

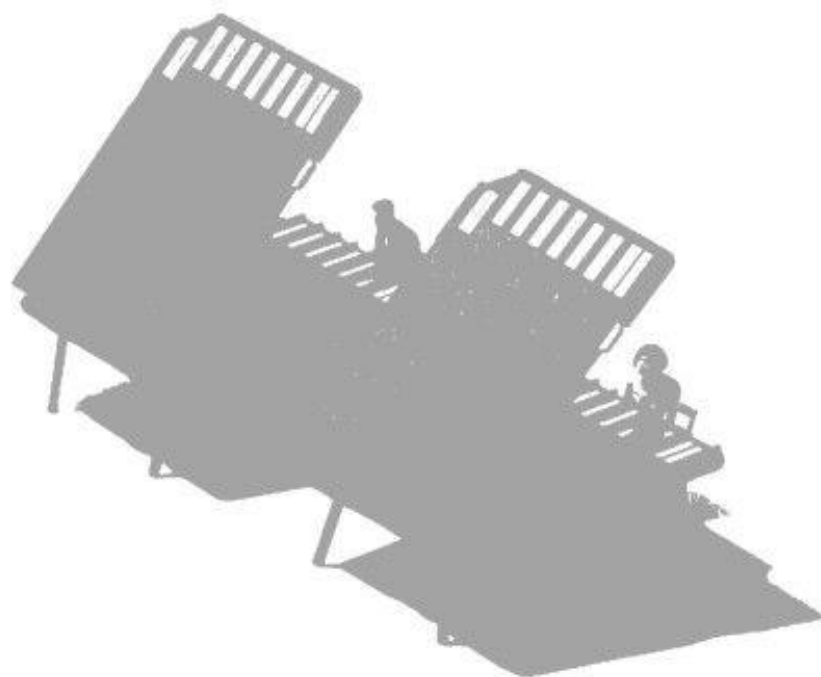


2019年第一学期版



大学计算机基础

期末小助手



仲英学业辅导中心出品



学辅公众号



学粉群 51

大学计算机基础 (C 语言版) 期末小助手

编写人员:(根据编写章节顺序排名) 机械 87 王雅康, 自动化 83 马英洪, 3D 打印 81 朱启翔, 越杰 81 (力学) 唐智亿, 数试 81 王辰扬, 自动化 74 陈子谦, 力学 71 颢孙宇翔
排版人员: 数试 81 王辰扬

感谢学业辅导中心各位工作人员与志愿者的努力工作, 使本资料可以按时完工. 由于编者们的能力与精力限制, 以及本资料是仲英学业辅导中心首次采用 \LaTeX 排版, 难免有错误之处. 如果同学们在本资料中发现错误, 请联系仲英学业辅导中心: **XJTUzyx-uefu@163.com**, 我们将在修订时予以更正.

从第 3 周开始, 每晚 **19:30-21:30**, 学辅志愿者在东 21 舍 118 学辅办公室值班, 当面为学弟学妹们答疑.

同时, 我们也有线上答疑平台——学粉群.

18 级学粉群: 646636875, 928740856;

19 级学粉群: 902493560,756433480.

期中考试与期末考试前, 我们还会举办考前讲座. 学辅还有新生专业交流会, 转专业交流会, 英语考试讲座等活动, 消息会在学粉群和公众号上公布, 欢迎同学们参与.

仲英书院学业辅导中心

2019 年 9 月 23 日

目录

第一章 计算机基本组成原理	7
1.1 引论 (背诵 ★★★)	7
1.1.1 计算机发展的四个阶段	7
1.1.2 微机系统组成	7
1.2 硬件系统 (背诵 ★★★)	8
1.3 软件系统 (背诵 ★★★)	8
1.4 主机	8
1.4.1 CPU (背诵 ★)	8
1.4.2 储存器: 内存和外存 (背诵 ★)	8
1.5 I/O 接口 (背诵 ★)	10
1.6 主机板 (背诵 ★★★)	10
1.6.1 芯片部分 (包括:CPU, 控制芯片组,BIOS)	10
1.6.2 扩展槽 (了解)	10
1.7 计算机的重要指标 (应用 ★)	11
1.8 图灵机 (一种思想模型) (背诵 ★★★)	11
1.9 练习题	12
第二章 系统软硬件构造	15
2.1 逻辑运算与逻辑门 (应用 ★★★★★)	15
2.1.1 学会判断是否为命题以及命题的真假	15
2.1.2 “与” 门和 “或” 门	15
2.1.3 “非” 运算	16
2.1.4 其它逻辑运算	16
2.2 触发器与加法器	17
2.2.1 基本概念 (背诵 ★★★★★)	17
2.2.2 半加法器与加法器的比较	17
2.2.3 半加法器和全加法器的性质	18
2.3 逻辑电路 (应用 ★)	19
2.4 触发器 (应用 ★)	20
2.4.1 RS 触发器	20

2.4.2	D 触发器	20
2.4.3	触发器的作用	20
2.5	冯诺伊曼结构与原理	21
2.5.1	冯诺依曼计算机结构 (背诵 ★★★)	21
2.5.2	指令和程序 (应用 ★★★)	21
2.6	冯诺依曼计算机基本原理 (背诵 ★★★)	23
2.6.1	冯诺依曼系统的局限性 (背诵 ★)	24
2.7	操作系统 (OS)	24
2.7.1	概念 (背诵 ★)	24
2.7.2	基本功能 (背诵 ★★★)	24
2.7.3	应用: 程序的并发执行 (背诵 ★)	24
2.7.4	应用: 进程 (应用 ★★★★★)	25
2.7.5	存储器管理的主要功能 (背诵 ★★★)	26
2.7.6	存储器扩充 (应用 ★★★)	27
第三章	二进制与编码	29
3.1	计算机与二进制	29
3.2	数制及其转换	30
3.2.1	计算机中的信息单位	30
3.2.2	进制表示法	31
3.2.3	进制转换	31
3.3	二进制数的表示和运算	32
3.3.1	数的表示	32
3.3.2	机器数的表示和运算	33
3.4	信息表示与编码 (非重点)	34
3.4.1	西文字符编码	34
3.4.2	汉字编码	35
第四章	计算机网络	37
4.1	计算机网络基础知识	37
4.1.1	概述	37
4.1.2	计算机网络发展 (了解)	37
4.1.3	计算机网络分类	38
4.1.4	TCP/IP 协议及其体系结构 (背诵 ★★★)	38
4.1.5	网络应用模式 (背诵 ★)	39
4.2	因特网	39
4.2.1	因特网的结构与组成	39
4.2.2	IP 地址和端口号 (背诵 ★★★)	39
4.2.3	子网和子网掩码 (应用 ★★★★★)	40

4.2.4	域名地址和 MAC 地址	41
4.2.5	练习题	41
第五章	C 语言语法基础	45
5.1	序与概述	45
5.2	C 语言的基本知识	46
5.2.1	C 语言程序的基本结构	46
5.3	代码编译与运行	47
5.4	C 语言的其它基本要素	48
5.4.1	标识符	48
5.4.2	空格	48
5.4.3	关键字	48
5.5	常见数据类型	49
5.5.1	常量	49
5.5.2	变量	52
5.6	运算符与表达式	52
5.7	控制结构	53
5.7.1	顺序结构	53
5.7.2	选择结构	54
5.7.3	循环结构	54
5.7.4	switch-case 语句	55
5.7.5	break-continue 语句	55
5.8	数组	56
5.8.1	数组的基本概念	56
5.8.2	一维数组	56
5.8.3	二维数组	56
5.8.4	字符串	56
5.9	结构体	57
5.9.1	结构体定义与成员引用	57
5.10	函数	57
5.10.1	函数的定义方式	57
5.10.2	函数调用	57
5.10.3	全局变量与局部变量	58
5.10.4	递归函数	58
5.10.5	库函数	60
5.11	指针	60
5.11.1	指针基础知识	60
5.11.2	指针初始化与运算	60
5.11.3	指向一维数组的指针	61

5.11.4 动态储存分配	62
第六章 数据结构与算法	63
6.1 数据与数据结构简介	63
6.2 线性表	64
6.2.1 线性表的逻辑结构 (应用 ★)	64
6.2.2 线性表的基本操作 (应用 ★★★)	64
6.2.3 代码实现 (应用 ★★★★★)	64
6.3 栈与队列	66
6.3.1 基本操作 (应用 ★★★★★)	66
6.3.2 代码实现 (应用 ★★★★★)	66
6.4 图与树	67
6.4.1 图的定义 (背诵 ★)	67
6.4.2 图的常用概念 (背诵 ★★★)	67
6.4.3 图的邻接矩阵表示 (应用 ★)	67
6.4.4 树的定义	68
6.4.5 树的基本概念	68
6.4.6 二叉树	68
6.4.7 树的遍历 (应用 ★★★★★)	69
6.5 算法的描述方法	69
6.6 排序算法简介	70
6.6.1 冒泡排序 (应用 ★★★★★)	70
6.6.2 选择排序 (应用 ★★★)	71
6.6.3 快速排序 (应用 ★★★)	72
6.7 查找算法与思想	74
6.7.1 顺序查找 (应用 ★★★)	74
6.7.2 折半查找 (应用 ★★★)	76
6.7.3 查找算法总结	78
6.8 递归与分治	85
6.8.1 定义 (背诵 ★)	85
6.8.2 应用举例 (应用 ★★★★★)	85
6.8.3 不合理递归 (应用 ★★★)	88
6.9 练习题	90
6.10 长图片与长代码	99
6.10.1 顺序表实现代码 SeqList.c	99
6.10.2 链表实现代码 LinkList.c	102
6.10.3 数组实现栈的代码 Stack.c	104
6.10.4 数组实现队列的代码 Queue.c	106
6.10.5 冒泡排序流程图	108

第一章 计算机基本组成原理

编者: 王雅康

1.1 引论 (背诵 ★★★)

1.1.1 计算机发展的四个阶段

电子管计算机 (第一台:1946 年)→ 晶体管计算机 → 集成电路 → 大规模或超大规模集成电路.

1.1.2 微机系统组成

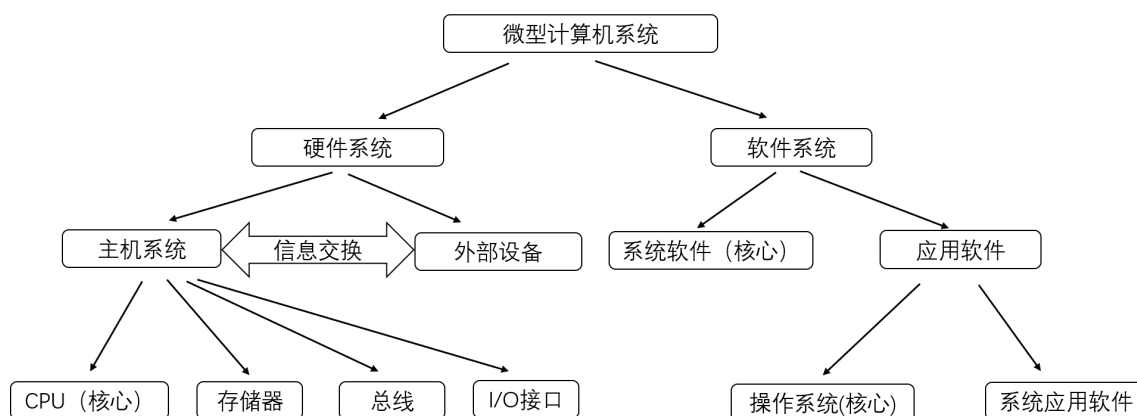


图 1.1: 微型计算机组成框构图

例 1.1.1 微型计算机系统的概念结构由 ____ 组成.

- A. 微处理器, 总线, 存储器, 输入输出设备, I/O 接口, 软件系统
- B. 微处理器, 总线, 存储器, 输入输出设备, I/O 接口, 主机
- C. 微处理器, 主机, 存储器, 输入输出设备, I/O 接口
- D. 微处理器, 存储器, 输入输出设备, I/O 接口, 软件系统

答案 A. 解析: 见上面框构图.

1.2 硬件系统 (背诵 ★★★)

1. 概念区分: 主机箱 \neq 主机.
2. 判断是否为主机部件的重要标志: 能否与处理器进行直接信息交流 (不通过接口)
3. 外部设备: 所有能够与计算机进行信息交换的设备 (通过接口)
4. 输入设备: 向计算机输入信息的设备. 输出设备: 接受计算机输出信息的设备.

注: 一个设备可以身兼两职. 如: 磁盘, 计算机向其写入数据时为输出设备, 读取数据时为输入设备

1.3 软件系统 (背诵 ★★★)

1. 系统软件功能: 管理, 监控, 维护计算机软硬件的软件.
2. 操作系统 (OS) 功能: 存储器管理, 文件管理, 进程管理, 设备管理等. 在计算机上运行的其他所有系统软件及各种应用程序都要依赖于 OS 的支持.
3. (背诵 ★★★★★) 软件系统的核心是系统软件, 系统软件的核心是操作软件.

1.4 主机

1.4.1 CPU (背诵 ★)

1. 运算器 (ALU): 执行指令, 进行各种算术和逻辑运算.
2. 内部寄存器组: 用于暂存运行数据, 避免 CPU 频繁访问存储器, 缩短运行时间.
3. 程序计数器 (PC): 用于指示下一条要取指令的地址. 使 CPU 按顺序执行程序, 处理文档.
4. 控制逻辑单元: 产生时序信号, 控制和协调整个 CPU 的工作.
5. 总线: CPU 内部数据的传输通道.

1.4.2 存储器: 内存和外存 (背诵 ★)

内存 (属于主机设备)

特点:

1. 可与 CPU 直接进行信息交换.

2. 存储速度快, 容量小, 单位字节容量价格高.
3. 断电后, 存储信息丢失.

区分 ROM 与 RAM: ROM (只读储存器) 在断电时数据不丢失; RAM (随机储存器) 在断电时数据会丢失.

高速缓冲储存器 (Cache) (了解) 是存在于主存与 CPU 之间的一级存储器, 容量比较小但速度比主存高得多, 接近于 CPU 的速度. 在计算机存储系统的层次结构中, 是介于中央处理器和主存储器之间的高速小容量存储器. 它和主存储器一起构成一级的存储器.

地址码 (应用 ☆) (以二进制表示) 地址码长度体现内存容量.

位和字节 (应用 ☆☆☆) 位 (一个二进制位, bit, b) 存储的最小数据单位; 字节 (Byte, B) 基本的数据单位. 一般一个字母占 1B, 一个汉字占 2B.

换算:

$$1\text{B} = 8\text{bit}$$

$$1\text{kB} = 2^{10}\text{B}$$

$$1\text{MB} = 2^{10}\text{kB} = 2^{20}\text{B}$$

$$1\text{GB} = 2^{10}\text{MB} = 2^{30}\text{B}$$

外存 (属于外设) (应用 ☆☆☆☆)

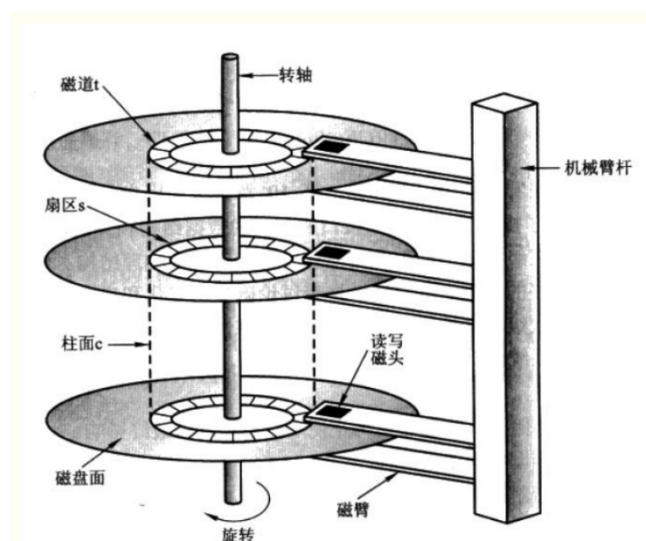


图 1.2: 机械硬盘原理图

对硬盘读写的基本单位是**扇区**。每扇区容量为 512B。整块硬盘的容量 = 磁头数 * 柱面数 * 扇区数 * 512B (公式套用即可)。

例 1.4.1 已知某硬盘的磁头数为 16, 柱面数为 4096, 扇区数为 64, 求该硬盘的容量。

解 该硬盘的容量 = $16 * 4096 * 64 * 512 = 2147483648 \text{B} = 2\text{GB}$ 。

1.5 I/O 接口 (背诵 ☆)

作用示意: CPU \longleftrightarrow I/O 接口 \longleftrightarrow 外设

功能: CPU 与外设的速度匹配, 信息的输入输出, 信息的转换, 总线隔离。

1.6 主机板 (背诵 ☆☆☆)

主机位于主机箱内, 主要包括芯片, 扩展槽和对外接口 3 种类型的部件。

1.6.1 芯片部分 (包括:CPU, 控制芯片组,BIOS)

典型的芯片组由南桥和北桥芯片两部分组成

1. 北桥芯片是芯片组的核心, 主要负责处理 CPU, 内存和显卡三者之间的“交通”。发热量较大, 故加装散热片。
2. 南桥芯片主要负责硬盘等存储设备和 PCI 之间的数据流通。

(了解)BIOS 芯片是可读写的只读存储器。系统 BIOS 程序的主要功能: 上电自检, 初始化, 系统设置。

1.6.2 扩展槽 (了解)

主板上包括内存插槽和总线接口插槽两大类。

内存插槽: 用于安装内存储器 (内存条)

总线接口插槽: CPU 通过系统总线与外设联系的通道。主要有 PCI 插槽, AGP 插槽, PCIE 插槽。

总线: 计算机中传输信息的通道。(按层次结构可分为: CPU 总线 (又称前端总线), 系统总线, 外设总线)

前端总线: 指从 CPU 引脚上引出的连接线, 用于实现 CPU 与主存储器, CPU 与 I/O 接口芯片, CPU 与控制芯片组等芯片之间的数据传输, 也用于系统中多个 CPU 之间的连接.

