] 可哈希 

[42] `map()` 

[43] JSON 

[44] PyCharm 

[45] Mypy 

[46] 官方文档 

[47] `lru_cache` 

[48] 官方文档 

[49] `array` 

[50] `argparse` 

[51] `bisect` 

[52] `collections` 

[53] `fractions` 

[54] `heapq` 

[55] `io` 

[56] `itertools` 

[57] `math` 

[58] `os.path` 

[59] `random` 

[60] `re` 

[61] `struct` 

[62] `sys` 

[63] 例题 洛谷 P4779【模板】单源最短路径（标准版） 

[64] Python Documentation 

[65] Python 官方中文教程 

[66] Learn Python3 In Y Minutes 

[67] Real Python Tutorials 

[68] 廖雪峰的 Python 教程 

[69] GeeksforGeeks: Python Tutorials 

# 4.8 Java 速成

## 关于 Java

Java 是一种广泛使用的计算机编程语言，拥有 **跨平台**、**面向对象**、**泛型编程** 的特性，广泛应用于企业级 Web 应用开发和移动应用开发.

## 环境安装

参见 JDK→3.6@p241.

## 基本语法

### 主函数

Java 类似 C/C++ 语言，需要一个函数（在面向对象中，这被称为方法）作为程序执行的入口点.

Java 的主函数的格式是固定的，形如：

```
class Test {
    public static void main(String[] args) {
        // 程序的代码
    }
}
```

一个打包的 Java 程序（名称一般是 `*.jar`）中可以有很多个类似的函数，但是当运行这个程序的时候，只有其中一个函数会被运行，这是定义在 `Jar` 的 `Manifest` 文件中的，在 OI 比赛中一般用不到关于它的知识.

### 注释

和 C/C++ 一样，Java 使用 `//` 和 `/* */` 分别注释单行和多行.

### 基本数据类型

| 类型名 | 意义 |
| --- | --- |
| boolean | 布尔类型 |
| byte | 字节类型 |
| char | 字符型 |
| double | 双精度浮点 |
| float | 单精度浮点 |
| int | 整型 |
| long | 长整型 |
| short | 短整型 |
| null | 空 |

### 声明变量

```
int a = 12; // 设置 a 为整数类型,并给 a 赋值为 12
String str = "Hello, OI-wiki"; // 声明字符串变量 str
char ch = 'W';
double PI = 3.1415926;
```

### final 关键字

`final` 含义是这是最终的、不可更改的结果，被 `final` 修饰的变量只能被赋值一次，赋值后不再改变.

```
final double PI = 3.1415926;
```

### 数组

```
// 有十个元素的整数类型数组
// 其语法格式为 数据类型[] 变量名 = new 数据类型[数组大小]
int[] ary = new int[10];
```

### 字符串

- 字符串是 Java 一个内置的类.

```
// 最为简单的构造一个字符串变量的方法如下
String a = "Hello";

// 还可以使用字符数组构造一个字符串变量
char[] stringArray = { 'H', 'e', 'l', 'l', 'o' };
String s = new String(stringArray);
```

### 包和导入包

Java 中的类（`Class`）都被放在一个个包（`package`）里面. 在一个包里面不允许有同名的类. 在类的第一行通常要说明这个类是属于哪个包的. 例如:

```
package org.oi-wiki.tutorial;
```

包的命名规范一般是: `项目所有者的顶级域.项目所有者的二级域.项目名称`.

通过 `import` 关键字来导入不在本类所属的包下面的类. 例如下面要用到的 `Scanner`:

```
import java.util.Scanner;
```

如果想要导入某包下面所有的类，只需要把这个语句最后的分号前的类名换成 `*`.

### 输入

可以通过 `Scanner` 类来处理命令行输入.

```
package org.oiwiki.tutorial;

import java.util.Scanner;

class Test {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in); // System.in 是输入流
        int a = scan.nextInt();
        double b = scan.nextDouble();
        String c = scan.nextLine();
    }
}
```

### 输出

可以对变量进行格式化输出.

| 符号 | 意义 |
| --- | --- |
| %f | 浮点类型 |
| %s | 字符串类型 |
| %d | 整数类型 |
| %c | 字符类型 |

```
class Test {
    public static void main(String[] args) {
        int a = 12;
        char b = 'A';
        double s = 3.14;
        String str = "Hello world";
        System.out.printf("%f\n", s);
        System.out.printf("%d\n", a);
        System.out.printf("%c\n", b);
        System.out.printf("%s\n", str);
    }
}
```

## 控制语句

Java 的流程控制语句与 C++ 是基本相同的.

### 选择

- if

```
class Test {
    public static void main(String[] args) {
        if ( /* 判断条件 */ ){
            // 条件成立时执行这里面的代码
        }
    }
}
```

- if...else

```
class Test {
    public static void main(String[] args) {
        if ( /* 判断条件 */ ) {
            // 条件成立时执行这里面的代码
        } else {
            // 条件不成立时执行这里面的代码
        }
    }
}
```

- if...else if...else

```
class Test {
    public static void main(String[] args) {
        if ( /* 判断条件 */ ) {
            //判断条件成立执行这里面的代码
        } else if ( /* 判断条件2 */ ) {
            // 判断条件2成立执行这里面的代码
        } else {
```

```
            // 上述条件都不成立执行这里面的代码
        }
    }
}
```

- switch...case

```
class Test {
    public static void main(String[] args) {
        switch ( /* 表达式 */ ){
          case /* 值 1 */:
              // 当表达式取得的值符合值 1 执行此段代码
              break; // 如果不加上 break 语句,会让程序按顺序往下执行直到 break
          case /* 值 2 */:
              // 当表达式取得的值符合值 2 执行此段代码
              break;
          default:
              // 当表达式不符合上面列举的值的时候执行这里面的代码
        }
    }
}
```

### 循环

- for

`for` 关键字有两种使用方法，其中第一种是普通的 `for` 循环，形式如下：

```
class Test {
    public static void main(String[] args) {
        for ( /* 初始化 */; /* 循环的判断条件 */; /* 每次循环后执行的步骤 */ ) {
            // 当循环的条件成立执行循环体内代码
        }
    }
}
```

第二种是类似 C++ 的 `foreach` 使用方法，用于循环数组或者集合中的数据，相当于把上一种方式中的循环变量隐藏起来了，形式如下：

```
class Test {
    public static void main(String[] args) {
        for ( /* 元素类型X */ /* 元素名Y */ : /* 集合Z */ ) {
            // 这个语句块的每一次循环时，元素Y分别是集合Z中的一个元素.
        }
    }
}
```

- while

```
class Test {
    public static void main(String[] args) {
        while ( /* 判定条件 */ ) {
            // 条件成立时执行循环体内代码
        }
    }
}
```

- do...while

```java
class Test {
    public static void main(String[] args) {
        do {
          // 需要执行的代码
        } while ( /* 循环判断条件 */ );
    }
}
```

## 注意事项

### 类名与文件名一致

创建 Java 源程序需要类名和文件名一致才能编译通过，否则编译器会提示找不到类. 通常该文件名会在具体 OJ 中指定.

例:

`Add.java`

```java
class Add {
    public static void main(String[] args) {
        // ...
    }
}
```

在该文件中需使用 `Add` 为类名方可编译通过.

# 4.9 Java 进阶

> ⚠ **注意**
>
> 以下内容均基于 Java JDK 8 版本编写，不排除在更高版本中有部分改动的可能性.

## 更高速的输入输出

`Scanner` 和 `System.out.print` 在最开始会工作得很好，但是在处理更大的输入的时候会降低效率，因此我们会需要使用一些方法来提高 IO 速度.

### 使用 Kattio + StringTokenizer 作为输入

最常用的方法之一是使用来自 Kattis 的 Kattio.java[1] 来提高 IO 效率. [1]这个方法会将 `StringTokenizer` 与 `PrintWriter` 包装在一个类中方便使用. 而在具体进行解题的时候（假如赛会/组织方允许）可以直接使用这个模板.

下方即为应包含在代码中的 IO 模板，由于 Kattis 的原 Kattio 包含一些并不常用的功能，下方的模板经过了一些调整（原 Kattio 使用 MIT 作为协议）.

```java
class Kattio extends PrintWriter {
    private BufferedReader r;
    private StringTokenizer st;
    // 标准 IO
    public Kattio() { this(System.in, System.out); }
    public Kattio(InputStream i, OutputStream o) {
        super(o);
```

[1]Input & Output - USACO Guide[2]

```
        r = new BufferedReader(new InputStreamReader(i));
    }
    // 文件 IO
    public Kattio(String intput, String output) throws IOException {
        super(output);
        r = new BufferedReader(new FileReader(intput));
    }
    // 在没有其他输入时返回 null
    public String next() {
        try {
            while (st == null || !st.hasMoreTokens())
                st = new StringTokenizer(r.readLine());
            return st.nextToken();
        } catch (Exception e) {}
        return null;
    }
    public int nextInt() { return Integer.parseInt(next()); }
    public double nextDouble() { return Double.parseDouble(next()); }
    public long nextLong() { return Long.parseLong(next()); }
}
```

而下方代码简单展示了 Kattio 的使用：

```
class Test {
    public static void main(String[] args) {
        Kattio io = new Kattio();
        // 字符串输入
        String str = io.next();
        // int 输入
        int num = io.nextInt();
        // 输出
        io.println("Result");
        // 请确保关闭 IO 流以确保输出被正确写入
        io.close();
    }
}
```

### 使用 StreamTokenizer 作为输入

在某些情况使用 `StringTokenizer` 会导致 MLE（Memory Limit Exceeded，超过内存上限），此时我们需要使用 `StreamTokenizer` 作为输入.

```
import java.io.*;
public class Main {
    // IO 代码
    public static StreamTokenizer in = new StreamTokenizer(new
BufferedReader(new InputStreamReader(System.in), 32768));
    public static PrintWriter out = new PrintWriter(new
OutputStreamWriter(System.out));
    public static double nextDouble() throws IOException { in.nextToken();
return in.nval; }
    public static float nextFloat() throws IOException { in.nextToken(); return
(float)in.nval; }
    public static int nextInt() throws IOException { in.nextToken(); return
(int)in.nval; }
```

```
    public static String next() throws IOException { in.nextToken(); return in.sval; }
    public static long nextLong() throws Exception { in.nextToken(); return (long)in.nval;}

    // 使用示例
    public static void main(String[] args) throws Exception {
        int n = nextInt();
        out.println(n);
        out.close();
    }
}
```

### Kattio + StringTokenizer 的方法与 StreamTokenizer 的方法之间的分析与对比

1. `StreamTokenizer` 相较于 `StringTokenizer` 使用的内存较少，当 Java 标程 MLE 时可以尝试使用 `StreamTokenizer`，但是 `StreamTokenizer` 会丢失精度，读入部分数据时会出现问题；
    - `StreamTokenizer` 源码存在 `Type`，该 `Type` 根据输入内容来决定类型，如果输入类似于 `123oi` 以 **数字开头** 的字符串，他会强制认为的类型是 `double` 类型，因此在读入中以 `double` 类型去读 `String` 类型便会抛出异常；
    - `StreamTokenizer` 在读入 `1e14` 以上大小的数字会丢失精度；
2. 在使用 `PrintWriter` 情况下，需注意在程序结束最后 `close()` 关闭输出流或在需要输出的时候使用 `flush()` 清除缓冲区，否则内容将不会被写入到控制台/文件中.
3. `Kattio` 是继承自 `PrintWriter` 类，自身对象具有了 `PrintWriter` 的功能，因此可以直接调用 `PrintWriter` 类的函数输出，同时将 `StringTokenizer` 作为了自身的成员变量来修改. 而第二种 `Main` 是同时将 `StreamTokenizer` 与 `PrintWriter` 作为了自身的成员变量，因此在使用上有些许差距.

综上所述，在大部分情况下，`StringTokenizer` 的使用处境要优越于 `StreamTokenizer`，在极端 MLE 的情况下可以尝试 `StreamTokenizer`，同时 `int` 范围以上的数据 `StreamTokenizer` 处理是无能为力的.

## BigInteger 与数论

`BigInteger` 是 Java 提供的高精度计算类，可以很方便地解决高精度问题.

### 初始化

`BigInteger` 常用创建方式有如下二种:

```
import java.io.PrintWriter;
import java.math.BigInteger;

class Main {
    static PrintWriter out = new PrintWriter(System.out);
    public static void main(String[] args) {
        BigInteger a = new BigInteger("12345678910");  // 将字符串以十进制的形式创建 BigInteger 对象
        out.println(a);  // a 的值为 12345678910
        BigInteger b = new BigInteger("1E", 16);  // 将字符串以指定进制的形式创建 BigInteger 对象
        out.println(b);  // b 的值为 30
        out.close();
    }
}
```

```
14
```

## 基本运算

以下均用 `this` 代替当前 `BigIntger`:

| 函数名 | 功能 |
| --- | --- |
| `abs()` | 返回 `this` 的绝对值 |
| `negate()` | 返回 `this` 的相反数 |
| `add(BigInteger val)` | 返回 `this` 和 `val` 的和 |
| `subtract(BigInteger val)` | 返回 `this` 和 `val` 的差 |
| `multiply(BigInteger val)` | 返回 `this` 和 `val` 的积 |
| `divide(BigInteger val)` | 返回 `this` 和 `val` 的商 |
| `remainder(BigInteger val)` | 返回 `this` 除以 `val` 的余数 |
| `mod(BigInteger val)` | 返回 `this` 对 `val` 取模的值 |
| `pow(int val)` | 返回 `this` 的 `val` 次方 |
| `and(BigInteger val)` | 返回 `this` 和 `val` 的按位与 |
| `or(BigInteger val)` | 返回 `this` 和 `val` 的按位或 |
| `not()` | 返回 `this` 的按位取反 |
| `xor(BigInteger val)` | 返回 `this` 和 `val` 的按位异或 |
| `shiftLeft(int n)` | 返回 `this` 左移 `n` 位 |
| `shiftRight(int n)` | 返回 `this` 右移 `n` 位 |
| `max(BigInteger val)` | 返回 `this` 与 `val` 的较大值 |
| `min(BigInteger val)` | 返回 `this` 与 `val` 的较小值 |
| `bitCount()` | 返回 `this` 的二进制中不包括符号位的 `1` 的个数 |
| `bitLength()` | 返回 `this` 的二进制中不包括符号位的长度 |
| `getLowestSetBit()` | 返回 `this` 的二进制中最右边的位置 |
| `compareTo(BigInteger val)` | 比较 `this` 和 `val` 值大小 |
| `toString()` | 返回 `this` 的十进制字符串表示形式 |
| `toString(int radix)` | 返回 `this` 的 `raidx` 进制字符串表示形式 |

使用案例如下:

```java
import java.io.PrintWriter;
import java.math.BigInteger;

public class Main {
    static PrintWriter out = new PrintWriter(System.out);
    static BigInteger a, b;

    static void abs() {
        out.println("abs:");
        a = new BigInteger("-123");
        out.println(a.abs());  // 输出 123
        a = new BigInteger("123");
        out.println(a.abs());  // 输出 123
    }

```

```
16      static void negate() {
17          out.println("negate:");
18          a = new BigInteger("-123");
19          out.println(a.negate());  // 输出 123
20          a = new BigInteger("123");
21          out.println(a.negate());  // 输出 -123
22      }
23
24      static void add() {
25          out.println("add:");
26          a = new BigInteger("123");
27          b = new BigInteger("123");
28          out.println(a.add(b));  // 输出 246
29      }
30
31      static void subtract() {
32          out.println("subtract:");
33          a = new BigInteger("123");
34          b = new BigInteger("123");
35          out.println(a.subtract(b));  // 输出 0
36      }
37
38      static void multiply() {
39          out.println("multiply:");
40          a = new BigInteger("12");
41          b = new BigInteger("12");
42          out.println(a.multiply(b));  // 输出 144
43      }
44
45      static void divide() {
46          out.println("divide:");
47          a = new BigInteger("12");
48          b = new BigInteger("11");
49          out.println(a.divide(b));  // 输出 1
50      }
51
52      static void remainder() {
53          out.println("remainder:");
54          a = new BigInteger("12");
55          b = new BigInteger("10");
56          out.println(a.remainder(b));  // 输出 2
57          a = new BigInteger("-12");
58          b = new BigInteger("10");
59          out.println(a.remainder(b));  // 输出 -2
60      }
61
62      static void mod() {
63          out.println("mod:");
64          a = new BigInteger("12");
65          b = new BigInteger("10");
66          out.println(a.mod(b));  // 输出 2
67          a = new BigInteger("-12");
68          b = new BigInteger("10");
69          out.println(a.mod(b));  // 输出 8
```

```
70      }
71
72      static void pow() {
73          out.println("pow:");
74          a = new BigInteger("2");
75          out.println(a.pow(10));  // 输出 1024
76      }
77
78      static void and() {
79          out.println("and:");
80          a = new BigInteger("3");  // 11
81          b = new BigInteger("5");  // 101
82          out.println(a.and(b));  // 输出 1
83      }
84
85      static void or() {
86          out.println("or:");
87          a = new BigInteger("2");  // 10
88          b = new BigInteger("5");  // 101
89          out.println(a.or(b));  // 输出 7
90      }
91
92      static void not() {
93          out.println("not:");
94          a = new BigInteger("2147483647");  // 01111111 11111111 11111111
    11111111
95          out.println(a.not());  // 输出 -2147483648 二进制为: 10000000 00000000
    00000000 00000000
96      }
97
98      static void xor() {
99          out.println("xor:");
100         a = new BigInteger("6");  // 110
101         b = new BigInteger("5");  // 101
102         out.println(a.xor(b));  // 011 输出 3
103     }
104
105     static void shiftLeft() {
106         out.println("shiftLeft:");
107         a = new BigInteger("1");
108         out.println(a.shiftLeft(10));  // 输出 1024
109     }
110
111     static void shiftRight() {
112         out.println("shiftRight:");
113         a = new BigInteger("1024");
114         out.println(a.shiftRight(8));  // 输出 4
115     }
116
117     static void max() {
118         out.println("max:");
119         a = new BigInteger("6");
120         b = new BigInteger("5");
121         out.println(a.max(b));  // 输出 6
```

```
122     }
123
124     static void min() {
125         out.println("min:");
126         a = new BigInteger("6");
127         b = new BigInteger("5");
128         out.println(a.min(b));  // 输出 5
129     }
130
131     static void bitCount() {
132         out.println("bitCount:");
133         a = new BigInteger("6");  // 110
134         out.println(a.bitCount());  // 输出 2
135     }
136
137     static void bitLength() {
138         out.println("bitLength:");
139         a = new BigInteger("6");  // 110
140         out.println(a.bitLength());  // 输出 3
141     }
142
143     static void getLowestSetBit() {
144         out.println("getLowestSetBit:");
145         a = new BigInteger("8");  // 1000
146         out.println(a.getLowestSetBit());  // 输出 3
147     }
148
149     static void compareTo() {
150         out.println("compareTo:");
151         a = new BigInteger("8");
152         b = new BigInteger("9");
153         out.println(a.compareTo(b));  // 输出 -1
154         a = new BigInteger("8");
155         b = new BigInteger("8");
156         out.println(a.compareTo(b));  // 输出 0
157         a = new BigInteger("8");
158         b = new BigInteger("7");
159         out.println(a.compareTo(b));  // 输出 1
160     }
161
162     static void toStringTest() {
163         out.println("toString:");
164         a = new BigInteger("15");
165         out.println(a.toString());  // 输出 15
166         out.println(a.toString(16));  // 输出 f
167     }
168
169     public static void main(String[] args) {
170         abs();
171         negate();
172         add();
173         subtract();
174         multiply();
175         divide();
```

```
        remainder();
        mod();
        pow();
        and();
        or();
        not();
        xor();
        shiftLeft();
        shiftRight();
        max();
        min();
        bitCount();
        bitLength();
        getLowestSetBit();
        compareTo();
        toStringTest();
        out.close();
    }
}
```

## 数学运算

以下均用 `this` 代替当前 `BigIntger`:

| 函数名 | 功能 |
| --- | --- |
| `gcd(BigInteger val)` | 返回 `this` 的绝对值与 `val` 的绝对值的最大公约数 |
| `isProbablePrime(int val)` | 返回一个表示 `this` 是否是素数的布尔值 |
| `nextProbablePrime()` | 返回第一个大于 `this` 的素数 |
| `modPow(BigInteger b, BigInteger p)` | 返回 `this` 的 `b` 次方模 `p` 的值 |
| `modInverse(BigInteger p)` | 返回 `this` 在模 `p` 意义下的乘法逆元 |

使用案例如下:

```
import java.io.PrintWriter;
import java.math.BigInteger;

public class Main {
    static PrintWriter out = new PrintWriter(System.out);
    static BigInteger a, b, p;

    static void gcd() {  // 最大公约数
        a = new BigInteger("120032414321432144212100");
        b = new BigInteger("240231431243123412432140");
        out.println(String.format("gcd(%s,
%s)=%s", a.toString(), b.toString(), a.gcd(b).toString()));  //
gcd(120032414321432144212100,240231431243123412432140)=20
    }

    static void isPrime() {  // 基于米勒罗宾判定该数是否是素数，参数越大准确性越高，复杂度
越高. 准确性为 (1-1/(val*2))
        a = new BigInteger("1200324143214321442127");
        out.println("a:" + a.toString());
        out.println(a.isProbablePrime(10) ? "a is prime" : "a is not
prime");  // a is not prime
```

```
18      }
19
20      static void nextPrime() {  // 找出该数的下一个素数
21          a = new BigInteger("1200324143214321442127");
22          out.println("a:" + a.toString());
23          out.println(String.format("a nextPrime is %s",
a.nextProbablePrime().toString()));  // a nextPrime is 1200324143214321442199
24      }
25
26      static void modPow() {  // 快速幂，比正常版本要快，内部有数学优化
27          a = new BigInteger("2");
28          b = new BigInteger("10");
29          p = new BigInteger("1000");
30          out.println(String.format("a:%s b:%s p:%s", a, b, p));
31          out.println(String.format("a^b mod p:%s", a.modPow(b, p).toString()));//
24
32      }
33
34      static void modInverse() {  // 逆元
35          a = new BigInteger("10");
36          b = new BigInteger("3");
37          out.println(a.modInverse(b));  // a ^ (p-2) mod p = 1
38      }
39
40      public static void main(String[] args) {
41          gcd();
42          isPrime();
43          nextPrime();
44          modPow();
45          modInverse();
46          out.close();
47      }
48  }
```

关于米勒罗宾相关知识可以查阅 Miller–Rabin 素性测试→9.11.3@p1179.

# 基本数据类型与包装数据类型

## 简介

由于基本类型没有面向对象的特征，为了他们参加到面向对象的开发中，Java 为八个基本类型提供了对应的包装类，分别是 `Byte`、`Double`、`Float`、`Integer`、`Long`、`Short`、`Character` 和 `Boolean`. 两者之间的对应关系如下：

| 基本数据类型 | 包装数据类型 |
| --- | --- |
| byte | Byte |
| short | Short |
| boolean | Boolean |
| char | Character |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |

### 区别

此处以 `int` 与 `Integer` 举例：

1. `Integer` 是 `int` 的包装类，`int` 则是 Java 的一种基本类型数据.
2. `Integer` 类型实例后才能使用，而 `int` 类型不需要.
3. `Integer` 实际对应的引用，当 `new` 一个 `Integer` 时，实际上生成了一个对象，而 `int` 则是直接存储数据.
4. `Integer` 的默认值是 `null`，可接受 `null` 和 `int` 类型的数据，`int` 默认值是 0，不能接受 `null` 类型的数据.
5. `Integer` 判定二个变量是否相同使用 `==` 可能会导致不正确的结果，只能使用 `equals()`，而 `int` 可以直接使用 `==`.

