

算法考试和考研机试

权威
宝典

算法笔记

胡凡 曾磊 主编

- ◎ 适合语言零基础的算法初学者
- ◎ 玩转研究生复试
上机考试和PAT甲乙级考试
- ◎ 突破CCF的CSP认证
或其他算法考试
- ◎ 海量例题实战解析
- ◎ 二维码内容实时互动更新
- ◎ 求职面试时的基础算法



机械工业出版社
CHINA MACHINE PRESS

有关此电子书的说明

本人可以帮助你找到你要的PDF电子书，计算机类，文学，艺术，设计，医学，理学，经济，金融等等。质量都很清晰，为方便读者阅读观看，每本100%都带可跳转的书签索引和目录，只要您提供给我书的相关信息，一般我都能找到，如果您有需求，请联系我 QQ1779903665。

PDF代找说明：

本人已经帮助了上万人找到了他们需要的PDF，其实网上有很多PDF，大家如果在网上不到的话，可以联系我QQ，大部分我都可以找到，而且每本100%带书签索引目录。因PDF电子书都有版权，请不要随意传播，如果您有经济购买能力，请尽量购买正版。

提供各种书籍的pd电子版代找服务，如果你找不到自己想要的书的pdf电子版，我们可以帮您找到，如有需要，请联系 QQ 1779903665.

备用:QQ 461573687

若以上联系方式失效，您可通过以下电子邮件获取有效联系方式。

E-mail：ebooksprite@163.com

E-mail：ebooksprite@foxmail.com

若您没有QQ通讯工具，请发送您的请求到 ebooksprite@gmail.com 与客服取得联系。

声明：本人只提供代找服务，每本100%索引书签和目录，因寻找和后期制作pdf电子书有一定难度，仅收取代找费用。如因PDF产生的版权纠纷，与本人无关，我们仅仅只是帮助你寻找到你要的pdf而已。

算 法 笔 记

胡凡 曾磊 主编



机 械 工 业 出 版 社

本书内容包括：C/C++快速入门、入门模拟、算法初步、数学问题、C++标准模板库（STL）、数据结构专题（二章）、搜索专题、图算法专题、动态规划专题、字符串专题、专题扩展。本书印有二维码，用来实时更新、补充内容及发布勘误的。

本书可作为计算机专业研究生入学考试复试上机、各类算法等级考试（如 PAT、CSP 等）的辅导书，也可作为“数据结构”科目的考研教材及辅导书内容的补充。本书还是学习 C 语言、数据结构与算法的入门辅导书，非常适合零基础的学习者对经典算法进行学习。

（编辑邮箱：jinacmp@163.com）

图书在版编目（CIP）数据

算法笔记 / 胡凡，曾磊主编. —北京：机械工业出版社，2016.7

ISBN 978-7-111-54009-0

I. ①算… II. ①胡… ②曾… III. ①电子计算机—算法理论 IV. ①TP301.6

中国版本图书馆 CIP 数据核字（2016）第 129818 号

机械工业出版社（北京市百万庄大街 22 号 邮政编码 100037）

策划编辑：吉 玲 责任编辑：吉 玲 吴晋瑜 王小东

封面设计：鞠 杨 责任印制：李 洋 责任校对：刘怡丹

北京振兴源印务有限公司印刷

2016 年 7 月第 1 版 · 第 1 次印刷

184mm×260mm · 30.75 印张 · 782 千字

标准书号：ISBN 978-7-111-54009-0

定价：65.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

电话服务

服务咨询热线：010-88361066

读者购书热线：010-68326294

010-88379203

封面无防伪标均为盗版

网络服务

机工官网：www.cmpbook.com

机工官博：weibo.com/cmp1952

金书网：www.golden-book.com

教育服务网：www.cmpedu.com

前言

最初打算写这本书是在自己刚考完研之后。那段时间，我每天都在浙江大学天勤考研群里给学弟学妹们答疑，在感受着他们的努力与进步的同时，自己仿佛又经历了一次考研，感慨颇多。渐渐地，出于兴趣，我感觉自己还能为他们做些什么，于是便萌生了写一些东西的想法。由于浙江大学机试就是 PAT 考试，因此一开始只是打算把 PAT 考试题目的题解都写一遍，但是在写作过程中慢慢发现，题解本身并不能给人带来太多的提高，而算法思想的理解和学习才是最为重要的。考虑到当时的算法入门书籍要么偏重于竞赛风格，要么偏重于面试风格，因此我便打算写一本适用于考研机试与 PAT 的算法书籍，以供考研的学弟学妹们学习。因为浙江机试的考试范围已经能覆盖大部分学校的机试范围，所以对于报考其他学校的同学也同样适用。

第一次试印的版本给当年浙江大学机试的平均分提高了十多分，反响不错。但我深知书中仍有许多不足，也有许多想要添加的内容没来得及加进去，因此便又花费了半年时间增加了许多内容。至此，本书已经覆盖了大部分基础经典算法，不仅可以作为考研机试和 PAT 的学习教材，对其他的一些算法考试（例如 CCF 的 CSP 考试）或者考研初试的数据结构科目的学习和理解也很有帮助，甚至仅仅想学习经典算法的读者也能从本书中学到许多知识。由于书中很多内容都来源于自己对算法的理解，因此最终把书名定为《算法笔记》。

本书希望让一个 C 语言零基础的读者能很好地进入本书的学习，因此在第 2 章设置了 C 语言的入门详解，使读者不必因自己不会 C 语言而有所担心，并且在对 C 语言的讲解中融入了部分 C++ 的特性内容，这样读者会更容易书写顺手的代码。第 3~5 章是入门部分，其中介绍了一些算法思想和数学问题，读者可从中学习到一些基础但非常重要的算法思想，并培养基本的思维能力和代码能力。第 6 章介绍了 C++ 标准模板库（STL）的常用内容和 `algorithm` 头文件下的常用函数，以帮助读者节省写代码的时间。第 7~12 章是进阶部分，其中介绍了各类经典数据结构、图算法以及较为进阶的重要算法，以使读者对经典算法和数据结构有较为深入的学习。第 13 章补充了一些上面没有介绍的内容，以帮助读者拓宽视野。

另外，书中印的二维码，是用来更新或补充书籍内容及发布本书勘误的。通过扫描本书的勘误和内容更新日志二维码，读者可以得到实时更新的相应内容。



最后，由于编者水平有限，尽管对本书进行了多次校对，书中可能仍有一些待改进的地方，敬请广大读者提出宝贵建议！

本书的适用范围

- 研究生复试上机考试
- PAT 甲级、乙级考试
- CCF 的 CSP 认证（或其他算法）

- 求职面试时的基础算法考试
- 考研初试数据结构科目
- 经典算法的入门学习

致谢

在本书写作过程中，得到了许多朋友给予的帮助，他们是鲁蕴铖、徐涵、王改革和周伟，他们在本书的内容、细节等方面给出了很多建设性的意见，在此表示衷心的感谢。

参加本书编写的人员还有：曾磊、唐晓瑜、庞志飞、冯杰、刘伟、王改革、柯扬斌、何世伟、朱逸晨、林炆平、杨晓海、庞博、张也、刘阳、吴联坤、于志超、朱清华、陈鸿翔、柴一平、李幸超、李邦鹏、范旭民、李疆、胡学军、厉月艳、朱华、鲁蕴铖、徐涵、王巨峰、金明健、刘欧、田唐昊。

感谢维护 PAT 的浙江大学陈越老师、维护 Codeup 的浙江传媒学院张浩斌老师，他们耐心回复了我关于 PAT 和 Codeup 的使用问题，使我能够更好地使用上面的题目作为例题和练习题。

感谢本书最初试印版本的读者，他们发现了书中的许多错误，并就本书的内容提了许多建议，使得本书更为完善。还有很多朋友对本书的写作十分关心，在他们的鼓励下，我才能在巨大的学业压力中最终完成本书，在此一并表示感谢。

感谢书链团队为本书提供的二维码与资源管理系统，它让本书成为一本可以动态添加内容的书籍，增强了本书的可扩展性。

最后，还要特别感谢机械工业出版社的吉玲编辑，在她的鼓励和帮助下，我顺利完成了本书的编写，并把更好的内容展现给读者。

胡凡

目 录

前言

第 1 章 如何使用本书	1
1.1 本书的基本内容	1
1.2 如何选择编程语言和编译器	1
1.3 在线评测系统	2
1.4 常见的评测结果	3
1.5 如何高效地做题	4
第 2 章 C/C++快速入门	5
2.1 基本数据类型	7
2.1.1 变量的定义	7
2.1.2 变量类型	7
2.1.3 强制类型转换	11
2.1.4 符号常量和 const 常量	12
2.1.5 运算符	14
2.2 顺序结构	17
2.2.1 赋值表达式	17
2.2.2 使用 scanf 和 printf 输入/输出	18
2.2.3 使用 getchar 和 putchar 输入/输出字符	23
2.2.4 注释	24
2.2.5 typedef	24
2.2.6 常用 math 函数	25
2.3 选择结构	28
2.3.1 if 语句	28
2.3.2 if 语句的嵌套	31
2.3.3 switch 语句	32
2.4 循环结构	34
2.4.1 while 语句	34
2.4.2 do... while 语句	35
2.4.3 for 语句	36
2.4.4 break 和 continue 语句	38
2.5 数组	39
2.5.1 一维数组	39
2.5.2 冒泡排序	41
2.5.3 二维数组	43
2.5.4 memset——对数组中每一个元素赋相同的值	46

2.5.5	字符数组	47
2.5.6	string.h 头文件	50
2.5.7	sscanf 与 sprintf	53
2.6	函数	55
2.6.1	函数的定义	55
2.6.2	再谈 main 函数	58
2.6.3	以数组作为函数参数	58
2.6.4	函数的嵌套调用	59
2.6.5	函数的递归调用	60
2.7	指针	61
2.7.1	什么是指针	61
2.7.2	指针变量	62
2.7.3	指针与数组	63
2.7.4	使用指针变量作为函数参数	65
2.7.5	引用	68
2.8	结构体 (struct) 的使用	70
2.8.1	结构体的定义	70
2.8.2	访问结构体内的元素	71
2.8.3	结构体的初始化	72
2.9	补充	74
2.9.1	cin 与 cout	74
2.9.2	浮点数的比较	75
2.9.3	复杂度	78
2.10	黑盒测试	80
2.10.1	单点测试	80
2.10.2	多点测试	80
第 3 章	入门篇 (1) —— 入门模拟	85
3.1	简单模拟	85
3.2	查找元素	87
3.3	图形输出	89
3.4	日期处理	91
3.5	进制转换	93
3.6	字符串处理	95
第 4 章	入门篇 (2) —— 算法初步	99
4.1	排序	99
4.1.1	选择排序	99
4.1.2	插入排序	100
4.1.3	排序题与 sort 函数的应用	101
4.2	散列	106
4.2.1	散列的定义与整数散列	106

4.2.2	字符串 hash 初步	109
4.3	递归	111
4.3.1	分治	111
4.3.2	递归	112
4.4	贪心	118
4.4.1	简单贪心	118
4.4.2	区间贪心	122
4.5	二分	124
4.5.1	二分查找	124
4.5.2	二分法拓展	131
4.5.3	快速幂	134
4.6	two pointers	137
4.6.1	什么是 two pointers	137
4.6.2	归并排序	139
4.6.3	快速排序	142
4.7	其他高效技巧与算法	146
4.7.1	打表	146
4.7.2	活用递推	147
4.7.3	随机选择算法	149
第 5 章	入门篇 (3) —— 数学问题	152
5.1	简单数学	152
5.2	最大公约数与最小公倍数	154
5.2.1	最大公约数	154
5.2.2	最小公倍数	156
5.3	分数的四则运算	156
5.3.1	分数的表示和化简	157
5.3.2	分数的四则运算	157
5.3.3	分数的输出	159
5.4	素数	159
5.4.1	素数的判断	160
5.4.2	素数表的获取	160
5.5	质因子分解	165
5.6	大整数运算	170
5.6.1	大整数的存储	170
5.6.2	大整数的四则运算	171
5.7	扩展欧几里得算法	176
5.8	组合数	181
5.8.1	关于 $n!$ 的一个问题	181
5.8.2	组合数的计算	183
第 6 章	C++ 标准模板库 (STL) 介绍	191

6.1	vector 的常见用法详解	191
6.2	set 的常见用法详解	197
6.3	string 的常见用法详解	202
6.4	map 的常用用法详解	213
6.5	queue 的常见用法详解	218
6.6	priority_queue 的常见用法详解	221
6.7	stack 的常见用法详解	227
6.8	pair 的常见用法详解	230
6.9	algorithm 头文件下的常用函数	232
6.9.1	max()、min()和 abs()	232
6.9.2	swap()	233
6.9.3	reverse()	233
6.9.4	next_permutation()	234
6.9.5	fill()	235
6.9.6	sort()	235
6.9.7	lower_bound()和 upper_bound()	242
第 7 章	提高篇 (1) —— 数据结构专题 (1)	245
7.1	栈的应用	245
7.2	队列的应用	251
7.3	链表处理	253
7.3.1	链表的概念	253
7.3.2	使用 malloc 函数或 new 运算符为链表结点分配内存空间	254
7.3.3	链表的基本操作	256
7.3.4	静态链表	260
第 8 章	提高篇 (2) —— 搜索专题	269
8.1	深度优先搜索 (DFS)	269
8.2	广度优先搜索 (BFS)	274
第 9 章	提高篇 (3) —— 数据结构专题 (2)	283
9.1	树与二叉树	283
9.1.1	树的定义与性质	283
9.1.2	二叉树的递归定义	284
9.1.3	二叉树的存储结构与基本操作	285
9.2	二叉树的遍历	289
9.2.1	先序遍历	289
9.2.2	中序遍历	290
9.2.3	后序遍历	291
9.2.4	层序遍历	292
9.2.5	二叉树的静态实现	298
9.3	树的遍历	302
9.3.1	树的静态写法	302