系统总线 (I/O 通道总线): 主机系统与外围设备之间的通信通道. **外设总线:** 计算机主机与外部设备接口总线.

1.7 计算机的重要指标 (应用 ★)

1. 主频 (主时钟频率): 一秒钟内发生的同步脉冲数, 单位兆赫 (MHz);
2. 运算速度;
3. 内存容量; 字长: 指 CPU 能够同时处理的二进制位数. 字节越长, 运算精度越高, 数据处理速度越快;
4. 外设的配置及扩展能力.

(了解) 衡量一个微处理器性能的高低, 最重要的是执行指令所用时间的多少. 而所用时间的多少与时钟速度和执行一条指令所需的时钟脉冲个数有关. 所以在同等情况下, CPU 的钟频越高, 运算速度越快.

1.8 图灵机 (一种思想模型) (背诵 ★★★)

(了解) **图灵机:** 一个按照确定, 有限的规则和步骤, 将输入信号进行变换后给出输出信息, 并在遇到停止状态时就结束工作的系统.

图灵机证明了: 任何能够被图灵机完成的工作都是可计算的.

图灵机是计算机的理论模型

纸带对应于计算机中的**存储器: 内存, 硬盘**
读写头对应于计算机中的**处理器 CPU: 运算器**
规则对应于**程序, 指令**
图灵机与计算机都具有**内部状态**

计算机不可能解决世界上的所有问题.

“不可解决性”反映在两方面: 无限步骤, 无限时间.

1.9 练习题

1. 以下哪种说法是错误的 ____
 - (a) 计算机系统是指包括外部设备在内的所有硬件
 - (b) 系统软件是管理、监控和维护计算机软硬件资源的软件
 - (c) 主板上的 BIOS 芯片中保存了控制计算机硬件的基本软件
 - (d) 现代微型计算机均采用了多总线系统结构
2. I/O 接口是指 ____
 - (a) 微处理器与存储器之间的接口
 - (b) 外部设备与存储器之间的接口
 - (c) 外部设备中用于与主机进行数据传输的接口
 - (d) 主机与外部设备之间的接口
3. 打开计算机的电源时, 首先执行的程序 ____
 - (a) 硬盘上的操作系统
 - (b) 内存中的操作系统
 - (c) 硬盘中的引导程序 (Boot Loader)
 - (d) 位于 BIOS 中的引导程序 (Boot Loader)
4. 在硬盘中 ____
 - (a) 每个文件都有固定的连续存储区域
 - (b) 每个文件都存放在某个固定的扇区中
 - (c) 每个文件都一定不存放在某个固定的扇区中
 - (d) 每个文件都不一定存放在连续的扇区中

参考答案

1. (a). (a) 项应为计算机系统是指包括外部设备在内的所有硬件和软件. 其余正确选项建议理解记忆.
2. (d). I/O 接口是将外设连接到系统总线上的一组逻辑电路的总称, 也称为外设接口. 实现 CPU 和外部设备之间频繁的信息交换.C 项错因为 I/O 接口不属于外部设备.
3. (d). 考查系统 BIOS 程序的上电自检功能.

4. (d). 硬盘中文件存储的物理位置不一定是连续的. 由于文件存放的地址是连续的, 并且计算机按照地址进行读写操作, 所以文件存放的逻辑位置是连续的.

第二章 系统软硬件构造

编者: 王雅康

2.1 逻辑运算与逻辑门 (应用 ★★★★★)

2.1.1 学会判断是否为命题以及命题的真假

2.1.2 “与”门和“或”门

“与”(用 \wedge 或者乘号 \cdot 表示) 和 “或”(用 \vee 或者加号 $+$ 表示) 的运算规则:

$$a \wedge b = \begin{cases} 1, & a = b = 1 \\ 0, & \text{etc.} \end{cases}, a \vee b = \begin{cases} 0, & a = b = 0 \\ 1, & \text{etc.} \end{cases}$$

一位的所有情况:

$$\begin{array}{llll} 0 \wedge 0 = 0 & 1 \wedge 0 = 0 & 0 \wedge 1 = 0 & 1 \wedge 1 = 1 \\ 0 \vee 0 = 0 & 1 \vee 0 = 1 & 0 \vee 1 = 1 & 1 \vee 1 = 1 \end{array}$$

对于多位的情况, 右对齐按位做与运算即可. 空位补 0.

例 2.1.1

$$\begin{array}{l} 11011010B \wedge 10010110B = 10010010B \\ 10101010B \wedge 11011101B = 10001000B \\ 11011010B \wedge 10010110B = 11011110B \\ 10101010B \wedge 11011101B = 11111111B \end{array}$$

在电路中,“与”运算相当于开关的串联电路. 仅当所有开关都闭合时, 电路才通电.“与”门和“或”门的逻辑符号如图 2.1 所示.

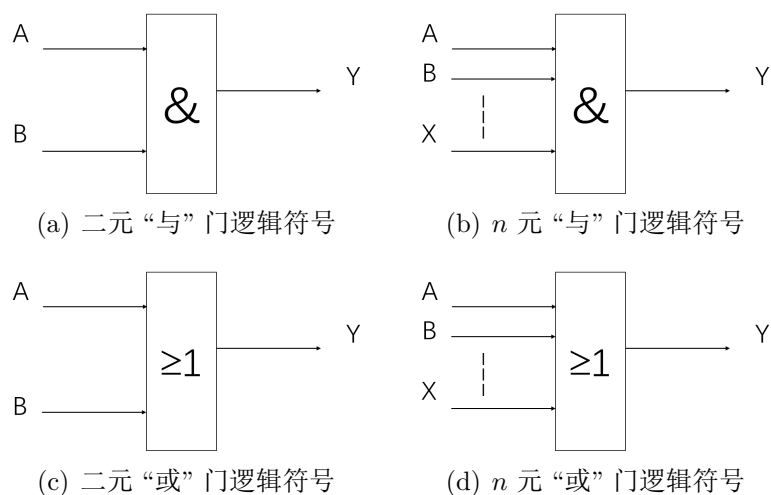


图 2.1: “与”门和“或”门的逻辑符号示意图

2.1.3 “非”运算

符号: 数值上面加一条横线, 如 $B = \bar{A}$.

运算法则: 按位取反 (0 转 1, 1 转 0). 一位的情况:

$$\bar{0} = 1 \quad \bar{1} = 0$$

多位的情况逐位操作即可.

例 2.1.2 $\overline{101101101} = 010010010$

“非”门的逻辑符号:

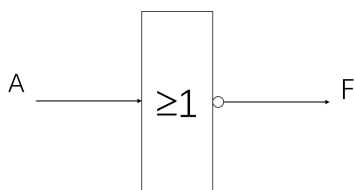
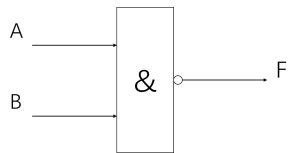
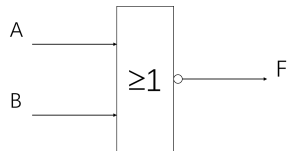
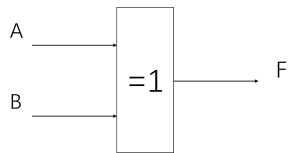
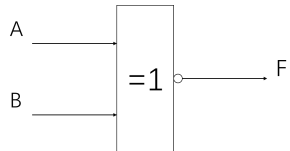


图 2.2: “非”门的逻辑符号示意图

2.1.4 其它逻辑运算

表 2.1 列举了其它常见的逻辑运算及其简单性质. 其中较为常用的是异或, 其它运算在本课程中出现的较少, 一般出现时也用其它运算 (与, 或, 非, 异或) 表示, 故读者只需做了解即可. 而对这四种常用的运算, 读者务必熟练掌握其计算与简单性质.

表 2.1: 其它逻辑运算一览表

名称	符号 (表达式)	运算规则	真值表	门电路逻辑符号
与非	$\overline{A \wedge B}, \overline{A \cdot B}$	先与后非	$\begin{array}{cc} \overline{0 \wedge 0} = 1 & \overline{0 \wedge 1} = 1 \\ \overline{1 \wedge 0} = 1 & \overline{1 \wedge 1} = 0 \end{array}$	
或非	$\overline{A \vee B}, \overline{A + B}$	先或后非	$\begin{array}{cc} \overline{0 \vee 0} = 1 & \overline{0 \vee 1} = 0 \\ \overline{1 \vee 0} = 0 & \overline{1 \vee 1} = 0 \end{array}$	
异或	$A \oplus B = \overline{A} \wedge B + A \wedge \overline{B}$	同 0 异 1	$\begin{array}{cc} 0 \oplus 0 = 0 & 0 \oplus 1 = 1 \\ 1 \oplus 0 = 1 & 1 \oplus 1 = 0 \end{array}$	
同或	$\overline{A \oplus b}$	同 1 异 0	$\begin{array}{cc} \overline{0 \oplus 0} = 1 & \overline{0 \oplus 1} = 0 \\ \overline{1 \oplus 0} = 0 & \overline{1 \oplus 1} = 1 \end{array}$	

2.2 触发器与加法器

2.2.1 基本概念 (背诵 ★★★★★)

1. 微处理器主要由控制器, 运算器和寄存器等三部分构成.
2. 所有程序的执行都是由运算器完成的.
3. 运算器的核心是算术逻辑单元 (ALU).
4. ALU 的基本功能部件是加法器.

2.2.2 半加法器与加法器的比较

表 2.2: 全/半加法器性质比较表

加法器类型	功能	输入	输出
半加法器	实现两个 1 位二进制数相加, 不考虑来自低位的进位	加数, 被加数	和, 进位
全加法器	实现两个 1 位二进制数相加, 考虑来自低位的进位	加数, 被加数, 低位的进位	和, 不进位

2.2.3 半加器和全加器的性质

电路图

半加器使用一个异或门和一个与门串联分别表示加法末位和进位, 全加器由两个半加器再加上一个“或”门构成 (只能实现 1 位二进制的加法运算). 它们的示意图如图 2.3 所示:

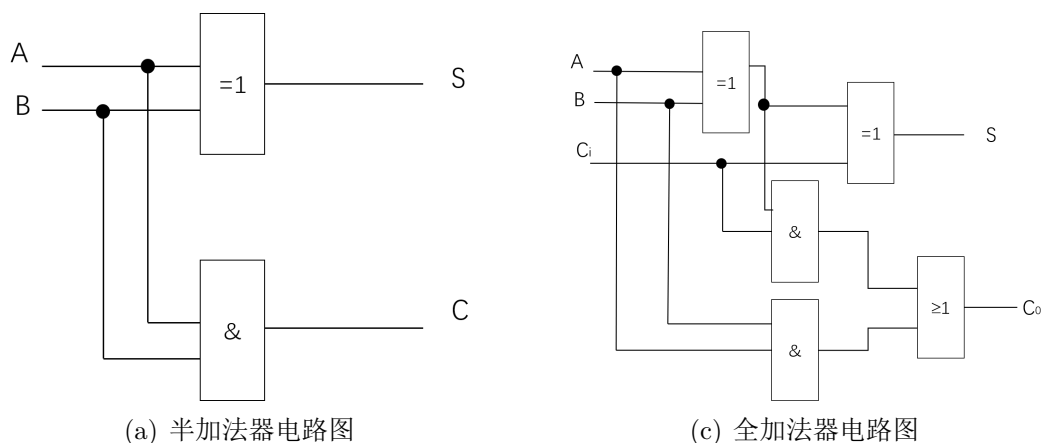


图 2.3: 半加器与全加器的电路图示意图

全加器有另一种设计, 由不同的电路实现, 具有相同的功能, 如图 2.4 所示:

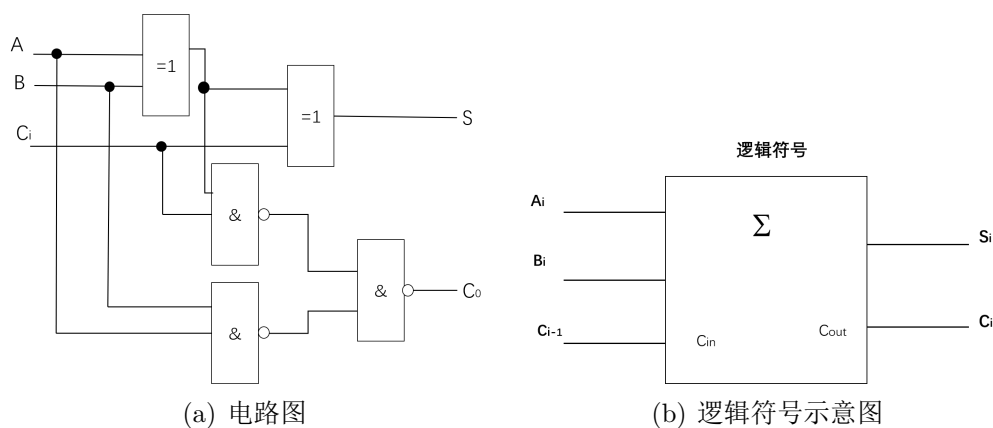


图 2.4: 全加器的另一种设计

电路中 A, B 表示两个加数, 全加器中的 C_i 表示进位, 输出的 S 和 C 分别表示加法结果和进位. 特别地, 在半加器中, 有 $S = A \oplus B, C = A \wedge B$.

逻辑关系 (真值表)

用真值表可以表示半加器与全加器的输入输出对应关系, 如表 2.3 所示:

表 2.3: 半加器与全加器的真值表

(a) 半加器				(b) 全加器				
输入		输出		输入			输出	
A	B	S	C	A	B	C	S	C_0
0	0	0	0	0	0	0	0	0
0	1	1	0	0	0	1	1	0
1	0	1	0	0	1	0	1	0
1	1	0	1	0	1	1	0	1
				1	0	0	1	0
				1	0	1	0	1
				1	1	0	0	1
				1	1	1	1	1

半加器与全加器的应用: 连波进位加法器

上面的半加器与全加器都只能计算一位二进制加法, 用 N 个 1 位加法器组成的连波进位加法器可以计算 N 位二进制数字之间的加减法, 图示如下:

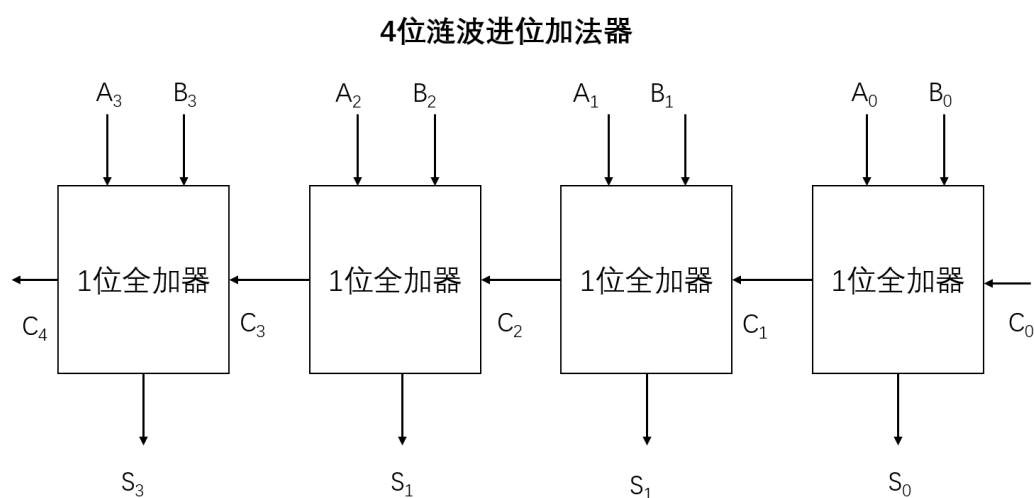


图 2.5: 4 位连波进位加法器图示

2.3 逻辑电路 (应用 ★)

组合逻辑电路: 任意时刻的输入仅仅取决于该时刻的输入, 与电路原来的状态无关 (没有记忆功能).