### 装箱与拆箱

此处以 `int` 与 `Integer` 举例：

`Integer` 的本质是对象，`int` 是基本类型，两个类型之间是不能直接赋值的. 需要转换时，应将基础类型转换为包装类型，这种做法称为装箱，反过来则称为拆箱.

```
// 基本类型
int value1 = 1;
// 装箱转换为包装类型
Integer integer = Integer.valueOf(value1);
// 拆箱转换为基本类型
int value2 = integer.intValue();
```

Java 5 引入了自动装箱拆箱机制：

```
Integer integer = 1;
int value = integer;
```

> ⚠ **注意**
>
> 虽然 JDK 增加了自动装箱拆箱的机制，但在声明变量时请选择合适的类型，因为包装类型 `Integer` 可以接受 `null`，而基本类型 `int` 不能接受 `null`. 因此，对使用 `null` 值的包装类型进行拆箱操作时，会抛出异常. 如下代码展示了这一行为.
>
> ```
> Integer integer = Integer.valueOf(null);
> integer.intValue();  // 抛出 java.lang.NumberFormatException 异常
>
> Integer integer = null;
> integer.intValue();  // 抛出 java.lang.NullPointerException 异常
> ```

## 继承

基于已有的设计创造新的设计，就是面向对象程序设计中的继承. 在继承中，新的类不是凭空产生的，而是基于一个已经存在的类而定义出来的. 通过继承，新的类自动获得了基础类中所有的成员，包括成员变量和方法，包括各种访问属性的成员，无论是 `public` 还是 `private`. 显然，通过继承来定义新的类，远比从头开始写一个新的类要简单快捷和方便. 继承是支持代码重用的重要手段之一.

在 Java 中，继承的关键字为 `extends`，且 Java 只支持单继承，但可以实现多接口.

在 Java 中，所有类都是 `Object` 类的子类.

子类继承父类，所有的父类的成员，包括变量和方法，都成为了子类的成员，除了构造方法．构造方法是父类所独有的，因为它们的名字就是类的名字，所以父类的构造方法在子类中不存在．除此之外，子类继承得到了父类所有的成员．

每个成员有不同的访问属性，子类继承得到了父类所有的成员，但是不同的访问属性使得子类在使用这些成员时有所不同：有些父类的成员直接成为子类的对外的界面，有些则被深深地隐藏起来，即使子类自己也不能直接访问．

下表列出了不同访问属性的父类成员在子类中的访问属性：

| 父类成员访问属性 | 在父类中的含义 | 在子类中的含义 |
| --- | --- | --- |
| `public` | 对所有类开放 | 对所有类开放 |
| `protected` | 只有包内其它类、自己和子类可以访问 | 只有包内其它类、自己和子类可以访问 |
| 缺省（`default`） | 只有包内其它类可以访问 | 如果子类与父类在同一个包内，只有包内其它类可以访问；否则相当于 `private`，不能访问 |
| `private` | 只有自己可以访问 | 不能访问 |

# 多态

在 Java 中当把一个对象赋值给一个变量时，对象的类型必须与变量的类型相匹配．但由于 Java 有继承的概念，便可重新定义为 **一个变量可以保存其所声明的类型或该类型的任何子类型**．

如果一个类型实现了接口，也可以称之为该接口的子类型．

Java 中保存对象类型的变量是多态变量．「多态」这个术语（字面意思是许多形态）是指一个变量可以保存不同类型（即其声明的类型或任何子类型）的对象．

多态变量：

1. Java 的对象变量是多态的，它们能保存不止一种类型的对象．
2. 它们可以保存的是声明类型的对象，或声明类型子类的对象．
3. 当把子类的对象赋给父类的变量的时候，就发生了向上转型．

# 泛型

泛型指在类定义时不设置类中的属性或方法参数的具体类型，而是在使用（或创建对象）时再进行类型的定义．泛型本质是参数化类型，即所操作的数据类型被指定为一个参数．

泛型提供了编译时类型安全检测的机制，该机制允许编译时检测非法类型．

# 接口

## 简介

接口（Interface）在 Java 中是一个抽象类型，是抽象方法的集合，通常以 `interface` 来声明．一个类通过实现接口的方式，从而来继承接口的抽象方法．

接口并不是类，编写接口的方式和类很相似，但是它们属于不同的概念．类描述对象的属性和方法．接口则包含类要实现的方法．

除非实现接口的类是抽象类，否则该类要定义接口中的所有方法．

接口无法被实例化，但是可以被实现．一个实现接口的类，必须实现接口内所描述的所有方法，否则就必须声明为抽象类．另外，在 Java 中，接口类型可用来声明一个变量，他们可以成为一个空指针，或是被绑定在一个以此接口实现的对象．

## 与类的区别

1. 接口不能用于实例化对象．

2. 接口没有构造方法.
3. 接口中所有的方法必须是抽象方法，Java 8 之后接口中可以使用 `default` 关键字修饰的非抽象方法.
4. 接口不能包含成员变量，除了 static 和 final 变量.
5. 接口不是被类继承了，而是要被类实现.
6. 接口支持多继承，类不支持多继承.

### 声明

```
[可见度] interface 接口名称 [extends 其他的接口名] {
        // 声明变量
        // 抽象方法
}
```

### 实现

```
...implements 接口名称[, 其他接口名称, 其他接口名称..., ...] ...
```

## Lambda 表达式

### 简介

lambda 表达式也可称为闭包，是 Java 8 的最重要的新特性.

lambda 表达式允许把函数作为一个方法的参数（函数作为参数传递进方法中）.

使用 lambda 表达式可以使代码变的更加简洁紧凑.

### 语法

- 可选类型声明：不需要声明参数类型，编译器可以统一识别参数值.
- 可选的参数圆括号：一个参数无需定义圆括号，但多个参数需要定义圆括号.
- 可选的大括号：如果主体包含了一个语句，就不需要使用大括号.
- 可选的返回关键字：如果主体只有一个表达式返回值则编译器会自动返回值，大括号需要指定表达式返回了一个数值.

lambda 表达式声明方式如下：

```
// 1. 不需要参数，返回值为 5
() -> 5

// 2. 接收一个参数（数字类型），返回其 2 倍的值
x -> 2 * x

// 3. 接受 2 个参数（数字）并返回他们的差值
(x, y) -> x – y

// 4. 接收 2 个 int 类型整数并返回他们的和
(int x, int y) -> x + y

// 5. 接受一个 String 对象并在控制台打印，不返回任何值（看起来像是返回 void）
(String s) -> System.out.print(s)
```

以字符串数组按长度排序的自定义比较器为例，lambda 表达式可以按如下形式应用.

```
import java.util.Arrays;
```

```
2
3  public class Main {
4      static PrintWriter out = new PrintWriter(System.out);
5
6      public static void main(String[] args) {
7          String[] plants = {"Mercury", "venus", "Earth", "Mars", "Jupiter",
   "Saturn", "Uranus", "Neptune"};
8          Arrays.sort(plants, (String first, String second) -> (first.length()
   - second.length()));
9          for (String word : plants) {
10             out.print(word + " ");
11         }
12         out.close();
13     }
14 }
```

也可以类似下面的例子在 lambda 表达式中使用多条语句.

```
1  import java.io.PrintWriter;
2  import java.util.Arrays;
3
4  public class Main {
5      static PrintWriter out = new PrintWriter(System.out);
6
7      public static void main(String[] args) {
8          String[] plants = {"Mercury", "venus", "Earth", "Mars", "Jupiter",
   "Saturn", "Uranus", "Neptune"};
9          Arrays.sort(plants, (first, second) ->
10         {
11             // 形参不写类型，可以从上下文判断出
12             int result = first.length() - second.length();
13             return result;
14         });
15         for (String word : plants) {
16             out.print(word + " ");
17         }
18         out.close();
19     }
20 }
```

其中，`->`是一个推导符号，表示前面的括号接收到参数，推导后面的返回值（其实就是传递了方法）.

## 函数式接口

1. 是一个接口，符合 Java 接口定义.
2. 只包含一个抽象方法的接口.
3. 因为只有一个未实现的方法，所以 lambda 表达式可以自动填上去.

函数式接口使用方式如下:

**输出长度为 2 的倍数的字符串**

```
1  import java.io.PrintWriter;
2
3  public class Main {
```

```
    static PrintWriter out = new PrintWriter(System.out);

    public static void main(String[] args) {
        String[] plants = {"Mercury", "venus", "Earth", "Mars", "Jupiter",
"Saturn", "Uranus", "Neptune"};
        Test test = s -> {  // lambda 表达式作为函数式接口的实例
            if (s.length() % 2 == 0) {
                return true;
            }
            return false;
        };
        for (String word : plants) {
            if (test.check(word)) {
                out.print(word + " ");
            }
        }
        out.close();
    }
}

interface Test {
    public boolean check(String s);
}
```

### 实现加减乘除四则运算

```
import java.io.PrintWriter;

public class Main {
    static PrintWriter out = new PrintWriter(System.out);

    public static double calc(double a, double b, Calculator util) {
        return util.operation(a, b);
    }

    public static void main(String[] args) {
        Calculator util[] = new Calculator[4];  // 定义函数式接口数组
        util[0] = (a, b) -> a + b;
        util[1] = (a, b) -> a - b;
        util[2] = (a, b) -> a * b;
        util[3] = (a, b) -> a / b;
        double a = 20, b = 15;
        for (Calculator c : util) {
            System.out.println(calc(a, b, c));
        }
        out.close();
    }
}

interface Calculator {
    public double operation(double a, double b);
```

```
26 }
```

## Collection

`Collection` 是 Java 中的接口，被多个泛型容器接口所实现．在这里，`Collection` 是指代存放对象类型的数据结构．

Java 中的 `Collection` 元素类型定义时必须为对象，不能为基本数据类型．

以下内容用法均基于 Java 里多态的性质，均是以实现接口的形式出现．

常用的接口包括 `List`、`Queue`、`Set` 和 `Map`．

### 容器定义

当定义泛型容器类时，需要在定义时指定数据类型．如果不指定数据类型，而当成 `Object` 类型随意添加数据，在 Java 8 中虽能编译通过，但会有很多警告风险．

例如，如下定义方式是安全的，容器中只接受 `Integer` 类型．

```
1 List<Integer> list1 = new LinkedList<>();
```

而如下定义方式会出现警告．

```
1 List list = new ArrayList<>();
2 list.add(1);
3 list.add(true);
4 list.add(1.01);
5 list.add(1L);
6 list.add("I am String");
```

因此，如果没有特殊需求的话不推荐第 2 种行为，编译器无法帮忙检查存入的数据是否安全．`list.get(index)` 取值时无法明确数据的类型（取到的数据类型都为 `Object`），需要手动转回原来的类型，稍有不慎可能出现误转型异常．

如果是明确了类型如 `List<Integer>`，此时编译器会检查放入的数据类型，只能放入整数的数据．声明集合变量时只能使用包装类型 `List<Integer>` 或者自定义的 `Class`，而不能是基本类型如 `List<int>`．

### List

#### ArrayList

`ArrayList` 是支持可以根据需求动态生长的数组，初始长度默认为 10．如果超出当前长度便扩容 $\frac{3}{2}$．

#### 初始化

```
1 import java.io.PrintWriter;
2 import java.util.ArrayList;
3 import java.util.List;
4
5 public class Main {
6     static PrintWriter out = new PrintWriter(System.out);
7
8     public static void main(String[] args) {
9         List<Integer> list1 = new ArrayList<>();  // 创建一个名字为 list1 的可自增数组，初始长度为默认值（10）
```

```
        List<Integer> list2 = new ArrayList<>(30);  // 创建一个名字为list2的可自增数组，初始长度为 30
        List<Integer> list3 = new ArrayList<>(list2);  // 创建一个名字为 list3 的可自增数组，使用 list2 里的元素和 size 作为自己的初始值
    }
}
```

### LinkedList

`LinkedList` 是双链表.

### 初始化

```
import java.io.PrintWriter;
import java.util.LinkedList;
import java.util.List;

public class Main {
    static PrintWriter out = new PrintWriter(System.out);

    public static void main(String[] args) {
        List<Integer> list1 = new LinkedList<>();  // 创建一个名字为 list1 的双链表
        List<Integer> list2 = new LinkedList<>(list1);  // 创建一个名字为 list2 的双链表，将 list1 内所有元素加入进来
    }
}
```

### 常用方法

以下均用 `this` 代替当前 `List<Integer>`:

| 函数名 | 功能 |
| --- | --- |
| `size()` | 返回 `this` 的长度 |
| `add(Integer val)` | 在 `this` 尾部插入 `val` 元素 |
| `add(int idx, Integer e)` | 在 `this` 的 `idx` 位置插入 `e` 元素 |
| `get(int idx)` | 返回 `this` 中第 `idx` 位置的值，若越界则抛出异常 |
| `set(int idx, Integer e)` | 修改 `this` 中第 `idx` 位置的值为 `e` |

使用案例及区别对比:

```
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class Main {
    static PrintWriter out = new PrintWriter(System.out);
    static List<Integer> array = new ArrayList<>();
    static List<Integer> linked = new LinkedList<>();

    static void add() {
        array.add(1);  // 时间复杂度为 O(1)
        linked.add(1);  // 时间复杂度为 O(1)
    }

```

```
    static void get() {
        array.get(10);  // 时间复杂度为 O(1)
        linked.get(10);  // 时间复杂度为 O(11)
    }

    static void addIdx() {
        array.add(0, 2);  // 最坏情况下时间复杂度为 O(n)
        linked.add(0, 2);  // 最坏情况下时间复杂度为 O(n)
    }

    static void size() {
        array.size();  // 时间复杂度为 O(1)
        linked.size();  // 时间复杂度为 O(1)
    }

    static void set() {  // 该方法返回值为原本该位置元素的值
        array.set(0, 1);  // 时间复杂度为 O(1)
        linked.set(0, 1);  // 最坏时间复杂度为 O(n)
    }

}
```

### 遍历

```
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

public class Main {
    static PrintWriter out = new PrintWriter(System.out);
    static List<Integer> array = new ArrayList<>();
    static List<Integer> linked = new LinkedList<>();

    static void function1() {  // 朴素遍历
        for (int i = 0; i < array.size(); i++) {
            out.println(array.get(i));  // 遍历自增数组，复杂度为 O(n)
        }
        for (int i = 0; i < linked.size(); i++) {
            out.println(linked.get(i));  // 遍历双链表，复杂度为 O(n^2)，因为 LinkedList 的 get(i) 复杂度是 O(i)
        }
    }

    static void function2() {  // 增强 for 循环遍历
        for (int e : array) {
            out.println(e);
        }
        for (int e : linked) {
            out.println(e);  // 复杂度均为 O(n)
        }
    }

```

```
30      static void function3() {  // 迭代器遍历
31          Iterator<Integer> iterator1 = array.iterator();
32          Iterator<Integer> iterator2 = linked.iterator();
33          while (iterator1.hasNext()) {
34              out.println(iterator1.next());
35          }
36          while (iterator2.hasNext()) {
37              out.println(iterator2.next());
38          }  // 复杂度均为 O(n)
39      }
40
41  }
```

> ⚠ **注意**
>
> 不要在 `for` 或 `foreach` 遍历 `List` 的过程中删除其中的元素，否则会抛出异常.
>
> 原因也很简单，`list.size()` 改变了，但在循环中已循环的次数却是没有随之变化. 原来预计在下一个 `index` 的数据因为删除的操作变成了当前 `index` 的数据，运行下一个循环时操作的会变为原来预计在下下个 `index` 的数据，最终会导致操作的数据不符合预期.

## Queue

### LinkedList

可以使用 `LinkedList` 实现普通队列，底层是链表模拟队列.

#### 初始化

```
1  Queue<Integer> q = new LinkedList<>();
```

`LinkedList` 底层实现了 `List` 接口与 `Deque` 接口，而 `Deque` 接口继承自 `Queue` 接口，所以 `LinkedList` 可以同时实现 `List` 与 `Queue`.

### ArrayDeque

可以使用 `ArrayDeque` 实现普通队列，底层是数组模拟队列.

#### 初始化

```
1  Queue<Integer> q = new ArrayDeque<>();
```

`ArrayDeque` 底层实现了 `Deque` 接口，而 `Deque` 接口继承自 `Queue` 接口，所以 `ArrayDeque` 可以实现 `Queue`.

### LinkedList 与 ArrayDeque 在实现 Queue 接口上的区别

1. 数据结构：在数据结构上，`ArrayDeque` 和 `LinkedList` 都实现了 Java Deque 双端队列接口. 但 `ArrayDeque` 没有实现了 Java List 列表接口，所以不具备根据索引位置操作的行为.
2. 线程安全：`ArrayDeque` 和 `LinkedList` 都不考虑线程同步，不保证线程安全.
3. 底层实现：在底层实现上，`ArrayDeque` 是基于动态数组的，而 `LinkedList` 是基于双向链表的.
4. 在遍历速度上：`ArrayDeque` 是一块连续内存空间，基于局部性原理能够更好地命中 CPU 缓存行，而 `LinkedList` 是离散的内存空间对缓存行不友好.
5. 在操作速度上：`ArrayDeque` 和 `LinkedList` 的栈和队列行为都是 $O(1)$ 时间复杂度，`ArrayDeque` 的入栈和入队有可能会触发扩容，但从均摊分析上看依然是 $O(1)$ 时间复杂度.

6. 额外内存消耗上：`ArrayDeque` 在数组的头指针和尾指针外部有闲置空间，而 `LinkedList` 在节点上增加了前驱和后继指针.

### PriorityQueue

`PriorityQueue` 是优先队列，默认是小根堆.

#### 初始化

```
Queue<Integer> q1 = new PriorityQueue<>();  // 小根堆
Queue<Integer> q2 = new PriorityQueue<>((x, y) -> {return y - x;});  // 大根堆
```

#### 常用方法

下表中队列定义为 `Queue<Integer>`.

| 函数名 | 功能 |
| --- | --- |
| `size()` | 返回当前队列长度 |
| `add(Integer val)` | 将 `val` 插入队列，如果插入时违反了队列的容量限制，将抛出异常 |
| `offer(Integer val)` | 将 `val` 插入队列，如果插入时违反了队列的容量限制，则插入失败，但不会抛出异常 |
| `isEmpty()` | 判断队列是否为空，为空则返回 `true` |
| `peek()` | 返回队头元素，若队列为空返回 `null` |
| `poll()` | 返回并删除队头元素，若队列为空返回 `null` |

使用案例及区别对比：

```
import java.io.PrintWriter;
import java.util.LinkedList;
import java.util.PriorityQueue;
import java.util.Queue;

public class Main {
    static PrintWriter out = new PrintWriter(System.out);
    static Queue<Integer> q1 = new LinkedList<>();
    static Queue<Integer> q2 = new PriorityQueue<>();

    static void add() {  // add 和 offer 功能上没有差距，区别是是否会抛出异常
        q1.add(1);  // 时间复杂度为 O(1)
        q2.add(1);  // 时间复杂度为 O(logn)
    }

    static void isEmpty() {
        q1.isEmpty();  // 时间复杂度为 O(1)
        q2.isEmpty();  // 空间复杂度为 O(1)
    }

    static void size() {
        q1.size();  // 时间复杂度为 O(1)
        q2.size();  // 返回 q2 的长度
    }

    static void peek() {
        q1.peek();  // 时间复杂度为 O(1)
        q2.peek();  // 时间复杂度为 O(logn)
    }

```

```
31      static void poll() {
32          q1.poll();  // 时间复杂度为 O(1)
33          q2.poll();  // 时间复杂度为 O(logn)
34      }
35  }
```

### 遍历

```
1  import java.io.PrintWriter;
2  import java.util.LinkedList;
3  import java.util.PriorityQueue;
4  import java.util.Queue;
5
6  public class Main {
7      static PrintWriter out = new PrintWriter(System.out);
8      static Queue<Integer> q1 = new LinkedList<>();
9      static Queue<Integer> q2 = new PriorityQueue<>();
10
11     static void test() {
12         while (!q1.isEmpty()) {  // 复杂度为 O(n)
13             out.println(q1.poll());
14         }
15         while (!q2.isEmpty()) {  // 复杂度为 O(nlogn)
16             out.println(q2.poll());
17         }
18     }
19
20 }
```

## Deque

`Deque` 是 `Java` 中的双端队列，我们通常用其进行队列的操作以及栈的操作.

### 主要函数

下表中队列定义为 `Deque<Integer>`.

| 函数名 | 功能 |
|---|---|
| `addFirst(Integer val)` | 将 `val` 插入队头，如果插入时违反了队列的容量限制，将抛出异常 |
| `offerFirst(Integer val)` | 将 `val` 插入队头，如果插入时违反了队列的容量限制，则插入失败，但不会抛出异常 |
| `removeFirst()` | 返回并删除队头元素，如果队列为空，将抛出异常 |
| `pollFirst()` | 返回并删除队头元素，如果队列为空，则返回 `null` |
| `peekFirst()` | 返回队头元素，如果队列为空，则返回 `null` |
| `push(Integer val)` | 将 `val` 插入队头，等效于 `addFirst` |
| `pop()` | 返回并删除队头元素，等效于 `removeFirst` |
| `remove()` | 删除队头元素，等效于 `removeFirst` |
| `poll()` | 删除队头元素，等效于 `pollFirst` |
| `addLast(Integer val)` | 将 `val` 插入队尾，如果插入时违反了队列的容量限制，将抛出异常 |
| `offerLast(Integer val)` | 将 `val` 插入队尾，如果插入时违反了队列的容量限制，则插入失败，但不会抛出异常 |
| `removeLast()` | 返回并删除队尾元素，如果队列为空，将抛出异常 |
| `pollLast()` | 返回并删除队尾元素，如果队列为空，则返回 `null` |

| | |
|---|---|
| `peekLast()` | 返回队尾元素，如果队列为空，则返回 `null` |
| `add(Integer val)` | 将 `val` 插入队尾，等效于 `addLast` |
| `offer(Integer val)` | 将 `val` 插入队尾，等效于 `offerLast` |

### 栈的操作

```
import java.util.ArrayDeque;
import java.util.Deque;

public class Main {
    static Deque<Integer> stack = new ArrayDeque<>();
    static int[] a = {1, 2, 3, 4, 5};

    public static void main(String[] args) {
        for (int v : a) {
            stack.push(v);
        }
        while (!stack.isEmpty()) { //输出 5 4 3 2 1
            System.out.println(stack.pop());
        }
    }
}

```

### 双端队列的操作

```
import java.util.ArrayDeque;
import java.util.Deque;

public class Main {
    static Deque<Integer> deque = new ArrayDeque<>();

    static void insert() {
        deque.addFirst(1);
        deque.addFirst(2);
        deque.addLast(3);
        deque.addLast(4);
    }

    public static void main(String[] args) {
        insert();
        while (!deque.isEmpty()) { //输出 2 1 3 4
            System.out.println(deque.poll());
        }
        insert();
        while (!deque.isEmpty()) { //输出 4 3 1 2
            System.out.println(deque.pollLast());
        }
    }
}
```

## Set

`Set` 是保持容器中的元素不重复的一种数据结构.

### HashSet

随机位置插入的 `Set`.

#### 初始化

```
Set<Integer> s1 = new HashSet<>();
```

### LinkedHashSet

保持插入顺序的 `Set`.