9.3.2	树的先根遍历	303
9.3.3	树的层序遍历	303
9.3.4	从树的遍历看 DFS 与 BFS	304
9.4	二叉查找树 (BST)	310
9.4.1	二叉查找树的定义	310
9.4.2	二叉查找树的基本操作	310
9.4.3	二叉查找树的性质	314
9.5	平衡二叉树 (AVL 树)	319
9.5.1	平衡二叉树的定义	319
9.5.2	平衡二叉树的基本操作	320
9.6	并查集	328
9.6.1	并查集的定义	328
9.6.2	并查集的基本操作	328
9.6.3	路径压缩	330
9.7	堆	335
9.7.1	堆的定义与基本操作	335
9.7.2	堆排序	339
9.8	哈夫曼树	342
9.8.1	哈夫曼树	342
9.8.2	哈弗曼编码	345
第 10 章	提高篇 (4) ——图算法专题	347
10.1	图的定义和相关术语	347
10.2	图的存储	348
10.2.1	邻接矩阵	348
10.2.2	邻接表	348
10.3	图的遍历	350
10.3.1	采用深度优先搜索 (DFS) 法遍历图	350
10.3.2	采用广度优先搜索 (BFS) 法遍历图	359
10.4	最短路径	367
10.4.1	Dijkstra 算法	367
10.4.2	Bellman-Ford 算法和 SPFA 算法	391
10.4.3	Floyd 算法	398
10.5	最小生成树	400
10.5.1	最小生成树及其性质	400
10.5.2	prim 算法	401
10.5.3	kruskal 算法	409
10.6	拓扑排序	414
10.6.1	有向无环图	414
10.6.2	拓扑排序	415
10.7	关键路径	417

10.7.1	AOV 网和 AOE 网	417
10.7.2	最长路径	419
10.7.3	关键路径	419
第 11 章	提高篇 (5) —— 动态规划专题	425
11.1	动态规划的递归写法和递推写法	425
11.1.1	什么是动态规划	425
11.1.2	动态规划的递归写法	425
11.1.3	动态规划的递推写法	426
11.2	最大连续子序列和	429
11.3	最长不下降子序列 (LIS)	432
11.4	最长公共子序列 (LCS)	434
11.5	最长回文子串	436
11.6	DAG 最长路	439
11.7	背包问题	442
11.7.1	多阶段动态规划问题	442
11.7.2	01 背包问题	443
11.7.3	完全背包问题	446
11.8	总结	447
第 12 章	提高篇 (6) —— 字符串专题	449
12.1	字符串 hash 进阶	449
12.2	KMP 算法	455
12.2.1	next 数组	456
12.2.2	KMP 算法	458
12.2.3	从有限状态自动机的角度看待 KMP 算法	463
第 13 章	专题扩展	465
13.1	分块思想	465
13.2	树状数组 (BIT)	470
13.2.1	lowbit 运算	470
13.2.2	树状数组及其应用	470
参考文献		481

第 1 章 如何使用本书

1.1 本书的基本内容

本书旨在让一个 C 语言零基础算法的学习者循序渐进地学习经典算法，因此在第 2 章对 C 语言的语法进行了详细的入门讲解，并在其中融入了部分 C++ 的特性，以便读者能够更容易地书写代码。

第 3~5 章是入门部分。其中第 3 章将初步训练读者最基本的编写代码能力，内容比较少，建议读者用较少的时间完成；第 4 章对常用的基本算法思想进行介绍，内容非常重要，建议读者多花一些时间仔细思考和训练；第 5 章是一些数学问题，其中 5.7 节的内容和 5.8 节的后半部分内容相对没有那么容易，读者可以选择性阅读。

第 6 章介绍了 C++ 标准模板库 (STL) 中的常用容器和 `algorithm` 头文件下的常用函数，通过学习本章，读者可以节省许多写代码的时间，把注意力更多地放在解决问题上。需要说明的是，此部分内容不难，读者不必对难度有所担心，但还应认真实现其中给出的实例。另外，部分内容可能会在前几章用到，因此，如果在前几章中需要用到本章的内容（需要使用时书中会给出说明），那么请在本章来阅读相关内容，然后再回头去继续学习。

第 7~12 章是进阶部分。其中第 7 章介绍了栈、队列和链表，第 8 章介绍了深度优先搜索和广度优先搜索，它们是树和图算法学习的基础，需要读者认真学习并掌握；第 9 章和第 10 章分别讲解了树和图的相关算法，它们是数据结构中非常重要的内容，在很多考试中也经常会出现，需要读者特别关注；第 11 章介绍了动态规划算法的几个经典模型，并进行了相应的总结；第 12 章对字符串 `hash` 和 `KMP` 算法进行了探讨。

第 13 章是在前面章节的基础上额外增加的内容，需要读者花一些时间来阅读并掌握。

根据不同的需要，读者可以不同的方式来学习本书。就研究生复试上机来说，不同的学校对机试的要求不同，因此需要根据报考学校的机考大纲来确定需要学习哪些内容。本书覆盖了大部分学校的机试内容，对于 PAT 乙级考试，前 7 章内容已经基本够用；对于 PAT 甲级考试，本书的大部分内容都要掌握（冷门考点会在“本章二维码”中给出）；对于 CCF 的 CSP 认证，本书能覆盖竞赛内容以外的考点（最后一题偶尔会涉及 ACM-ICPC 的内容，超出了本书范围）；对于考研初试数据结构科目，本书能帮助读者更好地理解各种数据结构与算法。

另外，本书还有一本配套习题集——《算法笔记上机训练实战指南》，书中给出了 PAT 乙级前 50 题、甲级前 107 题的详细题解。若读者想要对知识点的熟练度有进一步的提升，推荐同时使用本书的配套习题集（最新的考题会在本书的二维码中添加更新）。习题集的题目顺序是按本书的章节顺序编排的，并配有十分详细的题解，以便读者更有效地进行针对性训练。推荐使用“阅读一节本书的内容，然后做一节习题集对应小节的题目”的训练方式。

1.2 如何选择编程语言和编译器

很多考试都会限定程序的运行时间的上限，因此选择尽可能快的编程语言是非常重要的。

一般来说,可供选择的语言有 C、C++、Java 等,但是 Java 的执行比较慢,因此较常使用的是 C 或者 C++。考虑到 C++ 的语法向下兼容 C,并且 C 的输入输出语句比 C++ 的要快很多,因此我们可以在主体上使用 C 语言的语法;而 C++ 中有一些特性和功能非常好用(例如变量可以随时定义、拥有标准模板库 STL 等),因此在一定程度上我们可以混用部分 C++ 的语法(事实上,由于 C++ 向下兼容 C,因此一般都是在 C++ 中写 C 语言的语法,相关内容参见第 2 章)。

编译器的选择则因人而异、因现场环境而异。不同的考试可能提供不同的编译器,要根据具体情况来选择。但一般来说,可能出现的编译器有 VC 6.0、VS 系列、Dev-C++、C-Free、Code::Blocks、Eclipse 等,其中 VC 6.0 因为标准过于古老,很多语法在其中没办法通过编译,所以尽量不要使用;Dev-C++、C-Free、Code::Blocks 则是轻便好用的编译器,推荐使用,可以根据具体情况来选择;VS 系列是较为厚重的编译器,在没有其他轻便编译器可供选择的情况下使用;Eclipse 则更常用于 Java 代码的编写。

1.3 在线评测系统

在各类考试中,判断程序写得对不对,一般需要借助在线评测系统(Online Judge, OJ)。一般来说,在 OJ 上可以看到题目的题目描述、输入格式、输出格式、样例输入及样例输出。我们需要根据题目来写出相应的代码,然后提交给 OJ 进行评测。OJ 的后台会让程序运行很多组数据,并根据程序输出结果的正确与否返回不同的结果(具体的结果在 1.4 节中讲述)。注意:即便代码能通过样例,也不能说明是完全正确的,因为后台会有很多组数据,样例只是一个示范而已。

本书的例题与练习题将来自 PAT 与 codeup,下面对其分别介绍。

(1) PAT

PAT 的全称为 Programming Ability Test,是考察计算机程序设计能力的一个考试,目前分为乙级(Basic)、甲级(Advanced)和顶级(Top)三个难度层次(需要选择其中一个报名),难度依次递增,其中顶级将涉及大量 ACM-ICPC 竞赛的考点,报考的人数也相对较少。本书能够覆盖乙级和甲级的考试知识点,并且书中有很多例题都来自 PAT 甲、乙级的真题。

为便于区别,本书中来自乙级的题目将以 B 开头,例如 B1003 表示乙级的题号为 1003 的题目。PAT 乙级真题题库地址如下:

<http://www.patest.cn/contests/pat-b-practise>

同样的,本书中来自甲级的题目将以 A 开头,例如 A1066 表示甲级的题号为 1066 的题目。PAT 甲级真题题库地址如下:

<http://www.patest.cn/contests/pat-a-practise>

对于来自 PAT 甲、乙级的题目,只需要在网站上面注册一个账号即可提交,例如对 B1001 来说,在阅读完题目并写出相应的代码后,只需要单击最下方的“提交代码”,接着选择代码对应的语言(例如 C 或者 C++ 代码的提交都可以选择“C++(g++ 4.7.2)”这一项),然后把代码粘贴在编辑框内,单击“提交代码”即可,稍等片刻刷新页面即可得到程序评测的结果。注意: PAT 的评测方式为“单点测试”,即代码只需要能够处理一组数据的输入即可,后台会多次运行代码来测试不同的数据,然后对每组数据都返回相应的结果。“单点测试”的具体写法见 2.10.1 节。

(2) codeup

codeup 是一个有着很多题目的题库，当然它也是一个在线评测系统，本书的部分例题和练习题将来自于 codeup，其地址如下：

<http://www.codeup.cn/>

在注册一个账号之后，单击最上面一栏中的“题目集”即可进入题库，之后可以根据题目标题的关键字或者题号本身来搜索题目。当然，它对题目进行了归类，读者也可以直接从分类中选择自己想要训练的算法类型。

对 codeup 的题目而言，页面中同样有着题目描述、输入格式、输出格式、样例输入和样例输出。一旦写好了代码，只需要单击最下方的“提交”，然后选择代码对应的语言（例如 C 和 C++ 只需要一律选择 C++ 即可），接着在编辑框内粘贴代码，单击“提交”，过几秒后刷新页面即可得到程序评测的结果。注意：codeup 的评测方式为“多点测试”（除了第 2 章的 C 语言练习题外），即代码需要能够处理所有数据的输入，也就是说，后台只会运行代码一次来测试不同的数据，只有当所有数据都输出正确的结果时才会让程序通过，只要有一组数据错误，就会返回对应的错误类型。“多点测试”的具体写法见 2.10.2。

除了 PAT 和 codeup 之外，还有很多优秀的在线评测系统，但是大多数都是侧重于竞赛的，因此如果不是算法竞赛的话，一般不需要去它们上面做题。下面是几个较知名 OJ 的地址：

POJ: <http://poj.org/>

HDOJ: <http://acm.hdu.edu.cn/>

ZOJ: <http://acm.zju.edu.cn/>

CodeForces: <http://codeforces.com/>

UVa: uva.onlinejudge.org

ACdream: <http://acdream.info/>

1.4 常见的评测结果

(1) 答案正确 (Accepted, AC)

恭喜你！所提交的代码通过了数据！这个评测结果应该是大家最喜欢见到的，也非常好理解。如果是单点测试，那么每通过一组数据，就会返回一个 Accepted；如果是多点测试，那么只有当通过了所有数据时，才会返回 Accepted。

(2) 编译错误 (Compile Error, CE)

很显然，如果代码没有办法通过编译，那么就会返回 Compile Error。这时要先注意是不是选错了语言，然后再看本地的编译器能不能编译通过刚刚提交的代码，修改之后再次提交即可。

(3) 答案错误 (Wrong Answer, WA)

“答案错误”是比较令人懊恼的结果，因为这说明代码有漏洞或者算法根本就是错误的，只是恰好能过样例而已。不过有时可能是因为输出了一些调试信息导致的，那就删掉多余的输出内容再输出。当然，大部分情况下都需要认真检查代码的逻辑有没有问题。

(4) 运行超时 (Time Limit Exceeded, TLE)

由于每道题都会规定程序运行时间的上限，因此当超过这个限制时就会返回 TLE。一般来说，这一结果可能是由算法的时间复杂度过大而导致的，当然也可能是某组数据使得代码

中某处地方死循环了。因此，要仔细思考最坏时间复杂度是多少，或者检查代码中是否可能出现特殊数据死循环的情况。

(5) 运行错误 (Runtime Error, RE)

这一结果的可能性非常多，常见的有段错误（直接的原因是非法访问了内存，例如数组越界、指针乱指）、浮点错误（例如除数为 0、模数为 0）、递归爆栈（一般由递归时层数过深导致的）等。一般来说，需要先检查数组大小是否比题目的数据范围大，然后再去检查不可能有特殊数据可以使除数或模数为 0，有递归的情况则检查是否在大数据时递归层数太深。

(6) 内存超限 (Memory Limit Exceeded, MLE)

每道题目都会有规定程序使用的空间上限，因此如果程序中使用太多的空间，则会返回 MLE，例如数组太大一般最容易导致这个结果。

(7) 格式错误 (Presentation Error, PE)

这应该是最接近 Accepted 的错误了，基本是由多输出了空格或者换行导致的，稍作修改即可。

(8) 输出超限 (Output Limit Exceeded, OLE)

如果程序输出了过量的内容（一般是指过量非常多），那么就会返回 OLE。一般是由输出了大量的调试信息或者特殊数据导致的死循环输出导致的。

1.5 如何高效地做题

一般来说，按照算法专题进行集中性的题目训练是算法学习的较好方法，因为这可以一次性地对某个算法有一个较为深入且细致的训练，而随意乱做或是按题号从小到大地“刷”题目并不是一个很好的选择，也无法形成完整的知识体系。

如果在做一道题时暂时没有想法，那么可以先放着，跳过去做其他题目，过一段时间再回来重新做，或许就柳暗花明了（例如你可以设置一个未解题目的队列，每次有题目暂时不会做时就扔到队列里，然后隔三差五取出里面的题目再想一下，想不出来就再扔到队列里……）。当然，如果题目本身较难，做了很久也没有想法，也可以查看题解，知道做法之后再自己独立完成代码过程。