时序逻辑电路: 任意时刻的输入不仅取决于该时刻的输入, 还与以前的输入有关 (有记忆功能).

2.4 触发器 (应用 ☆)

触发器是构成计算机储存装置的基本单元.

2.4.1 RS 触发器

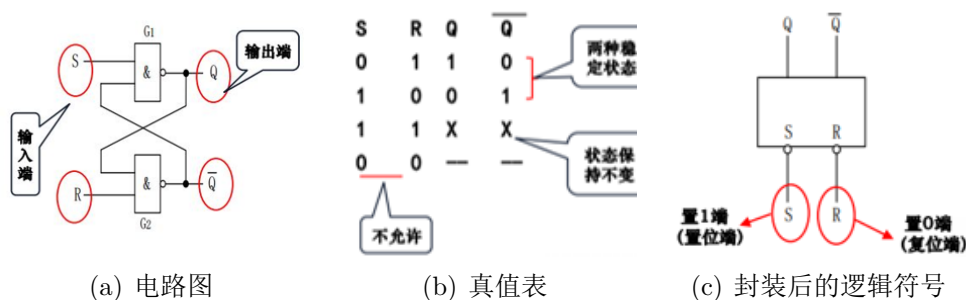


图 2.6: RS 触发器讲解图

2.4.2 D 触发器

在 RS 触发器基础上增加两个与非门. 下图为 D 触发器的示意图, 其中 $CP=0$ 时输出状态保持不变; $CP=1$ 时输出取决于 D 端状态.

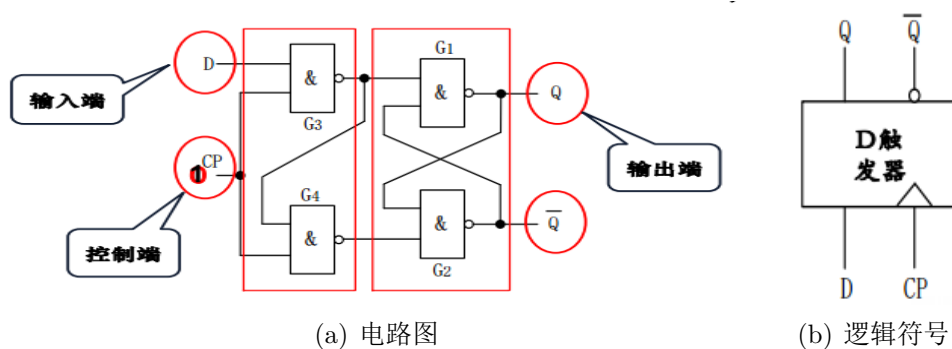


图 2.7: D 触发器讲解图

2.4.3 触发器的作用

1. 触发器是具有记忆功能的逻辑部件 [任何时候输出端都能保持一个确定的稳定状态 (0 或 1)].
2. 一个触发器能够存储 1 位二进制数.
3. 触发器是构成计算机存储装置的基本单元.

2.5 冯诺伊曼结构与原理

2.5.1 冯诺依曼计算机结构 (背诵 ★★★)

1. 采用二进制: 计算机中所有信息 (数据和指令) 统一用二进制表示.
2. 设计计算机硬件由五个部分构成: 运算器, 逻辑控制装置, 存储器, 输入设备, 输出设备.
3. 存储程序原理: 如图 2.8. 特点: 以运算器为核心, 所有信息的输入和输出都需要通过运算器.

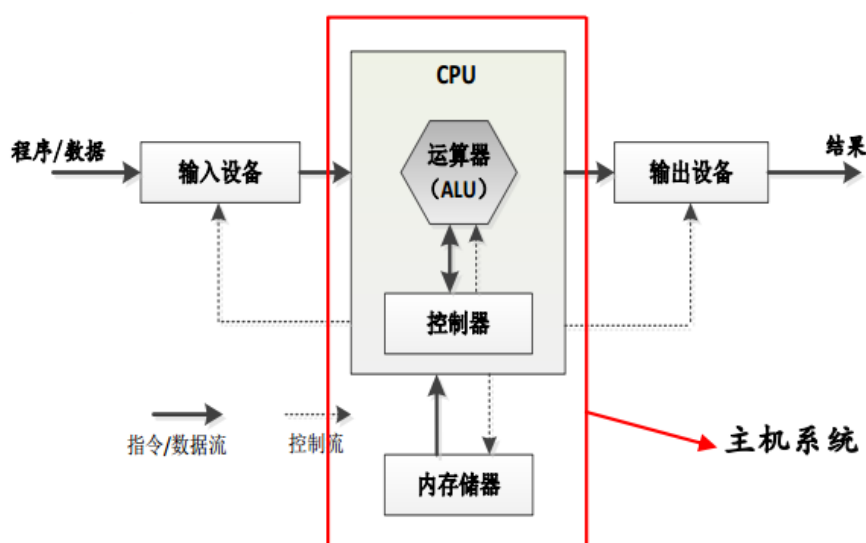


图 2.8: 储存程序原理示意图

2.5.2 指令和程序 (应用 ★★★)

概念

1. **指令**: 控制计算机完成某项操作的, 能够被计算机识别的“命令”(二进制形式描述的机器指令).
2. **指令系统**: 计算机能够识别的所有指令的集合.
3. **程序**: 按一定顺序组织在一起的指令序列.

指令格式

指令的格式为**操作码** + **操作数**. 其中操作码说明指令的功能, 操作数说明指令操作的对象.

程序计数器 (PC)

步骤:

1. PC 用来产生和存放下一条将要读取的指令的地址.
2. PC 每输出一次地址, 就指向内存的一个单元, CPU 将该单元的指令自动取出.
3. PC 中内容自动加 1, 准备读取下一条指令.

PC 的作用: 程序执行的“指挥棒”.PC 指向哪里,CPU 就到哪里取指令.

例 2.5.1 控制程序能够按照一定的顺序对一条条指令执行的主要部件 _____

- A. 通用寄存器
- B. 指令寄存器
- C. 堆栈指针
- D. 程序计数器

答案: D. 解析: 程序计数器的作用就是指挥 CPU 按顺序执行指令.

指令执行的过程

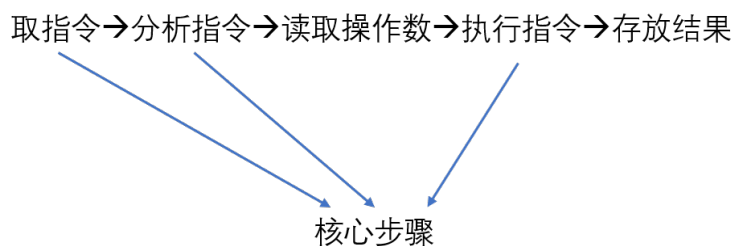


图 2.9: 指令执行过程示意图

例 2.5.2 处理器取指令是指 _____

- A. 将指令从内存读入 CPU
- B. 将指令从硬盘读入 CPU
- C. 将指令从寄存器读入运算器
- D. 将指令从硬盘读入运算器

答案 A. 解析: 需要记住 CPU 只从内存 (或高速缓存) 中读入数据.

两种执行方式比较

1. **顺序执行:** 一条指令执行完了再执行下一条指令.

执行方式: CPU 取指令 1 → 分析指令 1 → 执行指令 1 → 去指令 2 → 分析指令 2 → 执行指令 2 → ...

执行时间 = 取指令 + 分析指令 + 执行指令. 若设三部分的执行时间均为 Δt , 则执行 n 条指令时间 $T_0 = 3\Delta t$.

2. **并行执行:** 同时执行两条或多条指令.

执行方式:

```

取指令 1  → 分析指令 1  → 执行指令 1
              取指令 2  → 分析指令 2  → 执行指令 2
                      取指令 2  → 分析指令 2  → 执行指令 2
  
```

仅第 1 条指令需要 $3\Delta t$ 时间, 之后每经历 $1\Delta t$, 就有一条指令执行结束, 因此执行时间 $T = 3\Delta t + (n - 1)\Delta t$.

时间比较: 并行拥有更高的效率, 同时用于更高的复杂度. 相比顺序执行, 并行执行的优势用速率比表示:

$$S = \frac{t_{\text{顺序执行}}}{t_{\text{并行执行}}} = \frac{3n\Delta t}{3\Delta t + (n - 1)\Delta t} = \frac{3n}{n + 2} \rightarrow 3(n \rightarrow +\infty)$$

2.6 冯诺依曼计算机基本原理 (背诵 ★★★)

存储程序控制原理, 以运算器为核心, 采用二进制.

进一步可表示:

1. 将计算过程描述为由多条指令按一定顺序组成的程序, 并放入存储器保存;
2. 指令按其在存储器中存放的顺序执行;
3. 由控制器控制整个程序和数据的存取以及程序的执行.

例 2.6.1 冯诺依曼计算机的基本工作原理是 _____

- A 以运算器为核心
- B 存储数据原理
- C 存储指令原理
- D 存储程序原理

答案 D. **解析:** 基本工作原理: 将程序和数据存放到计算机内部的存储器中, 计算机在程序的控制下一步一步进行处理, 直到得出结果.

2.6.1 冯诺依曼系统的局限性 (背诵 ☆)

1. CPU 与存储器之间会有大量的数据交互, 造成总线瓶颈.
2. 指令的执行顺序由程序计数器控制, 使得即使有关数据已经准备好, 也必须逐条执行指令序列.
3. 指令的执行顺序由程序决定, 对一些大型的, 复杂的任务是比较困难;
4. 以运算器为中心, I/O 设备与存储器间的数据传送都要经过运算器, 使处理效率, 特别是对非数值数据的处理效率比较低.

2.7 操作系统 (OS)

2.7.1 概念 (背诵 ☆)

操作系统是一组控制和管理计算机软, 硬件资源, 为用户提供便捷使用计算机的程序集合; 是用户和计算机之间进行“交流”的界面.

2.7.2 基本功能 (背诵 ☆☆☆)

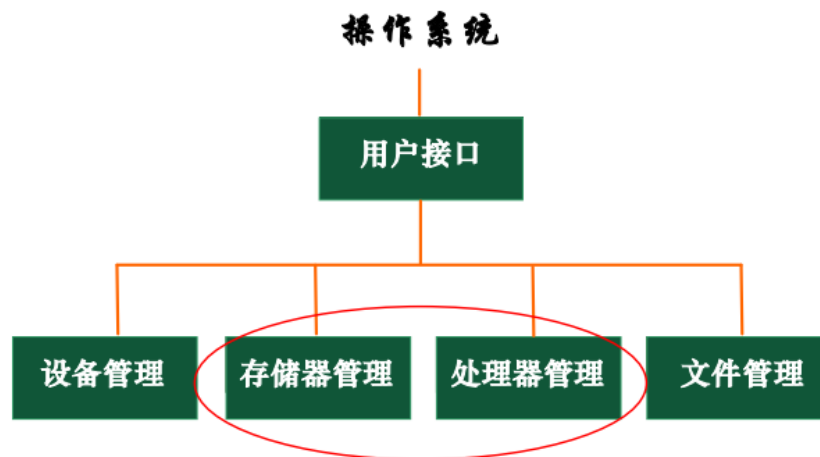


图 2.10: 五大基本功能示意图

主要功能对应:

2.7.3 应用: 程序的并发执行 (背诵 ☆)

特点: 结果不可再现 (计算结果与程序段的执行速度 (顺序) 有关)

多道程序在宏观上可“同时”执行 (只有 1 个 CPU, 微观上不可能同时执行) 系统资源为多道程序共享.

例 2.7.1 判断正误: 进程管理使得 CPU 在某一时刻可以同时执行多个程序.

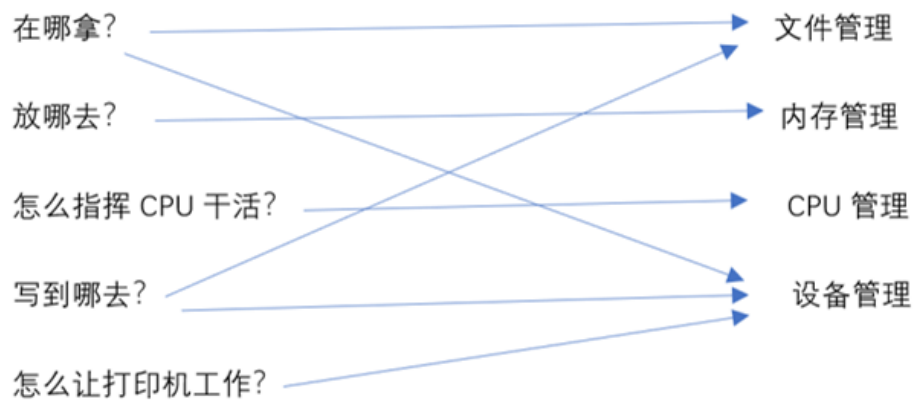


图 2.11: 部分主要功能对应示意图

答案 错. **解析**: 多道程序在宏观上可“同时”执行 (只有 1 个 CPU, 微观上不可能同时执行)

2.7.4 应用: 进程 (应用 ★★★★★)

概念

1. 进程是程序的一次执行过程. 是系统进行资源分配和调度的一个独立单位.(一个程序可以对应多个进程, 一个进程也可以对应多个程序)
2. 在多道程序环境中, 对系统内部资源的分配和管理是以进程为基本单位.
3. 任何程序要运行, 都必须为它创建进程.

进程与程序的关系

1. 进程是动态的, 程序是静态的: 进程是程序的一次执行, 程序是有序代码的集合
2. 进程是暂时的, 程序是永久的: 进程有生命周期, 会消亡, 程序可长期保存在外存储器上
3. 进程与程序密切相关: 同一程序的多次运行对应到多个进程; 一个进程可以通过调用激活多个程序.

进程的基本状态与状态转换

受资源的制约, 进程在其生命周期中的执行过程是间断性的. 有三种基本状态:

1. **就绪状态**: 已经获得了除 CPU 之外所必需的一切资源, 一旦分配到 CPU, 就可立即执行.
2. **运行状态**: 已经获得 CPU 及其它一切所需资源, 正在运行.

3. **等待状态:** 由于某种资源得不到满足, 进程运行受阻, 处于暂停状态, 等待分配到所需资源后, 再投入运行.

进程之间的状态转换如图 2.12 所示. 在整个生存周期内, 进程的状态处于不断变化中.

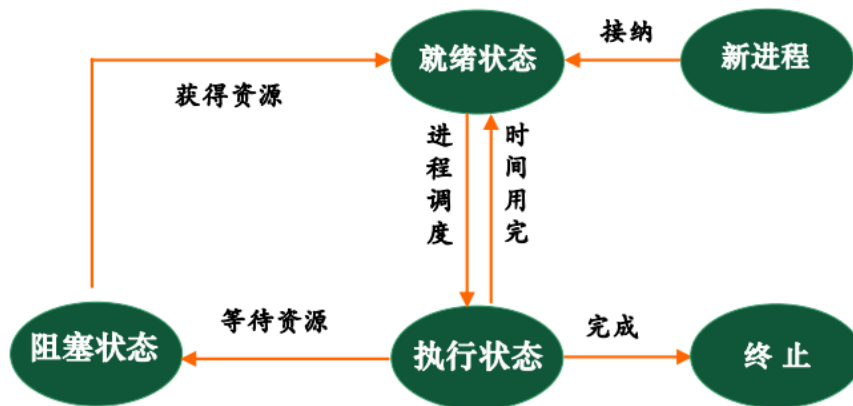


图 2.12: 进程状态转换示意图

例 2.7.2 当所需资源不能得到满足时, 当前进程将 _____

- A. 被结束
- B. 被挂起
- C. 继续运行
- D. 被保存到硬盘中

答案 B. **解析:** 考查进程的等待状态. 当所需资源不足时, 进程将进入等待状态.

2.7.5 存储器管理的主要功能 (背诵 ★★★)

1. 负责将程序从联机外存储器 (硬盘) 调入内存
地址变换: 将程序中的地址对应到内存中的地址
存储分配: 为程序分配相应的内存空间
2. 将硬盘和内存实行统一的管理 (存储器系统)
3. 存储扩充: 解决在小的存储空间中运行大程序的问题
4. 存储保护: 保护各类程序及数据区免遭破坏

2.7.6 存储器扩充 (应用 ★★★)

1. 主导思想: 如何在有限的内存空间中, 处理大于内存的程序.
2. 虚拟存储技术
 - (a) 将内存与部分硬磁盘统一在一起管理, 使其构成一个整体, 从而将部分外存空间作为内存使用.
 - (b) 从用户的角度, 相当于有一个容量足够大的内存空间. 用户可以在这个地址空间内编程 (存储扩充), 而完全不考虑内存的大小.