#### 初始化

```
Set<Integer> s2 = new LinkedHashSet<>();
```

### TreeSet

保持容器中元素有序的 `Set`，默认为升序.

#### 初始化

```
Set<Integer> s3 = new TreeSet<>();
Set<Integer> s4 = new TreeSet<>((x, y) -> {return y - x;});  // 降序
```

#### TreeSet 的更多使用

这些方法是 `TreeSet` 新创建并实现的，我们无法使用 `Set` 接口调用以下方法，因此我们创建方式如下:

```
TreeSet<Integer> s3 = new TreeSet<>();
TreeSet<Integer> s4 = new TreeSet<>((x, y) -> {return y - x;});  // 降序
```

下表中均用 `this` 代替当前 `TreeSet<Integer>`.

| 函数名 | 功能 |
| --- | --- |
| `first()` | 返回 `this` 中第一个元素，无则返回 `null` |
| `last()` | 返回 `this` 中最后一个元素，无则返回 `null` |
| `floor(Integer val)` | 返回 `this` 中小于等于 `val` 的第一个元素，无则返回 `null` |
| `ceiling(Integer val)` | 返回 `this` 中大于等于 `val` 的第一个元素，无则返回 `null` |
| `higher(Integer val)` | 返回 `this` 中大于 `val` 的第一个元素，无则返回 `null` |
| `lower(Integer val)` | 返回 `this` 中小于 `val` 的第一个元素，无则返回 `null` |
| `pollFirst()` | 返回并删除 `this` 中第一个元素，无则返回 `null` |
| `pollLast()` | 返回并删除 `this` 中最后一个元素，无则返回 `null` |

代码示例:

```
import java.util.TreeSet;

public class Main {
    static int[] a = {4,7,1,2,3,6};

    public static void main(String[] args) {
        TreeSet<Integer> set = new TreeSet<>();
        for(int v:a) {
            set.add(v);
        }
        Integer a2 = set.first();
```

```
        System.out.println(a2); //返回 1
        Integer a3 = set.last();
        System.out.println(a3); //返回 7
        Integer a4 = set.floor(5);
        System.out.println(a4); //返回 4
        Integer a5 = set.ceiling(6);
        System.out.println(a5); //返回 6
        Integer a6 = set.higher(7);
        System.out.println(a6); //返回 null
        Integer a7 = set.lower(2);
        System.out.println(a7); //返回 1
        Integer a8 = set.pollFirst();
        System.out.println(a8); //返回 1
        Integer a9 = set.pollLast();
        System.out.println(a9); //返回 7
    }
}
```

### Set 常用方法

| 函数名 | 功能 |
| --- | --- |
| `size()` | 返回当前集合的大小 |
| `add(Integer val)` | 将 `val` 插入集合 |
| `contains(Integer val)` | 判断集合中是否有元素 `val` |
| `addAll(Collection e)` | 将容器 `e` 里的所有元素添加进当前集合 |
| `retainAll(Collection e)` | 删除当前集合中未出现在容器 `e` 中的元素，即求当前集合与 `e` 的交集 |
| `removeAll(Collection e)` | 删除当前集合中出现在容器 `e` 中的元素，即求当前集合与 `e` 的差集 |

```
import java.io.PrintWriter;
import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.Set;

public class Main {
    static PrintWriter out = new PrintWriter(System.out);
    static Set<Integer> s1 = new HashSet<>();
    static Set<Integer> s2 = new LinkedHashSet<>();

    static void add() {
        s1.add(1);
    }

    static void contains() {  // 判断 set 中是否有元素值为 2，有则返回 true，否则返回 false
        s1.contains(2);
    }

    static void test1() {  // s1 与 s2 的并集
        Set<Integer> res = new HashSet<>();
        res.addAll(s1);
        res.addAll(s2);
    }
```

```
24
25      static void test2() {  // s1 与 s2 的交集
26          Set<Integer> res = new HashSet<>();
27          res.addAll(s1);
28          res.retainAll(s2);
29      }
30
31      static void test3() {  // 差集: s1 - s2
32          Set<Integer> res = new HashSet<>();
33          res.addAll(s1);
34          res.removeAll(s2);
35      }
36  }
```

### 遍历

```
1  import java.io.PrintWriter;
2  import java.util.HashSet;
3  import java.util.LinkedHashSet;
4  import java.util.Set;
5
6  public class Main {
7      static PrintWriter out = new PrintWriter(System.out);
8      static Set<Integer> s1 = new HashSet<>();
9      static Set<Integer> s2 = new LinkedHashSet<>();
10
11     static void test() {
12         for (int key : s1) {
13             out.println(key);
14         }
15         out.close();
16     }
17 }
```

## Map

`Map` 是维护键值对 `<Key, Value>` 的一种数据结构，其中 `Key` 唯一.

### HashMap

随机位置插入的 `Map`.

#### 初始化

```
1  Map<Integer, Integer> map1 = new HashMap<>();
```

### LinkedHashMap

保持插入顺序的 `Map`.

#### 初始化

```
1  Map<Integer, Integer> map2 = new LinkedHashMap<>();
```

### TreeMap

保持 `key` 有序的 `Map`，默认升序.

### 初始化

```
Map<Integer, Integer> map3 = new TreeMap<>();
Map<Integer, Integer> map4 = new TreeMap<>((x, y) -> {return y - x;});  // 降序
```

### 常用方法

以下均用 `this` 代替当前 `Map<Integer, Integer>`:

| 函数名 | 功能 |
| --- | --- |
| `put(Integer key, Integer value)` | 将 `<key, value>` 插入 `this` |
| `size()` | 返回 `this` 的大小 |
| `containsKey(Integer key)` | 判断 `this` 中是否有存在某个元素的键为 `key` |
| `get(Integer key)` | 返回 `this` 中键为 `key` 的元素对应的值 |
| `keySet()` | 将 `this` 中所有元素的键作为集合返回 |

使用案例:

```
import java.io.PrintWriter;
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.TreeMap;

public class Main {
    static PrintWriter out = new PrintWriter(System.out);

    static Map<Integer, Integer> map1 = new HashMap<>();
    static Map<Integer, Integer> map2 = new LinkedHashMap<>();
    static Map<Integer, Integer> map3 = new TreeMap<>();
    static Map<Integer, Integer> map4 = new TreeMap<>((x,y)->{return y-x;});

    static void put(){  // 将 key 为 1、value 为 1 的元素返回
        map1.put(1, 1);
    }
    static void get(){  // 将 key 为 1 的 value 返回
        map1.get(1);
    }
    static void containsKey(){  // 判断是否有 key 为 1 的键值对
        map1.containsKey(1);
    }
    static void KeySet(){
        map1.keySet();
    }
}
```

### 遍历

```
import java.io.PrintWriter;
import java.util.HashMap;
import java.util.Map;

public class Main {
    static PrintWriter out = new PrintWriter(System.out);

```

```
    static Map<Integer, Integer> map1 = new HashMap<>();

    static void print() {
        for (int key : map1.keySet()) {
            out.println(key + " " + map1.get(key));
        }
    }
}
```

当然，键值的类型也可以更改．例如 `Map` 也可以定义为：

```
Map<String, Set<Integer>> map = new HashMap<>();
```

## Arrays

`Arrays` 是 `java.util` 中对数组操作的一个工具类．方法均为静态方法，可使用类名直接调用．

### Arrays.sort()

`Arrays.sort()` 是对数组进行的排序的方法，主要重载方法如下：

```
import java.util.Arrays;
import java.util.Comparator;

public class Main {
    static int[] a = new int[10];
    static Integer[] b = new Integer[10];
    static int firstIdx, lastIdx;

    public static void main(String[] args) {
        Arrays.sort(a);  // 1
        Arrays.sort(a, firstIdx, lastIdx);  // 2
        Arrays.sort(b, new Comparator<Integer>() {  // 3
            @Override
            public int compare(Integer o1, Integer o2) {
                return o2 - o1;
            }
        });
        Arrays.sort(b, firstIdx, lastIdx, new Comparator<Integer>() {  // 4
            @Override
            public int compare(Integer o1, Integer o2) {
                return o2 - o1;
            }
        });
        // 由于 Java 8 后有 Lambda 表达式，第三个重载及第四个重载亦可写为
        Arrays.sort(b, (x, y) -> {  // 5
            return y - x;
        });
        Arrays.sort(b, (x, y) -> {  // 6
            return y - x;
        });
    }
}
```

序号所对应的重载方法含义：

1. 对数组 a 进行排序，默认升序．

2. 对数组 `a` 的指定位置进行排序，默认升序，排序区间为左闭右开 `[firstIdx, lastIdx)`.
3. 对数组 `a` 以自定义的形式排序，第二个参数 `-` 第一个参数为降序，第一个参数 `-` 第二个参数为升序，当自定义排序比较器时，数组元素类型必须为对象类型.
4. 对数组 `a` 的指定位置进行自定义排序，排序区间为左闭右开 `[firstIdx, lastIdx)`，当自定义排序比较器时，数组元素类型必须为对象类型.
5. 和 3 同理，用 Lambda 表达式优化了代码长度.
6. 和 4 同理，用 Lambda 表达式优化了代码长度.

**`Arrays.sort()` 底层函数**

1. 当 `Arrays.sort` 的参数数组元素类型为基本数据类型（`byte`、`short`、`char`、`int`、`long`、`double`、`float`）时，默认为 `DualPivotQuicksort`（双轴快排），复杂度最坏可以达到 $O(n^2)$.
2. 当 `Arrays.sort` 的参数数组元素类型为非基本数据类型时，则默认为 `legacyMergeSort` 和 `TimSort`（归并排序），复杂度为 $O(n \log n)$.

可以通过如下代码验证：

**Codeforces 1646B - Quality vs Quantity**[3]

有 $n$ 个整数，你需要将其分为两组，是否能存在某一组的长度小于另一组，同时和大于它.

**例题代码**

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.util.Arrays;
import java.util.StringTokenizer;

public class Main {
    static class FastReader {
        StringTokenizer st;
        BufferedReader br;

        public FastReader() {
            br = new BufferedReader(new InputStreamReader(System.in));
        }

        String next() {
            while (st == null || !st.hasMoreElements()) {
                try {
                    st = new StringTokenizer(br.readLine());
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            return st.nextToken();
        }

        int nextInt() {
            return Integer.parseInt(next());
```

```
30          }
31
32          long nextLong() {
33              return Long.parseLong(next());
34          }
35
36          double nextDouble() {
37              return Double.parseDouble(next());
38          }
39
40          String nextLine() {
41              String str = "";
42              try {
43                  str = br.readLine();
44              } catch (IOException e) {
45                  e.printStackTrace();
46              }
47              return str;
48          }
49      }
50
51      static PrintWriter out = new PrintWriter(System.out);
52      static FastReader in = new FastReader();
53
54      static void solve() {
55          int n = in.nextInt();
56          // 此处数组类型由 Integer 修改为 int 会导致 TLE
57          Integer[] a = new Integer[n + 10];
58          for (int i = 1; i <= n; i++) {
59              a[i] = in.nextInt();
60          }
61          Arrays.sort(a, 1, n + 1);
62          long left = a[1];
63          long right = 0;
64          int x = n;
65          for (int i = 2; i < x; i++, x--) {
66              left = left + a[i];
67              right = right + a[x];
68              if (right > left) {
69                  out.println("YES");
70                  return;
71              }
72          }
73          out.println("NO");
74      }
75
76      public static void main(String[] args) {
77          int t = in.nextInt();
78          while (t-- > 0) {
79              solve();
80          }
81          out.close();
```

```
82     }
83 }
```

### Arrays.binarySearch()

`Arrays.binarySearch()` 是对数组连续区间进行二分搜索的方法，前提是数组必须有序，时间复杂度为 $O(\log_n)$，主要重载方法如下：

```
import java.util.Arrays;

public class Main {
    static int[] a = new int[10];
    static Integer[] b = new Integer[10];
    static int firstIdx, lastIdx;
    static int key;

    public static void main(String[] args) {
        Arrays.binarySearch(a, key);  // 1
        Arrays.binarySearch(a, firstIdx, lastIdx, key);  // 2
    }
}
```

源码如下：

```
private static int binarySearch0(int[] a, int fromIndex, int toIndex, int key) {
    int low = fromIndex;
    int high = toIndex - 1;

    while (low <= high) {
        int mid = (low + high) >>> 1;
        int midVal = a[mid];

        if (midVal < key)
            low = mid + 1;
        else if (midVal > key)
            high = mid - 1;
        else
            return mid; // key found
    }
    return -(low + 1);  // key not found.
}
```

序号所对应的重载方法含义：

1. 从数组 a 中二分查找是否存在 `key`，如果存在，便返回其下标．若不存在，则返回一个负数．
2. 从数组 a 中二分查找是否存在 `key`，如果存在，便返回其下标，搜索区间为左闭右开 `[firstIdx,lastIdx)`．若不存在，则返回一个负数．

### Arrays.fill()

`Arrays.fill()` 方法将数组中连续位置的元素赋值为统一元素．其接受的参数为数组、`fromIndex`、`toIndex` 和需要填充的数．方法执行后，数组左闭右开区间 `[firstIdx,lastIdx)` 内的所有元素的值均为需要填充的数．

## Collections

`Collections` 是 `java.util` 中对集合操作的一个工具类. 方法均为静态方法, 可使用类名直接调用.

### Collections.sort()

`Collections.sort()` 底层原理为将其中所有元素转化为数组调用 `Arrays.sort()`, 完成排序后再赋值给原本的集合. 又因为 Java 中 `Collection` 的元素类型均为对象类型, 所以始终是归并排序去处理.

该方法无法对集合指定区间排序.

底层源码:

```
default void sort(Comparator<? super E> c) {
    Object[] a = this.toArray();
    Arrays.sort(a, (Comparator) c);
    ListIterator<E> i = this.listIterator();
    for (Object e : a) {
        i.next();
        i.set((E) e);
    }
}
```

### Collections.binarySearch()

`Collections.binarySearch()` 是对集合中指定区间进行二分搜索, 功能与 `Arrays.binarySearch()` 相同.

```
Collections.binarySearch(list, key);
```

该方法无法对指定区间进行搜索.

### Collections.swap()

`Collections.swap()` 的功能是交换集合中指定二个位置的元素.

```
 Collections.swap(list, i, j);
```

# 其他

## 数值比较问题

在 Java 中, 如果单纯是数值类型, `-0.0 = 0.0`. 若是对象类型, 则 `-0.0 != 0.0`. 如果尝试用 `Set` 统计斜率数量时, 这个问题就会带来麻烦. 提供的解决方式是在所有的斜率加入 `Set` 前将值增加 `0.0`.

```
import java.io.PrintWriter;

public class Main {
    static PrintWriter out = new PrintWriter(System.out);

    static void A() {
        Double a = 0.0;
        Double b = -0.0;
        out.println(a.equals(b));  // false
    }

    static void B() {
        Double a = 0.0;
        Double b = -0.0 + 0.0;
```

```
        out.println(a.equals(b));  // true
    }

    static void C() {
        double a = 0.0;
        double b = -0.0;
        out.println(a == b);  // true
    }


    public static void main(String[] args) {
        A();
        B();
        C();
        out.close();
    }
}
```

## 参考资料

[1] Kattio.java



[2] Input & Output - USACO Guide



[3] Codeforces 1646B - Quality vs Quantity



# 第五章

# 算法基础

# 5.1 算法基础简介

**算法**，顾名思义，即计算的方法．算法通常用于解决特定的计算任务，但与可以直接在计算机上运行的程序不同，算法使用数学化的描述，更加侧重于思想，可以被看作抽象的程序．同一个算法可以有许多种不同的实现方式，两个不同的程序里也可能使用了同一种算法．

本章介绍一些基础算法．当一个计算任务足够通用，在各种问题中广泛出现时，解决该任务的算法就会被更多的算法所调用，拥有相当的重要性和基础性．同时，越通用的算法，通常越是简洁，越近似于思想而非程序．本章中介绍的算法具备足够的优美及趣味性，在之后的进阶内容中也常常会出现．

# 5.2 复杂度

## 5.2.1 复杂度简介

**Authors:** linehk, persdre

时间复杂度和空间复杂度是衡量一个算法效率的重要标准．

### 基本操作数

同一个算法在不同的计算机上运行的速度会有一定的差别，并且实际运行速度难以在理论上进行计算，实际去测量又比较麻烦，所以我们通常考虑的不是算法运行的实际用时，而是算法运行所需要进行的基本操作的数量．

在普通的计算机上，加减乘除、访问变量（基本数据类型的变量，下同）、给变量赋值等都可以看作基本操作．

对基本操作的计数或是估测可以作为评判算法用时的指标．

### 时间复杂度

#### 定义

衡量一个算法的快慢，一定要考虑数据规模的大小．所谓数据规模，一般指输入的数字个数、输入中给出的图的点数与边数等等．一般来说，数据规模越大，算法的用时就越长．而在算法竞赛中，我们衡量一个算法的效率时，最重要的不是看它在某个数据规模下的用时，而是看它的用时随数据规模而增长的趋势，即 **时间复杂度**．

#### 引入

考虑用时随数据规模变化的趋势的主要原因有以下几点：

1. 现代计算机每秒可以处理数亿乃至更多次基本运算，因此我们处理的数据规模通常很大．如果算法 A 在规模为 $n$ 的数据上用时为 $100n$ 而算法 B 在规模为 $n$ 的数据上用时为 $n^2$，在数据规模小于 100 时算法 B 用时更短，但在一秒钟内算法 A 可以处理数百万规模的数据，而算法 B 只能处理数万规模的数据．在允许算法执行时间更久时，时间复杂度对可处理数据规模的影响就会更加明显，远大于同一数据规模下用时的影响；
2. 我们采用基本操作数来表示算法的用时，而不同的基本操作实际用时是不同的，例如加减法的用时远小于除法的用时．计算时间复杂度而忽略不同基本操作之间的区别以及一次基本操作与十次基本操作之间的区别，可以消除基本操作间用时不同的影响．

当然，算法的运行用时并非完全由输入规模决定，而是也与输入的内容相关．所以，时间复杂度又分为几种，例如：

1. 最坏时间复杂度，即每个输入规模下用时最长的输入对应的时间复杂度．在算法竞赛中，由于输入可以在给定的数据范围内任意给定，我们为保证算法能够通过某个数据范围内的任何数据，一般考虑最坏时间复杂度；

2. 平均（期望）时间复杂度，即每个输入规模下所有可能输入对应用时的平均值的复杂度（给定输入概率分布下期望用时的复杂度；均匀平均是常见特例）.

所谓「用时随数据规模而增长的趋势」是一个模糊的概念，我们需要借助下文所介绍的 **渐近符号** 来形式化地表示时间复杂度.

## 渐近符号的定义

渐近符号是函数的阶的规范描述. 简单来说, 渐近符号忽略了一个函数中增长较慢的部分以及各项的系数（在时间复杂度相关分析中，系数一般被称作「常数」），而保留了可以用来表明该函数增长趋势的重要部分.

一个简单的记忆方法是，含等于（非严格）用大写，不含等于（严格）用小写，相等是 $\Theta$，小于是 $O$，大于是 $\Omega$. 大 $O$ 和小 $o$ 原本是希腊字母 Omicron，由于字形相同，也可以理解为拉丁字母的大 $O$ 和小 $o$.

在英文中，词根「-micro-」和「-mega-」常用于表示 10 的负六次方（百万分之一）和六次方（百万），也表示「小」和「大」. 小和大也是希腊字母 Omicron 和 Omega 常表示的含义.

### 大 Θ 符号

对于函数 $f(n)$ 和 $g(n)$，$f(n)=\Theta(g(n))$，当且仅当 $\exists c_1,c_2,n_0>0$，使得 $\forall n\geq n_0, 0\leq c_1\cdot g(n)\leq f(n)\leq c_2\cdot g(n)$.

也就是说, 如果函数 $f(n)=\Theta(g(n))$, 那么我们能找到两个正数 $c_1,c_2$ 使得 $f(n)$ 被 $c_1\cdot g(n)$ 和 $c_2\cdot g(n)$ 夹在中间.

例如，$3n^2+5n-3=\Theta(n^2)$，这里的 $c_1,c_2,n_0$ 可以分别是 $2,4,100$；$n\sqrt{n}+n\log^5 n+m\log m+nm=\Theta(n\sqrt{n}+m\log m+nm)$, 这是因为 $\log^5 n=o(\sqrt{n})$, 故存在足够大的 $n_0$, 使被省略项不超过 $n\sqrt{n}$.

### 大 O 符号

Θ 符号同时给了我们一个函数的上下界, 如果只知道一个函数的渐近上界而不知道其渐近下界, 可以使用 $O$ 符号. $f(n)=O(g(n))$，当且仅当 $\exists c,n_0>0$，使得 $\forall n\geq n_0, 0\leq f(n)\leq c\cdot g(n)$.

研究时间复杂度时通常会使用 $O$ 符号，因为我们关注的通常是程序用时的上界，而不关心其用时的下界.

需要注意的是，这里的「上界」和「下界」是对于函数的变化趋势而言的，而不是对算法而言的. 算法用时的上界对应的是「最坏时间复杂度」而非大 $O$ 记号. 所以，使用 Θ 记号表示最坏时间复杂度是完全可行的，甚至可以说 Θ 比 $O$ 更加精确，而使用 $O$ 记号的主要原因，一是我们有时只能证明时间复杂度的上界而无法证明其下界（这种情况一般出现在较为复杂的算法以及复杂度分析），二是 $O$ 在电脑上输入更方便一些.

### 大 Ω 符号

同样的，我们使用 Ω 符号来描述一个函数的渐近下界. $f(n)=\Omega(g(n))$，当且仅当 $\exists c,n_0>0$，使得 $\forall n\geq n_0, 0\leq c\cdot g(n)\leq f(n)$.

### 小 o 符号

如果说 $O$ 符号相当于小于等于号，那么 $o$ 符号就相当于小于号.

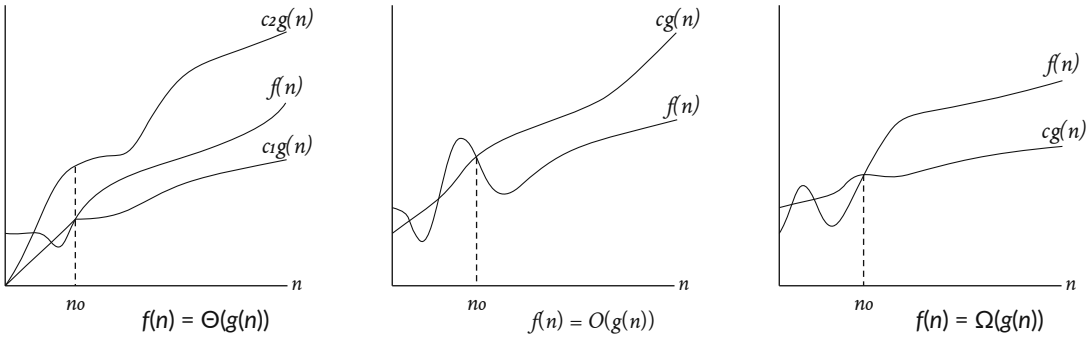
小 $o$ 符号大量应用于数学分析中, 函数在某点处的泰勒展开式拥有皮亚诺余项, 使用小 $o$ 符号表示严格小于，从而进行等价无穷小的渐近分析.

$f(n)=o(g(n))$，当且仅当对于任意给定的正数 $c$，$\exists n_0$，使得 $\forall n\geq n_0, 0\leq f(n)<c\cdot g(n)$.

### 小 ω 符号

如果说 Ω 符号相当于大于等于号，那么 $\omega$ 符号就相当于大于号.

$f(n)=\omega(g(n))$，当且仅当对于任意给定的正数 $c$，$\exists n_0$，使得 $\forall n\geq n_0, 0\leq c\cdot g(n)<f(n)$.




### 常见性质

- $f(n)=\Theta(g(n)) \iff f(n)=O(g(n)) \wedge f(n)=\Omega(g(n))$.
- $f_1(n)+f_2(n)=O(\max(f_1(n),f_2(n)))$.
- $f_1(n)\times f_2(n)=O(f_1(n)\times f_2(n))$.
- $\forall a>1, \log_a n=O(\log_2 n)$. 由换底公式可以得知，当底数为固定实数 $a>1$ 时，对数函数都具有相同的增长率，因此渐近时间复杂度中对数的底数一般省略不写.

## 简单的时间复杂度计算的例子

### `for` 循环

- C++