另外，在做题时也可以适当总结相似题目的解题方法，这也是进行专题训练时可以顺带完成的事，可起到事半功倍的效果。



本章二维码

第2章 C/C++快速入门

考虑到一些参加考研机试或其他算法考试的读者并没有学过 C 语言或者 C++，而在机考中最适合使用的就是这两个语言，因此编写了 C/C++入门的部分。该部分旨在让没有接触过 C 语言的读者能够快速上手编写 C 程序，因此对一些不影响实际编程的语法点采取了舍去的策略。如果读者备考的时间比较短且又没有学习过 C 语言，那么建议只要把本书提到的语法点掌握即可；如果读者想要系统、完整地学习 C 语言，则建议参考那些专门讲解 C 语言的书籍。

虽然是 C/C++的入门，但是本书主要侧重于 C 语言的讲解，这是因为 C++中大部分语法在机考中都是用不到的，并且 C++向下兼容 C，因此读者可以用 C++中一些很好用的特性来取代 C 语言中那些不太顺手的设定。

如果读者确实从未接触过编程语言，那么最好把下面接触到的程序和语句都自己尝试编写并运行一下，而不是仅仅是“看过看懂”的状态，因为在实际编程之前，任何“觉得自己会了”的想法都不过是“纸上谈兵”。

如果读者已经学习过 C 语言或者 C++，那么可以选择跳过这部分，但是最好可以大致浏览一下，看看有没有以前没有确切接触过的细节，例如“引用”、int 型范围、指针的错误写法、结构体的构造函数、浮点型的比较等。

注意：有些读者认为学过 C++之后就没有必要学 C 语言，甚至觉得 C 语言太麻烦而不学，这是不太正确的。因为就机考使用的语法而言，除了输入和输出部分，其余顺序结构、分支结构、循环结构、数组、指针都是几乎一样的，学习 C 语言并不会带来什么负担。对于让 C++使用者觉得麻烦的 scanf 函数和 printf 函数，虽然必须承认 cin 和 cout 可以不指定输入输出格式比较方便，但是 cin 和 cout 消耗的时间比 scanf 和 printf 多得多，很多题目可能输入还没结束就超时了。当然，读者可以在某次使用 cin 和 cout 超时，改成 scanf 和 printf 后通过的时候，痛下决心以后使用 scanf 和 printf。顺便指出，请不要同时在一个程序中使用 cout 和 printf，有时候会出问题。

最后再次强调，本书会使用一些代码作为举例，希望读者能够亲自照着输入并理解一下，这将对语言的学习大有帮助。如果代码中出现某些暂时还没有提到的语法，不妨只要先知道是什么意思，具体细节可以等后面提到再学。

下面开始介绍 C 语言的相关内容。

先来看一段 C 语言小程序：

```
#include <stdio.h>
int main(){
    int a, b;
    scanf("%d%d", &a, &b);
    printf("%d", a+b);
    return 0;
}
```


请读者在编译器中输入这段代码，并将其保存为.cpp 文件（C 语言的文件扩展名为.c，但是为了使用 C++中的一些好用的特性，请把文件扩展名改为 C++的文件扩展名.cpp）。

这个程序分为两个部分：头文件和主函数。

1. 头文件

在上面的代码中，`#include <stdio.h>`这一行就是头文件。其中，`stdio.h` 是标准输入输出库，如果在程序中需要输入输出，就需要加上这个头文件。不过一般来说程序都是需要输入输出的，所以基本上每一个 C 程序都需要加上头文件。

`stdio` 的全称是 `standard input output`，`h` 就是 `head` 的缩写，`.h` 是头文件的文件格式。我们可以这样理解：`stdio.h` 就是一个文件，这个文件中包含了一些跟输入输出有关的东西，如果程序需要输入输出，就要通过 `#include <×××>` 的写法来包含（`include`）这个头文件，这样才可以使用 `stdio.h` 这个文件里的输入输出函数。

既然 `stdio.h` 是负责输入输出，那么自然还会有负责其他功能的头文件。例如，`math.h` 负责一些数学函数，`string.h` 负责跟字符串有关的函数，则只需要在需要使用对应的函数时，将它们的头文件包含到这个程序中即可。

此外，在 C++ 的标准中，`stdio.h` 更推荐使用等价写法：`cstdio`，也就是在前面加一个 `c`，然后去掉 `h` 即可。所以 `#include <stdio.h>` 和 `#include <cstdio>` 的写法是等价的，`#include <math.h>` 和 `#include <cmath>` 等价，`#include <string.h>` 和 `#include <cstring>` 也等价。读者在程序中看到这种写法应当能明白它的意思。

2. 主函数

```
int main(){
    ...
    return 0;
}
```

上面的代码就是主函数。主函数是一个程序的入口位置，整个程序从主函数开始执行。一个程序最多只能有一个主函数。

下面来看一下省略号中的内容，读者暂时只需要大致了解每个语句的作用，因为会在后面仔细讲解这些语法。

```
int a, b;
```

这句话定义了两个变量 `a` 和 `b`，类型是 `int` 型（简单来说就是整数）。

```
scanf("%d%d", &a, &b);
```

`scanf` 用来读入数据，这条语句以 `%d` 的格式输入 `a` 和 `b`，其中 `%d` 就是 `int` 型的输入输出标识。简单来说，就是把 `a` 和 `b` 作为整数输入。

```
printf("%d", a + b);
```

`printf` 用来输出数据，这条语句计算 `a + b` 并以 `%d` 格式输出。上面说过，`%d` 就是 `int` 型的输入输出标识，所以就是把 `a + b` 作为整数输出。因此这段代码的主函数实现了输入两个数 `a` 和 `b` 然后输出 `a+b` 的功能。

接下来进入正题，讲解一下 C 语言中各个需要使用的语法。

声明：下文使用的代码请保存成.cpp 文件（即 C++ 文件），然后选择 C++ 语言（或 C++）进行提交。由于 C++ 向下兼容 C，因此采用这种方式可以尽可能防止一些因 C 与 C++ 之间的区分而导致的编译错误。

2.1 基本数据类型

2.1.1 变量的定义

变量是在程序运行过程中其值可以改变的量，需要在定义之后才可以使用，其定义格式如下：

变量类型 变量名；

并且，变量可以在定义的时候就赋初值：

变量类型 变量名 = 初值；

变量名一般来说可以任意取，只是需要满足几个条件：

- ① 不能是 C 语言标识符（标识符不多，比如 for、if、or 等都不能作为变量名，因为它们在 C 语言中本身有含义）。所以 ZJU、Love 等都可以用作变量名，但还是建议取有实际意义的变量名，这样可以提高程序的可读性。
- ② 变量名的第一个字符必须是字母或下画线，除第一个字符之外的其他字符必须是字母、数字或下画线。因此 abc、_zju123_ujz 是合法的变量名，6abc 是不合法的变量名。
- ③ 区分大小写，因此 Zju 和 zju 可以作为两个不同的变量名。

2.1.2 变量类型

一般来说，基本数据类型分为整型、浮点型、字符型，C++中又包括布尔型。每种类型里面又可以分为若干种类型（为了方便记忆，只列出常用的）。表 2-1 中列出了四种基本数据类型。

表 2-1 四种基本数据类型

	类型	取值范围	大致范围
整型	int	$-2147483648 \sim +2147483647$ (即 $-2^{31} \sim +(2^{31}-1)$)	$-2 \times 10^9 \sim 2 \times 10^9$
	long long	$-2^{63} \sim +(2^{63}-1)$	$-9 \times 10^{18} \sim 9 \times 10^{18}$
浮点型	float	$-2^{128} \sim +2^{128}$ (实际精度 6~7 位)	实际精度 6~7 位
	double	$-2^{1024} \sim +2^{1024}$ (实际精度 15~16 位)	实际精度 15~16 位
字符型	char	$-128 \sim +127$	$-128 \sim +127$
布尔型	bool	0(false) or 1(true)	0(false) or 1(true)

1. 整型

整型一般可以分为短整型 (short)、整型 (int) 和长整型 (long long)，其中短整型 (short) 一般用不到，此处不再赘述。下面介绍整型 (int) 和长整型 (long long)，其中整型 int 也被称为 long int，长整型 long long 也被称为 long long int。

① 对整型 int 来说，一个整数占用 32bit，也即 4Byte，取值范围是 $-2^{31} \sim +(2^{31}-1)$ 。如果对范围不太有把握，可以记住绝对值在 10^9 范围以内的整数都可以定义成 int 型。示例如下：

```
int num;
```

```
int num = 5;
```

② 对长整型 **long long** 来说，一个整数占用 64bit，也即 8Byte，取值范围是 $-2^{63} \sim +(2^{63}-1)$ ，也就是说，如果题目要求的整数取值范围超过 2147483647（例如 10^{10} 或者 10^{18} ），就得用 **long long** 型来存储。定义举例：

```
long long bignum;
long long bignum = 123456789012345LL;
```

注意：如果 **long long** 型赋大于 $2^{31}-1$ 的初值，则需要在初值后面加上 LL，否则会编译错误。

除此之外，对于整型数据，都可以在前面加个 **unsigned**，以表示无符号型，例如 **unsigned int** 和 **unsigned long long**，占用的位数和原先相同，但是把负数范围挪到正数上来了。也就是说，**unsigned int** 的取值范围是 $0 \sim 2^{32}-1$ ，**unsigned long long** 的取值范围是 $0 \sim 2^{64}-1$ 。一般来说，很少会出现必须使用 **unsigned int** 和 **unsigned long long** 的情况，因此初学者只需要熟练使用 **int** 和 **long long** 即可。

下面给出一个跟整型有关的程序：

```
#include <stdio.h>
int main(){
    int a = 1, b = 2;
    printf("%d", a + b);
    return 0;
}
```

这段代码首先定义了 **int** 型变量 **a** 和 **b**，并分别赋初值 1 和 2，然后使用 **printf** 输出了 **a+b**。其中 **%d** 是 **int** 型的输出格式。

输出结果：

3

简单来说，需要记住的是，看到题目要求 10^9 以内或者说 32 位整数，就用 **int** 型来存放；如果是 10^{18} 以内（例如 10^{10} ）或者说 64 位整数，就要用 **long long** 型来存放。

2. 浮点型

通俗来讲，浮点型就是小数，一般可以分为单精度（**float**）和双精度（**double**）。

① 对单精度 **float** 来说，一个浮点数占用 32bit，其中 1bit 作为符号位、8bit 作为指数位、23bit 作为尾数位（了解即可），可以存放的浮点数的范围是 $-2^{128} \sim +2^{128}$ ，但是其有效精度只有 6~7 位（由 2^{23} 可以得到，读者只需要知道 6~7 位有效精度即可）。这对一些精度要求比较高的题目是不合适的。定义举例：

```
float fl;
float fl = 3.1415;
```

② 对双精度 **double** 来说，一个浮点数占用 64bit，其中依照浮点数的标准，1bit 作为符号位、11bit 作为指数位、52bit 作为尾数位，可以存放的浮点数的范围是 $-2^{1024} \sim +2^{1024}$ ，其有效精度有 15~16 位，比 **float** 优秀许多。示例如下：

```
double db;
double db = 3.1415926536;
```

下面给出一个跟浮点型相关的程序：


```
#include <stdio.h>

int main(){
    double a = 3.14, b = 0.12;
    double c = a + b;
    printf("%f", c);
    return 0;
}
```

这段代码定义了 `double` 型变量 `a` 和 `b`，并分别赋初值 3.14 和 0.12，然后把 `a + b` 赋值给 `c`，最后输出 `c`。其中 `%f` 是 `float` 和 `double` 型的输出格式。

输出结果：

3.26

因此，对浮点型来说，只需要记住一点，不要使用 `float`，碰到浮点型的数据都应该用 `double` 来存储。

3. 字符型

(1) 字符变量和字符常量

字符型变量的定义方法如下：

```
char c;
char c = 'e';
```

如何理解字符常量？可以先从这么一个角度考虑：假设现在使用“`int num`”的方式定义了一个整型变量 `num`，那么 `num` 就是一个可以被随时赋值的变量；而对于一个整数本身（比如 5），它并不能被改变、不能被赋值，那么可将其称为“整型常量”。同理，如果是通过“`char c`”的方式定义了一个字符，那么 `c` 在这里就被称作“字符变量”，它可以被赋值；但是如果是一个字符本身（例如小写字母‘e’），它是一个没有办法被改变其值的东西，这和前面的整数 5 是一样的，则将其称为“字符常量”。事实上，对单个的字符‘Z’、‘J’、‘U’，都可以把它们称作“字符常量”。字符常量可以被赋值给字符变量，就跟整型常量可以被赋值给整型变量一样。

在 C 语言中，字符常量使用 **ASCII 码** 统一编码。标准 ASCII 码的范围是 0 ~ 127，其中包含了控制字符或通信专用字符（不可显示）和常用的可显示字符。在键盘上，通过敲击可以在屏幕上显示的字符就是可显示字符，比如 0 ~ 9、A ~ Z、a ~ z 等都是可显示字符，它们的 ASCII 码分别是 48 ~ 57、65 ~ 90、97 ~ 122，不过具体数字不需要记住，只要知道小写字母比大写字母的 ASCII 码值大 32 即可。

注意：字符常量必须用单引号标注起来，以区分是作为字符变量还是字符常量出现。正如上面的例子，使用 `char c` 的方式定义了字符变量 `c` 之后，如果出现了字符常量‘c’又不加单引号，那么就会产生误解。为此，在 C 语言中，字符常量（必须是单个字符）必须用单引号标注，例如上面提到的‘Z’、‘J’、‘U’就都使用了单引号标注，以表明它们是字符常量。

最后来看一个程序：

```
#include <stdio.h>

int main(){
    char c1 = 'z', c2 = 'j', c3 = 117;
    printf("%c%c%c", c1, c2, c3);
    return 0;
}
```

```
}
```

输出结果:

```
zju
```

有些读者可能会感到奇怪,为什么字符型变量 `c3` 可以被赋值整数 117,而且最后居然输出了字符'u'?其实在计算机内部,字符就是按 ASCII 码存储的,'u'的 ASCII 码就是 117,因此将 117 赋值给 `c3` 其实就是把 ASCII 码赋值给 `c3`。而且从代码中可以发现,在赋值时,117 并没有加单引号。因此这种写法是成立的。(最后说明一下, `%c` 是 `char` 型的输出格式)