例 2.7.3 计算机 _____

- A. 只能运行小于或等于其内存容量的程序
- B. 内存被当前程序全部占满时不能再运行新的程序
- C. 可与运行大于其内存容量的程序
- D. 内存不够时将先结束当前运行的程序再运行新的程序

答案 C. 解析: 考察虚拟存储技术. 计算机可以运行比其内存大的程序. 而对于 D 项, 程序在虚拟存储器环境中运行时, 是只将那些当前要运行的程序段装入内存, 其余部分则存留在外存中.

第三章 二进制与编码

编者: 马英洪

有必要说明一下, 在考试中, 这一部分内容几乎只要求计算, 知识点比较少, 一般是两道选择, 两道填空, 请大家将“内存”多分配一些给硬件和网络部分, 此处着重于理解.

3.1 计算机与二进制

在计算机内部, 所有的信息都采用二进制 (即 0 和 1) 形式存放.
使用二进制的原因:

1. 二进制只有 0 和 1 两个基本符号, 具有两种稳定状态的电子器件很容易找到, 产生两种稳定状态的电路也易于设计.
2. 二进制的算术运算规则简单.(加法和乘法各仅有三条运算规则)
3. 易于与十进制转换.
4. 适合逻辑运算.(逻辑运算的对象是真和假, 二进制的 1 和 0 刚好与之对应)

例 3.1.1 晶体管的导通与截止

在二极管电路中, 当 X 端电位为 0 时, 晶体二极管导通, 有电流通过电阻 R ; 当 X 端电位为 $+5V$ 时二极管将截止, R 上不会有电流通过. 根据欧姆定律知, 导通时 a 点电位近似于 $0V$ (低电平); 截止时因 $i = 0$, a 点电位为 $+5V$ (高电平). 如果周期性地使 X 端呈现 $0V$ 和 $+5V$, 则二极管就会周期性地导通和截止. 可将 $0V$ 用 0 表示, $+5V$ 用 1 表示, 上述过程就与二进制码对应了.

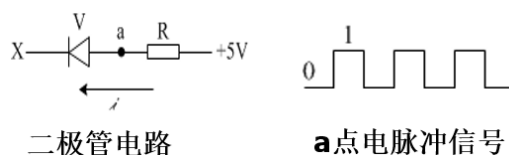


图 3.1: 二极管状态表示 0 或一

二进制加法运算规则 (减法可以转化为加法):

$$0 + 0 = 0 \quad 1 + 0 = 1 \quad 1 + 1 = 10$$

二进制乘法运算规则 (除法可转化为乘法):

$$0 \times 0 = 0 \quad 0 \times 1 = 0 \quad 1 \times 1 = 1$$

加减比较简单, 这里给出乘除法的实例:

例: $100111 \div 110 = ?$ $1011 \times 100 = ?$

$ \begin{array}{r} 110 \overline{) 100111} \\ \underline{110} \\ 0111 \\ \underline{110} \\ 0011 0 \\ \underline{11 } 0 \\ 0 \end{array} $	$ \begin{array}{r} 1011 \\ \times 100 \\ \hline 0000 \\ 0000 \\ 1011 \\ \hline 101100 \end{array} $
--	--

3.2 数制及其转换

计算机的硬件能直接识别的只有 0 和 1 构成的二进制数, 想要识别其他进制数, 则必须借助软件.

计算机中常用的计数制:

二进制 B, 十进制 D, 八进制 O, 十六进制 H

3.2.1 计算机中的信息单位

Bit, 比特/位是计算机中最小的信息单位, 只能存放一个二进制位, 即表示 0 或 1 两种状态.

Byte, 字节/B 是计算机处理数据的基本单位, 由八个二进制位构成, 同时也是存储空间大小的基本容量单位.

$$1Byte = 8Bit; \quad (3.1)$$

$$1KB = 1024B = 2^{10}B; \quad (3.2)$$

$$1TB = 1024GB = 1024^2MB = 1024^3KB = 1024^4B \quad (3.3)$$

3.2.2 进制表示法

字长 (字): 计算机一次能够同时 (并行) 处理的二进制位数.

字长越长, 计算机处理数据的速度越快.(早期计算机的字长为 8 位, 现在微机的并行处理能力一般为 64 位, 而大型机已达 128 位.)

十进制表示法:

1. 有 $0 \dots 9$ 共十个数字符号
2. 逢十进一

同理, n 进制数共有 $0 \sim n-1$ (从 10 开始依次用字母 A, B, \dots 来代替), n 个数字符号, 逢 n 进位.

举例:

$$\begin{cases} 126.43D = 1 * 10^2 + 2 * 10^1 + 6 * 10^0 + 4 * 10^{-1} + 3 * 10^{-2} \\ 1101.11B = 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 + 1 * 2^{-1} + 1 * 2^{-2} \\ A2.48H = 10 * 16^1 + 2 * 16^0 + 4 * 16^{-1} + 8 * 16^{-2} \\ 76.51O = 7 * 8^1 + 6 * 8^0 + 5 * 8^{-1} + 1 * 8^{-2} \end{cases} \quad (3.4)$$

一般的进制计算与均可用此公式:

$$(S_{n-1}S_{n-2} \dots S_0.S_{-1} \dots S_{-m})_K = S_{n-1} * K^{n-1} \dots + S_0 * K^0 + S_{-1} * K^{-1} \dots + S_{-m} * K^{-m} \quad (3.5)$$

3.2.3 进制转换

1. 非十进制数转十进制, 按照指数 (权) 展开求和即可.

2. 十进制转 k 进制, 整数与小数分别转换, 整数部分除 k 取余, 直到商为 0, 然后反序写出各位数, 小数部分乘 k 取整, 直到乘积为 0.0 或满足精度, 顺序写出取出的整数, 并在前面加上 0.

从上面的计算可以看出, 有限位其他进制数一定可以转为十进制数, 并且操作简单; 但十进制数不一定能正好转化为其他进制数, 并且转化过程相对复杂, 此处我们以二进制为例, 给出过程示范.

二进制与十六进制, 八进制之间的转换:

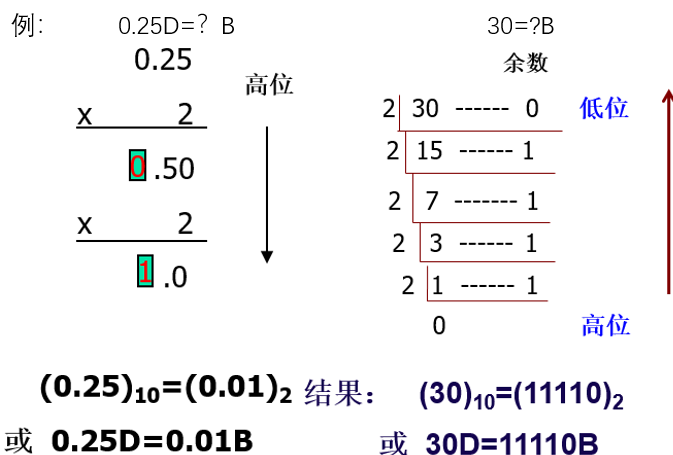


图 3.2: 二进制与十进制相互转换示意图

二进制 → 十六进制: 从个位数开始向左, 每四位算在一起, 转化为十六进制数, 最高位不足时用 0 补齐; 小数点后向右每四位算在一起, 转化为十六进制数, 将得到的数顺写即得相应的十六进制数.

$$(1001010.0110101)_2 = (4A.6A)_{16} \quad (3.6)$$

二进制 → 八进制: 将上述过程改成每三个数合并即可.

这样计算的原因就是 $2^4 = 16, 2^3 = 8$, 将三进制转化为九进制, 二十七进制也同理, 理解原理就可以.

3.3 二进制数的表示和运算

二进制数也有正负之分, 在计算机中, 正好用 0 表示, 负号用 1 表示.

3.3.1 数的表示

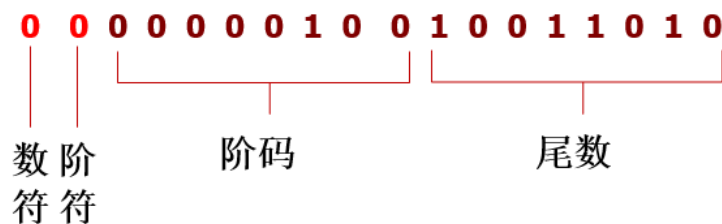
1. 定点表示, 即固定小数点位置, 通俗来讲, 就是明确小数点前有几位数, 小数点后有几位数, 带有很大的整数部分和很小的数部分的数在用这种方式存储时精度很容易受损.(定点数不考)
2. 浮点表示, 数中的小数点位置不定, 可采用阶码 E 和尾数 F 表示, 明显的优点就是表数范围大, 运算速度快, 准确.

表示形式: $x = \pm 2^E * F$

规格化浮点数: 尾数用纯小数, 阶码用整数表示.

例 3.3.1 将 $1001.101B$ 表示为浮点数.

规格化为 $0.1001101 * 2^4$, 阶码 4 的二进制表示为 100, 则:



溢出现象: 运算的数字超出字长时, 会产生溢出. (不考)

3.3.2 机器数的表示和运算

机器数: 计算机中存储和处理的二进制数. 分为无符号数和有符号数.

真值: 机器数表示的原本数据.

为方便起见, 以下示例均假设字长为 8.

机器数的三种表示方法: 原码, 反码, 补码.

原码

$X = +52$, 则 $[X]_{\text{原}} = 00110100$

$Y = -52$, 则 $[Y]_{\text{原}} = 10110100$

优点: 真值与其原码之间对应简单, 易于理解.

缺点: 计算机中原码加减运算较为困难, 0 的表示不唯一, 有正零和负零即 00000000 和 10000000.

表数范围: $[-127D, +127D]$

反码

对于一个数 X , X 非负, 则 $[X]_{\text{反}} = [X]_{\text{原}}$;

若 X 为负, 则 X 的原码除符号外, 全部取反即得反码.

例 3.3.2 若 $X = -52D = -0110100$, 则 $[X]_{\text{原}} = 10110100$, $[X]_{\text{反}} = 11001011$

例 3.3.3 已知机器数 X 的反码为 $[X]_{\text{反}} = 11111100$, 求其真值.

解

$$X = -0000011B = -3D$$

注意到 $[+0]_{\text{反}} = 00000000$, $[-0]_{\text{反}} = 11111111$, 0 的反码表示也是不唯一的, 引入反码概念只是为了方便下面引入补码概念.

表数范围: $[-127D, +127D]$

补码

对于一个数 X :

若 $x \geq 0$, $[X]_{\text{补}} = [X]_{\text{原}}$

若 $x < 0$, 则 $[X]_{\text{补}} = [X]_{\text{反}} + 1$

重点: 从机器数的补码还原其真值, 除符号位外, 其余各位按位取反后加一.

例 3.3.4 $[X]_{\text{补}} = 11111100$, 求 X 真值.

解 $[X]_{\text{真}} = -0000100B = -4D$.

$[+0]_{\text{补}} = 00000000$

$[-0]_{\text{补}} = 100000000 = 00000000(\text{溢出, 舍弃进位})$

例 3.3.5 机器数 10000000 在原码中为 -0 , 在反码中为 -127 , 在补码中为 -128 .

表数范围: $[-128D, +127D]$, 即 $[100000000, 011111111]$

运算规则:

$$[X + Y]_{\text{补}} = [X]_{\text{补}} + [Y]_{\text{补}} \quad (3.7)$$

$$[X - Y]_{\text{补}} = [X + (-Y)]_{\text{补}} = [X]_{\text{补}} + [-Y]_{\text{补}} \quad (3.8)$$

例 3.3.6 $X = +51, Y = +66$, 求 $[X - Y]_{\text{补}}$.

解 $[X]_{\text{补}} = [X]_{\text{原}} = 00110011$

$[-Y]_{\text{补}} = 10111110$

$[X - Y]_{\text{补}} = 11110001$

3.4 信息表示与编码 (非重点)

计算机中的信息有数值, 文字, 声音, 图像, 视频等多种形式, 但计算机能够直接识别的信息只有二进制, 将信息转化为二进制的过程即是编码.

需要掌握的仅仅是字符的编码即可, 多媒体等的编码不考.

3.4.1 西文字符编码

ASCII 码 (美国标准信息交换代码)

标准 ASCII 码: 7 位二进制码表示一个符号, 可表示 128 个字符, 最高位默认为 0.

扩展 ASCII 码: 八位表示一个字符.

重点: 大写字母与其对应的小写字母 ASCII 码之差为 32.

其他编码

Unicode 码: 通过字符编码, 每个字符用两个字节, 可表示 65536 个字符.

UTF-8 码

3.4.2 汉字编码

GB2313-80,BIG5 等

外码

输入码,(字音, 字形, 形音和数字编码).

机内码

汉字在计算机中的编码.

用于计算机之间或与终端之间信息交换时的汉字代码 (GB2312,GBK,GB18030). 由连续的两个字节组成, 每个字节七位有效, 最高位为 1.

机内码的构成:

机内码的高 8 位 = 区号 + 10100000

机内码的低 8 位 = 位号 + 10100000

例如: 汉字”啊“在 GB2312 汉字字符集中区位号为 1601 区号的二进制数:00010000 位号的二进制数:00000001 分别加上 10100000 后, 对应的机内码为:1011000010100001B, 即 B0A1H

输出编码

字形码; 确定一个汉字字形点阵的代码, 点阵中每个点对应一个二进制位.

矢量汉字: 保存为数学公式, 不变形.

第四章 计算机网络

编者: 朱启翔

网络部分重点在第一第二节, 局域网和网络安全一般考试不做要求, 了解即可. 要重点掌握网络拓扑结构,TCP/IP 协议,IP 地址,MAC 地址等等. 这一章考点, 考试考得都不会很难, 把知识点记清楚就行.

4.1 计算机网络基础知识

4.1.1 概述

计算机资源 (背诵 ★)

1. 硬件资源: 大量存储设备, 网络打印机等
2. 计算机资源: 超级计算机系统提供的大数据量, 高速运算能力
3. 软件资源: 各类软件
4. 数据资源: 各种统计数据等

4.1.2 计算机网络发展 (了解)

1. 以单片机为中心的联机系统:20 世纪 60 年代中期以前, 多台用户终端与中央计算机相连.
2. 分组交换网络的出现:20 世纪 60 年代中期开始, 共有两个演变过程, 第一个是仅仅将各联机系统的主机通过通信线路连接, 第二个是将数据通信任务分离, 交由通信处理机 (CCP), 主机仅作数据处理,CCP 组成的网络是通信子网, 主机组成的网络是资源子网
3. 网络体系结构标准化时代:20 世纪 70 年代以后,1983 年国际标准化组织推出 OSI 标准,TCP/IP 协议迅速普及
4. 因特网时代:20 世纪 90 年代

4.1.3 计算机网络分类

按覆盖区域分类 (背诵 ★)

1. 广域网

特点: 覆盖地理范围大 (几十千米到几千千米), 传输速率低, 传播延迟大, 适应大容量与突发性要求...

2. 局域网

特点: 覆盖地域小 (几米到几十千米), 传输速率高, 传播延迟小

3. 城域网

特点: 覆盖范围在广域网和局域网之间, 运行方式与局域网类似

按拓扑结构分类 (背诵 ★★★)

除了知道每一种拓扑结构的组成方式, 特点也要记忆, 第 5,6 考试一般不过多涉及.

1. **星型结构**: 优点: 易于构建, 扩充; 控制相对简单. 缺点: 对中心节点的可靠性要求比较高
2. **树形结构 (层次结构)**: 规模较大的网络一般都采用树形结构
3. **总线型结构**: 优点: 结构简单, 成本低. 缺点: 实时性较差
4. **环形结构**
5. **点到点部分连接的不规则形结构 (网状结构)**: 多用于广域网
6. **点对点的全互联结构**

4.1.4 TCP/IP 协议及其体系结构 (背诵 ★★★)

TCP/IP 协议有四个层次, 从上到下分别是应用层, 传输层, 网际层, 网络接口层.

1. **应用层**: 包含了很多面向应用的协议, 如简单邮件传输协议 (SMTP), 超文本传输协议 (HTTP), 文件传输协议 (FTP).
2. **传输层**: 两个主要传输协议: 无连接的用户数据报协议 (UDP, 不可靠传输协议), 面向连接的传输控制协议 (TCP, 可靠传输协议). 两者的主要区别在于, 前者只负责发送数据, 无需提前建立连接, 发后不管; 后者会建立连接, 与接受的主机进行多次反馈.
3. **网际层**: 又称互联网层或网络层. 为网络上不同主机提供通信服务, 及解决主机到主机的通信问题. 核心协议是 IP 协议.

4. 网络接口层: 网络接口层没有定义什么具体内容, 一般用 OSI 模型中的数据链路层和物理层代替.