```cpp
int n, m;
std::cin >> n >> m;
for (int i = 0; i < n; ++i) {
  for (int j = 0; j < n; ++j) {
    for (int k = 0; k < m; ++k) {
      std::cout << "hello world\n";
    }
  }
}
```

- Python

```python
n = int(input())
m = int(input())
for i in range(0, n):
    for j in range(0, n):
        for k in range(0, m):
            print("hello world")
```

- Java

```java
int n, m;
n = input.nextInt();
m = input.nextInt();
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        for (int k = 0; k < m; ++k) {
            System.out.println("hello world");
        }
    }
```

```
10 }
```

如果以输入的数值 $n$ 和 $m$ 的大小作为数据规模，则上面这段代码的时间复杂度为 $\Theta(n^2m)$.

### DFS

在对以邻接表存储的、$n$ 个点 $m$ 条边的 连通图→11.2@p2598 进行 DFS→11.4@p2612 时，由于每个节点和每条边都只会被访问常数次，复杂度为 $\Theta(n+m)$.

## 哪些量是常量

当我们要进行若干次操作时，如何判断这若干次操作是否影响时间复杂度呢？例如：

- C++

```cpp
constexpr int N = 100000;
for (int i = 0; i < N; ++i) {
  std::cout << "hello world\n";
}
```

- Python

```python
N = 100000
for i in range(0, N):
    print("hello world")
```

- Java

```java
final int N = 100000;
for (int i = 0; i < N; ++i) {
    System.out.println("hello world");
}
```

如果 $N$ 的大小不被看作输入规模，那么这段代码的时间复杂度就是 $O(1)$.

进行时间复杂度计算时，哪些变量被视作输入规模是很重要的，而所有和输入规模无关的量都被视作常量，计算复杂度时可当作 1 来处理.

需要注意的是，在进行时间复杂度相关的理论性讨论时，「算法能够解决任何规模的问题」是一个基本假设（当然，在实际中，由于时间和存储空间有限，无法解决规模过大的问题）. 因此，能在常量时间内解决数据规模有限的问题（例如，对于数据范围内的每个可能输入预先计算出答案）并不能使一个算法的时间复杂度变为 $O(1)$.

## 主定理

我们可以使用主定理（Master theorem）来快速求得关于递归算法的复杂度. 假设 $a \geq 1$、$b > 1$ 为常数，$f$ 非负，$T(1) = \Theta(1)$. 为简化讨论，下面额外假设 $n = b^L$，其中 $L$ 为非负整数. 其递推关系式如下：

$$T(n) = aT\left(\frac{n}{b}\right) + f(n), \qquad \forall n = b^L, L \geq 1.$$

那么

$$T(n) = \begin{cases} \Theta(n^{\log_b a}), & f(n) = O(n^{\log_b(a)-\epsilon}), \epsilon > 0, \\ \Theta(f(n)), & f(n) = \Omega(n^{\log_b(a)+\epsilon}), \epsilon \geq 0, \\ \Theta(n^{\log_b a}\log^{k+1} n), & f(n) = \Theta(n^{\log_b a}\log^k n), k \geq 0. \end{cases}$$

需要注意的是，这里的第二种情况还需要满足正则条件（regularity condition），即存在常数 $0 < c < 1$，使得对足够大的 $n$ 有 $af(n/b) \leq cf(n)$.

证明思路是将规模为 $n$ 的问题，分解为 $a$ 个规模为 $n/b$ 的问题，然后依次合并，直到合并到最高层. 每一次合并子问题，都需要花费 $f(n)$ 的时间.

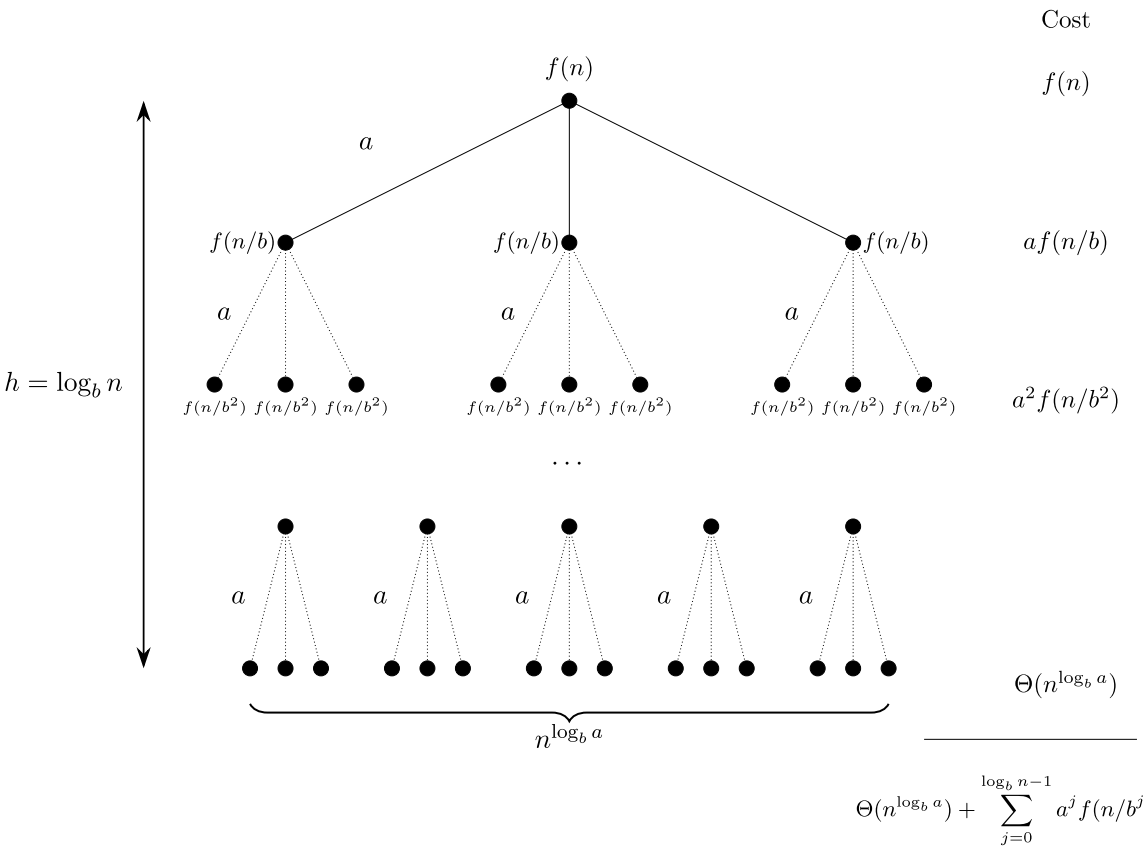
**证明**

依据上文提到的证明思路，具体证明过程如下：

对于第 0 层（最高层），合并子问题需要花费 $f(n)$ 的时间.

对于第 1 层（第一次划分出来的子问题），共有 $a$ 个子问题，每个子问题合并需要花费 $f\left(\frac{n}{b}\right)$ 的时间，所以合并总共要花费 $af\left(\frac{n}{b}\right)$ 的时间.

层层递推，我们可以画出如下递归树：




这棵树在第 $L = \log_b n$ 层到达终点，共有 $a^L = n^{\log_b a}$ 个叶子. 非叶层的编号为 $0, ..., L-1$，从而 $T(n) = \Theta(n^{\log_b a}) + g(n)$，其中 $g(n) = \sum_{j=0}^{L-1} a^j f(n/b^j)$.

对于第一种情况：$f(n) = O(n^{\log_b a - \epsilon})$，因此 $g(n) = O(n^{\log_b a})$.

对于第二种情况，首先根节点的成本 $T(n) = \Omega(f(n))$. 由正则条件，在子问题规模仍足够大时，第 $j$ 层的总成本至多为 $c^j f(n)$，这些层的成本之和为 $O(f(n))$. 剩余的底部常数层和叶子的总成本为 $O(n^{\log_b a}) = O(f(n))$，因此 $T(n) = \Theta(f(n))$.

对于第三种情况，令 $p = \log_b a$. 对规模足够大的子问题，第 $j$ 层的总成本为 $\Theta(n^p (L-j)^k)$，其中 $k \geq 0$ 为固定常数. 取固定的 $r_0 \geq 1$，使 $b^{r_0}$ 能达到该规模，由 $\sum_{r=r_0}^{L} r^k = \Theta(L^{k+1})$ 可知这些层的总成本为 $\Theta(n^p \log^{k+1} n)$. 其余底部常数层和叶子的总成本为 $O(n^p)$，加上后即得结论.

下面举几个例子来说明主定理如何使用.

1. $T(n) = 2T\left(\frac{n}{2}\right) + 1$，那么 $a = 2, b = 2, \log_2 2 = 1$，那么 $\epsilon$ 可以取值在 $(0, 1]$ 之间，从而满足第一种情况，所以 $T(n) = \Theta(n)$；
2. $T(n) = T\left(\frac{n}{2}\right) + n$，那么 $a = 1, b = 2, \log_2 1 = 0$，那么 $\epsilon$ 可以取值在 $(0, 1]$ 之间，从而满足第二种情况，所以 $T(n) = \Theta(n)$；

3. $T(n)=T\left(\frac{n}{2}\right)+\log n$，那么 $a=1,b=2,\log_2 1=0$，那么 $k$ 可以取值为 1，从而满足第三种情况，所以 $T(n)=\Theta(\log^2 n)$；
4. $T(n)=T\left(\frac{n}{2}\right)+1$，那么 $a=1,b=2,\log_2 1=0$，那么 $k$ 可以取值为 0，从而满足第三种情况，所以 $T(n)=\Theta(\log n)$.

### 均摊复杂度

详情可见 均摊复杂度→5.2.2@p549.

### 空间复杂度

类似地，算法所使用的空间随输入规模变化的趋势可以用 **空间复杂度** 来衡量.

### 计算复杂性

本文主要从算法分析的角度对复杂度进行了介绍，如果有兴趣的话可以在 计算复杂性→13.9@p3306 进行更深入的了解.

## 5.2.2 均摊复杂度

前置知识：时间复杂度→5.2.1@p544

本页面将介绍均摊复杂度的基础知识.

### 引入

均摊分析（Amortized Analysis）是一种用于分析算法和动态数据结构性能的技术. 它不仅仅关注单次操作的成本, 还通过评估一系列操作的平均成本, 为整体性能提供更加准确的评估. 均摊分析不涉及概率, 且只能确保最坏情况性能的每次操作耗费的平均时间，并不能确认系统的平均性能. 在最坏情况下，均摊分析通过将高成本操作的开销分摊到低成本操作上，确保整体操作的平均成本保持在合理范围内.

均摊分析通常采用三种主要分析方法：聚合分析、记账分析和势能分析. 这些方法各有侧重，分别适用于不同的场景，但它们的共同目标是通过均衡操作成本，优化数据结构在最坏情况下的整体性能表现.

### 内容

考虑一个可扩展的数组，例如 C++ 中的 `vector`，其初始容量为 $m=1$. 每次插入新元素时，如果数组已满，则需要将数组的大小加倍，然后将原数组中的元素复制到新数组中，最后插入新元素.

接下来，将以动态数组的插入操作为例，通过聚合分析、记账分析和势能分析三种方法，分析其均摊成本.

#### 聚合分析

聚合分析（Aggregate Analysis）通过计算一系列操作的总成本，并将其平均到每次操作上，从而得出每次操作的均摊时间复杂度.

以动态数组为例，首先，可以得到插入操作的两个关键成本：

- 如果数组未满，插入操作的成本为 $O(1)$.
- 如果数组已满, 则插入操作需要扩容, 扩容后复制元素的成本为 $O(m)$, 其中 $m$ 为当前数组的大小.

所以，为了计算 $n$ 次插入操作的总成本，可以将其分开为两部分计算：

1. **插入操作的成本**：每次插入新元素的直接成本是常数时间 $O(1)$，对于 $n$ 次操作，总成本是 $O(n)$；
2. **数组扩容的成本**：每次扩容涉及到复制原数组元素到新数组. 这些操作发生在数组大小为 2 的幂次时. 扩容时复制的旧容量为所有严格小于 $n$ 的二次幂之和，当 $n\geq 1$ 时，总复制成本为 $\sum_{i=0}^{\lceil\log_2 n\rceil-1}2^i=2^{\lceil\log_2 n\rceil}-1<2n$，总成本是 $O(n)$.

因此，该数组总的插入成本为 $O(n)$，均摊到每次操作的成本为 $O(1)$. 即使在最坏情况下，平均每次插入操作的成本依然是常数时间.

### 记账分析

记账法（Accounting Method）通过为每次操作预先分配一个固定的均摊成本来确保所有操作的总成本不超过这些预分配的成本总和. 记账法类似于一种 **费用前置支付** 的机制，其中较低成本的操作会存储部分费用，以支付未来高成本的操作.

以动态数组为例, 可以为每次插入操作分配一个固定的均摊成本, 以确保在需要扩容时已经预留了足够的费用.

1. **费用分配**:
    - 假设每次插入操作的实际成本为 1，均摊成本设为 3.
    - 其中 1 用于当前插入操作，2 用于未来可能的扩容操作.
2. **费用使用**:
    - 当数组已满时，需要进行扩容操作，实际成本为 $O(m)$，其中 $m$ 是当前数组的大小.
    - 假设扩容前数组的元素数量为 $n$, 由于原数组的后半部分 $n/2$ 个元素在插入时共预存了 $n$ 单位的均摊成本，恰好足够支付扩容操作的成本.

以下是一个具体的示例:

```
初始状态:
arr    = [1, 2, 3, 4]  // 初始数组
amount = [2, 2, 2, 2]  // 每个元素预存的费用

// 第一轮扩容：数组已满，需要扩容
arr    = [1, 2, 3, 4, null, null, null, null]  // 扩容后数组
amount = [2, 2, 0, 0, 0, 0, 0, 0]  // 3, 4的费用用于支付扩容

// 继续插入新元素，直至再次满载
arr    = [1, 2, 3, 4, 5, 6, 7, 8]  // 继续填充数组
amount = [2, 2, 0, 0, 2, 2, 2, 2]  // 新插入的元素同样预存费用

// 第二轮扩容：数组再次满载，需要更大的空间
arr    = [1, 2, 3, 4, 5, 6, 7, 8, null, null, null, null, null, null, null, null]  // 扩容后数组
amount = [2, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]  // 5, 6, 7, 8的费用用于支付扩容
```

以上过程表明, 每次插入操作所存储的均摊成本足够支付未来的扩容操作, 从而确保了每次操作的均摊成本维持在 $O(1)$.

### 势能分析

势能分析（Potential Method）通过定义一个势能函数（通常表示为 $\Phi$），度量数据结构的 **潜在能量**，即系统状态中的预留资源, 这些资源可以用来支付未来的高成本操作. 势能的变化用于平衡操作序列的总成本，从而确保整个算法的均摊成本在合理范围内.

#### 原理

首先，定义 **状态** $S$ 为某一时刻数据结构的状态，该状态可能包含元素数量、容量、指针等信息，其中定义初始状态为 $S_0$，即未进行任何操作时的状态.

其次，定义势能函数 $\Phi(S)$ 用于度量数据结构状态 $S$ 的势能，设初始状态为 $S_0$，则所有可达状态 $S$ 的势能均需满足 $\Phi(S) \geq \Phi(S_0)$[1][2][3]，有的资料会额外要求 $\Phi(S_0) = 0$[4]，实际上是不必要的.

---

[1] Clemson University - Amortized Analysis + Splay Trees[1]
[2] UC Irvine - Amortized Analysis[2]
[3] Stanford CS166 - Amortized Analysis[3]
[4] Cornell CS 3110 - Lecture 20: Amortized Analysis[4]

对于每个操作，其均摊成本 $\hat{c}$ 定义为：

$$\hat{c} = c + \Phi(S') - \Phi(S)$$

其中 $c$ 为操作的实际成本，$S$ 和 $S'$ 分别表示操作前后的数据结构状态．该公式表明，均摊成本等于实际成本加上势能的变化．如果操作增加了势能（即 $\Phi(S') > \Phi(S)$），则均摊成本上升；如果操作消耗了势能（即 $\Phi(S') < \Phi(S)$），则均摊成本下降．

我们可以通过势能函数来分析一系列操作的总成本．设 $S_1, S_2, ..., S_m$ 为从初始状态 $S_0$ 开始，经过 $m$ 次操作后产生的状态序列，$c_i$ 为第 $i$ 次操作的实际开销，那么第 $i$ 次操作的均摊成本 $p_i$ 为：

$$p_i = c_i + \Phi(S_i) - \Phi(S_{i-1})$$

因此，$m$ 次操作的总时间花销为：

$$\sum_{i=1}^{m} c_i = \sum_{i=1}^{m} p_i + \Phi(S_0) - \Phi(S_m)$$

由于 $\Phi(S) \geq \Phi(S_0)$，总时间花销的上界为：

$$\sum_{i=1}^{m} p_i \geq \sum_{i=1}^{m} c_i$$

因此，若 $p_i = O(T(n))$，则 $O(T(n))$ 是均摊复杂度的一个上界．

**示例：动态数组的扩容分析**

以动态数组 `vector` 的插入操作为例，记初始状态为 $h_0$，定义如下的势能函数 $\Phi(h)$：

$$\Phi(h) = 2n - m$$

其中，$n$ 是数组中的元素数量，$m$ 是数组的当前容量．这个势能函数的变化反映了用于支付未来扩容的预存费用的增减．在容量为 1 的空数组初态，$\Phi(h_0) = -1$；在仅插入、满时倍增的非空可达状态中，$m \leq 2n$，故 $\Phi(h) \geq 0 > \Phi(h_0)$．

1. **插入操作（无需扩容）**：
    - **操作成本**：$O(1)$，因为只需插入一个元素．
    - **势能变化**：插入后，元素数量增加 1，势能增加 2．
        - $\Phi(h') - \Phi(h) = 2(n+1) - m - (2n - m) = 2$．
    - **均摊成本**：$1 + 2 = 3$．
2. **插入操作（触发扩容）**：
    - 假设当前容量 $m = n$，插入一个新元素时触发扩容，新的容量变为 $2n$．
    - **操作成本**：$O(n)$，因为需要将所有元素复制到新数组中，并插入新元素．
    - **势能变化**：扩容并插入后的势能变化为 $2 - n$，当 $n > 2$ 时才下降．
        - $\Phi(h') - \Phi(h) = 2(n+1) - 2n - (2n - n) = 2 - n$．
    - **均摊成本**：$n + 1 + (2 - n) = 3$．

通过上述分析可以看出，尽管扩容操作的实际成本较高，但由于势能函数的设计，整体均摊成本仍然保持在常数级别 $O(1)$．

## 扩展示例：堆栈操作

堆栈操作是均摊分析的经典应用场景之一．假设从空栈开始，`pop` 仅用于非空栈，堆栈 `S` 支持以下三种操作：

| 操作 | 说明 | 实际成本 $c_i$ |
|---|---|---|
| `S.push(x)` | 将元素 $x$ 入栈 | 1 |
| `S.pop()` | 弹出栈顶元素 | 1 |
| `S.multi-pop(k)` | 最多弹出栈顶 $k$ 个元素 | $O(\min(\|S\|, k))$ |

我们将通过聚合分析、记账分析和势能分析三种方法来分析这些堆栈操作的均摊成本．

### 聚合分析法

聚合分析将计算所有操作的总成本，并将其平均分摊到每个操作上，从而得出均摊成本.

1. 对于 $n_{\text{push}}$ 次 `push(x)` 操作，每次的成本为 $O(1)$，因此总成本为 $O(n_{\text{push}})$；
2. 对于 $n_{\text{pop}}$ 次 `pop()` 操作，每次的成本为 $O(1)$，总成本为 $O(n_{\text{pop}})$；
3. 对于 $n_{\text{multi-pop}}$ 次 `multi-pop(k)` 操作，尽管每次的实际成本为 $O(\min(|S|,k))$，但这些操作弹出的元素数量不会超过之前 `push(x)` 的元素数量，因此总成本仍受 $n_{\text{push}}$ 的约束.

总操作次数为 $n = n_{\text{push}} + n_{\text{pop}} + n_{\text{multi-pop}}$，且 $n_{\text{push}} \leq n$. 每个元素最多入栈和出栈各一次，故按元素计费的总成本为 $O(n_{\text{push}}) = O(n)$；即使另计每次调用的常数检查开销，总成本仍为 $O(n)$，均摊为 $O(1)$.

### 记账分析法

记账分析为每次 `push(x)` 操作预留一部分费用，以支付未来可能的 `pop()` 或 `multi-pop(k)` 操作.

1. **`S.push(x)`**：假设每次 `push(x)` 操作的均摊成本为 2，其中 1 单位用于当前操作，另 1 单位存储为费用，用于支付未来的 `pop()` 或 `multi-pop(k)` 操作；
2. **`S.pop()`**：实际成本为 1，但由于之前的 `push(x)` 操作已为其预存了 1 单位费用，因此均摊成本为 0；
3. **`S.multi-pop(k)`**：每个弹出的元素的实际成本为 1，可以由之前该元素的 `push(x)` 操作预存的费用支付，因此均摊成本为 0.

通过以上分析，入栈操作预存的费用足以支付未来该元素的出栈操作，因此每次操作的均摊成本为 $O(1)$.

### 势能分析法

势能分析定义了一个势能函数来衡量堆栈的状态，并利用势能的变化来平衡操作成本.

1. **势能函数**：设 $\Phi(h)$ 为堆栈中的元素数量，即 $\Phi(h) = |S|$. 每个元素贡献 1 单位的势能；
2. **`S.push(x)`**：每次 `push(x)` 操作增加堆栈中的元素数量，势能增加 1，因此均摊成本为 $1 + 1 = 2$；
3. **`S.pop()`**：每次 `pop()` 操作减少堆栈中的元素数量，势能减少 1，因此均摊成本为 $1 - 1 = 0$；
4. **`S.multi-pop(k)`**：令 $t = \min(|S|, k)$，`multi-pop(k)` 实际弹出 $t$ 个元素，势能减少 $t$，按元素计费的均摊成本为 $t - t = 0$；计入调用开销则为 $O(1)$.

通过以上势能函数设计，`push(x)` 操作的均摊成本为 2，而 `pop()` 和 `multi-pop(k)` 操作的均摊成本为 0. 因此，所有堆栈操作的均摊成本均为 $O(1)$.

### 参考资料与注释

- Amortized Analysis - Wikipedia[5]

[1] Clemson University - Amortized Analysis + Splay Trees

[2] UC Irvine - Amortized Analysis

[3] Stanford CS166 - Amortized Analysis

[4] Cornell CS 3110 - Lecture 20: Amortized Analysis

[5] Amortized Analysis - Wikipedia

## 5.3 枚举

**Authors:** Early0v0, frank-xjh, Great-designer, ksyx, qiqistyle, Tiphereth-A , Saisyc, shuzhouliu, Xeonacid, xyf007

本页面将简要介绍枚举算法.

## 简介

枚举（Enumerate）是基于已有知识来猜测答案的一种问题求解策略.

枚举的思想是不断地猜测，从可能的集合中一一尝试，然后再判断题目的条件是否成立.

# 要点

## 给出解空间

建立简洁的数学模型.

枚举的时候要想清楚：可能的情况是什么？要枚举哪些要素?

## 减少枚举的空间

枚举的范围是什么？是所有的内容都需要枚举吗?

在用枚举法解决问题的时候，一定要想清楚这两件事，否则会带来不必要的时间开销.

## 选择合适的枚举顺序

根据题目判断. 例如，若题目要求最大的符合条件的素数，那自然是从大到小枚举比较合适.

# 例题

以下是一个使用枚举解题与优化枚举范围的例子.

**例题**

给定一个数组，其所有元素互不相同且均不为 0. 求该数组中和为 0 的有序数对个数.

**解题思路**

枚举两个数的代码很容易就可以写出来.

- C++