(2) 转义字符

上面提到,ASCII 码中有一部分是控制字符,是不可显示的。像换行、删除、Tab 等都是控制字符。那么在程序中怎样表示一个控制字符呢?对一些常用的控制字符,C 语言中可以用一个右斜线加一些特定的字母来表示。例如,换行通过“`\n`”来表示,Tab 键通过“`\t`”来表示。由于这种情况下斜线后面的字母失去了本身的含义,因此又称为“转义字符”。在实际做题时,比较常用的转义字符就只有下面两个,希望读者能够记住。

`\n` 代表换行

`\0` 代表空字符 NULL,其 ASCII 码为 0,请注意 `\0` 不是空格

再来看一个程序:

```
#include <stdio.h>

int main(){
    int num1 = 1, num2 = 2;
    printf("%d\n\n%d", num1, num2);
    printf("%c", 7);
    return 0;
}
```

输出结果:

```
1
```

```
2
```

可以发现,在第一个 `printf` 中使用了两个 `\n` 来换行,说明在 `num1` 输出后要连续换行两次再输出 `num2`,就得到了上面的输出结果。第二个 `printf` 则没有显示任何输出,因为 ASCII 为 7 的字符是控制字符,并且是控制响铃功能的控制字符,不出意外的话,计算机会响一下。

(3) 字符串常量

字符串是由若干字符组成的串,在 C 语言中没有单独一种基本数据类型可以存储(C++ 中有 `string` 类型),只能使用字符数组的方式。因此这里先介绍字符串常量。

上面提到,字符常量就是单个使用单引号标记的字符,那么此处的字符串常量则是由双引号标记的字符集,例如“`WoAiDeRenBuAiWo`”就是一个字符串常量。

字符串常量可以作为初值赋给字符数组,并使用 `%s` 的格式输出。

下面来看一个程序:

```
#include <stdio.h>

int main(){
    char str1[25] = "Wo ai de ren bu ai wo";
```

```

char str2[25] = "so sad a story it is.";
printf("%s, %s", str1, str2);
return 0;
}

```

输出结果:

```
Wo ai de ren bu ai wo, so sad a story it is.
```

在上面的代码中, `str1[25]`和 `str2[25]`均表示由 25 个 `char` 字符组合而成的字符集合, 可称其为字符数组。在 `printf` 中使用两个 `%s` 分别将它们输出。

最后指出, 不能把字符串常量赋值给字符变量, 因此 `char c = "abcd"` 的写法是不允许的。

4. 布尔型

布尔型在 C++ 中可以直接使用, 但在 C 语言中必须添加 `stdbool.h` 头文件才可以使用。布尔型变量又称为“bool 型变量”, 它的取值只能是 `true` (真) 或者 `false` (假), 分别代表非零与零。在赋值时, 可以直接使用 `true` 或 `false` 进行赋值, 或是使用整型常量对其进行赋值, 只不过整型常量在赋值给布尔型变量时会自动转换为 `true` (非零) 或者 `false` (零)。注意: “非零”是包括正整数和负整数的, 即 1 和 -1 都会转换为 `true`。但是对计算机来说, `true` 和 `false` 在存储时分别为 1 和 0, 因此如果使用 `%d` 输出 bool 型变量, 则 `true` 和 `false` 会输出 1 和 0。

下面来看一个例子 (请将文件扩展名设为 `.cpp`, 否则需要添加 `#include <stdbool.h>` 头文件):

```

#include <stdio.h>
int main(){
    bool flag1 = 0, flag2 = true;
    int a = 1, b = 1;
    printf("%d %d %d\n", flag1, flag2, a==b);
    return 0;
}

```

运行结果:

```
0 1 1
```

在上面的代码中, 把 `flag1` 赋值为 0, `flag2` 赋值为 `true` (也就是 1), 然后将它们输出。比较有疑问的是为什么把 `a == b` (`a` 等于 `b`) 当作整数输出的时候会是 1。事实上, 系统会把 `a == b` 作为一个条件, 判断其是否为真。由于 `a` 和 `b` 均为 1, 显然是相等的, 因此 `a == b` 为真 (`true`), 即输出 1。

2.1.3 强制类型转换

有时需要把浮点数的小数部分切掉来变成整数, 或是把整型变为浮点型来方便做除法 (因为整数除以整数在计算机中视为整除操作, 不会自动变为浮点数), 或是在其他很多情况下, 都会要用到强制类型转换, 即把一种数据类型转换成另一种数据类型。

强制类型转换的格式如下:

```
(新类型名) 变量名
```

这其实很简洁, 只需要把需要变成的类型用括号括着写在前面就行了。

下面给出一个例子来说明:

```
#include <stdio.h>

int main(){
    double r = 12.56;
    int a = 3, b = 5;
    printf("%d\n", (int)r);
    printf("%d\n", a / b);
    printf("%.1f", (double)a / (double)b);
    return 0;
}
```

输出结果:

```
12
0
0.6
```

这个例子使用了“(int)r”来把r强制转换成int型并输出,使用了“(double)a”把a强制转换成浮点型来做除法,输出格式中的“%.1f”是指保留一位小数输出。

需要指出的是,如果将一个类型的变量赋值给另一个类型的变量,却没有写强制类型转换操作,那么编译器将会自动进行转换。但是这并不是说任何时候都可以不用写强制类型转换,因为如果是在计算的过程中需要转换类型,那么就不能等它算完再在赋值的时候转换。

2.1.4 符号常量和 const 常量

符号常量通俗地讲就是“替换”,即用一个标识符来替代常量,又称为“宏定义”或者“宏替换”。其格式如下:

```
#define 标识符 常量
```

例如下面这个例子是把圆周率pi设置为3.14,注意:末尾不加分号:

```
#define pi 3.14
```

于是在程序中凡是使用pi的地方将在程序执行前全部自动替换为3.14。下面这个程序就用于计算半径为3的圆的近似面积:

```
#include <stdio.h>

#define pi 3.14

int main(){
    double r = 3;
    printf("%f\n", pi * r * r);
    return 0;
}
```

输出结果:

```
28.26
```

另一种定义常量的方法是使用const,其格式如下:

```
const 数据类型 变量名 = 常量;
```

仍然用pi来举例:

```
const double pi = 3.14;
```


下面的程序用以输出一个半径为 3 的圆的近似周长：

```
#include <stdio.h>
const double pi = 3.14;
int main(){
    double r = 3;
    printf("%f\n", 2 * pi * r);
    return 0;
}
```

输出结果：

18.84

这两种写法都被称为常量，这是因为它们一旦确定其值后就无法改变，例如 $\text{pi} = \text{pi} + 1$ 的写法就是不行的。这两种方法采用哪种都可，一般都不会出错，但推荐 `const` 的写法。

题外话：`define` 除了可以定义常量外，其实可以定义任何语句或片段。其格式如下：

`#define 标识符 任何语句或片段`

例如可以写一个这样的宏定义：

```
#define ADD(a, b) ((a)+(b))
```

这样就可以直接使用 `ADD(a,b)` 来代替 `a + b` 的功能：

```
#include <stdio.h>
#define ADD(a, b) ((a)+(b))
int main(){
    int num1 = 3, num2 = 5;
    printf("%d", ADD(num1, num2));
    return 0;
}
```

输出结果：

8

有读者会问，为什么要在上面加那么多括号呢？直接 `#define ADD(a, b) a + b` 不可以吗？或者，为保险起见，是否能写成 `#define ADD(a, b) (a + b)`？实际上必须加那么多括号，这是因为宏定义是直接将对应的部分替换，然后才进行编译和运行。因此像下面这种程序，就会出问题：

```
#include <stdio.h>
#define CAL(x) (x * 2 + 1)
int main(){
    int a = 1;
    printf("%d\n", CAL(a + 1));
    return 0;
}
```

输出结果：

4

这个结果跟一些读者预想的结果可能不太一致，读者可能觉得应该是 5 才对。实际上这

就是宏定义的陷阱，它把替换的部分直接原封不动替换进去，导致 $CAL(a+1)$ 实际上是 $(a+1 \times 2+1)$ ，也就是 $1+2+1=4$ ，而不是 $((a+1) \times 2+1)$ 。

总之，尽量不要使用宏定义来做除了定义常量以外的事情，除非给能加的地方都加上括号。

2.1.5 运算符

运算符就是用来计算的符号。常用的运算符有算术运算符、关系运算符、逻辑运算符、条件运算符、位运算符等。下面对每种运算符逐一进行解释。

1. 算术运算符

算术运算符有很多，比较常用的是下面几个：

- ① + 加法运算符：将前后两个数相加。
- ② - 减法运算符：将前后两个数相减。
- ③ * 乘法运算符：将前后两个数相乘。
- ④ / 除法运算符：取前面的数除以后面的数得到的商。
- ⑤ % 取模运算符：取前面的数除以后面的数得到的余数。
- ⑥ ++ 自增运算符：令一个整型变量增加 1。
- ⑦ -- 自减运算符：令一个整型变量减少 1。

这些运算符都有一些细节可说，不妨都来看看。

首先，①②③这 3 个运算符没有特别需要注意的问题，可以直接用，举例如下：

```
#include <stdio.h>

int main(){
    int a = 3, b = 4;
    double c = 1.23, d = 0.24;
    printf("%d %d\n", a + b, a - b);
    printf("%f\n", c * d);
    return 0;
}
```

输出结果：

```
7 -1
0.295200
```

对于除法运算符，需要注意的是，当被除数跟除数都是整型时，并不会得到一个 `double` 浮点型的数，而是直接舍去小数部分（即向下取整）。举例如下：

```
#include <stdio.h>

int main(){
    int a = 5, b = 4, c = 5, d = 6;
    printf("%d %d %d\n", a / b, a / c, a / d);
    return 0;
}
```

输出结果：

```
1 1 0
```

可以看到, 5/4 直接舍掉了小数部分变成了 1, 而 5/6 则直接变成了 0。

另外, 除数如果是 0, 会导致程序异常退出或是得到错误输出 “1.#INF00”, 因此在出现问题时请检查是否在某种情况下除数为零。

加减乘除四种运算符的优先级顺序和四则运算的优先级相同。

取模运算符返回被除数与除数相除得到的余数, 举例如下:

```
#include <stdio.h>

int main(){
    int a = 5, b = 3, c = 5;
    printf("%d %d\n", a % b, a % c);
    return 0;
}
```

输出结果:

```
2 0
```

与除法运算符一样, 除数不允许为 0, 因为当出问题的时候应先考虑除数是否有可能为零。取模运算符的优先级和除法运算符相同。

再来讨论**自增运算符**。自增运算符有两种写法: `i++`或`++i`。这两个都可以实现把 `i` 增加 1 的功能, 但是也有不同的地方。它们的区别在于 `i++`是先使用 `i` 再将 `i` 加 1, 而`++i`则是先将 `i` 加 1 再使用 `i`。

看起来是不是很绕? 来看下面的例子:

```
#include <stdio.h>

int main(){
    int a = 1, b = 1, n1, n2;
    n1 = a++;
    n2 = ++b;
    printf("%d %d\n", n1, a);
    printf("%d %d\n", n2, b);
    return 0;
}
```

输出结果:

```
1 2
```

```
2 2
```

首先看 `n1 = a++`: 这里 `n1` 先获得 `a` 的值, 再将 `a` 加 1, 因此 `n1` 和 `a` 分别为 1 和 2; 接着是 `n2 = ++b`: 先将 `b` 加 1, `n2` 再获得 `b` 的值, 因此 `n2` 和 `b` 分别为 2 和 2。

自减运算符和自增运算符一样, 也有 `i--`和`--i`这两种写法, 作用是将 `i` 减 1, 细节上和自增运算符相同。

2. 关系运算符

常用的关系运算符共有六种: `<`、`>`、`<=`、`>=`、`==`、`!=`, 它们所实现的功能及语法见表 2-2。

3. 逻辑运算符

常用的逻辑运算符有三种: `&&`、`||`、`!`, 分别对应 “与” “或” “非”, 它们所实现的功能

及语法见表 2-3。

表 2-2 六种关系运算符

运算符	含 义	语 法	返回值
<	小于	a < b	表达式成立时返回真(1, true), 不成立时返回假(0, false)
>	大于	a > b	
<=	小于等于	a <= b	
>=	大于等于	a >= b	
==	等于	a == b	
!=	不等于	a != b	

表 2-3 三种逻辑运算符

运算符	含 义	语 法	返回值
&&	与	a && b	ab 都真，则返回真 其他情况均返回假
	或	a b	ab 都假，则返回假 其他情况均返回真
!	非	!a	如果 a 为真，则返回假 如果 a 为假，则返回真

4. 条件运算符

条件运算符（ ? : ）是 C 语言中唯一的三目运算符，即需要三个参数的运算符，其格式如下：

A ? B : C;

其含义是：如果 A 为真，那么执行并返回 B 的结果；如果 A 为假，那么执行并返回 C 的结果。举一个例子来说明：

```
#include <stdio.h>
int main(){
    int a = 3, b = 5;
    int c = a > b ? 7 : 11;
    printf("%d\n", c);
    return 0;
}
```

输出结果：

11

在上述代码中，由于 a>b 不成立（3<5），因此返回冒号后面的 11，并将 11 赋值给 c。再举一个例子：

```
#include <stdio.h>
#define MAX(a, b) ((a) > (b) ? (a) : (b))
int main(){
    int a = 4, b = 3;
```



```
printf("%d\n", MAX(a, b));
return 0;
}
```

输出结果:

4

上述代码使用宏定义来定义了 MAX(a, b)结构, ((a) > (b) ? (a) : (b))的意思是当 a > b 时返回 a, 否则返回 b。这就实现了从两个数中取较大值的功能。