4.1.5 网络应用模式 (背诵 ★)

这一部分熟悉 C/S 模式和 B/S 模式即可, P2P 模式了解即可.

- C/S 模式: 客户/服务器模式, 有客户端
- B/S 模式: 浏览器/服务器模式, 无专门的客户端, 以网络浏览器为客户端

4.2 因特网

4.2.1 因特网的结构与组成

边缘部分: 有所有连接在因特网上的主机组成, 每一个主机都有一个 IP 地址, 用于区别不同主机.

核心部分: 由大量的网络和这些网络的路由器组成. 路由器的功能是将接受到的分组按最佳路径转发到另一个合适的网络.

4.2.2 IP 地址和端口号 (背诵 ★★★)

这一节的主要重点在于 IP 地址, 端口号一般了解就行.

IPv4 地址

IPv4 地址用三十二位二进制编码表示. 为了便于记忆, 人们将 32 位 IP 地址分为四个字节, 用等效十进制代替. IP 地址有 A, B, C, D, E, F 五类, 常用的 IP 地址是 A, B, C 三类. A 类给大型网络用, B 类分配给中等规模的网络, C 类给规模较小的局域网.

- A 类: 网络号有七位, 可使用的网络号有 $126(2^7 - 2)$ 个, 网络号全为 0 的 IP 地址和网络号为 127(01111111) 不允许使用. 每个 A 类地址网络允许最大主机数为 $16\,777\,214(2^{24} - 2)$
- B 类: 网络号有 14 位. 规定 128.0.0.0 不能使用, 可用网络号有 $16\,383(2^{14} - 1)$ 个, 最大主机数 $65\,534(2^{16} - 2)$.
- C 类: 网络号有 21 位, 规定 192.0.0.0 不能使用, 可用网络号有 $2\,097\,151(2^{21} - 1)$ 个. 最大主机数 $254(2^8 - 2)$

此外还有一部分私有地址不可使用, 私有地址用于没有合法的 IP 地址的单位或家庭用户自己组建局域网.

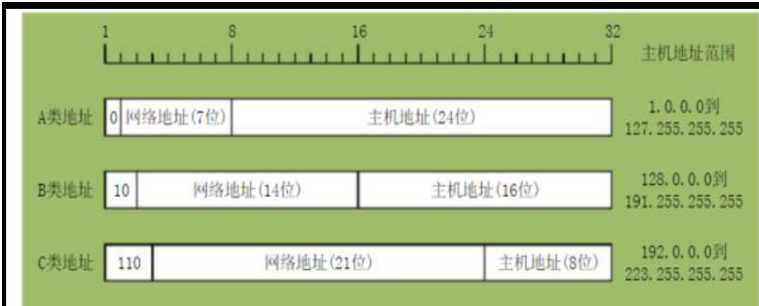


图 4.1: 三种 IP 地址示意图

以下形式的IP地址具有特殊的含义，不能作为主机的地址

全 0	全 0	本网络中的本主机
全 0	主机 号	本网络中的主机
全 1	全 1	局域网中的广播
网络号	全 1	对指定网络的广播
网络号	全 0	网络地址

图 4.2: 不可用的 IP 地址示意图

10.0.0.0 10.255.255.255(1 个 A 类地址块)
172.16.0.0 172.31.255.255(15 个 B 类地址块)
192.168.0.0 192.168.255.255(1 个 C 类地址块)

端口号

端口号是用来标识同一主机中不同进程的机制. 如果把 IP 地址看作港口的地址, 端口号就时港口中泊位的编号. 端口号是传输层和应用层之间数据交换的一种机制.

一些常见的 TCP 熟知的端口号所对应的应用层协议 (了解)

- 1. 21 FTP(文本传输协议)
- 2. 23 Talnet(远程登录)
- 3. 25 SMTP(邮件传输协议)
- 4. 80 HTTP(超文本传输协议)
- 5. 110 POP3(邮局协议)

4.2.3 子网和子网掩码 (应用 ★★★★★)

为了更有效的利用 IPv4 的地址空间, 将主机号的一定位数划分出来作为网络号的一部分, 划分出来的就是子网号. 这样,IP 地址就变成了网络-子网-主机三个部分组成

下面用一个例子演示如何划分, 有一个 C 类网络 202.117.58.0 划分为四个子网, 那么需要从主机号中取出两位作为子网号.

子网 1:202.117.8.00000000

子网 2:202.117.8.01000000

子网 3:202.117.8.10000000

子网 4:202.117.8.11000000

子网掩码

子网掩码是用来识别各个子网的, 也是一个 32 位二进制数, IP 地址的网络号和子网号部分, 子网掩码对应是 1, 对应于主机号部分, 子网掩码中相应为 0. 上一个例子中子网掩码就是 255.255.255.192(11000000)

要得到子网地址, 只需要把 IP 地址和子网掩码进行 ‘与’ 运算即可

4.2.4 域名地址和 MAC 地址

域名地址

IP 地址不便于记忆, 为了方便用户记忆和使用, 域名与 IP 地址进行了绑定. 例如西安交通大学的服务器域名是 www.xjtu.edu.cn, IP 地址为 202.117.0.13.

DNS 系统由域名空间, 域名服务器和解析程序组成. 域名服务器中存放着主机的域名与 IP 地址之间的对应关系, 因特网中有许多域名服务器, 分布在世界各地. 同时, 为了减少域名查找的开销, 每个域的域名服务器 (包括本地主机) 都会有一个 DNS 缓存, 将相关域名的查询记录保存一段时间.

MAC 地址

MAC 地址是固化在网络接口硬件中的地址, 也称物理地址或硬件地址, 用于标识一个主机的网络接口.

MAC 地址是一个 48 位的二进制编码, 高 24 位是设备生产厂商的编码, 后 24 位是产品序列号.

MAC 地址在数据链路层处理, IP 地址在网络层处理.

MAC 地址相当于身份证号, IP 地址相当于居住地址. 二者不存在绑定关系.

4.2.5 练习题

1. 局域网具有的几种典型的拓扑结构中一般不含

- (a) 星型
- (b) 环型
- (c) 总线型

- (d) 网状结构
- 2. 面向连接的数据传输过程包括三个阶段, 它们是
 - (a) 数据传输, 接收确认和关闭连接.
 - (b) 建立连接, 数据传输和关闭连接.
 - (c) 请求传输, 数据传输和接收确认.
 - (d) 联系好友, 开始聊天, 说“拜拜”
- 3. 如果一台主机从一个网络移到另一个网络, 则
 - (a) 需要修改它的域名, 但 IP 地址无需改变.
 - (b) 需要修改它的 IP 地址和域名.
 - (c) 需要修改它的 IP 地址, 但域名无需改变.
 - (d) 它的 IP 地址和域名都不需要改变
- 4. 关于因特网的传输层, 以下哪种说法是错误的?
 - (a) 传输层提供的数据传输服务是可信赖的.
 - (b) 传输层提供的数据传输服务不一定是可信赖的.
 - (c) 传输层建立一个 TCP 连接需要经过“三次握手”过程.
 - (d) 传输层提供了因特网中应用进程之间的数据传输服务
- 5. SMTP 是用于 ____ 的协议.
 - (a) 文件传输
 - (b) 网络管理
 - (c) 电子邮箱服务
 - (d) 电子邮件传输
- 6. 以下 C 类地址中, 哪个允许作为因特网中主机的 IP 地址?
 - (a) 202.117.35.255
 - (b) 202.117.35.200
 - (c) 202.117.35.0
 - (d) 192.168.1.20
- 7. TCP/IP 网络参考模型的层次包括.
 - (a) 应用层, 表示层, 网络层和网络接口层

- (b) 应用层, 传输层, 网络层和网络接口层
 - (c) 应用层, 网络层, 数据层和物理层
 - (d) 表示层, 传输层, 网络层和物理层
8. 假定某 FTP 服务器的域名为 x.y.z,IP 地址为 202.117.1.100. 使用浏览器访问该 FTP 服务器时, 以下哪一个 URL 是正确的?
- (a) FTP://x.y.z
 - (b) FTP://x.y.z:20
 - (c) HTTP://202.117.1.100
 - (d) FTP://202.117.1.100:20
9. 以下哪种说法是错误的?
- (a) 计算机网络可以仅包含两台计算机, 也可以包含成千上万台计算机.
 - (b) 除了主机外, 一个计算机网络也可以包含另外的计算机网络.
 - (c) 计算机网络不能由另外一些计算机网络所构成.
 - (d) 计算机网络提供的共享资源包括硬件资源, 软件资源, 计算资源和数据资源
10. 以下哪些是因特网的特征?(多选)
- (a) 由大量的网络互联而成
 - (b) 是一个分组交换网
 - (c) 采用了 TCP/IP 协议
 - (d) 是一个全球性的网络
11. DNS 系统用于以下哪项任务?
- (a) 将 IP 地址转换为 MAC 地址
 - (b) 将域名转换为 MAC 地址
 - (c) 将 IP 地址转换为 MAC 地址
 - (d) 将域名转换为 IP 地址
12. 以下 IP 地址中 ____ 和 ____ 属于同一子网 (子网掩码 255.255.192.0)
- (a) 150.20.115.133
 - (b) 150.20.190.2
 - (c) 150.20.192.59

(d) 150.20.215.133

13. 以下哪种说法是正确的?

- (a) MAC 地址在 TCP/IP 体系结构的网络接口层进行处理.
- (b) IP 地址在 TCP/IP 体系结构的网络接口层进行处理.
- (c) 没有域名系统, 因特网就无法工作.
- (d) MAC 地址在网络的物理层进行处理.

14. 以下哪种说法是错误的?(多选)

- (a) 一个计算机网络中, 各主机所具有的协议层次个数可以不同.
- (b) 网络中实际的物理通信在网络体系结构的最底层实现.
- (c) 网络体系结构的分层结构只是用于理论研究.
- (d) 网络体系结构的层次数越多, 网络功能就越强大.

参考答案 D B C A D B B A C (ABCD) D (CD) A (ACD)

解析 第一题, 网状结构一般用于广域网. 第四题, 传输层有两种协议, 一种是可靠传输协议, 一种是不可靠传输协议. 不可靠传输协议有出错的可能. 第六题, A 是对于指定网络的广播, C 是网络地址, D 是私有地址. 第八题, 题目中给出了域名和 IP 地址, 如果使用 IP 地址, 那么不需要任何前面的协议前缀.

第五章 C 语言语法基础

编者: 唐智亿

5.1 序与概述

学习完电脑基本知识, 网络等章节后, 大学计算机真正包含语言和实际操作的部分将在本节展开. 如笔者所说的“实际操作”, 本部分的内容光凭认真听课和记背作用不大, 真正能够掌握的方法只有不停地尝试, 出错和总结.

笔者的老师是本书的主编吴老师, 然而她本人并不是完全按照书上的内容进行讲解的, 结合 SPOC 和线下课的补充, 吴老师对课本内容做了极大的延伸. 笔者不才, 也不是特别认真的听课者, 只能结合自己的体会进行总结. 希望对之后学习的同学有所帮助.

本课所介绍的语言是 C 语言, 也是当今大家公认的最适合入门的一门语言之一. 与如今火爆的 Python 不同, 这是一门几乎未经封装, 语法复杂但较易懂的语言. 通过学习它的基本语法知识, 进行简单的编程操作, 学习者可以从计算机编译, 执行程序的原理的角度更好地理解计算机的工作原理. 本章对应课本得第 5-6 章, 但是有部分内容属于不讲解内容 (或不考核内容, 由于老师们讲的不一致, 以最后考核的为准), 在此说明:

5.5.3 类型修饰符

5.6.7 位运算

6.1.2 二维数组

6.1.3 多维数组

6.5.7 带参数的 main 函数

6.6 变量的储存类别

6.10.2 指向多维数组的指针

6.11.3 指向函数的指针

6.13 指向指针的指针

6.16 void 和 const 类型的指针

6.18 预处理命令

C 语言的部分最终考核分为平时作业 (平台上从第 5 周到 12/13 周完成的编程题), 上机期末考试, 其中期末考试占超过 70%.(笔者所在的班级有两次讨论课, 占一定比例) 考核方式遍布选择, 填空, 判断和编程大题, 具体分布是由系统题库抽题组成, 每套题难度一致, 但是侧重点可能有所不同.

由于 C 语言小知识点较多, 笔者难免会出现手误, 如果发现请及时向笔者所在的仲

英学业辅导中心反馈. 另外, 如果某道题目有其他好方法, 也可以通过其他途径直接联系笔者, 随时欢迎.

5.2 C 语言的基本知识

5.2.1 C 语言程序的基本结构

让我们从这段代码开始:

```
1 // 在屏幕上显示: Hello World!  
2 #include<stdio.h>  
3 int main()  
4 {  
5     printf("Hello World!\n");  
6     return 0;  
7 }
```

如果按照课本打出第一个程序 “Hello World” 的代码, 会发现代码中有通俗易懂的字符, 也有些并不明白的符号语句. 这个简单的例子恰好可以反映出 C 语言的基本结构, 包括预处理器指令, 函数, 变量 (这个例子里还没有出现), 语句 & 表达式以及注释.

第一行在程序中有特殊的颜色 (一般是绿色), 它称为注释, 内容不计入程序运行的部分, 即会被编译器忽略. 上例的注释由 “//” 和后面的解释说明接在一起, 但是换行后, 文字变为白色, 所以 “//” 的作用范围为单行. 还有一种注释由 “/*” 和 “/” 组成, 由一对 “/*” 和 “*/” 构成, 从 “/*” 开始到 “*/” 结束都是注释内容, 形式如下:

```
1 /* 这是一段注释  
2    换行结束 */
```

注释的一般作用是说明某个语句, 试想在几千行代码中, 程序编写着不可能随时都记着每一句的作用或者实现的能力, 所以必要的注释有利于程序编写.

“#include<stdio.h>” 是一种声明文件包含的预处理命令, 告诉 C 编译器在实际编译之前要包含 stdio.h 文件. 除此之外, 有很多函数的调用, 常量的声明也会以预处理命令的形式写在开头, 这个将写在后面的部分里.

C 语言中的函数和数学中的函数概念不同但都是代表某种功能或者特定算法的实现. 即使是最简单的程序, 也至少包含一个 main 函数, C 语言的程序就从 main 函数开始执行 (这个概念曾经考过选择题). main 前面的 int 说明了该函数返回值类型, 与后面的 “return” (在英文里大家也知道是返回的意思) 对应, “return 0” 表明该主函数接收到的返回值是 0.

除此之外, 函数还包括系统自带的库函数和用户自己定义的函数. 上例中的 main 函数包含了两条语句, 其中 printf 也是系统自带的一种函数, 称为输出函数, 表示显示后面的内容, 由两部分组成:

```
1 printf("%d",a);
```

与输出函数相对应的还有输入函数, 结构相似, 但有一点不同, 经常成为初学者报错的点, 一定要注意:

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

常见的输入输出格式如表 5.1 所示:

表 5.1: 常见输入输出格式一览表		
标识符	数据类型	意义
%lf	double	双精度实数
%f	float	单精度实数
%c	char	字符
%s	char[]	字符串
%g	float, double	实数最简格式, 不会输出无意义的 0

在基本结构的背后, 大家或许会好奇, 有一个统一的, 看起来非常没有用的符号不停地出现——“;”, 其实, 它代表的是一个语句的结束. 忘记打出的后果就是之后运行中出现报错.

5.3 代码编译与运行

我们知道计算机能识别的只有由“0”和“1”组成的机器语言, 稍微高级一点的汇编语言和正在学习的 C 语言为代表的高级语言都是计算机无法直接识别的, 因此后两者需要经过编译, 转化为电脑可以识别的“文字”.

编译结束后, 程序可以被执行.VS 有多种方式运行程序, 菜单栏直接有“本地 Windows 调试器”, 点击即可编译运行, 快捷键是 F5; 在菜单中选择调试-> 开始执行 (不调试) 也可以编译运行, 快捷键 Ctrl+F5. 有些情况下, 直接 F5 可能出现程序结果一秒闪退, 可以用后一种方法. 除此之外, 还有很多方便查看程序语法, 逻辑错误或者确定运行报错原因的方法, 编程者总会遇到程序报错, 有可能是编写时的逻辑错误或者敲代码时的手误, 有些时候能很快检查出来, 有些时候就需要其他手段帮助.F10 是一步一步执行但不进入用户自编写的函数, 而 F11 逐句执行且进入子函数. 通过 F10 和 F11 的运行, 设置查看中间变量 (运行中选择窗口设置), 可以轻松地找出思路中的漏洞或者程序中的问题.

5.4 C 语言的其它基本要素

5.4.1 标识符

C 标识符是用来标识变量, 函数, 或任何其他用户自定义项目的名称. 一个标识符以大写字母 A-Z 或小写字母 a-z 或下划线 “_” 开始, 后跟零个或多个字母, 下划线和数字 (0-9).