```cpp
for (int i = 0; i < n; ++i)
  for (int j = 0; j < n; ++j)
    if (a[i] + a[j] == 0) ++ans;
```

- Python

```python
for i in range(n):
    for j in range(n):
        if a[i] + a[j] == 0:
            ans += 1
```

- Java

```cpp
for (int i = 0; i < n; ++i)
  for (int j = 0; j < n; ++j)
    if (a[i] + a[j] == 0) ++ans;
```

来看看枚举的范围如何优化. 若 $(a_i, a_j)$ 满足条件，则 $(a_j, a_i)$ 也满足条件；又因为元素均不为 0，满足条件时必有 $i \neq j$. 因此可以只枚举 $j < i$，使每个无序配对只统计一次，再将结果乘 2，得到有序数对个数. 代码如下：

- C++

```cpp
for (int i = 0; i < n; ++i)
  for (int j = 0; j < i; ++j)
    if (a[i] + a[j] == 0) ++ans;
ans *= 2;
```

- Python

```python
for i in range(n):
    for j in range(i):
        if a[i] + a[j] == 0:
            ans += 1
ans *= 2
```

- Java

```java
for (int i = 0; i < n; ++i)
    for (int j = 0; j < i; ++j)
        if (a[i] + a[j] == 0) ++ans;
ans *= 2;
```

不难发现这里已经减少了 $j$ 的枚举范围，减少了这段代码的时间开销.

我们可以在此之上进一步优化.

两个数是否都一定要枚举出来呢? 枚举其中一个数之后，题目的条件已经确定了其他的要素（另一个数）的条件，如果能找到一种方法直接判断题目要求的那个数是否存在，就可以省掉枚举后一个数的时间了. 较为进阶地，在数据范围允许的情况下，我们可以使用桶[1]记录遍历过的数.

- C++

```cpp
#include <cstring>
constexpr int MAXN = 100000;  // 此处 MAXN 是数组内元素的界

int solve(int n, int a[]) {
  bool met[MAXN * 2 + 1];  // 创建一个能装下 [-MAXN, MAXN] 的桶
  memset(met, 0, sizeof(met));
  int ans = 0;
  for (int i = 0; i < n; ++i) {
    if (met[MAXN - a[i]]) ++ans;  // 如果桶内有想要的元素，答案加一
    met[MAXN + a[i]] = true;  // 无论如何，都要把当前元素放进桶里
  }
  return ans * 2;
}

```

- Python

```python
met = [False] * (MAXN * 2 + 1)
for i in range(n):
    if met[MAXN - a[i]]:
        ans += 1
    met[a[i] + MAXN] = True
ans *= 2
```

- Java

```java
boolean[] met = new boolean[MAXN * 2 + 1];
for (int i = 0; i < n; ++i) {
    if (met[MAXN - a[i]]) ++ans;
    met[MAXN + a[i]] = true;
}
ans *= 2;
```

[1] 桶排序→5.7.10@p598 以及 主元素问题→13.14@p3323 以及 Stack Overflow 上对桶数据结构的讲解[1]（英文）

## 复杂度分析

- 时间复杂度分析：对 $a$ 数组遍历了一遍就能完成题目要求，当 $n$ 足够大的时候时间复杂度为 $O(n)$.
- 空间复杂度分析：$O(n + \max\{|\ x\ | : x \in a\})$.

## 习题

- 2811: 熄灯问题 - OpenJudge[2]

## 参考资料与注释

[1] Stack Overflow 上对桶数据结构的讲解 

[2] 2811: 熄灯问题 - OpenJudge 

# 5.4 模拟

本页面将简要介绍模拟算法.

## 简介

模拟就是用计算机来模拟题目中要求的操作.

模拟题目通常具有码量大、操作多、思路繁复的特点. 由于它码量大，经常会出现难以查错的情况，如果在考试中写错是相当浪费时间的.

## 技巧

写模拟题时，遵循以下的建议有可能会提升做题速度:

- 在动手写代码之前，在草纸上尽可能地写好要实现的流程.
- 在代码中，尽量把每个部分模块化，写成函数、结构体或类.
- 对于一些可能重复用到的概念，可以统一转化，方便处理：如，某题给你「YY-MM-DD 时：分」把它抽取到一个函数，处理成秒，会减少概念混淆.
- 调试时分块调试. 模块化的好处就是可以方便地单独调试某一部分.
- 写代码的时候一定要思路清晰，不要想到什么写什么，要按照落在纸上的步骤写.

实际上，上述步骤在解决其它类型的题目时也是很有帮助的.

## 例题详解

**Climbing Worm**[1]

一只长度不计的蠕虫位于 $n$ 英寸深的井的底部. 它每次向上爬 $u$ 英寸，但是必须休息一次才能再次向上爬. 在休息的时候，它滑落了 $d$ 英寸. 之后它将重复向上爬和休息的过程. 蠕虫爬出井口需要至少爬多少次? 如果蠕虫爬完后刚好到达井的顶部，我们也视作蠕虫已经爬出井口.

**解题思路**

本题保证能爬出井口，即 $u \geq n$ 或 $u > d$. 在此条件下可直接模拟. 用一个循环重复蠕虫的爬井过程，当攀爬的长度超过或者等于井的深度时跳出.

**参考代码**

- C++

```
```

```cpp
#include <iostream>

int main() {
  int n = 0, u = 0, d = 0;
  std::cin >> u >> d >> n;
  int time = 0, dist = 0;
  while (true) {  // 用死循环来枚举
    dist += u;
    time++;
    if (dist >= n) break;  // 满足条件则退出死循环
    dist -= d;
  }
  std::cout << time << '\n';  // 输出得到的结果
  return 0;
}

```

- Python

```python
u, d, n = map(int, input().split())
time = dist = 0
while True:  # 用死循环来枚举
    dist += u
    time += 1
    if dist >= n:  # 满足条件则退出死循环
        break
    dist -= d
print(time)  # 输出得到的结果

```

- Java

```java
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        int u = input.nextInt();
        int d = input.nextInt();
        int n = input.nextInt();
        int time = 0, dist = 0;
        while (true) {  // 用死循环来枚举
            dist += u;
            time++;
            if (dist >= n) {
                break;  // 满足条件则退出死循环
            }
            dist -= d;
        }
        System.out.println(time);   // 输出得到的结果
        input.close();
    }
}

```

## 习题

- 「NOIP2014」生活大爆炸版石头剪刀布 - Universal Online Judge[2]
- 「OpenJudge 3750」魔兽世界[3]
- 「SDOI2010」猪国杀 - LibreOJ[4]

## 外部参考资料

[1] Climbing Worm

[2]「NOIP2014」生活大爆炸版石头剪刀布 - Universal Online Judge

[3]「OpenJudge 3750」魔兽世界

[4]「SDOI2010」猪国杀 - LibreOJ

# 5.5 递归 & 分治

**Authors:** fudonglai, AngelKitty, labuladong

本页面将介绍递归与分治算法的区别与结合运用.

## 递归

### 定义

递归（Recursion），在数学和计算机科学中是指在函数的定义中使用函数自身的方法，在计算机科学中还额外指一种通过重复将问题分解为同类的子问题而解决问题的方法.

### 引入

要理解递归，就得先理解什么是递归.

递归的基本思想是某个函数直接或者间接地调用自身，这样原问题的求解就转换为了许多性质相同但是规模更小的子问题. 求解时只需要关注如何把原问题划分成符合条件的子问题，而不需要过分关注这个子问题是如何被解决的.

以下是一些有助于理解递归的例子：

1. 什么是递归？→5.5@p557
2. 如何给一堆数字排序？答：分成两半，先排左半边再排右半边，最后合并就行了，至于怎么排左边和右边，请重新阅读这句话.
3. 你今年几岁？答：去年的岁数加一岁，1999 年我出生.
4. 


递归在数学中非常常见. 例如，集合论对自然数的正式定义是：1 是一个自然数，每个自然数都有一个后继，这一个后继也是自然数.

递归代码最重要的两个特征：结束条件和自我调用. 自我调用是在解决子问题，而结束条件定义了最简子问题的答案.

```
int func(传入数值) {
  if (终止条件) return 最小子问题解;
```

```
  return func(缩小规模);
}
```

## 为什么要写递归

1. 结构清晰，可读性强．例如，分别用不同的方法实现 归并排序→5.7.8@p592：
   - C++

```cpp
// 不使用递归的归并排序算法
template <typename T>
void merge_sort(vector<T>& a) {
  int n = a.size();
  for (int seg = 1; seg < n; seg = seg + seg)
    for (int start = 0; start < n - seg; start += seg + seg)
      merge(a, start, start + seg - 1, std::min(start + seg + seg - 1, n - 1));
}

// 使用递归的归并排序算法
template <typename T>
void merge_sort(vector<T>& a, int front, int end) {
  if (front >= end) return;
  int mid = front + (end - front) / 2;
  merge_sort(a, front, mid);
  merge_sort(a, mid + 1, end);
  merge(a, front, mid, end);
}

```

   - Python

```python
# 不使用递归的归并排序算法
def merge_sort_iterative(a):
    n = len(a)
    seg = 1
    while seg < n:
        start = 0
        while start < n - seg:
            merge(a, start, start + seg - 1, min(start + seg + seg - 1, n - 1))
            start = start + seg + seg
        seg = seg + seg


# 使用递归的归并排序算法
def merge_sort(a, front, end):
    if front >= end:
        return
    mid = front + (end - front) // 2
    merge_sort(a, front, mid)
    merge_sort(a, mid + 1, end)
    merge(a, front, mid, end)


```

两份代码中的`merge(a, front, mid, end)`均原地合并同一数组的两个有序闭区间`[front,mid]`和`[mid+1,end]`.

2. 练习分析问题的结构. 当发现问题可以被分解成相同结构的小问题时, 递归写多了就能敏锐发现这个特点，进而高效解决问题.

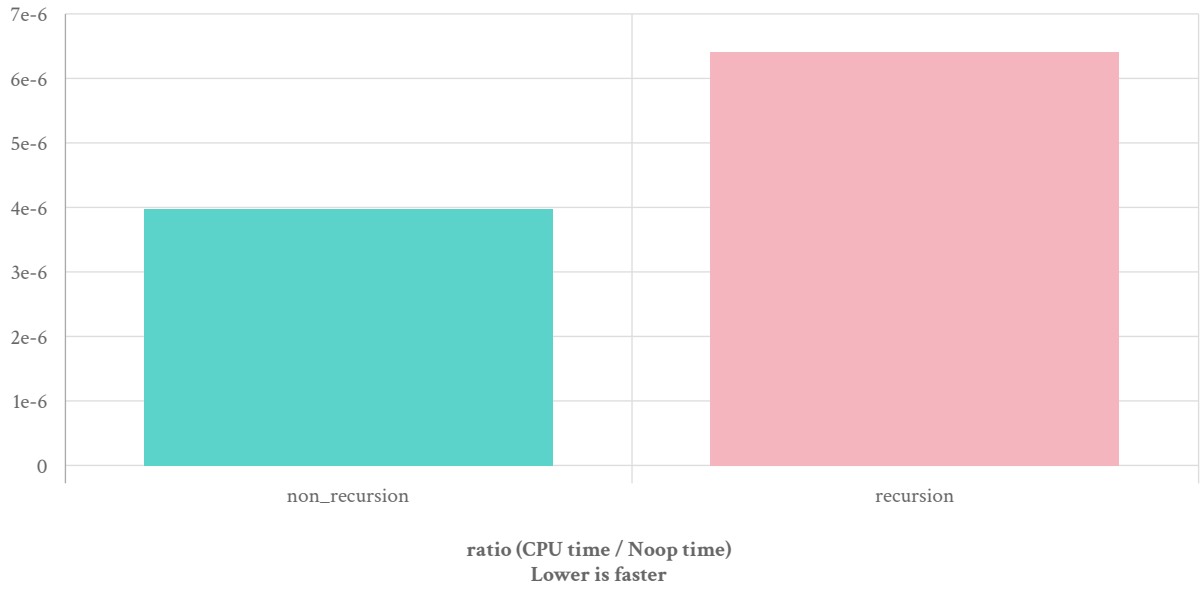
## 递归的缺点

在程序执行中，递归是利用堆栈来实现的. 每当进入一个函数调用，栈就会增加一层栈帧，每次函数返回，栈就会减少一层栈帧. 而栈不是无限大的，当递归层数过多时，就会造成 **栈溢出** 的后果.

递归实现可能需要额外的调用栈，而递推实现可以只使用常数辅助空间. 例如，给定一个链表头，计算它的长度:

```cpp
// 典型的递推遍历框架
int size(Node *head) {
  int size = 0;
  for (Node *p = head; p != nullptr; p = p->next) size++;
  return size;
}

// 递归遍历，每层处理一个结点
int size_recursion(Node *head) {
  if (head == nullptr) return 0;
  return size_recursion(head->next) + 1;
}
```




[1]

## 递归的优化

主页面: 搜索优化→6.12@p708 和 记忆化搜索→7.3@p720

比较初级的递归实现可能递归次数太多，容易超时. 这时需要对递归进行优化. [1]

[1] labuladong 的算法小抄 - 递归详解[2]

# 分治

## 定义

分治（Divide and Conquer），字面上的解释是「分而治之」，就是把一个复杂的问题分成两个或更多的相同或相似的子问题，直到最后子问题可以直接求解，原问题的解即子问题的解的合并.

## 过程

分治算法的核心思想就是「分而治之」.

大概的流程可以分为三步：分解 -> 解决 -> 合并.

1. 分解原问题为结构相同的子问题；
2. 分解到某个容易求解的边界之后，进行递归求解；
3. 将子问题的解合并成原问题的解.

分治法能解决的问题一般有如下特征：

- 该问题的规模缩小到一定的程度就可以容易地解决.
- 该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质，利用该问题分解出的子问题的解可以合并为该问题的解.
- 该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子问题.

> ⚠ **注意**
>
> 如果各子问题是不独立的，则分治法要重复地解公共的子问题，也就做了许多不必要的工作. 此时虽然也可用分治法，但一般用 动态规划→7.2@p714 较好.

以归并排序为例. 假设实现归并排序的函数名为 `merge_sort`. 明确该函数的职责，即 **对传入的一个数组排序**. 这个问题显然可以分解. 给一个数组排序等于给该数组的左右两半分别排序，然后合并成一个数组.

```
void merge_sort(一个数组) {
  if (可以很容易处理) return;
  merge_sort(左半个数组);
  merge_sort(右半个数组);
  merge(左半个数组, 右半个数组);
}
```

传给它半个数组，那么处理完后这半个数组就已经被排好了. 注意到，`merge_sort` 与二叉树的后序遍历模板极其相似. 因为分治算法的套路是 **分解 -> 解决（触底）-> 合并（回溯）**，先左右分解，再处理合并，回溯就是在退栈，即相当于后序遍历.

`merge` 函数的实现方式与两个有序链表的合并一致.

# 要点

## 写递归的要点

**明白一个函数的作用并相信它能完成这个任务，千万不要跳进这个函数里面企图探究更多细节，**否则就会陷入无穷的细节无法自拔，人脑能压几个栈啊.

以遍历二叉树为例.

```
void traverse(TreeNode* root) {
  if (root == nullptr) return;
  traverse(root->left);
  traverse(root->right);
}
```

这几行代码就足以遍历任何一棵二叉树了．对于递归函数 `traverse(root)`，只要相信给它一个根节点 `root`，它就能遍历这棵树．所以只需要把这个节点的左右节点再传给这个函数就行了．

同样扩展到遍历一棵 $N$ 叉树．与二叉树的写法一模一样．不过，对于 $N$ 叉树，显然没有中序遍历．

```cpp
void traverse(TreeNode* root) {
  if (root == nullptr) return;
  for (auto child : root->children) traverse(child);
}
```

## 区别

### 递归与枚举的区别

递归和枚举的区别在于：枚举是横向地把问题划分，然后依次求解子问题；而递归是把问题逐级分解，是纵向的拆分．

### 递归与分治的区别

递归是一种编程技巧，一种解决问题的思维方式；分治算法很大程度上是基于递归的，解决更具体问题的算法思想．

## 例题详解

**437. 路径总和 III**[3]

给定一个二叉树，它的每个结点都存放着一个整数值．

找出路径和等于给定数值的路径总数．

路径不需要从根节点开始，也不需要在叶子节点结束，但是路径方向必须是向下的（只能从父节点到子节点）．

二叉树不超过 1000 个节点，且节点数值范围是 $[-10^6, 10^6]$ 的整数．

示例：

```
root = [10,5,-3,3,2,null,11,3,-2,null,1], sum = 8

      10
     /  \
    5   -3
   / \    \
  3   2   11
 / \   \
3  -2   1

返回 3. 和等于 8 的路径有:

1.  5 -> 3
2.  5 -> 2 -> 1
3. -3 -> 11
```

```cpp
// divide-and-conquer_1.h
// 二叉树结点的定义
struct TreeNode {
  int val;
```

```
  TreeNode *left;
  TreeNode *right;

  TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

```

**参考代码**

```
#include "divide-and-conquer_2.h"

// 统计以 node 为起点、向下延伸且权值和为 sum 的路径.
int count(TreeNode *node, int sum) {
  if (node == nullptr) return 0;
  return (node->val == sum) + count(node->left, sum - node->val) +
         count(node->right, sum - node->val);
}

// 分别统计从当前结点出发、完全位于左子树和完全位于右子树的路径.
int pathSum(TreeNode *root, int sum) {
  if (root == nullptr) return 0;
  return count(root, sum) + pathSum(root->left, sum) +
         pathSum(root->right, sum);
}

```

**题目解析**

题目看起来很复杂，不过代码却极其简洁.

首先明确，递归求解树的问题必然是要遍历整棵树的，所以二叉树的遍历框架（分别对左右子树递归调用函数本身）必然要出现在主函数 pathSum 中. 那么对于每个节点，它们应该干什么呢？它们应该看看，自己和它们的子树包含多少条符合条件的路径. 好了，这道题就结束了.

按照前面说的技巧，根据刚才的分析来定义清楚每个递归函数应该做的事：

`pathSum` 函数：给定一个节点和一个目标值，返回以这个节点为根的树中，和为目标值的路径总数.

`count` 函数：给定一个节点和一个目标值，返回以这个节点为根的树中，能凑出几个以该节点为路径开头，和为目标值的路径总数.

**参考代码**

```
#include "divide-and-conquer_2.h"

// 统计以 node 为起点、向下延伸且权值和为 sum 的路径.
int count(TreeNode *node, int sum) {
  if (node == nullptr) return 0;
  return (node->val == sum) + count(node->left, sum - node->val) +
```

```
        count(node->right, sum - node->val);
}

// 分别统计从当前结点出发、完全位于左子树和完全位于右子树的路径.
int pathSum(TreeNode *root, int sum) {
  if (root == nullptr) return 0;
  return count(root, sum) + pathSum(root->left, sum) +
         pathSum(root->right, sum);
}

```

还是那句话，**明白每个函数能做的事，并相信它们能够完成.**

总结下，pathSum 函数提供了二叉树遍历框架，在遍历中对每个节点调用 count 函数（这里用的是先序遍历，不过中序遍历和后序遍历也可以）. count 函数也是一个二叉树遍历，用于寻找以该节点开头的目标值路径.

## 习题

- LeetCode 上的递归专题练习[4]
- LeetCode 上的分治算法专项练习[5]

## 参考资料与注释

[1] 二者的对比 

[2] labuladong 的算法小抄 - 递归详解 

[3] 437. 路径总和 III 

[4] LeetCode 上的递归专题练习 

[5] LeetCode 上的分治算法专项练习 

# 5.6 贪心

本页面将简要介绍贪心算法.

## 引入

贪心算法（greedy algorithm），是用计算机来模拟一个「贪心」的人做出决策的过程. 这个人十分贪婪，每一步行动总是按某种指标选取最优的操作. 而且他目光短浅，总是只看眼前，并不考虑以后可能造成的影响.

可想而知，并不是所有的时候贪心法都能获得最优解，所以一般使用贪心法的时候，都要确保自己能证明其正确性.

## 解释

### 适用范围

贪心算法在有最优子结构的问题中尤为有效. 最优子结构的意思是问题能够分解成子问题来解决，子问题的最优解能递推到最终问题的最优解. [1]

[1]贪心算法 - 维基百科，自由的百科全书[1]

### 证明

常见的证明思路包括交换论证和数学归纳法，两者也可以结合使用.

1. 交换论证：从任意一个最优解出发，通过有限次保持可行性且不使目标值变差的交换或替换，将其变为贪心算法得到的解，从而证明贪心解也是最优解. 常用反证法书写证明.
2. 归纳法：先算得出边界情况（例如 $n=1$）的最优解 $F_1$，然后再证明：对于每个 $n$，$F_{n+1}$ 都可以由 $F_n$ 推导出结果.

## 要点

### 常见题型

在提高组难度以下的题目中，最常见的贪心有两种.

- 将 XXX 按照某某顺序排序，然后按某种顺序（例如从小到大）选择.
- 每次取 XXX 中最大/小的东西，并更新 XXX.（有时取最大/小值的操作可以优化，比如用优先队列维护）

二者的区别在于前一种一定是离线的，可以先处理后选择；后一种可能是在线的，需要边处理边选择.

### 排序解法

用排序法/邻项交换法常见的情况是输入一个包含几个（一般一到两个）权值的数组，通过排序然后遍历模拟计算的方法求出最优值.

**例题 NOIP 2012 国王游戏**[2]

恰逢 H 国国庆，国王邀请 $n$ 位大臣来玩一个有奖游戏. 首先，他让每个大臣在左、右手上面分别写下一个整数，国王自己也在左、右手上各写一个整数. 然后，让这 $n$ 位大臣排成一排，国王站在队伍的最前面. 排好队后，所有的大臣都会获得国王奖赏的若干金币，每位大臣获得的金币数分别是：排在该大臣前面的所有人的左手上的数的乘积除以他自己右手上的数，然后向下取整得到的结果.

国王不希望某一个大臣获得特别多的奖赏，所以他想请你帮他重新安排一下队伍的顺序，使得获得奖赏最多的大臣，所获奖赏尽可能少. 注意，国王的位置始终在队伍的最前面.

**解题思路**

设当前排列中第 $i$ 个大臣左右手上的数分别为 $a_i, b_i$. 考虑通过邻项交换法推导贪心策略.

先忽略向下取整，用 $s$ 表示第 $i$ 个大臣前面所有人的 $a_i$ 的乘积. 交换前两人之中的最大奖赏为

$$\frac{s}{b_i b_{i+1}} \max(b_{i+1}, a_i b_i), \tag{1}$$

交换后两人之中的最大奖赏为

$$\frac{s}{b_i b_{i+1}} \max(b_i, a_{i+1} b_{i+1}). \tag{2}$$

若 $a_i b_i \le a_{i+1} b_{i+1}$，由于 $b_{i+1} \le a_{i+1} b_{i+1}$，故 (1) 式不大于 (2) 式. 向下取整是单调的，且 $\max(\lfloor u \rfloor, \lfloor v \rfloor) = \lfloor \max(u, v) \rfloor$，所以两人交换顺序后最大奖赏不会变小. 因此按 $a_i b_i$ 升序排序即为最优解.

实现的时候我们将输入的两个数用一个结构体来保存并重载运算符：