5. 位运算符

位运算符有六种, 见表 2-4。不过相对上面的几种运算符来说, 位运算符使用得较少, 读者可能常用的是左移运算符。由于 int 型的上限为 $2^{31} - 1$, 因此有时程序中无穷大的数 INF 可以设置成 $(1 \ll 31) - 1$ (注意: 必须加括号, 因为位运算符的优先级没有算术运算符高)。但是一般更常用的是 $2^{30} - 1$, 因为它可以避免相加超过 int 的情况。注意: 如果把 $2^{30} - 1$ 写成二进制的形式就是 0x3fffffff, 因此下面两个式子是等价的。

```
const int INF = (1 << 30) - 1;
const int INF = 0x3fffffff;
```

表 2-4 六种位运算符

运算符	含 义	语 法	效 果
<<	左移	a << x	整数 a 按二进制位左移 x 位
>>	右移	a >> x	整数 a 按二进制位右移 x 位
&	位与	a & b	整数 a 和 b 按二进制对齐, 按位进行与运算 (除了 11 得 1, 其他均为 0)
	位或	a b	整数 a 和 b 按二进制对齐, 按位进行或运算 (除了 00 得 0, 其他均为 1)
^	位异或	a ^ b	整数 a 和 b 按二进制对齐, 按位进行异或运算 (相同为 0, 不同为 1)
~	位取反	~ a	整数 a 的二进制的每一位进行 0 变 1、1 变 0 的操作

练习

Codeup Contest ID: 100000565
地址: <http://codeup.cn/contest.php?cid=100000565>。



本节二维码

2.2 顺序结构

2.2.1 赋值表达式

在 C 语言中可以使用等号 “=” 来实现赋值操作:

```
int n = 5;
```

```
n = 6;
```

上面的第一个语句在定义变量时将 5 赋值给 int 型变量 n，然后在第二个语句中又把 6 赋值给了 n。而如果要给多个变量赋同一个值，可以使用连续等号的方法：

```
int n, m;
n = m = 5;
```

另外，等号右边也可以是一个表达式，例如：

```
#include <stdio.h>
int main(){
    int n = 3 * 2 + 1;
    int m = (n > 6) && (n < 8);
    n = n + 2;
    printf("%d %d\n", n, m);
    return 0;
}
```

输出结果：

```
9 1
```

上面的第一个语句将一个四则运算表达式的结果 7 赋值给了 n，而第二个语句判断 $n > 6$ 和 $n < 8$ 同时成立，因此将返回值 1 赋值给了 m。接着 $n = n + 2$ 将 $n + 2$ 赋值给 n，使得 n 变成 9。

最后，赋值运算符可以通过将其他运算符放在前面来实现赋值操作的简化。例如， $n += 2$ 的意思即为 $n = n + 2$ ，而 $n \times = 3$ 的意思即为 $n = n \times 3$ 。下面再举个例子：

```
#include <stdio.h>
int main(){
    int n = 12, m = 3;
    n /= m + 1;
    m %= 2;
    printf("%d %d\n", n, m);
    return 0;
}
```

输出结果：

```
3 1
```

上面的代码中， $n /= m + 1$ 等价于 $n = n / (m + 1)$ ，因此结果是 3；而 $m \% = 2$ 等价于 $m = m \% 2$ ，因此结果是 1。当然，初学者应当尽量写成 $n /= (m + 1)$ 的形式，这样可以避免因为基础不好而产生一些错误。

这种复合赋值运算符在程序中会被经常使用，并且可以加快编译速度、提高代码可读性，因此初学者即便没办法马上接受，也应尽量去学着写。

2.2.2 使用 scanf 和 printf 输入/输出

C 语言的 stdio.h 库函数中提供了 scanf 函数和 printf 函数，分别对应输入和输出。学会这两个函数是 C 语言学习中必不可少的。

1. scanf 函数的使用

scanf 是输入函数，其格式如下：

```
scanf("格式控制", 变量地址);
```

这看起来似乎有些抽象，不过其实很好理解。举个例子：

```
scanf("%d", &n);
```

其中，双引号里面是一个%d，表示通过这个 scanf 用户需要输入一个 int 型的变量。那这个变量输入后存在哪里呢？就是后面给出的 n。也就是说，通过这个 scanf，把输入的一个整数存放在 int 型变量 n 中。

接下来解释&n 前面的&。在 C 语言中，变量在定义之后，就会在计算机内存中分配一块空间给这个变量，该空间在内存中的地址称为变量的地址。为了得到变量的地址，需要在变量前加一个&（称为取地址运算符），也就是“&变量名”的写法。

既然%d 是 int 型变量的格式符，那么其他类型的变量自然也有对应的格式符。表 2-5 列出了常见数据类型的 scanf 格式符。

表 2-5 常见数据类型变量的 scanf 格式符

数据类型	格式符	举 例
int	%d	scanf("%d", &n);
long long	%lld	scanf("%lld", &n);
float	%f	scanf("%f", &f);
double	%lf	scanf("%lf", &db);
char	%c	scanf("%c", &c);
字符串（char 数组）	%s	scanf("%s", str);

应该会有读者注意到，表 2-5 对字符数组的举例中，数组名 str 前面并没有&取地址运算符。这是因为数组比较特殊，数组名称本身就代表了这个数组第一个元素的地址，所以不需要再加取地址运算符。也许读者现在对数组还没有较清晰的概念，后面介绍到数组的时候再次提到这一点，现在只需要记住，在 scanf 中，除了 char 数组整个输入的情况不加&之外，其他变量类型都需要加&。

那么，如果有类似 13:45:20 这种 hh:mm:ss 的时间需要输入，应该怎么做？事实上，可以使用下面代码的方法：

```
int hh, mm, ss;
scanf("%d:%d:%d", &hh, &mm, &ss);
```

可以看到，双引号内使用%d:%d:%d 的写法跟输入格式 hh:mm:ss 是一样的，只是把 hh、mm、ss 的部分换成了%d，以告诉计算机此处输入的是 int 型。这给读者一个启示：scanf 的双引号内的内容其实就是整个输入，只不过把数据换成它们对应的格式符并把变量的地址按次序写在后面而已。因此，如果要输入 12, 18.23, t 这种格式的数据，那么就把 12 替换成%d、18.23 替换成%lf、t 替换成%c 即可。

```
int a;
double b;
char c;
scanf("%d,%lf,%c", &a, &b, &c);
```

另外，如果要输入“3 4”这种用空格隔开的两个数字，两个%d之间可以不加空格：

```
int a, b;
scanf("%d%d", &a, &b);
```

可以不加空格的原因是，除了%c外，scanf对其他格式符（如%d）的输入是以空白符（即空格、Tab）为结束判断标志的，因此除非使用%c把空格按字符读入，其他情况都会自动跳过空格。另外，字符数组使用%s读入的时候以空格跟换行为读入结束的标志，如下面的代码所示：

```
#include <stdio.h>
int main(){
    char str[10];
    scanf("%s", str);
    printf("%s",str);
    return 0;
}
```

输入数据：

abcd efg

输出结果：

abcd

再次强调，scanf的%c格式是可以读入空格跟换行的，因此下面的例子中字符c是一个空格，请读者认真研究这个例子：

```
#include <stdio.h>
int main() {
    int a;
    char c, str[10];
    scanf("%d%c%s", &a, &c, str);
    printf("a=%d,c=%c,str=%s", a, c, str);
    return 0;
}
```

输入数据：

1 a bcd

输出结果：

a=1,c= ,str=a

特别提醒：初学者特别容易在写scanf时漏写&，因此如果在输入数据后程序异常退出，要马上考虑是否是在scanf中漏写了&。

2. printf函数的使用

在C语言中，printf函数用来输出。与scanf函数类似，printf函数的格式如下：

```
printf("格式控制", 变量名称);
```

由此可见，printf的双引号中的部分和scanf的用法是相同的，但是后面并不像scanf那样需要给出变量地址，而是直接跟上变量名称就行了，例如下面的例子：

```
int n = 5;
```



```
printf("%d", n);
```

如果上面的代码中使用 `scanf` 来输入 `n`，那么就需要在 `scanf` 中使用 `&n`，但是 `printf` 只需要填写 `n` 就可以了。

和 `scanf` 一样，表 2-6 中列出了各种常见数据类型对应的 `printf` 格式符，其中只有一个和 `scanf` 不同。

表 2-6 常见数据类型的 printf 格式符

数据类型	格式符	举 例
int	%d	printf("%d", n);
long long	%lld	printf("%lld", n);
float	%f	printf("%f", f1);
double	%f	printf("%f", db);
char	%c	printf("%c", c);
字符串(char 数组)	%s	printf("%s", str);

由表 2-6 可见，对于 `double` 类型的变量，其输出格式变成了 `%f`，而在 `scanf` 中却是 `%lf`。在有些系统中如果把输出格式写成 `%lf` 倒也不会出错，不过尽量还是按标准来。另外，不要因为 `float` 的 `scanf` 和 `printf` 的格式符都是 `%f` 比较好记而偷懒用 `float`，因为 `float` 的精度较低，如下面的代码所示：

```
#include <stdio.h>

int main(){
    float f1 = 8765.4, f2 = 8765.4;
    double d1 = 8765.4, d2 = 8765.4;
    printf("%f\n%f\n", f1 * f2, d1 * d2);
    return 0;
}
```

输出结果：

```
76832244.007969
76832237.160000
```

可以发现，两个 `float` 类型的浮点数相乘，精度在整数部分就已经不准确了，完全不能满足要求。所以，建议用 `double` 型。

在 `printf` 中也可以使用转义字符（其实 `scanf` 里也可以，只是一般用不到），因此如果想在必要的地方换行，可以加上 “`\n`”：

```
#include <stdio.h>

int main(){
    printf("abcd\nefg\n\nhijklmn");
    return 0;
}
```

输出结果：

```
abcd
efg
```

hijklmn

另外，如果想要输出"%\”，则需要前面再加一个%或\，例如下面的代码：

```
printf("%%");
printf("%\\");
```

最后介绍三种实用的输出格式，另外有一些格式在平时并不常用，此处不再赘述。

(1) %md

%md 可以使不足 m 位的 int 型变量以 m 位进行右对齐输出，其中高位用空格补齐；如果变量本身超过 m 位，则保持原样。

来看一个实例：

```
#include <stdio.h>

int main(){
    int a = 123, b = 1234567;
    printf("%5d\n", a);
    printf("%5d\n", b);
    return 0;
}
```

输出结果：

```
123
1234567
```

可以看见，123 有三位数字，不足五位，因此前面自动用两个空格填充，使整个输出凑足五位；而 1234567 已经大于五位，因此仍然直接输出。

(2) %0md

%0md 只是在 %md 中间多加了 0。和 %md 的唯一不同点在于，当变量不足 m 位时，将在前面补足够数量的 0 而不是空格。

下面是一个例子：

```
#include <stdio.h>

int main(){
    int a = 123, b = 1234567;
    printf("%05d\n", a);
    printf("%05d\n", b);
    return 0;
}
```

输出结果：

```
00123
1234567
```

这里 123 的前面并不是用空格补齐，而是使用 0 补齐。这个格式在某些题中非常适用。

(3) %.mf

%.mf 可以让浮点数保留 m 位小数输出，这个“保留”使用的是精度的“四舍六入五成双”规则（具体细节不必掌握）。很多题目都会要求浮点数的输出保留××位小数（或是精确

到小数点后××位)，就是用这个格式来进行输出（如果是四舍五入，那么需要用到后面会介绍的 **round** 函数）。示例如下：

```
#include <stdio.h>

int main(){
    double d1 = 12.3456;
    printf("%.0f\n", d1);
    printf("%.1f\n", d1);
    printf("%.2f\n", d1);
    printf("%.3f\n", d1);
    printf("%.4f\n", d1);
    return 0;
}
```

输出结果：

```
12
12.3
12.35
12.346
12.3456
```

2.2.3 使用 **getchar** 和 **putchar** 输入/输出字符

getchar 用来输入单个字符，**putchar** 用来输出单个字符，在某些 **scanf** 函数使用不便的场合可以使用 **getchar** 来输入字符。

来看下面的例子：

```
#include <stdio.h>

int main(){
    char c1, c2, c3;
    c1 = getchar();
    getchar();
    c2 = getchar();
    c3 = getchar();
    putchar(c1);
    putchar(c2);
    putchar(c3);
    return 0;
}
```

输入数据：

```
abcd
```

输出结果：

```
acd
```

此处第一个字符'a'被 **c1** 接收；第二个字符'b'虽然被接收，但是没有将它存储在某个变量

中；第三个字符'c'被 c2 接收；第四个字符'd'被 c3 接收。之后，连续三次 putchar 将把 c1、c2、c3 连续输出。而如果输入"ab"，然后按<Enter>键，再输入'c'，再按<Enter>键，输出结果会是这样：

```
a
c
```

这是因为 getchar 可以识别换行符，所以 c2 实际上储存的是换行符\n，因此在 a 和 c 之间会有一个换行出现。

2.2.4 注 释

注释是 C/C++中常用到的，用来在需要进行注解的语句旁边对语句进行解释。在程序编译的时候会自动跳过该部分，不执行这些被注释的内容。C/C++的注释有两种：

(1) 使用 “/**/” 注释

/**/对 “/*” 跟 “*/” 之间的内容进行注释，且可以注释若干连续行的内容，示例如下：

```
#include <stdio.h>
int main(){
    int a, b;
    scanf("%d%d", &a, &b);
    /*a++;
    b++;
    a = a * 2; */
    printf("%d %d\n", a, b);
    return 0;
}
```

这样在 “/*” 跟 “*/” 之间的内容就都不会被执行了。

(2) 使用 “//” 注释

“//” 可用以注释一行中在该符号之后的所有内容，效果仅限于该行，示例如下：

```
#include <stdio.h>
int main(){
    int a, b;
    scanf("%d%d", &a, &b);
    a++; //将 a 自增
    b++; //将 b 自增
    //a = a * 2;
    printf("%d %d\n", a, b);
    return 0;
}
```