C 标识符内不允许出现特殊标点字符, 比如 “&”, “*”. 另外, C 是区分大小写的编程语言. 因此, 在 C 中, Teacher 和 teacher 是两个不同的标识符. 以数字开头, 出现非法字符或出现 “-”(减号) 都是不允许的.

5.4.2 空格

只包含空格的行, 被称为空白行, 会被编译器直接忽略.

一般情况下, 空格用于描述空白符, 制表符, 换行符和注释. 空格分隔语句的各个部分, 让编译器能识别语句中的某个元素 (比如 int) 在哪里结束, 下一个元素在哪里开始. 比如后面学到的变量定义: `int a=2;` “int” 和 “a” 之间的空格就是分隔两个元素的作用, 如果连在一起: `inta=2;` 就会报错.

5.4.3 关键字

关键字是 C 语言中的保留字, 代表一些特殊的含义, 不能被定义为标识符. C 语言中的常用关键字如表 5.2 所示.

表 5.2: C 语言中的常用关键字表

关键字	含义	关键字	含义
auto	自动变量的声明	int	整型变量的声明
break	跳出循环	long	长整型变量的声明
case	用于 switch 语句的分支	register	寄存器变量的声明
char	字符型变量的声明	return	子程序返回语句
const	只读变量的声明	short	短整型变量的声明
continue	结束循环进入下一个	sizeof	计算所占字节数的运算符
default	用于 switch 语句分支	static	静态变量的声明
do	用于循环体 do-while	struct	声明结构体类型
double	双精度浮点型变量	switch	开关语句
else	与条件语句 if 连用	typedef	命名新的数据类型
float	浮点型变量	unsigned	无符号类型变量的声明
for	循环语句的标志	void	空类型, 代表无返回值/无参数等
if	条件语句的标志	while	用于循环语句

5.5 常见数据类型

C 语言的数据形式有两种: 变量和常量, 使用之前需要先声明。

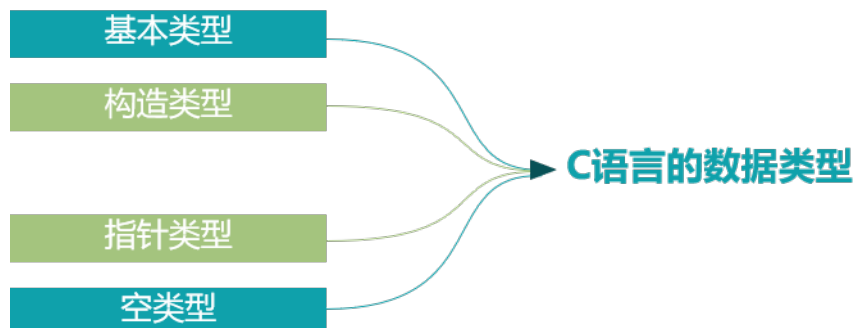


图 5.1: 数据类型分类示意图

基本类型包括: 数值类型和字符类型。字符型数据占 1 个字节, 8 个二进制位, 其中第 7 位是符号位 (从第 0 位开始), 数值范围是 $-2^7 \sim 2^7 - 1$; 短整型占 2 个字节, 表示范围 $-2^{15} \sim 2^{15} - 1$; 整型和长整型占 4 个字节, 表示范围为 $-2^{31} \sim 2^{31} - 1$; 浮点类型占 4 个字节, 一般 7 位有效数字, 表示范围约为 $-3.4 \times 10^{-38} \sim 3.4 \times 10^{38}$; 双精度浮点类型占用 8 个字节, 范围约为 $-1.7 \times 10^{-308} \sim 1.7 \times 10^{308}$ 。

5.5.1 常量

字面量类型与表示

字面量是一种特殊的常量, 指“就是字面意思”的量。主要包括整数, 字符常量, 字符串常量等。例如以下都是字面量:

1, 2.56, 'a', "Hello, world!"

整型: 整型常量就是整数。在 C 语言中允许将整数表示为十进制, 八进制和十六进制。十进制没有前缀, 八进制以 0 (零) 开头作为前缀, 十六进制以 '0X' (零 X) 或 '0x' (零 x) 开头为前缀 (10, 11, ..., 16 分别表示为 A, B, ..., F)。如:

155=0233=0X9B

实型 (浮点型): 实数直接使用小数方式表示。特别地, 没有小数部分的量也应该加上 .0, 否则视为整型。实数可以使用科学计数法表示, 如 2.1×10^{-5} 可以表示成 2.1e-5。

字符型: 表示字符时, 可以用单引号 + 这个字符本身表示字符。如 'a' 表示字符 a, ASCII 码为 97; '2' 表示字符 2 (不是数字 2), ASCII 码为 50。有一些反斜杠开头的字符有特殊的含义, 称为转义字符。C 语言中的常用转义字符如表 5.3 所示, 其中必须熟练掌握 `\n`, `\0`, `\'`, `\"` 和 `\\`; 其它转义字符的使用在本课程中基本不出现, 也基本不会考。

表 5.3: C 语言中的常用转义字符表

字符	意义	字符	意义
<code>\n</code>	换行 (光标移到下一行同一列) ¹	<code>\0</code>	零字符 (ASCII 码为 0) ²
<code>\r</code>	回车 (光标移到当前行第一列)	<code>\'</code>	单引号'
<code>\b</code>	退格 (光标移到当前行上一列)	<code>\"</code>	双引号"
<code>\t</code>	制表符 (键盘上的 tab)	<code>\\</code>	反斜线\
<code>\f</code>	换页	<code>\ddd</code>	ASCII 码为八进制数 <i>ddd</i> 的字符
<code>\a</code>	警告 (发出提示音)	<code>\xhh</code>	ASCII 码为十六进制数 <i>xhh</i> 的字符

字符串 在 C 语言中, 字符串以 **字符数组** 的形式储存和使用. 非常特殊的是, 在储存时, 字符串的末尾会自动添加一个 `\0` 表示字符串的结束. 这个字符不会在输出时被显示, 使用 `strlen` 计算字符串长度时不会被计算, 但是会占数组中的一位. 因此定义字符数组时其长度最好大于实际被使用的长度, 防止这一位 0 污染未被使用的内存.

符号常量和常变量

符号常量的定义与使用方法是

```
1 #include<stdio.h>
2 #define MAX 10000
3 // 定义了符号常量MAX,注意结尾没有分号
4 int main()
5 {
6     int a=MAX;
7     // 像变量一样使用符号常量,可以赋值,当然也可以参与表达式
8     printf("%d",a);
9     return 0;
10 }
```

这里要特别声明 4 点:

- 1. 符号常量的定义不是语句, 结尾没有分号. 它本身不是变量, 不会占用内存. 这种 `#` 开头的行 (包括前面见过的 `#include<...>`) 被称为“预处理命令”, 在编译之前就会被逐条处理, 感兴趣的读者可以自己查询相关书籍.
- 2. 符号常量绝对不是规定一个量的值, 编译器对它的处理方法 (以上面的定义为例) 是在编译之前直接查找下文中出现的所有 `MAX` (一如既往地区分大小写) 然后替换为 `100000`, 最后把 `define` 语句删掉.

¹我们在 Windows 系统中编程时使用的 `\n` 的意思和它实际的意思完全不相同, 变为了“光标移动到下一行第一列”, 即 `\n\r`. 这与一个名为“CRLF”的规则有关, 感兴趣的读者可自行查阅相关资料.

²整数 0 与字符 `'\0'` 区别不大, 和字符 `'0'` (ASCII 码 48) 完全不同.

3. 符号常量的取值不一定是数字, 可以是一串任意字符 (不能带空格和换行), 都会按 2 的规则进行替换. 如以下程序:

```
1 #include<stdio.h>
2 #define FINISH printf("program ended.");
3 int main()
4 {
5     int a;
6     scanf("%d",&a);
7     printf("%d\n",a);
8     FINISH
9 }
```

运行结果为 (加粗表示输入)

125

125

program ended.

这表明, 第 8 行的 FINISH 被原样替换为了 `printf("program ended.");` (分号也被换进来了, 这也是为什么前面说不能加分号). 特别地, 标识符, 关键字等其它元素或它们的一部分如果被定义成符号常量的话, 在编译前也会被替换 (显然这会破坏你的程序), 因此使用符号常量要特别小心**命名冲突**, 它可能造成非常隐晦但致命的错误.

4. 按照 C 语言语法, 符号常量可以任意命名 (当然不能有空格和换行), 但按照编程规范, 符号常量的命名一般使用**全大写** (原因见 4, 最大程度避免命名冲突). 不要使用下划线开头命名符号常量, 这种符号常量一般是头文件里定义的特殊常量.

常变量是一种特殊的变量, 声明方式是在变量类型之前或之后加关键字 `const`. 例如以下程序:

```
1 #include<stdio.h>
2 int main()
3 {
4     int const a=2; // 定义整型常变量 a
5     printf("%d",a); // 常变量就是变量, 可以按变量的方式使用
6     a=3;           // 编译器在这一行报错, 常变量的值不能改变.
7     return 0;
8 }
```

常变量也是变量, 可以把它当成变量任意使用. 理所当然的, 常变量和同类型普通变量占有相同的内存. 唯一的不同是常变量**必须赋初值**, 之后值不能改变.

当题目中没有明确要求时, 建议尽量不要使用常量 (驾驭不住容易犯错).

5.5.2 变量

变量的定义

定义变量的格式为变量类型 + 变量名 (+ 赋初值). 例如

```
1 int j; // 定义了整型变量j
2 double var1 = 0; // 定义了双精度变量var1并赋初值0
3 char c1, c2='a', c3=0, c4, c5; // 可以同时定义多个变量, 逗号隔开
```

定义指针变量时, 必须在每个变量之前都加上指针标志 “*” 以声明这个变量是指针. 例如如下两个语句:

```
1 char* p, q;
2 char *a, *b;
```

字符指针的类型是 `char*`, 但使用 `char*` 定义变量的第一行中, `p` 被定义成了字符指针而 `q` 被定义成了字符变量; 第二个语句使用 `char` 作为类型名, 在每个变量之前都加了指针符号 “*”, 定义的 `a`, `b` 都被定义为了字符指针.

typedef 命令

`typedef` 命令的作用是为已有的类型声明新的名字. 格式是 `typedef + 已有类型 + 新的名字`. 如

```
1 typedef double Real; // 给double起了别名real
```

5.6 运算符与表达式

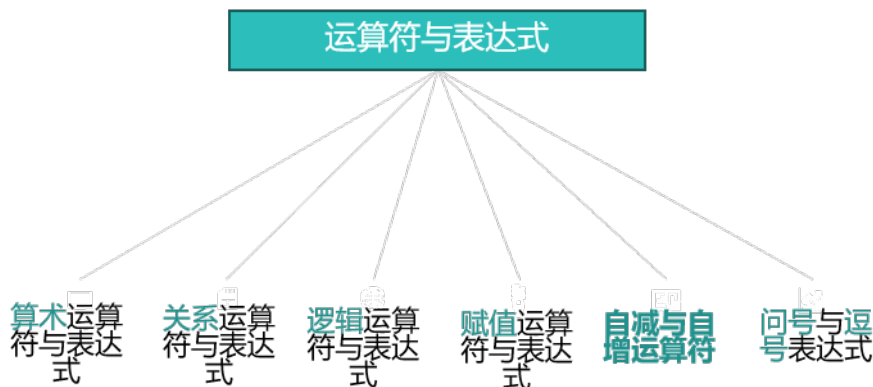


图 5.2: 运算符与表达式分类关系图

注意:

1. 算术运算符的优先级高于关系运算符;
2. 关系运算符的优先级高于逻辑运算符, 逻辑运算符中 “&&” 先于 “||” 的优先级;
3. 绝对不允许连续不等号, 例如 $i < j \leq k$ 这样的结构绝对不允许, 它会
4. 自增自减运算优先级较高, 高于算术和逻辑运算符;
5. 赋值本身也是表达式, 是有值的. 例如 $i = j * k$, 会将 $j * k$ 赋值给 i , 整个表达式的值为 $j * k$. 所以会有经典的字符串读取结构 `while((c=getchar())!='\n')` (正是利用了赋值表达式的值).
6. 赋值运算符 “=” 与等于 “==” 是完全不一样的. 这个错误编译器不会报错, 可能会按完全不同的方式解释, 千万要小心.(曾经作为考点, 新手编程必过之坑)
7. 赋值表达式 $i = j = m * n$ 会被这样解释: 首先计算表达式 $j = m * n$ 的值 (赋值被作为表达式处理, 值就是赋值等号右边的值), 再把得到的值赋给 i . 这个表达式被执行后, i, j 都会被赋值 $m * n$, 整个表达式的值为 $m * n$.
8. 自减与自增运算符有两种对应的表达式:
以自增为例: $++i$ 与 $i++$ 单独使用没有区别 (比如在循环结束需要给 $i+1$, 两者等价), 但在逻辑判断中, 前者表示的其实是 i (加法前的值), 而后者是 $i+1$ (加法执行后的值)(考点);
9. 三目运算符的形式为 $A ? B : C$, 值为 B (如果 A 非 0) 或 C (如果 A 为 0). 相当于

```
1 if(condition_A) value_B;  
2 else value_C;
```
10. 不同数据类型的混合运算情况下, 级别低的会转化成级别高的, 从高到低分别是: `double`, `float`, `long`, `int`, `char`;(考点)
11. 数据类型可以通过 “(数据类型)+ 表达式” 强制转换, 用来防止丢失小数部分.

5.7 控制结构

5.7.1 顺序结构

将两个语句顺序排列即可.

5.7.2 选择结构

选择结构的格式是:

```
1 if(<表达式1>)
2     <表达式2>;
3 else
4     <表达式3>;
```

或者

```
1 if(<表达式1>)
2     <表达式2>;
3 else if(<表达式3>)
4     <表达式4>;
5 else
6     <表达式5>;
```

通常如果只有两种情况, 该格式可以缩写成 (建议不要省略 “{}”, 使用它可以减少出错率, 便于检查):

```
1 if(<表达式1>)
2 {
3     <表达式2>;
4 }
5 <表达式3>;
```

5.7.3 循环结构

while 循环

while 循环有两种, 一种称为当型循环结构 (即 while), 一种称为直到型循环结构 (即 do-while), 两种区别不大, 但是要注意, do-while 结尾 while 后面有 “;”. (曾经出现过考点) do-while 型循环的结构如下:

```
1 do
2 {
3     <循环体>;
4 }
5 while(<表达式1>;)
```

while 型循环的结构如下:

```
1 while(<表达式1>)
```

```
2 {  
3     <循环体>;  
4 }
```

for 循环

for 比较灵活, 在数组的输入输出, 一般终止条件简单的情况下使用较多, 与 while 比较相似, 还可以方便地指定循环次数. 其结构如下:

```
1 for(<表达式1>; <表达式2>; <表达式3>;)  
2 {  
3     <循环体>;  
4 }
```

5.7.4 switch-case 语句

switch-case 语句的结构

```
1 switch(<整型表达式>)  
2 {  
3     case <数值1>:  
4     case <数值2>:  
5     case <数值3>:  
6     ...  
7     default:  
8     ...  
9 }
```

其中 default 可以省略. 如果需要在某种情况下跳出循环, 加上 “break” 即可, 每种 case 的情况数值不能相同.

5.7.5 break-continue 语句

break 可以跳出循环, 结束整个循环过程, 而 continue 仅结束本次循环, 不终止整个过程.

5.8 数组

5.8.1 数组的基本概念

数组用来存储一个固定大小的相同类型元素的顺序集合, 也遵循“先定义, 后使用”的原则, 定义后内存分配连续多个存储单元储存数组中的元素. 数组分为一维数组, 二维数组和 multidimensional 数组.

5.8.2 一维数组

一维数组用于存放一行或一列数据, 定义方法如下:

```
1 <类型名><数组名>[<常量表达式>;
```

数组的下标是从 0 开始的, 即 `a[0]` 是第一个元素 (考点), 各元素通过下标区分, 下标也同时确定了元素的位置.

声明数组也可以同时对其初始化, 如:

```
1 int a[]={1,2,3,4,5};
```

但数组不能整体赋值, 只能对单个元素赋值, 为数组元素键入值一般采用循环结构:

```
1 int a[5]; //5是数组长度
2 for(int i=0;i<5;i++)
3 {
4     scanf("%d",&a[i]);
5 }
```

5.8.3 二维数组

二维数组类似于矩阵, 有行数和列数, 基本形式是:

```
1 <类型名><数组名>[<常量表达式1><常量表达式2>;
```

第一个元素是左上角的 `a[0][0]`, 以行次序优先进行内存分配.

5.8.4 字符串

字符串的输入仍然可以使用 `scanf` 函数, 但在 VS 里有一种特殊的函数 `gets` 函数也可以用来输入字符串, 二者的区别是, `scanf` 不能输入包含空格的字符串而 `gets` 可以. 输出可以使用 `printf` 直接输出.