```
struct uv {
  int a, b;

  bool operator<(const uv& x) const { return 1LL * a * b < 1LL * x.a * x.b; }
```

```
};
```

## 后悔解法

思路是先临时接受新选项；若发生约束冲突，就在已选集合中撤销按贪心准则最差的选项．策略仍需正确性证明．

**例题 「USACO09OPEN」工作调度 Work Scheduling[3]**

约翰的工作日从 0 时刻开始，有 $10^9$ 个单位时间．在任一单位时间，他都可以选择编号 1 到 $N$ 的 $N(1 \le N \le 10^5)$ 项工作中的任意一项工作来完成．工作 $i$ 的截止时间是 $D_i(1 \le D_i \le 10^9)$，完成后获利是 $P_i(1 \le P_i \le 10^9)$．在给定的工作利润和截止时间下，求约翰能够获得的利润最大为多少．

**解题思路**

1. 先假设每一项工作都做，将各项工作按截止时间排序后入队；
2. 在判断第 `i` 项工作做与不做时，若其截止时间符合条件，则将其与队中报酬最小的元素比较，若第 `i` 项工作报酬较高（后悔），则 `ans += a[i].p - q.top()`．

   用优先队列（小根堆）来维护队首元素最小．
3. 当 `a[i].d<=q.size()` 可以这么理解从 0 开始到 `a[i].d` 这个时间段只能做 `a[i].d` 个任务，而若 `q.size()>=a[i].d` 说明完成 `q.size()` 个任务时间大于等于 `a[i].d` 的时间，所以当第 `i` 个任务获利比较大的时候应该把最小的任务从优先级队列中换出．

**参考代码**

- C++

```cpp
#include <algorithm>
#include <cmath>
#include <cstring>
#include <iostream>
#include <queue>
using namespace std;

struct f {
  long long d;
  long long p;
} a[100005];

bool cmp(f A, f B) { return A.d < B.d; }

// 小根堆维护最小值
priority_queue<long long, vector<long long>, greater<long long>> q;

int main() {
  long long n, i;
  cin >> n;
  for (i = 1; i <= n; i++) {
    cin >> a[i].d >> a[i].p;
  }
```

```
  sort(a + 1, a + n + 1, cmp);
  long long ans = 0;
  for (i = 1; i <= n; i++) {
    if (a[i].d <= (int)q.size()) {  // 超过截止时间
      if (q.top() < a[i].p) {       // 后悔
        ans += a[i].p - q.top();
        q.pop();
        q.push(a[i].p);
      }
    } else {  // 直接加入队列
      ans += a[i].p;
      q.push(a[i].p);
    }
  }
  cout << ans << endl;
  return 0;
}

```

- Python

```
from heapq import heappush, heapreplace

a = [tuple(map(int, input().split())) for _ in range(int(input()))]
a.sort(key=lambda job: job[0])  # 按截止时间升序排列

ans = 0  # 记录总收益
q = []  # 小根堆维护最小值
for d, p in a:
    if d <= len(q):  # 超过截止时间
        if q[0] < p:  # 后悔
            ans += p - heapreplace(q, p)
    else:  # 直接加入队列
        ans += p
        heappush(q, p)
print(ans)

```

**复杂度分析**

- 空间复杂度：当输入 $n$ 个任务时使用 $n$ 个 $a$ 数组元素，优先队列中最差情况下会储存 $n$ 个元素，则空间复杂度为 $O(n)$.
- 时间复杂度：`std::sort` 的时间复杂度为 $O(n\log n)$，维护优先队列的时间复杂度为 $O(n\log n)$，综上所述，时间复杂度为 $O(n\log n)$.

## 与动态规划的区别

常规贪心依赖可证明安全的局部选择；动态规划则维护各状态的最优值并比较候选转移.

## 习题

- 「USACO1.3」Barn Repair[4]
- 洛谷 P2123 皇后游戏[5]

- LeetCode 上标签为贪心算法的题目[6]

## 参考资料与注释

[1] 贪心算法 - 维基百科，自由的百科全书 

[2] NOIP 2012 国王游戏 

[3]「USACO09OPEN」工作调度 Work Scheduling 

[4]「USACO1.3」Barn Repair 

[5] 洛谷 P2123 皇后游戏 

[6] LeetCode 上标签为贪心算法的题目 

# 5.7 排序

## 5.7.1 排序简介

本页面将简要介绍排序算法.

### 定义

**排序算法**（Sorting algorithm）是一种将一组特定的数据按某种顺序进行排列的算法. 排序算法多种多样，性质也大多不同.

### 性质

#### 稳定性

稳定性是指相等的元素经过排序之后相对顺序是否发生了改变.

拥有稳定性这一特性的算法会让原本有相等键值的记录维持相对次序，即如果一个排序算法是稳定的，当有两个相等键值的记录 $R$ 和 $S$，且在原本的列表中 $R$ 出现在 $S$ 之前，在排序过的列表中 $R$ 也将会是在 $S$ 之前.

基数排序、计数排序、插入排序、冒泡排序、归并排序是稳定排序.

选择排序、堆排序、快速排序、希尔排序不是稳定排序.

#### 时间复杂度

主页面：复杂度→5.2.1@p544

时间复杂度用来衡量一个算法的运行时间和输入规模的关系，通常用 $O$ 表示.

简单计算复杂度的方法一般是统计「简单操作」的执行次数，有时候也可以直接数循环的层数来近似估计.

时间复杂度分为最优时间复杂度、平均时间复杂度和最坏时间复杂度. OI 竞赛中要考虑的一般是最坏时间复杂度，因为它代表的是算法运行水平的下界，在评测中不会出现更差的结果了.

对任意 $n$ 个互异元素进行排序的比较算法，最坏情况下需要 $\Omega(n \log n)$ 次比较.

当然也有不是 $O(n \log n)$ 的. 例如，计数排序→5.7.5@p574 的时间复杂度是 $O(n + w)$，其中 $w$ 代表输入数据的值域大小.

以下是几种排序算法的比较.




### 空间复杂度

与时间复杂度类似，空间复杂度用来描述算法空间消耗的规模．一般来说，空间复杂度越小，算法越好．

### 外部链接

- 排序算法 - 维基百科，自由的百科全书[1]

### 外部参考资料

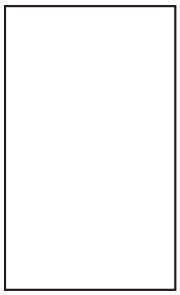
[1] 排序算法 - 维基百科，自由的百科全书

## 5.7.2 选择排序

本页面将简要介绍选择排序．

### 定义

选择排序（Selection sort）是一种简单直观的排序算法．它的工作原理是每次找出第 $i$ 小的元素（也就是 $A_{i..n}$ 中最小的元素），然后将这个元素与数组第 $i$ 个位置上的元素交换．



## 性质

### 稳定性

选择排序的稳定性取决于其具体实现.

倘若使用链表实现，由于链表的任意位置插入和删除均为 $O(1)$，故无需使用 swap（交换两个元素）操作：每次从未排序部分选择最小元素（若有多个，选取第一个）后，将其插入到未排序部分的第一个元素之前，这样就能够保证稳定性.

数组实现一般会使用 `swap` 将最小元素移到已排序部分，以此减少移动次数，所以不稳定.

下面给出的实现示例均是基于数组元素的交换，因此均为 **不稳定的**.

### 时间复杂度

选择排序的最优时间复杂度、平均时间复杂度和最坏时间复杂度均为 $O(n^2)$.

## 代码实现

### 伪代码

```
1 Input. An array A consisting of n elements.
2 Output. A will be sorted in nondecreasing order.
3 Method.
4 for i ← 1 to n − 1
5       ith ← i
6       for j ← i + 1 to n
7             if A[j] < A[ith]
8                   ith ← j
9       swap A[i] and A[ith]
```

- C++

```cpp
#include <utility>

void selection_sort(int* a, int n) {
  for (int i = 1; i < n; ++i) {
    int ith = i;
    for (int j = i + 1; j <= n; ++j) {
      if (a[j] < a[ith]) {
        ith = j;
      }
    }
    std::swap(a[i], a[ith]);
  }
}

```

- Python

```python
def selection_sort(a, n):
    for i in range(1, n):
        ith = i
        for j in range(i + 1, n + 1):
            if a[j] < a[ith]:
                ith = j
        a[i], a[ith] = a[ith], a[i]


```

- Java

```java
// arr 数组的下标从 1 开始
static void selection_sort(int[] arr, int n) {
    for (int i = 1; i < n; i++) {
        int ith = i;
        for (int j = i + 1; j <= n; j++) {
            if (arr[j] < arr[ith]) {
                ith = j;
            }
        }
        // swap
        int temp = arr[i];
        arr[i] = arr[ith];
        arr[ith] = temp;
```

```
    }
}
```

### 5.7.3 冒泡排序

本页面将简要介绍冒泡排序.

#### 定义

冒泡排序（Bubble sort）是一种简单的排序算法. 由于在算法的执行过程中，较小的元素像是气泡般慢慢「浮」到数列的顶端，故叫做冒泡排序.

#### 过程

它的工作原理是每次检查相邻两个元素, 如果前面的元素与后面的元素满足给定的排序条件, 就将相邻两个元素交换. 当没有相邻的元素需要交换时，排序就完成了.

经过 $i$ 次扫描后，数列的末尾 $i$ 项必然是最大的 $i$ 项，因此扫描 $n-1$ 遍足以使数组有序.

#### 性质

##### 稳定性

冒泡排序是一种稳定的排序算法.

##### 时间复杂度

在序列完全有序时，冒泡排序只需遍历一遍数组，不用执行任何交换操作，时间复杂度为 $O(n)$.

在最坏情况下，冒泡排序要执行 $\dfrac{(n-1)n}{2}$ 次交换操作，时间复杂度为 $O(n^2)$.

冒泡排序的平均时间复杂度为 $O(n^2)$.

#### 代码实现

##### 伪代码

```
1   Input. An array A consisting of n elements.
2   Output. A will be sorted in nondecreasing order stably.
3   Method.
4   flag ← True
5   while flag
6       flag ← False
7       for i ← 1 to n − 1
8           if A[i] > A[i + 1]
9               flag ← True
10              Swap A[i] and A[i + 1]
```

- C++

```cpp
// 假设数组的大小是 n + 1，冒泡排序从数组下标 1 开始
void bubble_sort(int *a, int n) {
  bool flag = true;
  while (flag) {
    flag = false;
    for (int i = 1; i < n; ++i) {
      if (a[i] > a[i + 1]) {
        flag = true;
        int t = a[i];
        a[i] = a[i + 1];
```

```
        a[i + 1] = t;
      }
    }
  }
}

```

- Python

```python
def bubble_sort(a, n):
    flag = True
    while flag:
        flag = False
        for i in range(1, n):
            if a[i] > a[i + 1]:
                flag = True
                a[i], a[i + 1] = a[i + 1], a[i]


```

- Java

```java
// 假设数组的大小是 n + 1，冒泡排序从数组下标 1 开始
static void bubble_sort(int[] a, int n) {
    boolean flag = true;
    while (flag) {
        flag = false;
        for (int i = 1; i < n; i++) {
            if (a[i] > a[i + 1]) {
                flag = true;
                int t = a[i];
                a[i] = a[i + 1];
                a[i + 1] = t;
            }
        }
    }
}
```

## 5.7.4 插入排序

本页面将简要介绍插入排序.

### 定义

插入排序（Insertion sort）是一种简单直观的排序算法. 它的工作原理为将待排列元素划分为「已排序」和「未排序」两部分，每次从「未排序的」元素中选择一个插入到「已排序的」元素中的正确位置.

一个与插入排序相同的操作是打扑克牌时，从牌桌上抓一张牌，按牌面大小插到手牌后，再抓下一张牌.



## 性质

### 稳定性

插入排序是一种稳定的排序算法.

### 时间复杂度

插入排序的最优时间复杂度为 $O(n)$，在数列几乎有序时效率很高.

插入排序的最坏时间复杂度和平均时间复杂度都为 $O(n^2)$.

## 代码实现

### 伪代码

1 **Input**. An array $A$ consisting of $n$ elements.
2 **Output**. $A$ will be sorted in nondecreasing order stably.
3 **Method**.
4 **for** $i \leftarrow 2$ **to** $n$
5 $\quad key \leftarrow A[i]$
6 $\quad j \leftarrow i-1$
7 $\quad$ **while** $j > 0$ **and** $A[j] > key$
8 $\quad\quad A[j+1] \leftarrow A[j]$
9 $\quad\quad j \leftarrow j-1$
10 $\quad A[j+1] \leftarrow key$

- C++

```cpp
void insertion_sort(int arr[], int len) {
  for (int i = 1; i < len; ++i) {
    int key = arr[i];
    int j = i - 1;
    while (j >= 0 && arr[j] > key) {
      arr[j + 1] = arr[j];
      j--;
    }
    arr[j + 1] = key;
  }
}

```

- Python

```python
def insertion_sort(arr, n):
    for i in range(1, n):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key


```

- Java

```java
public static void insertionSort(int[] arr) {
    for (int i = 1; i < arr.length; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

```

### 折半插入排序

插入排序还可以通过二分算法优化性能，在排序元素数量较多时优化的效果比较明显.

#### 时间复杂度

折半插入排序用二分查找确定插入位置, 将总比较次数降为 $O(n \log n)$, 但最坏情况下仍需移动 $\Theta(n^2)$ 个元素，因此最坏时间复杂度仍为 $\Theta(n^2)$.

#### 代码实现

- C++

```cpp
void insertion_sort(int arr[], int len) {
  if (len < 2) return;
  for (int i = 1; i != len; ++i) {
    int key = arr[i];
    auto index = upper_bound(arr, arr + i, key) - arr;
    // 使用 memmove 移动元素，比使用 for 循环速度更快，时间复杂度仍为 O(n)
    memmove(arr + index + 1, arr + index, (i - index) * sizeof(int));
    arr[index] = key;
  }
}
```

## 5.7.5 计数排序

前置知识：前缀和→5.8@p617

> ⚠ **提醒**
>
> 本页面要介绍的不是 **基数排序**→5.7.6@p578.

本页面将简要介绍计数排序.

## 定义

计数排序（Counting sort）是一种线性时间的排序算法.

## 过程

计数排序的工作原理是使用一个额外的数组 $C$，其中第 $i$ 个元素是待排序数组 $A$ 中值等于 $i$ 的元素的个数，然后根据数组 $C$ 来将 $A$ 中的元素排到正确的位置. [1]

它的工作过程分为三个步骤:

1. 计算每个数出现了几次;
2. 求出每个数出现次数的 前缀和→5.8@p617;
3. 利用出现次数的前缀和，从右至左计算每个数的排名.

### 计算前缀和的原因

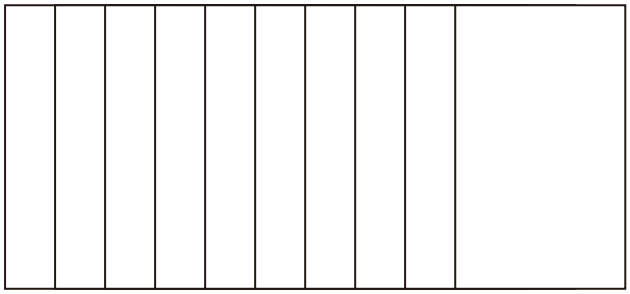
按频数重建可以排序纯键值并处理重复值; 若要稳定重排原始记录, 可利用频数的前缀和确定输出位置.

我们通过为额外数组 $C$ 中的每一项计算前缀和，结合每一项的数值，就可以为重复元素确定一个唯一排名:

额外数组 $C$ 中每一项的数值即是该 key 值下重复元素的个数，而该项的前缀和即是排在最后一个的重复元素的排名.

如果按照 $A$ 的逆序进行排列，那么显然排序后的数组将保持 $A$ 的原序（相同 key 值情况下），也即得到一种稳定的排序算法.

[1]计数排序 - 维基百科，自由的百科全书[1]



## 性质

### 稳定性

计数排序是一种稳定的排序算法.

### 时间复杂度

计数排序的时间复杂度为 $O(n+w)$，其中 $w$ 代表待排序数据的值域大小.

## 代码实现

### 伪代码

**Input**. An array $A$ consisting of $n$ positive integers no greater than $w$.
**Output**. Array $A$ after sorting in nondecreasing order stably.
**Method**.
**for** $i \leftarrow 0$ **to** $w$
    $cnt[i] \leftarrow 0$
**for** $i \leftarrow 1$ **to** $n$
    $cnt[A[i]] \leftarrow cnt[A[i]] + 1$
**for** $i \leftarrow 1$ **to** $w$
    $cnt[i] \leftarrow cnt[i] + cnt[i-1]$
**for** $i \leftarrow n$ **downto** 1
    $B[cnt[A[i]]] \leftarrow A[i]$
    $cnt[A[i]] \leftarrow cnt[A[i]] - 1$
**return** $B$

- C++

```cpp
#include <cstring>
constexpr int MAXN = 1010;
constexpr int MAXW = 100010;

int cnt[MAXW], b[MAXN];

int* counting_sort(int* a, int n, int w) {
  memset(cnt, 0, sizeof(cnt));
  for (int i = 1; i <= n; ++i) ++cnt[a[i]];
  for (int i = 1; i <= w; ++i) cnt[i] += cnt[i - 1];
  for (int i = n; i >= 1; --i) b[cnt[a[i]]--] = a[i];
  return b;
}

```

- Python

```python
def counting_sort(a, n, w):
    b = [0] * (n + 1)
    cnt = [0] * (w + 1)
    for i in range(1, n + 1):
        cnt[a[i]] += 1
    for i in range(1, w + 1):
        cnt[i] += cnt[i - 1]
    for i in range(n, 0, -1):
        b[cnt[a[i]]] = a[i]
        cnt[a[i]] -= 1
    return b


```

## 参考资料与注释

[1] 计数排序 - 维基百科, 自由的百科全书



## 5.7.6 基数排序

> ⚠ **提醒**
>
> 本页面要介绍的不是 **计数排序**→5.7.5@p574.

本页面将简要介绍基数排序.

### 定义

基数排序（Radix sort）是一种非比较型的排序算法，最早用于解决卡片排序的问题. 基数排序将待排序的元素拆分为 $k$ 个关键字，逐一对各个关键字排序后完成对所有元素的排序.

如果是从第 1 关键字到第 $k$ 关键字顺序进行比较，则该基数排序称为 MSD（Most Significant Digit first）基数排序；

如果是从第 $k$ 关键字到第 1 关键字顺序进行比较，则该基数排序称为 LSD（Least Significant Digit first）基数排序.

### k - 关键字元素的比较

下面用 $a_i$ 表示元素 $a$ 的第 $i$ 关键字.

假如元素有 $k$ 个关键字，对于两个元素 $a$ 和 $b$，默认的比较方法是：

- 比较两个元素的第 1 关键字 $a_1$ 和 $b_1$，如果 $a_1 < b_1$ 则 $a < b$，如果 $a_1 > b_1$ 则 $a > b$，如果 $a_1 = b_1$ 则进行下一步.
- 比较两个元素的第 2 关键字 $a_2$ 和 $b_2$，如果 $a_2 < b_2$ 则 $a < b$，如果 $a_2 > b_2$ 则 $a > b$，如果 $a_2 = b_2$ 则进行下一步.
- ……
- 比较两个元素的第 $k$ 关键字 $a_k$ 和 $b_k$，如果 $a_k < b_k$ 则 $a < b$，如果 $a_k > b_k$ 则 $a > b$，如果 $a_k = b_k$ 则 $a = b$.

例子：

- 如果对自然数进行比较，将自然数按个位对齐后往高位补齐 0，则一个数字从左往右数第 $i$ 位数就可以作为第 $i$ 关键字.
- 如果对字符串基于字典序进行比较，一个字符串从左往右数第 $i$ 个字符就可以作为第 $i$ 关键字.
- C++ 自带的 `std::pair` 与 `std::tuple` 的默认比较方法与上述的相同.

### MSD 基数排序

基于 k - 关键字元素的比较方法，可以想到：先比较所有元素的第 1 关键字，就可以确定出各元素大致的大小关系；然后对 **具有相同第 1 关键字的元素**，再比较它们的第 2 关键字……以此类推.

由于是从第 1 关键字到第 $k$ 关键字顺序进行比较，由上述思想导出的排序算法称为 MSD（Most Significant Digit first）基数排序.

#### 算法流程

将待排序的元素拆分为 $k$ 个关键字，先对第 1 关键字进行稳定排序，然后对于每组 **具有相同关键字的元素** 再对第 2 关键字进行稳定排序（递归执行）……最后对于每组 **具有相同关键字的元素** 再对第 $k$ 关键字进行稳定排序.

一般而言，我们默认基数排序是稳定的，所以在 MSD 基数排序中，我们也仅仅考虑借助 **稳定算法**（通常使用计数排序）完成内层对关键字的排序.

正确性参考上文 k - 关键字元素的比较.

## 参考代码

### 对自然数排序

下面是递归式 MSD 基数排序，处理 $[0, 2^{32}-1]$ 内至多 `MAXN` 个整数．初始 `digit=10`；修改基数时，须同步修改 `RADIX` 和 `powRADIX`.

**参考代码**

```
constexpr unsigned MAXN = 1000;  // 要排序的数的个数
constexpr unsigned RADIX = 10;   // 基数
constexpr unsigned powRADIX[10] = {1,         10,        100,     1000,
                                   10000,     100000,    1000000, 10000000,
                                   100000000, 1000000000};  // RADIX 的幂

unsigned get_digit(unsigned value, int digit)  // 提取第 digit 位
{
  return (value / powRADIX[digit]) % RADIX;
}

void MSD_radix_sort(unsigned* begin, unsigned* end, int digit)
// 表示现在 [begin,end) 内的元素（10 进制下）前若干位都相同
// 只有最后 digit 位（第 digit-1 到 0 位）需要进行排序
// 调用示例：MSD_radix_sort(a,a+n,10)
{
  if (begin >= end)  // 空区间
  {
    return;
  }
  /**计数排序（个人写法仅供参考）**/
  static unsigned cnt[RADIX + 1],
      tmp[MAXN + 5];  // 由于不同层递归不会同时使用 cnt,tmp
                      // 数组（每层使用完毕才会调用下一层），使用 static
                      // 关键字可以节省空间
  vector<unsigned> beg;
  beg.resize(RADIX + 1);  // 对 beg 的访问可能冲突，使用局部变量
  for (int i = 0; i <= RADIX; i++) {
    cnt[i] = beg[i] = 0;  // 清空是一个好习惯
  }
  for (unsigned* it = begin; it != end; it++)  // 计数
  {
    int bitVal = get_digit(*it, digit - 1);
    cnt[bitVal] += 1;
  }
  beg[0] = 0;  // 计算每个数码开始存储的位置（偏移量）
  for (int i = 1; i <= RADIX; i++) {
    beg[i] = beg[i - 1] + cnt[i - 1];
  }
  // 多计算 beg[RADIX] 是因为这样可以直接定义 i 对应的范围是 [beg[i],beg[i+1])
  // 而不用担心那个 beg[i+1] 越界
  for (int i = 0; i < RADIX; i++) {
    cnt[i] = 0;
  }
  for (unsigned* it = begin; it != end; it++)  // 将计数排序结果放入 tmp
  {
```