在上面的代码中，“将 a 自增”“将 b 自增”“a = a * 2” 都被注释了。

2.2.5 typedef

typedef 是一个很有用的东西，它能给复杂的数据类型起一个别名，这样在使用中就可以

用别名来代替原来的写法。例如，当数据类型是 `long long` 时，就可以像下面的例子这样用 `LL` 来代替 `long long`，避免因在程序中出现大量的 `long long` 而降低编码效率。

```
#include <stdio.h>

typedef long long LL;    //给 long long 起个别名 LL

int main() {
    LL a = 123456789012345, b = 234567890123456;    //直接使用 LL
    printf("%lld\n", a + b);
    return 0;
}
```

输出结果：

```
358024679135801
```

2.2.6 常用 math 函数

C 语言提供了很多实用的数学函数，如果要使用，需要在程序开头加上 `math.h` 头文件。下面是几个比较常用的数学函数，需要读者掌握一下。

1. fabs(double x)

该函数用于对 `double` 型变量取绝对值，示例如下：

```
#include <stdio.h>
#include <math.h>

int main(){
    double db = -12.56;
    printf("%.2f\n", fabs(db));
    return 0;
}
```

输出结果：

```
12.56
```

2. floor(double x)和 ceil(double x)

这两个函数分别用于 `double` 型变量的向下取整和向上取整，返回类型为 `double` 型，示例如下：

```
#include <stdio.h>
#include <math.h>

int main(){
    double db1 = -5.2, db2 = 5.2;
    printf("%.0f %.0f\n", floor(db1), ceil(db1));
    printf("%.0f %.0f\n", floor(db2), ceil(db2));
    return 0;
}
```

输出结果：

```
-6 -5
```

```
5 6
```

3. pow(double r, double p)

该函数用于返回 r^p ，要求 r 和 p 都是 `double` 型，示例如下：

```
#include <stdio.h>
#include <math.h>

int main(){
    double db = pow(2.0, 3.0);
    printf("%f\n", db);
    return 0;
}
```

输出结果：

8.000000

4. sqrt(double x)

该函数用于返回 `double` 型变量的算术平方根，示例如下：

```
#include <stdio.h>
#include <math.h>

int main(){
    double db = sqrt(2.0);
    printf("%f\n", db);
    return 0;
}
```

输出结果：

1.414214

5. log(double x)

该函数用于返回 `double` 型变量的以自然对数为底的对数，示例如下：

```
#include <stdio.h>
#include <math.h>

int main(){
    double db = log(1.0);
    printf("%f\n", db);
    return 0;
}
```

输出结果：

0.000000

顺带一提，C 语言中没有对任意底数求对数的函数，因此必须使用换底公式来将不是以自然对数为底的对数转换为以 e 为底的对数，即 $\log_a b = \log_e b / \log_e a$ 。

6. sin(double x)、cos(double x)和 tan(double x)

这三个函数分别返回 `double` 型变量的正弦值、余弦值和正切值，参数要求是弧度制，示例如下：

```
#include <stdio.h>
#include <math.h>
```

```
const double pi = acos(-1.0);

int main(){
    double db1 = sin(pi * 45 / 180);
    double db2 = cos(pi * 45 / 180);
    double db3 = tan(pi * 45 / 180);
    printf("%f, %f, %f\n", db1, db2, db3);
    return 0;
}
```

输出结果:

0.707107, 0.707107, 1.000000

此处把 π 定义为精确值 $\text{acos}(-1.0)$ (因为 $\cos(\pi) = -1$)。

7. asin(double x)、acos(double x)和 atan(double x)

这三个函数分别返回 double 型变量的反正弦值、反余弦值和反正切值, 示例如下:

```
#include <stdio.h>
#include <math.h>

int main(){
    double db1 = asin(1);
    double db2 = acos(-1.0);
    double db3 = atan(0);
    printf("%f, %f, %f\n", db1, db2, db3);
    return 0;
}
```

输出结果:

1.570796, 3.141593, 0.000000

8. round(double x)

该函数用于将 double 型变量 x 四舍五入, 返回类型也是 double 型, 需进行取整, 示例如下:

```
#include <stdio.h>
#include <math.h>

int main(){
    double db1 = round(3.40);
    double db2 = round(3.45);
    double db3 = round(3.50);
    double db4 = round(3.55);
    double db5 = round(3.60);
    printf("%d, %d, %d, %d, %d\n", (int)db1, (int)db2, (int)db3, (int)db4,
(int)db5);
    return 0;
}
```

输出结果:

3, 3, 4, 4, 4

练习

Codeup Contest ID: 100000566

地址: <http://codeup.cn/contest.php?cid=100000566>。

本节二维码

2.3 选择结构

2.3.1 if 语句

在编程时,经常会碰到需要根据某个条件是否为真来决定执行哪个语句的情况,这时就需要用到 if 语句。if 语句的格式如下:

```
if(条件 A) {  
    ...  
}
```

也就是说,当条件 A 为真时,执行省略号的内容。示例如下:

```
#include <stdio.h>  
int main() {  
    int n = 5;  
    if(n > 3) {  
        n = 9;  
        printf("%d\n", n);  
    }  
    return 0;  
}
```

输出结果:

9

上面的实例用以判断 $n > 3$ 是否为真,如果为真,就令 n 为 9,并输出 n 。

if 语句当条件满足时会执行其中的内容,但如果当条件不满足时也有语句需要执行,则应当使用 else,即如下格式:

```
if(条件 A) {  
    ...  
} else {  
    ...  
}
```

这样当条件 A 成立时就会执行第一个省略号中的内容，当条件 A 不成立时则执行第二个省略号中的内容。示例如下：

```
#include <stdio.h>

int main() {
    int n = 2;
    if(n > 3) {
        n = 9;
        printf("%d\n", n);
    } else {
        printf("%d\n", n);
    }
    return 0;
}
```

输出结果：

2

如果省略号中的内容只有一个语句，那么可以去掉大括号，使外观简洁一些。不过这样做有可能会使某些复杂情况的实际逻辑跟自己的想法出现偏差。所以，一般只有在明确不会出错的情况下才可以将大括号去掉。

另外，如果需要在 else 的分支下再根据某个条件来选择不同的语句，那么可以使用 else if 的写法，即

```
if(条件 A) {
    ...
} else if(条件 B) {
    ...
} else {
    ...
}
```

这样就会先判断条件 A 是否成立，如果不成立，则判断条件 B 是否成立，如果还不成立，才会执行最后一个省略号的内容，示例如下：

```
#include <stdio.h>

int main() {
    int n = 2;
    if(n > 3) {
        n = 9;
        printf("%d\n", n);
    } else if(n > 2) {
        printf("%d\n", n + 1);
    } else {
        printf("%d\n", n);
    }
}
```

```
    return 0;
}
```

输出结果:

2

最后学习一个技巧。在 if 条件中, 如果表达式是 “!= 0” 或 “== 0”, 那么可以采用比较简单的写法:

(1) 如果表达式是 “!= 0”, 则可以省略 “!= 0”。示例如下:

```
#include <stdio.h>

int main() {
    int n = 0, m = 5;
    if(n) {
        printf("n is not zero!\n");
    } else {
        printf("n is zero!\n");
    }
    if(m) {
        printf("m is not zero!\n");
    } else {
        printf("m is zero!\n");
    }
    return 0;
}
```

输出结果:

```
n is zero!
m is not zero!
```

在上述代码中, if(n)的写法其实就是 if(n != 0), 这里由于 if 条件语句接收的是括号中表达式的“真”或“假”, 也即 1 或 0, 而 n 本身作为一个整数, 当 n 为 0 时, 则相当于为“假”, 当 n 不为 0 时, 则相当于为“真”, 因此直接在 if 中填写这种表达式就可以直接作为真假判断 (例如填写 n+m 也是可以的, 这时会判断 n+m 是否为 0)。

(2) 如果表达式为 “== 0”, 则可以省略 “== 0”, 并在表达式前添加非运算符 “!”。示例如下:

```
#include <stdio.h>

int main() {
    int n = 0, m = 5;
    if(!n) {
        printf("n is zero!\n");
    } else {
        printf("n is not zero!\n");
    }
    if(!m) {
```



```

    printf("m is zero!\n");
} else {
    printf("m is not zero!\n");
}
return 0;
}

```

输出结果:

```

n iszero!
m is not zero!

```

上面 `if(!n)` 的写法就等价于 “`if(n == 0)`”。前面介绍过，非运算符的作用是将后面的表达式值真假颠倒。由于 `if(n)` 表示 `if(n != 0)`，因此 `if(!n)` 就表示 `if(n == 0)`。

初学者可能对这两个小技巧会不太适应，但其确实可以简化写法。希望读者在读到相应的程序时能够明白这种写法的意思。

2.3.2 if 语句的嵌套

if 语句的嵌套是指在 if 或者 else 的执行内容中使用 if 语句，其格式如下：

```

if(条件 A) {
    ...
    if(条件 B) {
        ...
    }else{
        ...
    }
    ...
}else{
    ...
}

```

按照上述代码，当条件 A 成立时，会执行其大括号内的语句，执行期间碰到另一个 if 语句，当条件 B 成立或非成立时执行不同的语句。示例如下：

```

#include <stdio.h>
int main(){
    int n = 3, m = 5;
    if(n < 5){
        if(m < 5){
            printf("%d\n", m + n);
        }else{
            printf("%d\n", m - n);
        }
    }else{
        printf("haha\n");
    }
}

```



```

    }
    return 0;
}

```

输出结果:

2

2.3.3 switch 语句

switch 语句在分支条件较多时会显得比较精练，但是在分支条件较少时用得并不多。其格式如下：

```

switch(表达式){
    case 常量表达式 1:
        ...
        break;
    case 常量表达式 2:
        ...
        break;
    case 常量表达式 n:
        ...
        break;
    default:
        ...
}

```

示例如下：

```

#include <stdio.h>
int main(){
    int a = 1, b = 2;
    switch(a + b){
        case 2:
            printf("%d\n", a);
            break;
        case 3:
            printf("%d\n", b);
            break;
        case 4:
            printf("%d\n", a + b);
            break;
        default:
            printf("sad story\n");
    }
    return 0;
}

```

}

输出结果:

2

在上面的示例中, 以 $a + b$ 作为需要判断的表达式: 当 $a + b$ 为 2、3、4 时各自有需要输出的东西, 而其他情况则输出 sad story。因为实际上 $a + b = 3$, 所以选择 case 3 这条分支, 输出了 b。另外, 可以注意到, 每个 case 下属的语句都没有使用大括号将它们括起来, 这是由于 case 本身默认把两个 case 之间的内容全部作为上一个 case 的内容, 因此不用加大括号。

还应该注意, 每个 case 的最后一个语句都是 break。这个 break 有什么作用呢? 不妨把所有 break 都删掉, 再输出结果看看:

```
#include <stdio.h>

int main(){
    int a = 1, b = 2;
    switch(a + b){
        case 2:
            printf("%d\n", a);
        case 3:
            printf("%d\n", b);
        case 4:
            printf("%d\n", a + b);
        default:
            printf("sad story\n");
    }
    return 0;
}
```

输出结果:

2

3

sad story

此时会发现, 删去 break 语句后, 程序把 case 3 以下的所有语句都输出了。由此可见, break 的作用在于可以结束当前 switch 语句, 如果将其删去, 则程序将会从第一个匹配的 case 开始执行语句, 直到其下面的所有语句都执行完毕才会退出 switch。

练习

Codeup Contest ID: 100000567

地址: <http://codeup.cn/contest.php?cid=100000567>。

本节二维码

2.4 循环结构

2.4.1 while 语句

现在有一个问题：如何用计算机求解 $1 + 2 + \dots + 100$ ？可能可以直接用公式算，但是如果一定要让计算机依次累加来计算，是不是得写一串很长的加法式子？自然不用。C 语言中提供了“循环”的实现方式，即只需要让一个变量从 1 循环自增直到 100，将中间的每个数字都累加起来，就可以得到正确结果，而 while 就是实现循环的三种方式之一。

while 的格式如下：

```
while(条件 A){
    ...
}
```

可以看到，while 的格式非常简洁，并且跟 if 语句十分相像——只要条件 A 成立，就反复执行省略号的内容。如果不加大括号，则 while 循环只作用于 while 后的第一个完整语块（例如分号）。

以本节开始的问题为例，可以先令 $n = 1$ ， $sum = 0$ ，然后以 $n \leq 100$ 作为循环条件，每次把 n 加到 sum 上，再使 n 自增：

```
#include <stdio.h>

int main(){
    int n = 1, sum = 0;
    while(n <= 100){
        sum = sum + n;
        n++;
    }
    printf("sum = %d\n", sum);
    return 0;
}
```

输出结果：

```
sum = 5050
```

另外，while 条件判断的是真假，因此在条件语句中的小技巧在此同样适用：

- ① 如果表达式是 “ $\neq 0$ ”，则可以省略 “ $\neq 0$ ”。
- ② 如果表达式为 “ $= 0$ ”，则可以省略 “ $= 0$ ”，并在表达式前添加非运算符 “!”。

示例如下：

```
#include <stdio.h>

int main() {
    int n = 12345, count = 0;
    while(n) { //相当于 while(n != 0)
        count = count + n % 10;
        n = n / 10;
    }
}
```



```

    }
    printf("%d\n", count);
    return 0;
}

```

输出结果:

15

上述程序实现了将 n 的每一位数字相加, 即 $1 + 2 + 3 + 4 + 5 = 15$ 。while 循环中每次通过 $n \% 10$ 获取当前 n 的最低位, 之后通过 $n = n / 10$ 将最低位抹去。while 循环直到 n 变为 0 时停止, 得到的 `count` 即为需要的结果。

2.4.2 do...while 语句

do...while 语句和 while 语句相似, 但是它们的格式是上下颠倒的:

```

do{
    ...
}while(条件A);

```

do...while 语句会先执行省略号中的内容一次, 然后才判断条件 A 是否成立。如果条件 A 成立, 继续反复执行省略号的内容, 直到某一次条件 A 不再成立, 则退出循环。

还是 $1 + 2 + \dots + 100$ 的求和问题, 写法如下 (注意: while 的末尾是有分号的):

```

#include <stdio.h>

int main(){
    int n = 1, sum = 0;
    do{
        sum = sum + n;
        n++;
    }while(n <= 100);
    printf("sum = %d\n", sum);
    return 0;
}

```

输出结果:

5050

这样看来, while 和 do...while 是不是等价的呢? 因为上面的例子看上去连循环条件都一样。其实不是的, do...while 语句和 while 的不同之处在于: do...while 会先执行循环体一次, 然后才去判断循环条件是否为真, 这就使得 do...while 语句的实用性远不如 while, 因为用户碰到的大部分情况都需要能处理在某些数据下不允许进入循环的情况。示例如下:

```

#include <stdio.h>

int main(){
    int n;
    scanf("%d", &n);
    do{
        printf("1");
    }
}

```

```

        n--;
    }while(n > 0);
    return 0;
}

```

在这个例子中，需要实现这样一个功能：对输入的非负整数 n ，输出 n 个 1。如果采用 `do...while` 语句的写法，当读入的 n 大于 0 时都可以很好地实现功能；但是当读入的 n 恰好为 0 时，理论上不应该输出，但是 `do...while` 会先执行一次循环体，然后才去判断，这就会输出一个 1，显然不符合题意。当然，用户可以修改判断条件或者对 $n = 0$ 进行特判，但是这对一些复杂的程序逻辑来说会增加思维难度。相比较来说，直接用 `while` 就可以更直接地完成功能：

```

#include <stdio.h>
int main(){
    int n;
    scanf("%d", &n);
    while(n > 0){
        printf("1");
        n--;
    }
    return 0;
}

```

2.4.3 for 语句

`for` 语句的使用频率是三种循环语句中最高的，其常见格式如下：

```

for(表达式 A; 表达式 B; 表达式 C){
    ...
}

```

初学者不必对 `for` 语句中的三个表达式心生畏惧，其实这样写反而“简洁”，后文会对此进行说明。先来解释这个格式的意思：

- ① 在 `for` 循环开始前，首先执行表达式 A。
- ② 判断表达式 B 是否成立：若成立，执行省略号内容；否则，退出循环。
- ③ 在省略号内容执行完毕后，执行表达式 C，之后回到②。

为了理解上面的格式，下面举一个较为常用的特例：

```

for(循环变量赋初值; 循环条件; 循环变量改变){
    ...
}

```

这个 `for` 循环的逻辑是：先给要循环的变量赋初值，然后反复判断循环条件是否成立；如果不成立，则退出循环；如果成立，则执行省略号部分的内容，执行完毕后改变循环变量的值（如加 1），并重新判断循环变量是否成立，如此反复，示例如下：

```

#include <stdio.h>
int main(){

```



```
int i, sum = 0;
for(i = 1; i <= 100; i++){
    sum = sum + i;
}
printf("sum = %d\n", sum);
return 0;
}
```

输出结果:

```
sum = 5050
```

不妨把 for 循环提出来查看:

```
for(i = 1; i <= 100; i++){
    sum = sum + i;
}
```

这个 for 循环的逻辑是这样的:

① 令 $i = 1$ 。

② 判断 $i \leq 100$ 是否成立: 若成立, 则令 $\text{sum} = \text{sum} + 1$, 并在之后执行 $i++$ 使 i 变为 2。

③ 判断 $i \leq 100$ 是否成立: 若成立, 则令 $\text{sum} = \text{sum} + 2$, 并在之后执行 $i++$ 使 i 变为 3。

...

④ 当 $i = 100$ 时, 判断 $i \leq 100$ 是否成立: 若成立, 则令 $\text{sum} = \text{sum} + 100$, 并在之后执行 $i++$ 使 i 变为 101。

⑤ 判断 $i \leq 100$ 是否成立: 若不成立, 则退出循环。

于是就有 $\text{sum} = 1 + 2 + \dots + 100 = 5050$ 。

初学者可能会认为 for 语句的写法比较复杂, 没有 while 好写, 其实不然, 因为这三个表达式其实在 while 语句中全都出现了:

```
int i = 1, sum = 0;
while(i <= 100){
    sum = sum + i;
    i++;
}
```

由此发现, 一开始定义变量的 $i = 1$ 就是在给循环变量赋初值, 而 $i \leq 100$ 则是循环条件, 在循环体执行完毕后的 $i++$ 就是在给循环变量自增以进行下一次循环, 所以 for 语句只是把这三个表达式都放在同一行了, 这样反而可以使逻辑更加清晰, 也更方便检查。读者一定要学会写 for 语句, 因为在大部分不太简单的题目里都需要用到。另外, for 语句下如果只有一个语块, 则可以不加大括号, 不过一般还是加上比较好, 可以省去很多潜在的错误。

特别提醒: 在 C 语言中不允许在 for 语句的表达式 1 里定义变量 (例如 `int i` 的写法是不允许的), 但是在 C++ 中可以, 因此下面这种写法需要把文件保存为 .cpp 文件才能通过编译:

```
for(int i = 1; i <= 100; i++){
    sum = sum + i;
}
```

显然, 随时定义临时变量才更符合用户的习惯, 因此要习惯把文件保存为 .cpp 文件而不

是.c文件，并在提交程序时选择C++语言提交。由于C++是向下兼容C的，C的程序可以在C++中运行，但是C++中的一些特性不允许在C语言中运行。总而言之，在训练中，请尽量将文件的扩展名保存为.cpp。

2.4.4 break 和 continue 语句

break 在前面讲解 switch 的时候已经提到过：它可以强制退出 switch 语句。而事实上 break 同样适用于循环，即在需要的场合下直接退出循环（前面介绍的三种循环语句都可以）。示例如下：

```
#include <stdio.h>
int main(){
    int n, sum = 0;
    for(int i = 1; i <= 100; i++){
        sum = sum + i;
        if(sum >= 2000) break;
    }
    printf("sum = %d\n", sum);
    return 0;
}
```

输出结果：

```
sum = 2016
```

上面代码实现了在 $1 + 2 + 3 + \dots + 100$ 的过程中，输出总和第一次超过 2000 时的 sum 值，这就需要在循环体中加一条 if 条件语句来使 $\text{sum} \geq 2000$ 时退出 for 循环。

continue 的作用跟 break 有点相似，它可以在需要的地方临时结束循环的当前轮回，然后进入下一个轮回，示例如下：

```
#include <stdio.h>
int main(){
    int sum = 0;
    for(int i = 1; i <= 5; i++){
        if(i % 2 == 1) continue;
        sum = sum + i;
    }
    printf("sum = %d\n", sum);
    return 0;
}
```

输出结果：

```
sum = 6
```

在这段代码的 for 循环中，当满足 $i \% 2 == 1$ （即 i 为奇数）时执行 continue，即可将该句以下的部分直接切断不执行，执行 i++ 后进入下一层循环。为了使 continue 的过程更为清晰，下面对这段代码的执行过程进行罗列：

① $i = 1$: $i \% 2 == 1$ ，因此 continue 执行，于是后面的语句都不执行，i++ 后进入下层

循环。

② $i = 2$: $i \% 2 = 0$, 因此 `continue` 不执行, $\text{sum} = \text{sum} + i$ 得 $\text{sum} = 2$, $i++$ 后进入下层循环。

③ $i = 3$: $i \% 2 = 1$, 因此 `continue` 执行, 于是后面的语句都不执行, $i++$ 后进入下层循环。

④ $i = 4$: $i \% 2 = 0$, 因此 `continue` 不执行, $\text{sum} = \text{sum} + i$ 得 $\text{sum} = 6$, $i++$ 后进入下层循环。

⑤ $i = 5$: $i \% 2 = 1$, 因此 `continue` 执行, 于是后面的语句都不执行, $i++$ 后进入下层循环。

⑥ $i = 6$: 不满足 $i \leq 5$ 的条件, 退出 `for` 循环。

至此, `break` 跟 `continue` 的用法都已经介绍完毕。这两个语句在编程时会频繁用到, 请读者务必掌握它们的用法。

练习

Codeup Contest ID: 100000568

地址: <http://codeup.cn/contest.php?cid=100000568>。



本节二维码

2.5 数 组

2.5.1 一维数组

数组就是把相同数据类型的变量组合在一起而产生的数据集合。众所周知, 每个变量在内存中都有对应的存放地址, 而数组就是从某个地址开始连续若干个位置形成的元素集合。

一维数组的定义格式如下:

数据类型 数组名 [数组大小];

注意: 数组大小必须是整数常量, 不可以是变量。几种常见数据类型的一维数组定义举例:

```
int a[10];
double db[2333];
char str[100000];
bool HashTable[1000000];
```

这样就可以把 `int a[10]` 理解为定义了十个 `int` 型数据, 且以下面的格式访问:

数组名称 [下标]

还需要知道, 在定义了长度为 `size` 的一维数组后, 只能访问下标为 `0 ~ size - 1` 的元素。例如定义 `int a[10]` 之后, 允许正常访问的元素是 `a[0]`、`a[1]`、 \dots 、`a[9]` (见图 2-1), 而不允许访问 `a[10]`, 在初学时要特别注意这点。

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
------	------	------	------	------	------	------	------	------	------

图 2-1 数组元素下标范围

下面讲述一维数组的初始化。一维数组的初始化，需要给出用逗号隔开的从第一个元素开始的若干个元素的初值，并用大括号括住。后面未被赋初值的元素将会由不同编译器内部实现的不同而被赋以不同的初值（可能是很大的随机数），而一般情况默认初值为 0。

示例如下：

```
#include <stdio.h>

int main(){
    int a[10] = {5, 3, 2, 6, 8, 4};
    for(int i = 0; i < 10; i++){
        printf("a[%d] = %d\n", i, a[i]);
    }
    return 0;
}
```

输出结果：

```
a[0] = 5
a[1] = 3
a[2] = 2
a[3] = 6
a[4] = 8
a[5] = 4
a[6] = 0
a[7] = 0
a[8] = 0
a[9] = 0
```

上面的程序对数组 a 的前六个元素进行了赋初值，而后面没有赋值的部分默认赋为 0。但是如果数组一开始没有赋初值，数组中的每个元素都可能会是一个随机数，并不一定默认为 0。因此，如果想要给整个数组都赋初值 0，只需要把第一个元素赋为 0，或者只用一个大括号来表示。

```
int a[10] = {0};
int a[10] = {};
```

数组中每个元素都可以被赋值、被运算，可以被当作普通变量进行相同的操作。如果根据一些条件，可以不断让后一位的结果由前一位或前若干位计算得来，那么就把这种做法称为递推。递推可以分为顺推和逆推两种。下面的程序实现了输入 a[0]，并将数组中后续元素都赋值为其前一个元素的两倍的功能，属于顺推。

```
#include <stdio.h>

int main(){
    int a[10];
    scanf("%d", &a[0]);
```



```

for(int i = 1; i < 10; i++){
    a[i] = a[i - 1] * 2;
}
for(int i = 0; i < 10; i++){
    printf("a[%d] = %d\n", i, a[i]);
}
return 0;
}

```

当输入 1 时，输出结果如下：

```

a[0] = 1
a[1] = 2
a[2] = 4
a[3] = 8
a[4] = 16
a[5] = 32
a[6] = 64
a[7] = 128
a[8] = 256
a[9] = 512

```

2.5.2 冒泡排序

排序是指将一个无序序列按某个规则进行有序排列，而冒泡排序是排序算法中最基础的一种。现给出一个序列 a ，其中元素的个数为 n ，要求将它们按从小到大的顺序排序。

冒泡排序的本质在于交换，即每次通过交换的方式把当前剩余元素的最大值移动到一端，而当剩余元素减少为 0 时，排序结束。为了使排序过程更加清楚，举一个例子。

现在有一个数组 a ，其中有 5 个元素，分别为 $a[0]=3$ 、 $a[1]=4$ 、 $a[2]=1$ 、 $a[3]=5$ 、 $a[4]=2$ ，要求把它们按从小到大的顺序排列。下面的过程中，每趟将最大数交换到最右边：

(1) 第一趟。

$a[0]$ 与 $a[1]$ 比较（3 与 4 比较）， $a[1]$ 大，因此不动，此时序列为{3, 4, 1, 5, 2};

3	4	1	5	2
---	---	---	---	---

$a[1]$ 与 $a[2]$ 比较（4 与 1 比较）， $a[1]$ 大，因此把 $a[1]$ 和 $a[2]$ 交换，此时序列为{3, 1, 4, 5, 2};

3	1	4	5	2
---	---	---	---	---

$a[2]$ 与 $a[3]$ 比较（4 与 5 比较）， $a[3]$ 大，因此不动，此时序列为{3, 1, 4, 5, 2};

3	1	4	5	2
---	---	---	---	---

$a[3]$ 与 $a[4]$ 比较（5 与 2 比较）， $a[4]$ 大，因此把 $a[3]$ 和 $a[4]$ 交换，此时序列为{3, 1, 4, 2, 5};

3	1	4	2	5
---	---	---	---	---

由此，第一趟排序结束，共进行了四次比较。

(2) 第二趟。

$a[0]$ 与 $a[1]$ 比较（3 与 1 比较）， $a[0]$ 大，因此把 $a[0]$ 和 $a[1]$ 交换，此时序列为{1, 3, 4, 2, 5};

1	3	4	2	5
---	---	---	---	---

$a[1]$ 与 $a[2]$ 比较（3 与 4 比较）， $a[2]$ 大，因此不动，此时序列为{1, 3, 4, 2, 5};

1	3	4	2	5
---	---	---	---	---

$a[2]$ 与 $a[3]$ 比较（4 与 2 比较）， $a[2]$ 大，因此把 $a[2]$ 和 $a[3]$ 交换，此时序列为{1, 3, 2, 4, 5};

1	3	2	4	5
---	---	---	---	---

由此，第二趟排序结束，共进行了三次比较。

(3) 第三趟。

$a[0]$ 与 $a[1]$ 比较（1 与 3 比较）， $a[1]$ 大，因此不动，此时序列为{1, 3, 2, 4, 5};

1	3	2	4	5
---	---	---	---	---

$a[1]$ 与 $a[2]$ 比较（3 与 2 比较）， $a[1]$ 大，因此把 $a[1]$ 和 $a[2]$ 交换，此时序列为{1, 2, 3, 4, 5};

1	2	3	4	5
---	---	---	---	---

由此，第三趟排序结束，共进行了两次比较。

(4) 第四趟。

$a[0]$ 与 $a[1]$ 比较（1 与 2 比较）， $a[2]$ 大，因此不动，此时序列为{1, 2, 3, 4, 5};