字符串有许多系统自带的库函数, 常用的有: `strlen`(求字符串长度), `strcpy`(字符串复制), `strcat`(字符串连接), `strrev`(字符串反转), `strcmp`(字符串比较) 等.

上述字符串系统自带的函数需要添加头文件 `# include<string.h>` 才可以使用, 否则因为系统无法识别而报错.(曾经考过)

5.9 结构体

5.9.1 结构体定义与成员引用

结构体定义方法如下

```
1 struct <结构体类型名>
2 {
3     <结构体类型成员变量说明语句表>
4 };
```

引用其中的成员的方法:

```
1 <结构体变量名>.<结构体成员变量名>
```

5.10 函数

5.10.1 函数的定义方式

函数的定义方式如下:

```
1 <函数值类型标识符> 函数名 (<形式参数表>)
2 {
3     <函数体>
4 }
```

函数名是编程者自己命名的, 也应该符合规范; 类型标识符与函数有无返回值有关, 有返回值的函数一般最后要加入:

```
1 return <表达式>;
```

一是释放储存空间, 二是传送函数值. 返回值类型要与函数类型一致, 否则会出错. 有些函数没有返回值, 如 `void` 类型, 此时不应出现 `return` 语句.

形式参数是连接函数和外界的窗口, 必须指明类型和名字; 函数体是一个分程序, 定义的变量只能在该函数中使用.

5.10.2 函数调用

函数中的形式参数要与外界联系, 就需要承接的实参, 它们之间应当是一一对应的关系. 调用程序中的函数一般是, 程序运行到另一个函数的位置, 停止当前函数的运行进

程, 保存下一条指令的地址, 从子函数的入口进入, 遇到 return 返回保存的地址继续执行.

编写及调用函数一般有两种编程方法: 第一种, 在 main 函数之前声明函数原型, 在主程序中调用, 在主程序后写出函数名和函数体; 第二种, 在主程序前写出函数, 后续直接使用. 一般后者较不容易出错.

5.10.3 全局变量与局部变量

变量在程序中的有限适用范围——作用域是不同的, 这样就划分出全局变量和局部变量两种. 函数内部的变量称为局部变量, 它的作用域仅限于函数内部, 离开该函数就无法继续使用. main 函数中定义的变量也只能在 main 函数中使用, 在其他函数中使用就会报错.

一般情况下, 允许在不同的函数中使用相同的变量名, 因为作用范围不同, 不会发生冲突. 形参变量也是一种局部变量. 实参给形参传值的过程也就是给局部变量赋值的过程.

全局变量因为供所有函数使用, 颇为方便, 初学者喜欢使用, 事实上这种变量的滥用会破坏程序的模块化, 占用内存空间, 不利于调试.

```
1 #include<stdio.h>
2 int a; //全局变量a
3 int fx(int a) //形式参数a
4 {
5     ...
6 }
7 int main()
8 {
9     a=0; //全局变量的赋值
10    ...
11    return 0;
12 }
```

5.10.4 递归函数

定义一个函数, 如果它调用了自身, 就称为递归函数, 很多数学问题用递归函数可以解决. 但在后面也会发现, 递归函数的复杂度是相当高的, 这就意味着, 当递归次数太大时, 电脑就会因为无法处理而崩溃.

下面是一个例子:

```
1 #include<stdio.h>
2 int fac(int n)
```

```
3 {
4     if (n < 0)
5         return -1;
6     else if (n == 0)
7         return 1;
8     else
9         return(n *fac(n - 1));
10 }
11 int main()
12 {
13     int n;
14     printf("请输入一个整数:");
15     scanf("%d", &n);
16     int a;
17     a = fac(n);
18     printf("%d!=" ,n);
19     printf("%d", a);
20     return 0;
21 }
```

运行该程序, 当输入一个较大的数字, 程序输出了“0”, 递归次数太多, 电脑无法处理得出值.

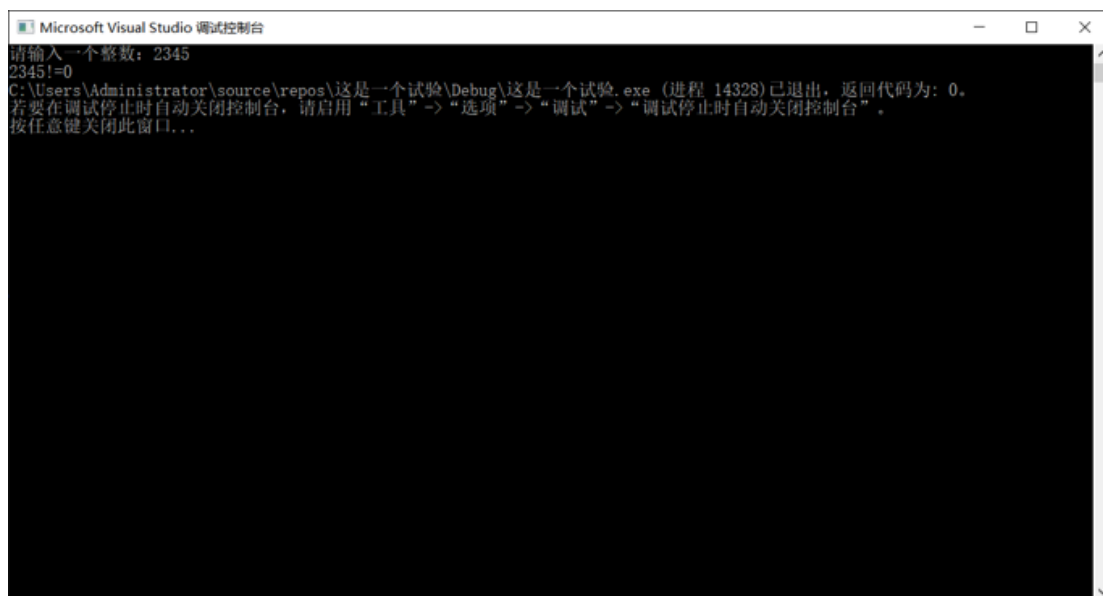


图 5.3: 递归过深导致爆栈崩溃

5.10.5 库函数

相信 `math`, `string`, `stdlib`, `time` 等已经成为大家常用的头文件了. C 语言还有许多功能多样的库函数, 相信在未来深入了解后, 大家会对库函数的使用越来越娴熟的.

5.11 指针

指针是一些同学和老师公认的最复杂最难理解的一部分, 事实上, 因为指针是 C 语言和 C++ 语言特有的一类方便操作的产物, 理解它的原理距离熟练使用它就不远了. 指针是内存地址的代言人, 这意味着指针的操作就是对地址直接调配操作.

5.11.1 指针基础知识

地址就是每一个内存单元的编号, 在计算机基础知识里有关于内存的详细介绍, 在这里不过多提及. 通过 C 语言, 可以知道地址的编码, 通过变量前添加 “&” 就可以知道地址编码. C 语言还规定一个数组的地址就是它的第一个元素的地址, 用数组名直接表示 (没有 “&”), 函数地址也直接用函数名.

“*” 是指针符号, 用以返回指向的变量的值, 表示如下:

```
1 数据类型 *指针变量名;
```

指针有不同的类型. 以整型指针为例, 该指针是将它指向的地址所代表的字节和其后的三个字节 (连续的 4 个字节) 作为一个数据单元处理的. 指针除了可以指向各种单一数值变量, 还可以指向数组, 函数, 甚至指针.

指针的实际操作中初始化类似于给数组赋值 0 的操作:

```
1  int *p=NULL;
```

5.11.2 指针初始化与运算

如何给让指针变量指向某一个已知变量是一个容易犯错的点, 主要有以下两种方法, 第一种:

```
1  int a=10;
2  int *p;
3  p=&a;
```

第二种:

```
1  int *p=a;
2  a=10;
```

这两种方法都可以让指针变量 p 指向 a.

指针变量简便的作用在这种情况下并没有很好地体现出来, 下面举一个经典的例子, 也是课本上, 老师讲解中重要的例子:(曾经考过)

例 5.11.1 交换两个变量的值.

这个例子大家都听说过, 之前看似正确的函数没有成功的将形参中的改变传递到实参里, 而指针真正可以实现.

我们假定函数名为 swap, 操作的两个整型变量分别是 x 和 y, 同时, 函数 swap 里有一个中间局部变量称为 tmp, 用以交换形式参数 x,y, 那么, 主函数如何实现交换呢?

这就需要在函数里定义两个指针变量 xp,yp, 那么操作就变成了地址的交换. 使 xp 指向 x,yp 指向 y, 利用中间变量交换指针即可. 具体如下:

```
1 #include<stdio.h>
2 void swap(int *xp, int *yp)
3 {
4     int tmp;
5     tmp = *xp;
6     *xp = *yp;
7     *yp = tmp;
8 }
9 int main()
10 {
11     int x, y;
12     x = 10;
13     y = 5;
14     printf("交换前:x=%d y=%d\n", x, y);
15     swap(&x, &y);
16     printf("交换后:x=%d y=%d", x, y);
17     return 0;
18 }
```

5.11.3 指向一维数组的指针

数组的地址就是第一个元素的地址, 如果用指针变量指向数组, 则可以直接写为:

```
1 int *ptr;
2 int a[3]={1,2,3};
3 ptr=a;
```

因为数组元素连续依次存储, 要获取末位的地址, 可以如下操作:

```
1 int *qtr;  
2 int len=strlen(a); //需要头文件string  
3 qtr=ptr+len;
```

5.11.4 动态储存分配

malloc 函数

```
1 (类型说明符 *)malloc(size)
```

在动态存储区开辟长度为 size 字节的连续区域, 函数的返回值的首地址.

calloc 函数

```
1 (类型说明符 *)calloc(n,size)
```

在动态存储区开辟 n 块长度为 size 字节的连续区域, 返回值也是首地址, 分配的内存被自动初始化为 0.

free

```
1 free(void *ptr);
```

释放 ptr 指向的空间.free 函数需要适时使用, 因为用户开辟的空间不会自动释放, 否则会造成不必要的浪费.

第六章 数据结构与算法

编者: 王辰扬

6.1 数据与数据结构简介

数据 (背诵 ★): 描述客观事物的信息符号的集合. 能被输入计算机储存, 能被程序处理与输出. 包括**数值** (整数, 实数等), **字符串** (英文字母, 汉字等组成), 表格, 图像, 声音等.

数据类型 (背诵 ★): 具有相同特性数据的集合. 数据类型决定了数据的性质 (如: 取值范围, 操作运算等). **常用 (基本) 数据类型 (应用 ★★★):** **整型, 浮点型, 字符型.**

数据结构 (背诵 ★): 研究数据及数据元素之间的关系关系. **分类 (应用 ★★★):**

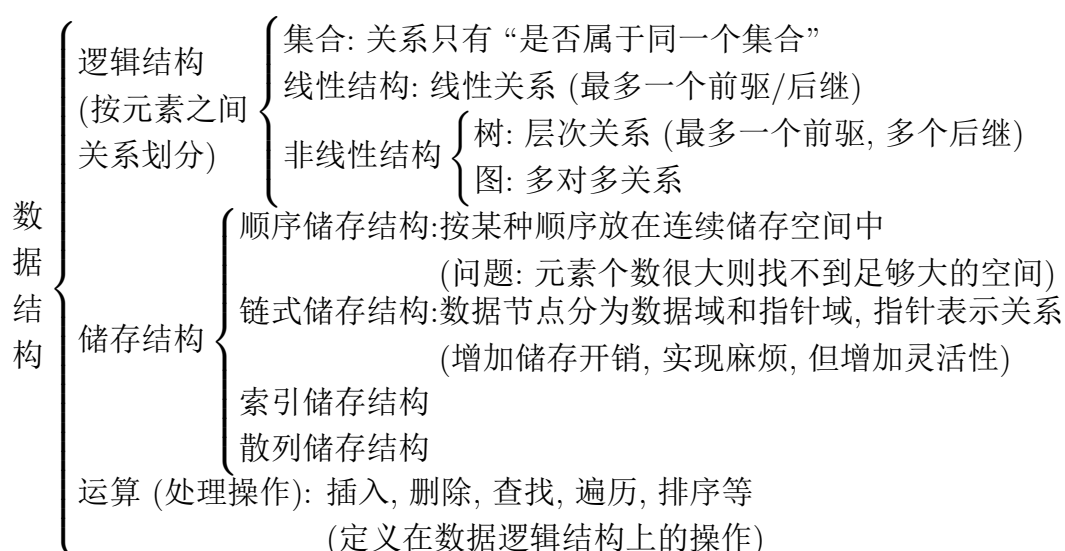


图 6.1: 数据结构分类示意图

6.2 线性表

6.2.1 线性表的逻辑结构 (应用 ★)

线性表是一种线性结构, 是一个含有 $n \geq 0$ 个结点的有限序列. 其中 ($n > 0$ 的) 有且仅有 1 个开始结点 (第一个结点) 没有前驱但有 1 个后继, 有且仅有一个终端结点 (最后一个结点) 没有后继但有 1 个后继. 其它结点都有且仅有 1 个前驱和一个后继. 一般线性表可以表示成线性序列 k_1, k_2, \dots, k_n , 其中 k_1 为开始结点, k_n 为终端结点. 数据元素 k_i 的意义在不同情况下各不相同, 但同一线性表的各数据元素必定具有相同的数据类型和长度.

6.2.2 线性表的基本操作 (应用 ★★★)

1. 置空表: 将整张表 L 清空, 表长度 n 置为 0.
2. 求表长: 求出表 L 中数据元素个数, 此处记为 $Length(L)$.
3. 取表 L 中 (特定位置 i 处) 元素.
4. 取元素的前驱 (后继): 当 $2 \leq i \leq Length(L)$ ($1 \leq i \leq Length(L) - 1$) 时返回 k_{i-1} (k_{i+1}).
5. 定位: 求数据元素 x 在表 L 中第一次出现的位置, 若 x 不在 L 中则返回 0 值.
6. 插入元素: 在线性表 L 的第 i 个位置插入元素 x (原先的元素顺次后移), 并且表长加 1.
7. 删除元素: 删除表 L 中的第 i 个位置的元素 (原先的元素顺次前移), 并且表长减 1.

6.2.3 代码实现 (应用 ★★★★★)

线性表按储存结构划分有两种实现形式: 顺序 (称为顺序表) 和链式 (称为链表). 顺序表可以借助数组很简单的实现, 实现代码如代码 SeqList.c 所示. 链表需要先定义结点 (一个数据域 + 一个指针域), 再整体实现链表. 实现代码如代码 LinkList.c 所示. 这两篇代码见第 6.10 节.

对于代码的考察通常不会以独立编写完整线性表的形式出现, 而更多的以补充局部代码或实现线性表部分功能以解决某个问题的形式出现. 下面举两个例子:

例 6.2.1 程序填空.

已知某单链表的表头 *Head* 和结点 *Node* 定义为

```
1 typedef struct
2 {
3     double data;
```



```

4     Node *next;
5 } Node;
6 typedef struct
7 {
8     Node *head;
9     int length;
10 } LinkList;

```

补全此单链表的插入节点函数 *DeleteNode* 中的 *A*, *B*, *C*, *D* 部分 (它们分别位于第 13, 16, 17 行的注释处).

A: _____

B: _____

```

1 int InsertElement(LinkList *lst, int pos, double Data)
2 { //插入元素函数,方便起见从0开始计数
3     if(pos<0||pos>lst->length)return -1;
4     Node *p=lst->head;
5     if(pos==0)
6     {
7         Node *q=(Node*)malloc(sizeof(Node));
8         q->data=Data;
9         q->next=lst->head->next;
10        lst->head=q;
11        return 0;
12    }
13    for( /* A */ ;i<lst->length;i++)p=p->next;
14    Node *q=(Node*)malloc(sizeof(Node));
15    q->data=Data;
16    q->next= /* B */ ;
17    return 0;
18 }

```

答案 A: int i=1

B: p->Next

分析: 第一个空要定指针 *p* (指向插入点的前一位) 的位置. 初始时的指向是第 0 位的前一位 (相当于表头), 考虑移动位数可以得到循环应该跑 $length - 1$ 次, 故此处罕见的出现了循环变量从 1 开始取值. B 空的填写可以从第 0 位的特殊情况得到提示. 注意此处指针 *p*, *q* 所指的都是数据元素, 都不能释放空间, 不要画蛇添足.

6.3 栈与队列

6.3.1 基本操作 (应用 ★★★★★)

栈和队列是两种特殊的线性表. 它们的结构和线性表完全相同, 但定义了完全不同的操作.

栈只允许在一端插入和删除元素. 允许插入和删除元素的一端称为栈顶, 另一端称为栈底. 一般插入元素称为入栈, 删除元素称为出栈. 显然先入栈的元素一定后出栈. 没有元素的栈称为空栈. 栈的特点被总结为先进后出 (First In Last Out, FILO).