```
    unsigned bitVal = get_digit(*it, digit - 1);  // 提取第 digit - 1 位
    // 按原顺序遍历，当前是第 cnt[bitVal]+1 个在第 digit-1 位为 bitVal 的元素
    tmp[beg[bitVal] + cnt[bitVal]] = *it;
    cnt[bitVal]++;
  }
  for (unsigned* it = begin; it != end; it++)  // 将 tmp 拷贝回原数组
  {
    *it = tmp[it - begin];
  }
  /**递归排序各桶的下一位**/
  if (digit == 1)  // 已经是最低位
  {
    return;
  }
  for (int i = 0; i < RADIX; i++)  // 递归排序下一位
  {
    MSD_radix_sort(begin + beg[i], begin + beg[i + 1], digit - 1);
  }
}

```

**对字符串排序**

下面是递归式 MSD 基数排序，对仅含小写英文字母的 `std::string` 按字典序排序：

**参考代码**

```
constexpr int MAXN = 1000;

int get_digit(string* arr, int i, int dig) {
  if (arr[i][dig] == '\0') return 0;
  return arr[i][dig] - 'a' + 1;
}

void MSD_radix_sort_string_base(string* arr, int* begin, int* end,
                                int digit)  // 主体函数
// 对仅由小写字母构成的字符串排序
// 参数表示现在 [begin,end) 内的元素 [0,digit) 位都相同
// 现在从 digit 位开始进行排序
// 调用示例：MSD_radix_sort_string(a, a + n)
// 和上一份代码几乎完全相同，所以注释较少
// 为节省空间和时间，我们对下标数组进行排序，比较函数依旧比较字符串对应位
{
  if (begin >= end) return;
  static int tmp[MAXN + 5];
  static int cnt[28];
  vector<int> beg;
  beg.resize(28);
  // Q: 为什么开 28?
  // A: 0=空字符，1-26=a-z，27=额外空间（防止越界）
  for (int i = 0; i < 28; i++) cnt[i] = beg[i] = 0;
```

```
25    for (int* it = begin; it != end; it++) cnt[get_digit(arr, *it, digit)]
  += 1;
26    beg[0] = 0;
27    for (int i = 1; i <= 27; i++) beg[i] = beg[i - 1] + cnt[i - 1];
28    for (int i = 0; i < 28; i++) cnt[i] = 0;
29    for (int* it = begin; it != end; it++) {
30      int bitVal = get_digit(arr, *it, digit);
31      tmp[beg[bitVal] + cnt[bitVal]] = *it;
32      cnt[bitVal]++;
33    }
34    for (int* it = begin; it != end; it++) *it = tmp[it - begin];
35    // 如果已经是空字符了就没必要递归了，所以递归 1~26
36    for (int i = 1; i <= 26; i++)
37      MSD_radix_sort_string_base(arr, begin + beg[i], begin + beg[i + 1],
38                                 digit + 1);
39  }
40
41  int label[MAXN + 5];
42
43  void MSD_radix_sort_string(string* begin, string* end)  // 调用接口
44  {
45    static string tmp[MAXN + 5];
46    int n = end - begin;
47    for (int i = 0; i < n; i++) {
48      label[i] = i;
49      tmp[i] = *(begin + i);
50    }
51    MSD_radix_sort_string_base(tmp, label, label + n, 0);
52    for (int i = 0; i < n; i++) {
53      *(begin + i) = tmp[label[i]];
54    }
55  }
56
```

由于两个字符串的比较很容易冲上 $O(n)$ 的线性复杂度，因此在字符串排序这件事情上，MSD 基数排序比大多数基于比较的排序算法在时间复杂度和实际用时上都更加优秀.

### 与桶排序的关系

前置知识：桶排序→5.7.10@p598

桶排序需要其它的排序算法来完成对每个桶内部元素的排序. 但实际上, 完全可以对每个桶继续执行桶排序，直至某一步桶的元素数量 $\leq 1$.

因此 MSD 基数排序的另一种理解方式是：使用桶排序实现的桶排序.

也因此, 可以提出 MSD 基数排序在时间常数上的一种优化方法: 假如到某一步桶的元素数量 $\leq B$ ($B$ 是自己选的常数)，则直接执行插入排序然后返回，降低递归次数.

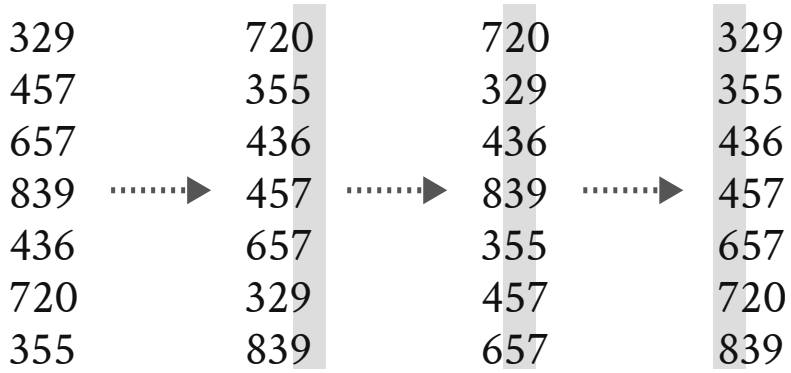
## LSD 基数排序

MSD 基数排序从第 1 关键字到第 $k$ 关键字顺序进行比较，为此需要借助递归或迭代来实现，时间常数还是较大，而且在比较自然数上还是略显不便.

而将递归的操作反过来：从第 $k$ 关键字到第 1 关键字顺序进行比较，就可以得到 LSD（Least Significant Digit first）基数排序，不使用递归就可以完成的排序算法.

### 算法流程

将待排序的元素拆分为 $k$ 个关键字，然后先对 **所有元素** 的第 $k$ 关键字进行稳定排序，再对 **所有元素** 的第 $k-1$ 关键字进行稳定排序，再对 **所有元素** 的第 $k-2$ 关键字进行稳定排序.....最后对 **所有元素** 的第 1 关键字进行稳定排序，这样就完成了对整个待排序序列的稳定排序.




LSD 基数排序也需要借助一种 **稳定算法** 完成内层对关键字的排序. 同样的，通常使用计数排序来完成.

LSD 基数排序的正确性可以参考 《算法导论（第三版）》第 8.3-3 题的解法[1] 或参考下面的解释：

### 正确性

回顾一下 k - 关键字元素的比较方法，

- 假如想通过 $a_1$ 和 $b_1$ 就比较出两个元素 $a$ 和 $b$ 的大小，则需要提前知道通过比较 $a_2$ 和 $b_2$ 得到的结论，以便于应对 $a_1 = b_1$ 的情况.
- 而想通过 $a_2$ 和 $b_2$ 就比较出两个元素 $a$ 和 $b$ 的大小，则需要提前知道通过比较 $a_3$ 和 $b_3$ 得到的结论，以便于应对 $a_2 = b_2$ 的情况.
- .....
- 而想通过 $a_{k-1}$ 和 $b_{k-1}$ 就比较出两个元素 $a$ 和 $b$ 的大小，则需要提前知道通过比较 $a_k$ 和 $b_k$ 得到的结论，以便于应对 $a_{k-1} = b_{k-1}$ 的情况.
- $a_k$ 和 $b_k$ 可以直接比较.

现在，将顺序反过来：

- $a_k$ 和 $b_k$ 可以直接比较.
- 而知道通过比较 $a_k$ 和 $b_k$ 得到的结论后，就可以得到比较 $a_{k-1}$ 和 $b_{k-1}$ 的结论.
- .....
- 而知道通过比较 $a_2$ 和 $b_2$ 得到的结论后，就可以得到比较 $a_1$ 和 $b_1$ 的结论.
- 而知道通过比较 $a_1$ 和 $b_1$ 得到的结论后，就最终得到了比较 $a$ 和 $b$ 的结论.

在这个过程中，对每个关键字边比较边重排元素的顺序，就得到了 LSD 基数排序.

### 伪代码

```
1 Input. An array A consisting of n elements, where each element has k keys.
2 Output. Array A will be sorted in nondecreasing order stably.
3 Method.
4 for i ← k down to 1
5       sort A into nondecreasing order by the i -th key stably
```

### 参考代码

下面是使用 LSD 基数排序实现的对 k - 关键字元素的排序.

**参考代码**

```cpp
constexpr int N = 100010;
constexpr int W = 100010;
constexpr int K = 100;

int n, w, k, cnt[W];

struct Element {
  int key[K];
} a[N], b[N];

void counting_sort(int p) {
  memset(cnt, 0, sizeof(cnt));
  for (int i = 1; i <= n; ++i) ++cnt[a[i].key[p]];
  for (int i = 1; i <= w; ++i) cnt[i] += cnt[i - 1];
  // 为保证排序的稳定性，此处循环i应从n到1
  // 即当两元素关键字的值相同时，原先排在后面的元素在排序后仍应排在后面
  for (int i = n; i >= 1; --i) b[cnt[a[i].key[p]]--] = a[i];
  memcpy(a, b, sizeof(a));
}

void radix_sort() {
  for (int i = k; i >= 1; --i) {
    // 借助计数排序完成对关键字的排序
    counting_sort(i);
  }
}

```

实际上并非必须从后往前枚举才是稳定排序，只需对 `cnt` 数组进行等价于 `std::exclusive_scan` 的操作即可.

**例题 洛谷 P1177【模板】排序**[2]

给出 $n$ 个正整数，从小到大输出.

```cpp
#include <algorithm>
#include <iostream>
#include <utility>

void radix_sort(int n, int a[]) {
  int *b = new int[n];  // 临时空间
  int *cnt = new int[1 << 8];
  int mask = (1 << 8) - 1;
  int *x = a, *y = b;
  for (int i = 0; i < 32; i += 8) {
    for (int j = 0; j != (1 << 8); ++j) cnt[j] = 0;
    for (int j = 0; j != n; ++j) ++cnt[x[j] >> i & mask];
    for (int sum = 0, j = 0; j != (1 << 8); ++j) {
      // 等价于 std::exclusive_scan(cnt, cnt + (1 << 8), cnt, 0);
      sum += cnt[j], cnt[j] = sum - cnt[j];
    }
```

```
    for (int j = 0; j != n; ++j) y[cnt[x[j] >> i & mask]++] = x[j];
    std::swap(x, y);
  }
  delete[] cnt;
  delete[] b;
}

int main() {
  std::ios::sync_with_stdio(false);
  std::cin.tie(nullptr);
  int n;
  std::cin >> n;
  int *a = new int[n];
  for (int i = 0; i < n; ++i) std::cin >> a[i];
  radix_sort(n, a);
  for (int i = 0; i < n; ++i) std::cout << a[i] << " \n"[i == n - 1];
  delete[] a;
  return 0;
}

```

## 性质

### 稳定性

如果对内层关键字的排序是稳定的，则 MSD 基数排序和 LSD 基数排序都是稳定的排序算法.

### 时间复杂度

通常而言，基数排序比基于比较的排序算法（比如快速排序）要快. 但由于需要额外的内存空间，因此当内存空间稀缺时，原地置换算法（比如快速排序）或许是个更好的选择. [2]

以下时间界假设提取单个关键字和移动一个元素（或其下标）均花费 $O(1)$ 时间. 一般来说，如果每个关键字的值域都不大，就可以使用 计数排序→5.7.5@p574 作为内层排序，LSD 和 MSD 的时间复杂度分别为

$$O\left(kn+\sum_{i=1}^{k} w_i\right), \qquad O\left(kn+\sum_{i=1}^{k} b_i w_i\right),$$

其中 $b_i$ 为第 $i$ 层实际执行计数排序的桶数，$w_i$ 为第 $i$ 关键字的值域大小. 如果关键字值域很大，就可以直接使用基于比较的 $O(nk\log n)$ 排序而无需使用基数排序了.

### 空间复杂度

下面讨论除输入以外的辅助空间，假设每个元素或其下标占用 $O(1)$ 空间. 使用计数排序时，LSD 可以复用长度为 $n$ 的临时数组和计数数组，辅助空间为 $O(n+\max_i w_i)$. 递归式 MSD 若复用临时数组，但在每层递归中保留桶的边界，则辅助空间为 $O(n+\sum_{i=1}^{k} w_i)$. 当各位的值域大小固定时，两者分别简化为 $O(n)$ 和 $O(n+k)$. 这里 $k$ 是关键字个数.

## 参考资料与注释

[1]《算法导论（第三版）》第 8.3-3 题的解法

[2] 洛谷 P1177【模板】排序

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*(3rd ed.). MIT Press and McGraw-Hill, 2009. ISBN 978-0-262-03384-8. "8.3 Radix sort", pp. 199.

## 5.7.7 快速排序

本页面将简要介绍快速排序.

### 定义

快速排序（Quicksort），又称分区交换排序（partition-exchange sort），简称「快排」，是一种被广泛运用的排序算法.

### 基本原理与实现

#### 过程

快速排序的工作原理是通过 分治→5.5@p557 的方式来将一个数组排序.

快速排序分为三个过程：

1. 将数列划分为两部分（要求保证相对大小关系）；
2. 递归到两个子序列中分别进行快速排序；
3. 不用合并，因为此时数列已经完全有序.

和归并排序不同，第一步并不是直接分成前后两个序列，而是在分的过程中要保证相对大小关系. 具体来说，第一步是要把数列分成两个部分，然后保证前一个子数列中的数都不大于后一个子数列中的数. 为了保证平均时间复杂度，一般是随机选择一个数 $m$ 来当做两个子数列的分界.

之后，维护一前一后两个指针 $p$ 和 $q$，依次考虑当前的数是否放在了应该放的位置（前还是后）. 如果当前的数没放对，比如说如果后面的指针 $q$ 遇到了一个比 $m$ 小的数，那么可以交换 $p$ 和 $q$ 位置上的数，再把 $p$ 向后移一位. 当前的数的位置全放对后，再移动指针继续处理，直到两个指针相遇.

其实，快速排序没有指定应如何具体实现第一步，不论是选择 $m$ 的过程还是划分的过程，都有不止一种实现方法.

第三步中的序列已经分别有序且第一个序列中的数都不大于第二个序列中的数，所以直接拼接起来就好了.

以下是固定选取主元的基础实现：非递归 C++ 实现选取当前子数组的末元素，递归 C++ 实现和 Python 实现选取首元素. 它们没有随机化选取主元，在有序输入等情况下可能退化为 $O(n^2)$.

- C++
  - 非递归实现[3]

```cpp
struct Range {
  int start, end;

  Range(int s = 0, int e = 0) { start = s, end = e; }
};

template <typename T>
void quick_sort(T arr[], const int len) {
  if (len <= 0) return;
  vector<Range> r(len);
  int p = 0;
  r[p++] = Range(0, len - 1);
  while (p) {
    Range range = r[--p];
    if (range.start >= range.end) continue;
    T mid = arr[range.end];
```

[3]算法实现/排序/快速排序 - 维基教科书，自由的教学读本[1]

```
    int left = range.start, right = range.end - 1;
    while (left < right) {
      while (arr[left] < mid && left < right) left++;
      while (arr[right] >= mid && left < right) right--;
      std::swap(arr[left], arr[right]);
    }
    if (arr[left] >= arr[range.end])
      std::swap(arr[left], arr[range.end]);
    else
      left++;
    r[p++] = Range(range.start, left - 1);
    r[p++] = Range(left + 1, range.end);
  }
}

```

- 递归实现

```
template <typename T>
int partition(T A[], int low, int high) {
  T pivot = A[low];
  while (low < high) {
    while (low < high && pivot <= A[high]) --high;
    A[low] = A[high];
    while (low < high && A[low] <= pivot) ++low;
    A[high] = A[low];
  }
  A[low] = pivot;
  return low;
}

template <typename T>
void quickSort(T A[], int low, int high) {
  if (low < high) {
    int pivot = partition(A, low, high);
    quickSort(A, low, pivot - 1);
    quickSort(A, pivot + 1, high);
  }
}

template <typename T>
void quickSort(T A[], int len) {
  quickSort(A, 0, len - 1);
}

```

- Python[4]

```
def quick_sort(alist, first, last):
    if first >= last:
        return
    mid_value = alist[first]
    low = first
    high = last
```

[4] 算法实现/排序/快速排序 - 维基教科书，自由的教学读本[2]

```
    while low < high:
        while low < high and alist[high] >= mid_value:
            high -= 1
        alist[low] = alist[high]
        while low < high and alist[low] < mid_value:
            low += 1
        alist[high] = alist[low]
    alist[low] = mid_value
    quick_sort(alist, first, low - 1)
    quick_sort(alist, low + 1, last)


```

## 性质

### 稳定性

快速排序是一种不稳定的排序算法.

### 时间复杂度

快速排序的最优时间复杂度和平均时间复杂度为 $O(n\log n)$，最坏时间复杂度为 $O(n^2)$.

对于最优情况, 每一次选择的分界值都是序列的中位数, 此时算法时间复杂度满足的递推式为 $T(n)=2T(\frac{n}{2})+\Theta(n)$，由主定理，$T(n)=\Theta(n\log n)$.

对于最坏情况，每一次选择的分界值都是序列的最值，此时算法时间复杂度满足的递推式为 $T(n)=T(n-1)+\Theta(n)$，累加可得 $T(n)=\Theta(n^2)$.

以下期望分析假设键值互异，且每次从当前子数组等概率随机选取主元.

**证明**

下面我们来证明这种情况下算法的时间复杂度是 $O(n\log n)$.

**引理1:** 当对 $n$ 个元素的数组进行快速排序时, 假设在划分元素时曾经比较的不同元素对数为 $X$, 则快速排序的时间复杂度是 $O(n+X)$.

由于在每次划分元素的过程中，都会选择一个元素作为分界，所以划分元素的过程至多发生 $n$ 次. 又由于划分元素的过程中每对元素只比较常数次, 比较工作量和其他基础操作的次数在一个数量级，所以总时间复杂度是 $O(n+X)$ 的.

设 $a_i$ 为原数组中第 $i$ 小的数，定义 $A_{i,j}$ 为 $\{a_i,a_{i+1},...,a_j\}$，$X_{i,j}$ 是一个取值为 0 或者 1 的离散随机变量表示在排序过程中 $a_i$ 是否和 $a_j$ 发生比较.

显然每次选取的分界值是不同的，而元素只会和分界值比较，所以不同比较对数

$$X=\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}X_{i,j}$$

由期望的线性性,

$$
\begin{aligned}
E[X] &= E\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{i,j}\right] \\
&= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E[X_{i,j}] \\
&= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} P(a_i \text{ 和 } a_j \text{ 比较})
\end{aligned}
$$

**引理 2：** $a_i$ 和 $a_j$ 比较的充要条件是 $a_i$ 或 $a_j$ 是集合 $A_{i,j}$ 中第一个被选中的分界值.

先证必要性，即若 $a_i$ 和 $a_j$ 都不是集合 $A_{i,j}$ 中第一个被选中的分界值，则 $a_i$ 不和 $a_j$ 比较.

若 $a_i$ 和 $a_j$ 都不是集合 $A_{i,j}$ 中第一个被选中的分界值，则一定存在一个 $x$ 满足 $i < x < j$，使得 $a_x$ 是 $A_{i,j}$ 中第一个被选中的分界值. 在以 $a_x$ 为分界值的划分中，$a_i$ 和 $a_j$ 被划分到数组的两个不同的子序列中，所以之后 $a_i$ 和 $a_j$ 一定不会比较. 又因为元素只和分界值比较，所以 $a_i$ 和 $a_j$ 在此次划分前和划分中没有比较. 所以 $a_i$ 不和 $a_j$ 比较.

再证充分性，即若 $a_i$ 或 $a_j$ 是集合 $A_{i,j}$ 中第一个被选中的分界值，则 $a_i$ 和 $a_j$ 比较.

不失一般地，假设 $a_i$ 是集合 $A_{i,j}$ 中第一个被选中的分界值. 由于 $A_{i,j}$ 中没有其他数选为分界值，所以 $A_{i,j}$ 中的元素都在数组的同一子序列中. 在以 $a_i$ 为分界值的划分中，$a_i$ 和当前子序列中所有元素都进行了比较，所以 $a_i$ 和 $a_j$ 进行了比较.

考虑计算 $P(a_i \text{ 和 } a_j \text{ 比较})$. 在 $A_{i,j}$ 中某个元素被选为分界值之前，$A_{i,j}$ 中的元素都在数组的同一子序列中. 所以 $A_{i,j}$ 中每个元素都会被等可能地第一个被选为分界值. 由于 $A_{i,j}$ 中有 $j-i+1$ 个元素，由引理 2，

$$
P(a_i \text{和} a_j \text{ 比较}) = P(a_i \text{或} a_j \text{ 是集合 } A_{i,j} \text{ 中第一个被选中的分界值}) = \frac{2}{j-i+1}
$$

所以

$$
\begin{aligned}
E[X] &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} P(a_i \text{ 和 } a_j \text{ 比较}) \\
&= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1}\sum_{k=2}^{n-i+1} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} O(\log n) \\
&= O(n \log n)
\end{aligned}
$$

由此，快速排序的期望时间复杂度为 $O(n \log n)$.

随机选取主元能够降低遭遇最坏划分的概率. 此外，快速排序的内存访问遵循局部性原理，这有助于其在实践中取得较好的性能. [5]

## 优化

### 朴素优化思想

如果仅按照上文所述的基本思想来实现快速排序（或者是直接照抄模板）的话，那一般无法通过 P1177【模板】快速排序[4] 这道模板题，因为可以构造数据使朴素快速排序退化为 $O(n^2)$.

[5] C++ 性能榨汁机之局部性原理 - I'm Root lee ![3]

所以，我们需要对朴素快速排序思想加以优化．较为常见的优化思路有以下三种[6]．

- 通过 **三数取中（即选取第一个、最后一个以及中间的元素中的中位数）** 的方法来选择两个子序列的分界元素（即比较基准）．这样可以避免极端数据（如升序序列或降序序列）带来的退化．
- 当序列较短时，使用 **插入排序** 的效率更高．
- 每趟排序后，**将与分界元素相等的元素聚集在分界元素周围**，这样可以避免极端数据（如序列中大部分元素都相等）带来的退化．

下面列举了几种较为成熟的快速排序优化方式．

### 三路快速排序

#### 定义

三路快速排序（3-way Quicksort）将元素按与主元的大小关系分成三部分．它的算法思想基于 荷兰国旗问题[6] 的解法．

#### 过程

与原始的快速排序不同，三路快速排序在随机选取分界点 $m$ 后，将待排数列划分为三个部分：小于 $m$、等于 $m$ 以及大于 $m$．这样做即实现了将与分界元素相等的元素聚集在分界元素周围这一效果．

#### 性质

三路快速排序在处理含有多个重复值的数组时，效率远高于原始快速排序．其最佳时间复杂度为 $O(n)$．

#### 实现

三路快速排序实现起来非常简单，下面给出三路快排的参考实现．

- C++