1	2	3	4	5
---	---	---	---	---

由此，第四趟排序结束，共进行了一次比较。

至此，已经无法再继续比较，序列已经有序，冒泡排序结束。

下面来看看如何实现冒泡排序。先来学习如何交换两个数。一般来说，交换两个数需要借助中间变量，即先定义中间变量 `temp` 存放其中一个数 `a`，然后再把另一个数 `b` 赋值给已被转移数据的 `a`，最后再把存有 `a` 的中间变量 `temp` 赋值给 `b`，这样 `a` 和 `b` 就完成了交换。下面这段代码就实现了这个功能：

```
#include <stdio.h>

int main(){
    int a = 1, b = 2;
    int temp = a;
    a = b;
    b = temp;
    printf("a = %d, b = %d\n", a, b);
    return 0;
}
```

输出结果：

```
a = 2, b = 1
```

然后来实现冒泡排序。从上面的例子中可以发现，整个过程执行 $n-1$ 趟，每一趟从左到右依次比较相邻的两个数，如果大的数在左边，则交换这两个数，当该趟结束时，该趟最大数被移动到当前剩余数的最右边。具体实现如下：

```
#include <stdio.h>

int main(){
    int a[10] = {3, 1, 4, 5, 2};
    for(int i = 1; i <= 4; i++){ //进行 n - 1 趟
        //第 i 趟时从 a[0] 到 a[n - i - 1] 都与它们下一个数比较
        for(int j = 0; j < 5 - i; j++){
            if(a[j] > a[j + 1]){ //如果左边的数更大，则交换 a[j] 和 a[j + 1]
                int temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
            }
        }
    }

    for(int i = 0; i < 5; i++){
        printf("%d ", a[i]);
    }

    return 0;
}
```

输出结果：

```
1 2 3 4 5
```

第二个 for 循环的理解：从前面的例子可以看出，第一趟从 $a[0]$ 到 $a[3]$ 都需要与下一个数比较，第二趟从 $a[0]$ 到 $a[2]$ 都需要与下一个数比较，第三趟从 $a[0]$ 到 $a[1]$ 都需要与下一个数比较，第四趟只有 $a[0]$ 需要与下一个数比较。因此很容易找到规律，即当第 i 趟时，从 $a[0]$ 到 $a[n-i-1]$ 都需要与下一个数比较。

2.5.3 二维数组

二维数组其实就是一维数组的扩展，其格式如下：

数据类型 数组名[第一维大小][第二维大小];

下面是常见的数据类型的二维数组定义：

```
int a[5][6];
double db[10][10];
char [256][256];
bool vis[1000][1000];
```

二维数组中元素的访问和一维数组类似，只需要给出第一维和第二维的下标：

数组名[下标 1][下标 2]

需要注意的是，对定义为 `int a[size1][size2]` 的二维数组，其第一维的下标取值只能是 $0 \sim$

(size1 - 1)，其第二维的下标取值只能是 0 ~ (size2 - 1)。

怎么理解二维数组？其实可以把二维数组当作一维数组的每一个元素都是一个一维数组，例如可以将定义为 `int a[5][6]` 的二维数组看作五个长度为 6 的一维数组（见图 2-2）。

a[0]	a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]	a[0][5]
a[1]	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]	a[1][5]
a[2]	a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]	a[2][5]
a[3]	a[3][0]	a[3][1]	a[3][2]	a[3][3]	a[3][4]	a[3][5]
a[4]	a[4][0]	a[4][1]	a[4][2]	a[4][3]	a[4][4]	a[4][5]

图 2-2 数组 `a[5][6]`

和一维数组一样，二维数组也可以在定义时进行初始化。二维数组在初始化的时候，需要按第一维的顺序依次用大括号给出第二维初始化情况，然后将它们用逗号分隔，并用大括号全部括住，而在这些被赋初值的元素之外的部分将被默认赋值为 0。举一个例子会比较容易理解：

```
#include <stdio.h>
int main(){
    int a[5][6] = {{3, 1, 2}, {8, 4}, {}, {1, 2, 3, 4, 5}};
    for(int i = 0; i < 5; i++){
        for(int j = 0; j < 6; j++){
            printf("%d", a[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

输出结果：

```
3 1 2 0 0 0
8 4 0 0 0 0
0 0 0 0 0 0
1 2 3 4 5 0
0 0 0 0 0 0
```

可以看到，第 1、2、4 行都赋予了初值，第 3 行使用大括号 `{}` 跳过了（注意：如果不写大括号是无法通过编译的）。剩下的部分均被默认赋为 0。

下面这个程序用以将两个二维数组对应位置的元素相加，并将结果存放到另一个二维数组中：

```
#include <stdio.h>
int main(){
    int a[3][3], b[3][3];
    for(int i = 0; i < 3; i++){
        for(int j = 0; j < 3; j++){
            scanf("%d", &a[i][j]);    //输入二维数组 a 的元素
```

```

    }
}
for(int i = 0; i < 3; i++){
    for(int j = 0; j < 3; j++){
        scanf("%d", &b[i][j]);    //输入二维数组 b 的元素
    }
}
int c[3][3];
for(int i = 0; i < 3; i++){
    for(int j = 0; j < 3; j++){
        c[i][j] = a[i][j] + b[i][j];    //对应位置元素相加并放到二维数组 c 中
    }
}
for(int i = 0; i < 3; i++){
    for(int j = 0; j < 3; j++){
        printf("%d ", c[i][j]);    //输出二维数组 c 的元素
    }
    printf("\n");
}
return 0;
}

```

输入如下两个 3×3 的矩阵:

```

1 2 3
4 5 6
7 8 9
1 4 7
2 5 8
3 6 9

```

输出结果:

```

2 6 10
6 10 14
10 14 18

```

特别提醒: 如果数组大小较大 (大概 10^6 级别), 则需要将其定义在主函数外面, 否则会使程序异常退出, 原因是函数内部申请的局部变量来自系统栈, 允许申请的空间较小; 而函数外部申请的全局变量来自静态存储区, 允许申请的空间较大。例如下面的代码就把 10^6 大小的数组定义在了主函数外面。

```

#include <stdio.h>
int a[1000000];
int main(){
    for(int i = 0; i < 1000000; i++){

```



```

        a[i] = i;
    }
    return 0;
}

```

最后再提一下多维数组，即维度高于二维的数组。多维数组跟二维数组类似，只是把维度增加了若干维，其使用方法和二维数组基本无二。例如下面这段代码就定义了一个三维数组，并在输入的基础上使每个元素都增加了 1：

```

#include <stdio.h>

int main(){
    int a[3][3][3];
    for(int i = 0; i < 3; i++){
        for(int j = 0; j < 3; j++){
            for(int k = 0; k < 3; k++){
                scanf("%d", &a[i][j][k]);    //输入三维数组 a 的元素
                a[i][j][k]++;    //自增
            }
        }
    }

    for(int i = 0; i < 3; i++){
        for(int j = 0; j < 3; j++){
            for(int k = 0; k < 3; k++){
                printf("%d\n", a[i][j][k]);    //输出三维数组 a 的元素
            }
        }
    }

    return 0;
}

```

2.5.4 memset——对数组中每一个元素赋相同的值

如果需要对数组中每一个元素赋以相同的值，例如对数组初始化为 0 或是其他的一些数，就可能要使用相关的函数。一般来说，给数组中每一个元素赋相同的值有两种方法：**memset** 函数和 **fill** 函数，其中 **fill** 函数在第 6 章 STL 的 **algorithm** 头文件中介绍，这里先介绍 **memset** 函数。

memset 函数的格式为：

```
memset(数组名, 值, sizeof(数组名));
```

不过也要记住，使用 **memset** 需要在程序开头添加 **string.h** 头文件，且只建议初学者使用 **memset** 赋 0 或 -1。这是因为 **memset** 使用的是按字节赋值，即对每个字节赋同样的值，这样组成 **int** 型的 4 个字节就会被赋成相同的值。而由于 0 的二进制补码为全 0，-1 的二进制补码为全 1，不容易弄错。如果要对数组赋其他数字（例如 1），那么请使用 **fill** 函数（但 **memset** 的执行速度快）。示例如下：


```

#include <stdio.h>
#include <string.h>
int main(){
    int a[5] = {1, 2, 3, 4, 5};
    //赋初值 0
    memset(a, 0, sizeof(a));
    for(int i = 0; i < 5; i++){
        printf("%d ", a[i]);
    }
    printf("\n");
    //赋初值-1
    memset(a, -1, sizeof(a));
    for(int i = 0; i < 5; i++){
        printf("%d ", a[i]);
    }
    printf("\n");
    return 0;
}

```

输出结果:

0 0 0 0 0

-1 -1 -1 -1 -1

读者不妨把 `memset` 里面的 `-1` 改为 `1`, 看看结果会有什么不同。另外, 对二维数组或多维数组的赋值方法也是一样的 (仍然只需要写数组名), 不需要改变任何东西。

2.5.5 字符数组

1. 字符数组的初始化

和普通数组一样, 字符数组也可以初始化, 其方法也相同, 示例如下:

```

#include <stdio.h>
int main(){
    char str[15] = {'G', 'o', 'o', 'd', ' ', 's', 't', 'o', 'r', 'y', '!'};
    for(int i = 0; i < 11; i++){
        printf("%c", str[i]);
    }
    return 0;
}

```

输出结果:

Good story!

除此之外, 字符数组也可以通过直接赋值字符串来初始化 (仅限于初始化, 程序其他位置不允许这样直接赋值整个字符串), 示例如下:

```

#include <stdio.h>

```

```
int main(){
    char str[15] = "Good Story!";
    for(int i = 0; i < 11; i++){
        printf("%c", str[i]);
    }
    return 0;
}
```

输出结果:

Good story!

2. 字符数组的输入输出

字符数组就是 `char` 数组，当维度是一维时可以当作“字符串”。当维度是二维时可以当作字符串数组，即若干字符串。字符数组的输入除了使用 `scanf` 外，还可以用 `getchar` 或者 `gets`；其输出除了使用 `printf` 外，还可以用 `putchar` 或者 `puts`。下面对上述几种方式分别进行介绍：

(1) `scanf` 输入，`printf` 输出

`scanf` 对字符类型有 `%c` 和 `%s` 两种格式（`printf` 同理，下同），其中 `%c` 用来输入单个字符，`%s` 用来输入一个字符串并存在字符数组里。`%c` 格式能够识别空格跟换行并将其输入，而 `%s` 通过空格或换行来识别一个字符串的结束。示例如下：

```
#include <stdio.h>
int main(){
    char str[10];
    scanf("%s", str);
    printf("%s", str);
    return 0;
}
```

输入下面这个字符串：

TAT TAT TAT

输出结果：

TAT

可以看到，`%s` 识别空格作为字符串的结尾，因此后两个 TAT 不会被读入。另外，`scanf` 在使用 `%s` 时，后面对应数组名前面是不需要加 `&` 取地址运算符的。

(2) `getchar` 输入，`putchar` 输出

`getchar` 和 `putchar` 分别用来输入和输出单个字符，这点在之前已经说过，这里简单举一个二维字符数组的例子：

```
#include <stdio.h>
int main(){
    char str[5][5];
    for(int i = 0; i < 3; i++){
        for(int j = 0; j < 3; j++){
            str[i][j] = getchar();
        }
    }
}
```

```

    getchar();    //这句是为了把输入中每行末尾的换行符吸收掉
}
for(int i = 0; i < 3; i++){
    for(int j = 0; j < 3; j++){
        putchar(str[i][j]);
    }
    putchar('\n');
}
return 0;
}

```

输入下面的字符矩阵:

```

^ ^
- -
^ ^
- -
^ ^
- -

```

输出的结果和输入相同。

(3) gets 输入, puts 输出

gets 用来输入一行字符串 (注意: gets 识别换行符\n 作为输入结束, 因此 scanf 完一个整数后, 如果要使用 gets, 需要先用 getchar 接收整数后的换行符), 并将其存放于一维数组 (或二维数组的一维) 中; puts 用来输出一行字符串, 即将一维数组 (或二维数组的一维) 在界面上输出, 并紧跟一个换行。示例如下:

```

#include <stdio.h>
int main(){
    char str 1[20];
    char str 2[5][10];
    gets(str 1);
    for(int i = 0; i < 3; i++){
        gets(str 2[i]);
    }
    puts(str 1);
    for(int i = 0; i < 3; i++){
        puts(str 2[i]);
    }
    return 0;
}

```

输入下面四个字符串:

```

WoAiDeRenBuAiWo
QAQ
T_T
WoAiNi

```

这段代码通过 gets(str1) 将第一个字符串存入字符数组 str 1 中, 然后通过 for 循环将后三

有关此电子书的说明

本人可以帮助你找到你要的PDF电子书，计算机类，文学，艺术，设计，医学，理学，经济，金融等等。质量都很清晰，为方便读者阅读观看，每本100%都带可跳转的书签索引和目录，只要您提供给我书的相关信息，一般我都能找到，如果您有需求，请联系我 QQ1779903665。

PDF代找说明：

本人已经帮助了上万人找到了他们需要的PDF，其实网上有很多PDF，大家如果在网上不到的话，可以联系我QQ，大部分我都可以找到，而且每本100%带书签索引目录。因PDF电子书都有版权，请不要随意传播，如果您有经济购买能力，请尽量购买正版。

提供各种书籍的pd电子版代找服务，如果你找不到自己想要的书的pdf电子版，我们可以帮您找到，如有需要，请联系 QQ 1779903665.

备用:QQ 461573687

若以上联系方式失效，您可通过以下电子邮件获取有效联系方式。

E-mail : ebooksprite@163.com

E-mail : ebooksprite@foxmail.com

若您没有QQ通讯工具，请发送您的请求到 ebooksprite@gmail.com 与客服取得联系。

声明：本人只提供代找服务，每本100%索引书签和目录，因寻找和后期制作pdf电子书有一定难度，仅收取代找费用。如因PDF产生的版权纠纷，与本人无关，我们仅仅只是帮助你寻找到你要的pdf而已。