```cpp
// 模板的 T 参数表示元素的类型，此类型需要定义小于（<）运算
template <typename T>
// arr 为需要被排序的数组，len 为数组长度
void quick_sort(T arr[], const int len) {
  if (len <= 1) return;
  // 随机选择基准（pivot）
  const T pivot = arr[rand() % len];
  // i：当前操作的元素下标
  // arr[0, j)：存储小于 pivot 的元素
  // arr[k, len)：存储大于 pivot 的元素
  int i = 0, j = 0, k = len;
  // 完成一趟三路快排，将序列分为：
  // 小于 pivot 的元素 | 等于 pivot 的元素 | 大于 pivot 的元素
  while (i < k) {
    if (arr[i] < pivot)
      swap(arr[i++], arr[j++]);
    else if (pivot < arr[i])
      swap(arr[i], arr[--k]);
    else
      i++;
  }
  // 递归完成对于两个子序列的快速排序
  quick_sort(arr, j);
  quick_sort(arr + k, len - k);
}

```

[6] 三种快速排序以及快速排序的优化[5]

- Python[7]

```python
def quick_sort(arr, l, r):
    if l >= r:
        return
    random_index = random.randint(l, r)
    pivot = arr[random_index]
    arr[l], arr[random_index] = arr[random_index], arr[l]
    i = l + 1
    j = l
    k = r + 1
    while i < k:
        if arr[i] < pivot:
            arr[i], arr[j + 1] = arr[j + 1], arr[i]
            j += 1
            i += 1
        elif arr[i] > pivot:
            arr[i], arr[k - 1] = arr[k - 1], arr[i]
            k -= 1
        else:
            i += 1
    arr[l], arr[j] = arr[j], arr[l]
    quick_sort(arr, l, j - 1)
    quick_sort(arr, k, r)


```

### 内省排序

#### 定义

内省排序（Introsort，Introspective sort）[8] 是快速排序和 堆排序→5.7.9@p596 的结合，由 David Musser 于 1997 年发明．内省排序其实是对快速排序的一种优化，保证了最差时间复杂度为 $O(n \log n)$．

#### 性质

内省排序限制快速排序的划分深度，上限通常取 $2\lfloor \log_2 n \rfloor$[9]，超过限制时就转换为堆排序．这样既保留了快速排序内存访问的局部性，又可以防止快速排序在某些情况下性能退化为 $O(n^2)$．

#### 实现

从 2000 年 6 月起，SGI C++ STL 的 `stl_algo.h` 中 `sort()` 函数的实现采用了内省排序算法．

## 线性找升序下标为 k 的数

在下面的代码示例中，升序下标为 $k$ 的数被定义为序列排成升序时，第 $k$ 个位置上的数（编号从 0 开始）．

找升序下标为 $k$ 的数（K-th order statistic），最简单的方法是先排序，然后直接找到升序下标为 $k$ 的位置的元素．这样做的时间复杂度是 $O(n \log n)$，对于这个问题来说很不划算．

我们可以借助快速排序的思想解决这个问题．考虑快速排序的划分过程，在快速排序的「划分」结束后，数列 $A_p \cdots A_r$ 被分成了 $A_p \cdots A_q$ 和 $A_{q+1} \cdots A_r$，此时可以按照左边元素的个数（$q-p+1$）和 $k$ 的大小关系来判断是只在左边还是只在右边递归地求解．

---

[7] 算法实现/排序/快速排序 - 维基教科书，自由的教学读本[7]

[8] introsort[8]

[9] 例如 libstdc++ 14.2.0：`__introsort_loop` 和 `__sort`[9]．实际上将深度限制为 $O(\log n)$ 并在达到上限时转用堆排序，即可保证总时间复杂度为 $O(n \log n)$．

和快速排序一样，该方法的时间复杂度依赖于每次划分时选择的分界值. 如果采用随机选取分界值的方式，可以证明在期望意义下，程序的时间复杂度为 $O(n)$.

**参考实现**

```cpp
// 模板的 T 参数表示元素的类型，此类型需要定义小于（<）运算
template <typename T>
// arr 为查找范围数组，rk 为需要查找的排名（从 0 开始），len 为数组长度
T find_kth_element(T arr[], int rk, const int len) {
  if (len <= 1) return arr[0];
  // 随机选择基准（pivot）
  const T pivot = arr[rand() % len];
  // i：当前操作的元素下标
  // arr[0, j)：存储小于 pivot 的元素
  // arr[k, len)：存储大于 pivot 的元素
  int i = 0, j = 0, k = len;
  // 完成一趟三路快排，将序列分为：
  // 小于 pivot 的元素 | 等于 pivot 的元素 | 大于 pivot 的元素
  while (i < k) {
    if (arr[i] < pivot)
      swap(arr[i++], arr[j++]);
    else if (pivot < arr[i])
      swap(arr[i], arr[--k]);
    else
      i++;
  }
  // 根据要找的排名与两条分界线的位置，去不同的区间递归查找升序下标为 k 的数
  // 如果小于 pivot 的元素个数比 k 多，则升序下标为 k 的元素一定是一个小于 pivot
  // 的元素
  if (rk < j) return find_kth_element(arr, rk, j);
  // 否则，如果小于 pivot 和等于 pivot 的元素加起来也没有 k 多，
  // 则升序下标为 k 的元素一定是一个大于 pivot 的元素
  else if (rk >= k)
    return find_kth_element(arr + k, rk - k, len - k);
  // 否则，pivot 就是升序下标为 k 的元素
  return pivot;
}

```

### 中位数中的中位数

中位数中的中位数（英文：Median of medians），提供了一种确定性的选择划分过程中分界值的方法，从而让「找升序下标为 $k$ 的数」的算法在最坏情况下也能实现线性时间复杂度.

该算法的流程如下：

1. 将序列划分为 $\lceil n/5 \rceil$ 组，每组至多五个元素，保留不足五个的尾组；
2. 排序每组并取中位数，偶数长度时统一取较小的中位数；
3. 递归选择这些中位数的中位数作为主元，按小于、等于、大于主元三路划分，只递归目标所在的严格小于或严格大于区间.

#### 时间复杂度证明

设中位数的组数为 $g = \lceil n/5 \rceil$. 主元两侧（含等号）各有至少 $\lfloor g/2 \rfloor$ 个组中位数；排除尾组后，每个完整组至少贡献三个不大于或不小于主元的元素. 因此两侧各有至少 $3n/10 - O(1)$ 个这样的元素，严格小于

和严格大于区间的大小均至多 $7n/10+O(1)$. 等值区不再递归. 分组、求各组中位数和划分共花费 $O(n)$. 综上，我们可以列出这样的不等式：

$$T(n)\le T(\lceil n/5\rceil)+T(7n/10+O(1))+O(n).$$

取固定常数 $a,b$，使右边两个递归规模之和至多 $9n/10+a$，非递归成本至多 $bn$. 当 $n\ge 20a$ 时，该和至多 $19n/20$. 在归纳假设 $T(m)\le cm$ 下，只需取 $c\ge 20b$ 并增大 $c$ 以覆盖有限个基例，就有 $T(n)\le 19cn/20+bn\le cn$. 因此最坏时间复杂度为 $O(n)$.

### 参考资料与注释

[1] 算法实现/排序/快速排序 - 维基教科书, 自由的教学读本

[2] 算法实现/排序/快速排序 - 维基教科书，自由的教学读本

[3] C++ 性能榨汁机之局部性原理 - I'm Root lee !

[4] P1177【模板】快速排序

[5] 三种快速排序以及快速排序的优化

[6] 荷兰国旗问题

[7] 算法实现/排序/快速排序 - 维基教科书，自由的教学读本

[8] introsort

[9] libstdc++ 14.2.0: `__introsort_loop` 和 `__sort`

## 5.7.8 归并排序

本页面将介绍归并排序及其计算逆序对的应用.

### 定义

归并排序（merge sort[1]）是高效的基于比较的稳定排序算法.

### 性质

归并排序基于分治思想将数组分段排序后合并，时间复杂度在最优、最坏与平均情况下均为 $\Theta(n\log n)$，空间复杂度为 $\Theta(n)$.

归并排序可以只使用 $\Theta(1)$ 的辅助空间，但为便捷通常使用与原数组等长的辅助数组.

### 过程

#### 合并

归并排序最核心的部分是合并（merge）过程：将两个有序的数组 `a` 和 `b` 合并为一个有序数组 `c`.

从左往右枚举 `a[i]` 和 `b[j]`，每次将较小元素写入 `c[k]`；当一个数组遍历完毕时，将另一个数组的剩余元素追加到 `c` 中.

为保证排序的稳定性，前段首元素小于或等于后段首元素时（`a[i] <= b[j]`）而非小于时（`a[i] < b[j]`）就要作为最小值放入 `c[k]`.

#### 实现

- C/C++
  - 数组实现

```cpp
void merge(const int* a, size_t aLen, const int* b, size_t bLen, int* c) {
  size_t i = 0, j = 0, k = 0;
  while (i < aLen && j < bLen) {
    if (b[j] < a[i]) {  // <!> 先判断 b[j] < a[i]，保证稳定性
      c[k] = b[j];
      ++j;
    } else {
      c[k] = a[i];
```

```
      ++i;
    }
    ++k;
  }
  // 此时一个数组已空，另一个数组非空，将非空的数组并入 c 中
  for (; i < aLen; ++i, ++k) c[k] = a[i];
  for (; j < bLen; ++j, ++k) c[k] = b[j];
}

```

- 指针实现

```
void merge(const int* aBegin, const int* aEnd, const int* bBegin,
           const int* bEnd, int* c) {
  while (aBegin != aEnd && bBegin != bEnd) {
    if (*bBegin < *aBegin) {
      *c = *bBegin;
      ++bBegin;
    } else {
      *c = *aBegin;
      ++aBegin;
    }
    ++c;
  }
  for (; aBegin != aEnd; ++aBegin, ++c) *c = *aBegin;
  for (; bBegin != bEnd; ++bBegin, ++c) *c = *bBegin;
}

```

也可使用 `<algorithm>` 库的 `std::merge` 函数，用法与上述指针式写法相同.

- Python

```
def merge(a, b):
    i, j = 0, 0
    c = []
    while i < len(a) and j < len(b):
        # <!> 先判断 b[j] < a[i]，保证稳定性
        if b[j] < a[i]:
            c.append(b[j])
            j += 1
        else:
            c.append(a[i])
            i += 1
    # 此时一个数组已空，另一个数组非空，将非空的数组并入 c 中
    c.extend(a[i:])
    c.extend(b[j:])
    return c


```

### 分治法实现归并排序

1. 当数组长度为 1 时，该数组就已经是有序的，不用再分解.

2. 当数组长度大于 1 时，该数组很可能不是有序的. 此时将该数组分为两段，再分别检查两个数组是否有序（用第一条）. 如果有序，则将它们合并为一个有序数组；否则对尚未有序的数组重复第二条，再合并.

用数学归纳法可以证明该流程可以将一个数组转变为有序数组.

为保证排序的复杂度，通常将数组分为尽量等长的两段（$mid = \left\lfloor \frac{l+r}{2} \right\rfloor$）.

**实现**

注意下面的代码所表示的区间分别是 $[l, r)$，$[l, mid)$，$[mid, r)$.

- C/C++

```cpp
void merge_sort(int* a, int l, int r) {
  if (r - l <= 1) return;
  // 分解
  int mid = l + ((r - l) >> 1);
  merge_sort(a, l, mid);
  merge_sort(a, mid, r);
  // 合并
  std::vector<int> tmp(r - l);  // 只为本次合并的区间分配缓冲区
  merge(a + l, a + mid, a + mid, a + r, tmp.data());
  for (int i = l; i < r; ++i) a[i] = tmp[i - l];
}

```

- Python

```python
def merge_sort(a, ll, rr):
    if rr - ll <= 1:
        return
    # 分解
    mid = (rr + ll) // 2
    merge_sort(a, ll, mid)
    merge_sort(a, mid, rr)
    # 合并
    a[ll:rr] = merge(a[ll:mid], a[mid:rr])


```

**倍增法实现归并排序**

已知当数组长度为 1 时，该数组就已经是有序的.

将数组全部切成长度为 1 的段.

从左往右依次合并两个长度为 1 的有序段，得到一系列长度 $\leq 2$ 的有序段；

从左往右依次合并两个长度 $\leq 2$ 的有序段，得到一系列长度 $\leq 4$ 的有序段；

从左往右依次合并两个长度 $\leq 4$ 的有序段，得到一系列长度 $\leq 8$ 的有序段；

……

重复上述过程直至数组只剩一个有序段，该段就是排好序的原数组.

> **为什么是 $\leq$ 而不是 $=$**
>
> 数组的长度很可能不是 $2^x$，此时在最后就可能出现长度不完整的段，可能出现最后一个段是独立的情况.

### 实现

- C/C++

```cpp
void merge_sort(int* a, size_t n) {
  std::vector<int> tmp(n);  // 合并缓冲区与输入等长
  for (size_t seg = 1; seg < n; seg <<= 1) {
    for (size_t left1 = 0; left1 < n - seg;
         left1 += seg + seg) {  // n - seg: 如果最后只有一个段就不用合并
      size_t right1 = left1 + seg;
      size_t left2 = right1;
      size_t right2 = std::min(left2 + seg, n);  // <!> 注意最后一个段的边界
      merge(a + left1, a + right1, a + left2, a + right2,
            tmp.data() + left1);  // pointer-style merge
      for (size_t i = left1; i < right2; ++i) a[i] = tmp[i];
    }
  }
}

```

- Python

```python
def merge_sort_iterative(a):
    seg = 1
    while seg < len(a):
        for l1 in range(0, len(a) - seg, seg + seg):
            r1 = l1 + seg
            l2 = r1
            r2 = l2 + seg
            a[l1:r2] = merge(a[l1:r1], a[l2:r2])
        seg <<= 1


```

## 逆序对

相关阅读和参考实现：逆序对→9.8@p1138

逆序对是 $i < j$ 且 $a_i > a_j$ 的有序数对 $(i, j)$.

排序后的数组无逆序对. 归并排序的合并操作中, 每次后段首元素被作为当前最小值取出时, 前段剩余元素个数之和即是合并操作减少的逆序对数量; 故归并排序计算逆序对数量的时间复杂度为 $\Theta(n \log n)$. 此外, 逆序对计数还可以通过树状数组或线段树解决, 时间复杂度也是 $O(n \log n)$; 这一算法的详细解释参见 树状数组→10.12@p2078 相应描述. 两种算法的参考实现都在 逆序对→9.8@p1138 章节.

## 外部链接

- Merge Sort - GeeksforGeeks[2]
- 归并排序 - 维基百科, 自由的百科全书[3]
- 逆序对 - 维基百科, 自由的百科全书[4]

## 外部参考资料

[1] merge sort

[2] Merge Sort - GeeksforGeeks

[3] 归并排序 - 维基百科, 自由的百科全书

[4] 逆序对 - 维基百科, 自由的百科全书

## 5.7.9 堆排序

本页面将简要介绍堆排序.

### 定义

堆排序（Heapsort）是指利用 二叉堆→10.7.2@p2026 这种数据结构所设计的一种排序算法. 堆排序的适用数据结构为数组.

### 过程

堆排序的本质是建立在堆上的选择排序.

#### 排序

首先建立大顶堆，然后将堆顶的元素取出，作为最大值，与数组尾部的元素交换，并维持残余堆的性质；

之后将堆顶的元素取出，作为次大值，与数组倒数第二位元素交换，并维持残余堆的性质；

以此类推，在第 $n-1$ 次操作后，整个数组就完成了排序.

#### 在数组上建立二叉堆

从根节点开始，依次将每一层的节点排列在数组里.

于是有数组中下标为 `i` 的节点，对应的父结点、左子结点和右子结点如下：

```
iParent(i) = (i - 1) / 2;
iLeftChild(i) = 2 * i + 1;
iRightChild(i) = 2 * i + 2;
```

### 性质

#### 稳定性

同选择排序一样，堆排序中的交换可能改变相等元素的相对顺序，因此是不稳定的排序算法.

#### 时间复杂度

堆排序的最坏时间复杂度为 $O(n\log n)$.

#### 空间复杂度

由于可以在输入数组上建立堆，所以这是一个原地算法.

### 实现

- C++

```cpp
void sift_down(int arr[], int start, int end) {
  // 计算父结点和子结点的下标
  int parent = start;
  int child = parent * 2 + 1;
  while (child <= end) {  // 子结点下标在范围内才做比较
    // 先比较两个子结点大小，选择最大的
    if (child + 1 <= end && arr[child] < arr[child + 1]) child++;
    // 如果父结点比子结点大，代表调整完毕，直接跳出函数
    if (arr[parent] >= arr[child])
      return;
    else {  // 否则交换父子内容，子结点再和孙结点比较
      swap(arr[parent], arr[child]);
```

```
      parent = child;
      child = parent * 2 + 1;
    }
  }
}

void heap_sort(int arr[], int len) {
  // 从最后一个节点的父节点开始 sift down 以完成堆化（heapify）
  for (int i = (len - 1 - 1) / 2; i >= 0; i--) sift_down(arr, i, len - 1);
  // 先将第一个元素和已经排好的元素前一位做交换，再重新调整（刚调整的元素之前的元素），直到排序完毕
  for (int i = len - 1; i > 0; i--) {
    swap(arr[0], arr[i]);
    sift_down(arr, 0, i - 1);
  }
}

```

- Python

```
def sift_down(arr, start, end):
    # 计算父结点和子结点的下标
    parent = int(start)
    child = int(parent * 2 + 1)
    while child <= end:  # 子结点下标在范围内才做比较
        # 先比较两个子结点大小，选择最大的
        if child + 1 <= end and arr[child] < arr[child + 1]:
            child += 1
        # 如果父结点比子结点大，代表调整完毕，直接跳出函数
        if arr[parent] >= arr[child]:
            return
        else:  # 否则交换父子内容，子结点再和孙结点比较
            arr[parent], arr[child] = arr[child], arr[parent]
            parent = child
            child = int(parent * 2 + 1)


def heap_sort(arr, len):
    # 从最后一个节点的父节点开始 sift down 以完成堆化（heapify）
    i = (len - 2) // 2
    while i >= 0:
        sift_down(arr, i, len - 1)
        i -= 1
    # 先将第一个元素和已经排好的元素前一位做交换，再重新调整（刚调整的元素之前的元素），直到排序完毕
    i = len - 1
    while i > 0:
        arr[0], arr[i] = arr[i], arr[0]
        sift_down(arr, 0, i - 1)
        i -= 1


```

### 外部链接

- 堆排序 - 维基百科，自由的百科全书[1]

### 外部参考资料

[1] 堆排序 - 维基百科, 自由的百科全书

## 5.7.10 桶排序

本页面将简要介绍桶排序.

### 定义

桶排序（Bucket sort）是排序算法的一种，适用于待排序数据值域较大但分布比较均匀的情况.

### 过程

桶排序按下列步骤进行：

1. 设置一个定量的数组当作空桶；
2. 遍历序列，并将元素一个个放到对应的桶中；
3. 对每个不是空的桶进行排序；
4. 从不是空的桶里把元素再放回原来的序列中.

### 性质

#### 稳定性

如果使用稳定的内层排序，并且将元素插入桶中时不改变元素间的相对顺序，那么桶排序就是一种稳定的排序算法.

由于每块元素不多，一般使用插入排序. 此时桶排序是一种稳定的排序算法.

#### 时间复杂度

设 $n$ 为元素个数，$k$ 为桶数. 若元素独立且均匀落桶，桶排序的平均时间复杂度为 $O(n+n^2/k+k)$（将值域平均分成 $k$ 块 + 排序 + 重新合并元素），当 $k \approx n$ 时为 $O(n)$. [10]

桶排序的最坏时间复杂度为 $O(n^2+k)$.

### 实现

下面 `a[]` 下标从 1 开始，要求 $0 \le a_i \le w$、$0 \le n < N$，其中 $w$ 是最大键值的上界.

- C++

```cpp
constexpr int N = 100010;

int n, w, a[N];
vector<int> bucket[N];

void insertion_sort(vector<int>& A) {
  for (int i = 1; i < A.size(); ++i) {
    int key = A[i];
    int j = i - 1;
    while (j >= 0 && A[j] > key) {
      A[j + 1] = A[j];
      --j;
```

[10] Bucket sort - Wikipedia[1]

```
    }
    A[j + 1] = key;
  }
}

void bucket_sort() {
  if (n == 0) return;
  auto bucket_size = unsigned(w / n) + 1;
  for (int i = 0; i < n; ++i) {
    bucket[i].clear();
  }
  for (int i = 1; i <= n; ++i) {
    bucket[a[i] / bucket_size].push_back(a[i]);
  }
  int p = 0;
  for (int i = 0; i < n; ++i) {
    insertion_sort(bucket[i]);
    for (int j = 0; j < bucket[i].size(); ++j) {
      a[++p] = bucket[i][j];
    }
  }
}

```

- Python

```
N = 100010
w = n = 0
a = [0] * N
bucket = [[] for i in range(N)]


def insertion_sort(A):
    for i in range(1, len(A)):
        key = A[i]
        j = i - 1
        while j >= 0 and A[j] > key:
            A[j + 1] = A[j]
            j -= 1
        A[j + 1] = key


def bucket_sort():
    if n == 0:
        return
    bucket_size = w // n + 1
    for i in range(0, n):
        bucket[i].clear()
    for i in range(1, n + 1):
        bucket[a[i] // bucket_size].append(a[i])
    p = 0
    for i in range(0, n):
        insertion_sort(bucket[i])
        for j in range(0, len(bucket[i])):
            p += 1
```

```
30             a[p] = bucket[i][j]
31
32
```

### 参考资料与注释

[1] Bucket sort - Wikipedia

## 5.7.11 希尔排序

本页面将简要介绍希尔排序.

### 定义

希尔排序（Shell sort），也称为缩小增量排序法，是 插入排序→5.7.4@p572 的一种改进版本．希尔排序以它的发明者希尔（Donald Shell）命名.

### 过程

排序对不相邻的记录进行比较和移动:

1. 将待排序序列分为若干子序列（每个子序列的元素在原始数组中间距相同）;
2. 对这些子序列进行插入排序;
3. 减小每个子序列中元素之间的间距，重复上述过程直至间距减少为 1.

为便于分析，下面给出单次间距为 $h$ 的插入排序伪代码，记为 InsertionSort$(h)$．数组 $A$ 的下标从 1 到 $n$，$v$ 保存当前待插入的值；希尔排序按间距从大到小调用这一过程.

```
1 for j ← h + 1 to n
2     v ← A_j
3     i ← j − h
4     while i ≥ 1 and A_i > v
5         A_{i+h} ← A_i
6         i ← i − h
7     A_{i+h} ← v
```

### 性质

#### 稳定性

希尔排序是一种不稳定的排序算法.

#### 时间复杂度

希尔排序的最好时间依赖间距序列和是否提前退出；下面给出的没有提前退出的 $3h+1$ 实现最好为 $\Theta(n \log n)$.

希尔排序的平均时间复杂度和最坏时间复杂度与间距序列的选取有关. 设间距序列为 $H$, 下面给出 $H$ 的两种经典选取方式，这两种选取方式均使得排序算法的复杂度降为 $o(n^2)$ 级别.

**命题 1**

若间距序列为 $H = \{2^k - 1 \mid k = 1, 2, ..., \lfloor \log_2 n \rfloor\}$ (从大到小), 则希尔排序算法的时间复杂度为 $O(n^{3/2})$.