队列允许在一端插入元素, 另一端删除元素. 进行插入的一端称为队尾, 进行删除的一端称为队头. 没有元素的队列称为空队列. 队列的特点被总结为先进先出 (First In First Out, FIFO)

栈和队列的基本操作如下:

1. 求栈 (队列) 的长度.
2. 判断栈 (队列) 是否为空.
3. 清空栈 (队列).
4. 入栈 (入队): 向将新数据元素添加到栈顶 (队尾), 栈 (队列) 发生变化.
5. 出栈 (出队): 将栈顶元素 (队头元素) 从栈 (队列) 中取出, 栈 (队列) 发生变化. 这种操作一般返回弹出元素的值.
6. 取栈顶 (队头) 元素: 返回栈顶元素 (队头元素) 的值, 栈 (队列) 不发生变化.

6.3.2 代码实现 (应用 ★★★★★)

使用数组实现栈和队列的代码 Stack.c 和 Queue.c 所示. 这两篇代码详见第 6.10 节.

下面讲解两个实现过程中的小技巧:

我们会发现, 在栈中从来都是固定栈底, 让栈顶上下浮动. 因此可以用栈底控制整个栈. 而队列中栈顶和栈底都会浮动, 所有需要额外创建内存范围的上限和下限这两个参数来控制队列的范围, 防止溢出.

细心的同学可能会发现, 队头的浮动会导致队头之前的内存全部浪费. 为了避免这种浪费, 笔者个人建议的处理方法是取地址在对最大长度模意义下的余数 (下限为起点 0).

理解这两点可以让读者更快的搭起栈和队列的架构, 在考场上节省编程时间.

6.4 图与树

注：这一部分主要强调理解，能够认识图和树的结构，理解简单的操作即可，几乎不考察代码实现。建议大家复习时过一遍书或小助手，让概念在脑内形成印象即可。不推荐花费大量时间，更不推荐死记硬背。

6.4.1 图的定义 (背诵 ★)

图是由结点 (或称顶点) 和连接结点的边所构成的图形。可记为 $G = \langle V, E \rangle$, 其中 $V = \{v_1, v_2, \dots, v_n\}$ 为顶点集合, $E = \{(v_i, v_j) | \dots\}$ 为边的集合。如果图为有向图 (即每条边带有方向), 则 E 记为 $E = \{\langle v_i, v_j \rangle | \dots\}$ 。

6.4.2 图的常用概念 (背诵 ★★★)

1. 结点, 边。
2. 无向边, 有向边: 不区分起点终点的边称为无向边, 区分起点终点的边称为有向边。
3. 无向图, 有向图: 所有边都是无向边的图称为无向图, 所有边都是有向边的图称为有向图。
4. 结点的度: 无向图中和一个结点关联的边的个数称为结点的度。有向图中由结点指向外的边的个数称为结点的出度, 由其它点指向结点的边的个数称为结点的入度。
5. 带权图, 带权边: 每条边都有一个非负实数对应的图称为带权图, 由非负实数与之对应的边称为带权边, 这个非负实数称为边的权 (在一些书上边的权可以为负)。

6.4.3 图的邻接矩阵表示 (应用 ★)

含 n 个顶点 v_1, v_2, \dots, v_n 的图 G 可以用一个 n 级矩阵 $A = (a_{ij})_n$ 表示。对于无向图,

$$a_{ij} = a_{ji} = \begin{cases} 1 & \text{(无向图) 或边 } (v_i, v_j) \text{ 的权值 (有向图), } v_i, v_j \text{ 有边相连} \\ 0, & v_i, v_j \text{ 没有边相连} \end{cases}$$

对于有向图,

$$a_{ij} = \begin{cases} 1 & \text{(无向图) 或边 } \langle v_i, v_j \rangle \text{ 的权值 (有向图), 存在边 } \langle v_i, v_j \rangle \\ 0, & \text{不存在边 } \langle v_i, v_j \rangle \end{cases}$$

这个用来表示图 G 的矩阵 A 称为图 G 的邻接矩阵。

6.4.4 树的定义

树是一种特殊的有向图. 满足以下条件的有向图称为树:

1. 存在唯一的结点 r , 其入度为 0.
2. 除 r 外其他结点的入度均为 1.
3. r 到图中其它结点均有路可达.

6.4.5 树的基本概念

1. 根节点: 入度为 0 的结点.
2. 叶子结点: 出度为 0 的结点.
3. 分枝结点: 出度不为 0 的结点.
4. m 叉树: 所有结点的出度均小于等于 m 的树.
5. 完全 m 叉树: 除叶子结点外, 每个结点的出度均为 m 的树.
6. 兄弟, 父亲, 儿子: 同一层次的结点称为兄弟, 上一层次的结点称为父亲, 下一层次的结点成为儿子.
7. 结点的层数 (深度): 从根到该结点的距离 (经过的边数).
8. 结点的高度: 从该节点到叶子结点的距离最大值.
9. 树的高度 (深度): 距根最远的叶子结点的层数.

注意: 整棵树的高度和深度是相等的, 但结点的高度和深度未必相等. 深度是从根结点向下数, 而高度是从叶子节点向上数.

6.4.6 二叉树

二叉树的定义与性质 (背诵 ★★★★★)

二叉树是每个结点最多有两个子树的结构. 二叉树上每个节点的两个子树一般分别被称为左子树和右子树.

二叉树的常用性质:

1. 第 i 层最多有 2^{i-1} 个结点.
2. 深度为 k 的二叉树的结点数最少为, 最多为 $2^k - 1$ (取等时称为满二叉树).

注意: 普通树上每个结点的子树 (孩子结点) 都是等价的, 没有先后顺序之分. 但是二叉树上结点的子树 (孩子结点) 有左右之分, 也就有了顺序. 二叉树的左右子树 (左右孩子结点) 绝对不能颠倒顺序.

完全二叉树和满二叉树 (背诵 ★★★★★)

满二叉树: 深度为 k 且有 $2^k - 1$ 个结点的二叉树.

完全二叉树: 深度为 k , 结点数为 n 且每个结点都与深度为 k 的满二叉树中编号从 1 至 n 一一对应的二叉树.

6.4.7 树的遍历 (应用 ★★★★★)

所谓遍历, 即按一定的顺序依次访问所有结点 (不重不漏). 树的访问有两种方法:

先根遍历: 先访问根结点, 再依次访问每个子树. 访问每个子树时, 也按照先根节点再子树的方式访问.

后根遍历: 先依次访问每个子树, 后访问根节点. 访问每个子树时, 也按照先子树后根节点的方式访问.

6.5 算法的描述方法

常用的四种描述方法 (背诵 ★)

1. 自然语言描述: 最直接, 没有语法语义障碍, 容易理解. 但是冗长不够简明, 可能出现含义不严格不确定的情况.
2. 计算机语言描述: 计算机可以直接执行, 程序设计的最终目标. 编写存在困难, 复杂算法需要中间过渡 (流程图或伪代码).
3. 伪代码描述: 介于自然语言和计算机语言之间. 使用自然语言和计算机语言构成 (尽可能地融入编程语言的函数和语法). 这种表示方法易于理解, 易于编写, 易于转化为计算机语言, 因而相当常用.
4. 流程图描述: 使用几种几何图形, 线条和文字表示不同的操作和处理步骤. 形象直观, 简洁清晰, 易于理解. ANSI 规定了五种常用流程图符号 (起始框/结束框, 处理框, 判断框, 流向线, 连接点), 一般只允许用标准的流程图符号作图.

算法的时间/空间复杂度 (应用 ★)

时间复杂度 $f(n)$ 为算法运行的时间频度 $T(n)$ (语句执行次数) 的同阶无穷大, 空间复杂度为所用辅助空间大小的一个同阶无穷大, 一般用大 O 表示. 常见的时间/空间复杂度 (从小到大): 常数阶 $O(1)$, 对数阶 $O(\log n)$, 线性阶 $O(n)$, 平方阶 $O(n^2)$, 立方阶 $O(n^3)$, k 次方阶 $O(n^k)$, 指数阶 $O(2^n)$, 阶乘阶 $O(n!)$ 等.

只需要理解时间/空间复杂度分别用来估计算法执行的时间和空间, 能估计简单程序 (暴力数数的过来) 的时空复杂度即可, 不需要在这上面浪费时间.

6.6 排序算法简介

6.6.1 冒泡排序 (应用 ★★★★★)

注: 绝对的重点, 必须熟练代码实现!

主要思想

两两比较待排序关键字, 发现两个相邻记录次序相反就进行交换, 直到没有反序记录为止.

算法步骤 (流程图)

冒泡排序算法的执行步骤如图 6.6(a) 所示. 可以计算得出其时间复杂度为 $O(n^2)$. 图 6.6(b) 是另一种实现冒泡排序的方法, 此方法需要使用一个游标变量探测在一趟比较中是否发生交换, 与不使用游标的方法在本质上是等价的. 这两张流程图见第 6.10 节.

代码实现

按照上面流程图中的描述, 可以写出冒泡排序的 C 语言代码实现如代码 BubbleSort.c 所示.

```
1  typedef int DataType; //通用起见数据类型记为DataType
2  void BubbleSort(DataType *lst, int n)
3  {    //冒泡排序函数,降序,lst是待排序数组,n是数组长度
4      for (int i = 0; i < n; i++)
5      {    //使用循环遍历数组
6          for (int j = n - 1; j > i; j--)
7          {
8              if (lst[j] < lst[j - 1])
9              {    //发现反序即进行交换
10                 DataType temp = lst[j];
11                 lst[j] = lst[j - 1];
12                 lst[j - 1] = temp;
13             }
14         }
15     }
16 }
```

BubbleSort.c

下面给出另一种实现冒泡排序的方式. 基本想法是: 如果整个数组已经有序, 再次遍历时就不会出现任何交换. 因此可以定义一个游标变量来探测是否交换, 然后反复搜索数组, 如果无交换直接退出循环即可. 流程图如 6.6(b) 所示, 实现代码如 BubbleSortExtra.c 所示. 这种实现方式和上面的实现方式都有人用, 读者可自行分辨好坏并作出选择.

```
1 typedef int DataType;  
2 void BubbleSortExtra(DataType *lst, int n)  
3 {    //冒泡排序函数(降序)  
4     int sign = 1; //探测交换的游标,1表示有交换,0表示没有交换  
5     while (sign == 1) //有交换则继续循环搜索  
6     {  
7         sign = 0; //初始时游标置0表示未交换  
8         for (int i = 0; i < n - 1; i++)  
9         {  
10             if (lst[i] < lst[i + 1])  
11             {    //发现反序就进行交换,同时设置游标  
12                 DataType temp = lst[i];  
13                 lst[i] = lst[i + 1];  
14                 lst[i + 1] = temp;  
15                 sign = 1;  
16             }  
17         }  
18     }  
19 }
```

BubbleSortExtra.c

上面这段代码含有一个常用的小技巧: 使用一个游标变量探测某种行为是否发生或以什么样的方式发生, 然后根据游标变量带回的值决定以何种方式进行下一步操作. 这个技巧经常用来带回特殊状态或异常. 使用时一定要注意游标初始状态的设置.

6.6.2 选择排序 (应用 ★★★)

主要思想

每次直接顺序查找待排序部分的最大值 (以降序为例), 放到这部分的最前面, 然后继续排序剩下的部分, 如此往复, 最终就可以使得数组有序.

代码实现

由于选择排序算法较为简单, 故省略流程图, 直接用代码表示, 如代码 SelectSort.c 所示. 选择排序的时间复杂度为 $O(n^2)$.

```
1 typedef int DataType;
2 void SelectSort(DataType *lst, int n)
3 { //选择排序函数,降序,lst是待排序数组,n是数组长度
4     for (int i = 0; i < n; i++)
5     {
6         int maxlab = i; //记录最大元素下标,初始设为第一个数
7         for (int j = i; j < n; j++)
8         { //顺序查找最大值,记录最大值所在下标
9             if (lst[j] > lst[maxlab]) maxlab = j;
10        }
11        //每次搜索完成后将最大值与排序部分最前面的值交换
12        DataType temp = lst[i];
13        lst[i] = lst[maxlab];
14        lst[maxlab] = temp;
15    }
16 }
```

SelectSort.c

这种算法也可使用指针作为循环变量进行编写, 比上面的代码更直观, 但是指针操作要求更高的控制技巧. 感兴趣的读者可以自己尝试.

6.6.3 快速排序 (应用 ★★★)

快速排序是常用的排序算法中速度最快的之一, 平均时间复杂度可达到 $O(n \log n)$. 而因为采用了更深刻的递归思想, 难度比冒泡排序和选择排序大. 但是快速排序思想十分简明, 掌握思想并记住核心步骤即可轻松实现.

主要思想

通过一趟排序 (一次划界) 将要排序的数据分割成独立的两部分, 其中一部分的所有数据比另外一部分的所有数据都要小, 然后再按此方法对这两部分数据分别进行快速排序, 整个排序过程可以递归进行, 一次将整个序列变为有序序列.

算法步骤

快速排序是一种递归算法, 并不方便使用流程图表示. 这里使用自然语言描述快速排序算法如下:

第一步: 任取数组中一个元素为**关键元素** (关键元素一般取第一个元素, 最后一个元素或随机取一个元素)

第二步: 将数组中比关键元素大的元素全部放到关键元素左边, 比关键元素小的全部放到关键元素右边 (这个过程称为“一趟排序”), 从而关键元素将整个数组划分成了左子数组和右子数组.

第三步: 对左子数组和右子数组分别进行以上过程的快速排序, 如果有数组长度为 1 或 0 则不对该数组排序.

第四步: 所有子数组的排序完成后整个数组的排序就完成了.

为了使上面的过程更具体, 举一个具体的例子:

例 6.6.1 使用快速排序, 对数组 $A = \{3, 7, 1, 6, 0, 8\}$ 进行排序.

解 取 3 为关键数据 (加粗), 同时标记数组未处理的最后一个数据 (上方横线):

$3, 7, 1, 6, 0, \bar{8}$

向前搜索. 7 比 3 大, 换到最后, 同时把后面的一个数字换到前面:

$3, 8, 1, 6, \bar{0}, 7$

8 比 3 大, 换到最后, 同时把后面的一个数字换到前面:

$3, 0, 1, \bar{6}, 8, 7$

0 比 3 小, 换到前面:

$0, 3, 1, \bar{6}, 8, 7$

1 比 3 小, 换到前面:

$0, 1, 3, \bar{6}, 8, 7$

6 比 3 大, 换到后面:

$0, 1, \bar{3}, 6, 8, 7$

此时关键数据 3 的位置已经被确定, 数组被剖成了 0, 1 和 6, 8, 7 两个待排序子数组. 分别对两个数组按以上过程快速排序 (子数组长度 ≤ 1 时不再继续排序), 可得最终的有序数组为

$0, 1, 3, 6, 8, 7$

快速排序的平均时间复杂度为 $O(n \log n)$, 最坏时间复杂度为 $O(n^2)$.

代码实现

快速排序的代码实现如代码 QuickSort.c 所示. 算法的要点: 主函数始末游标有序则先划界后递归, 划界函数先取关键值和两个位置, 后双向缩小序列, 保证大的放前小的放

后 (降序, 升序反之), 最后关键值放回数组, 关键值位置返回.

```
1 typedef int DataType;
2 void QuickSort(DataType *lst, int beginloc, int endloc)
3 { //快速排序函数
4     int QSKeyLoc(DataType *lst, int beginloc, int endloc);
5     if(beginloc < endloc)
6     {
7         int keyloc = QSKeyLoc(lst, beginloc, endloc);
8         QuickSort(lst, beginloc, keyloc - 1);
9         QuickSort(lst, keyloc + 1, endloc);
10    }
11 }
12 int QSKeyLoc(DataType *lst, int beginloc, int endloc)
13 { //快速排序辅助函数, 用于执行一次划界
14     int keyloc = beginloc, lastloc = endloc;
15     DataType keydata = lst[keyloc];
16     while(keyloc < lastloc)
17     {
18         while(keyloc < lastloc && lst[lastloc] <= keydata) lastloc--;
19         lst[keyloc] = lst[lastloc];
20         while(keyloc < lastloc && lst[keyloc] >= keydata) keyloc++;
21         lst[lastloc] = lst[keyloc];
22     }
23     lst[keyloc] = keydata;
24     return keyloc;
25 }
```

QuickSort.c

6.7 查找算法与思想

6.7.1 顺序查找 (应用 ★★★)

基本思想

从数组的第一个元素开始, 逐个把元素的关键字和给定值比较. 若某个元素的关键字值和给定值相等, 则查找成功. 否则, 若直至第 n 个记录都不相等, 说明不存在符合条件的数据元素, 查找失败.

算法流程图与代码实现

假设要搜索的表为 L , 要查找的数据为 A , 算法的流程图如图 6.2 所示, 实现代码可依据流程图非常直接的写出, 如程序 SeqSearch.c 所示.

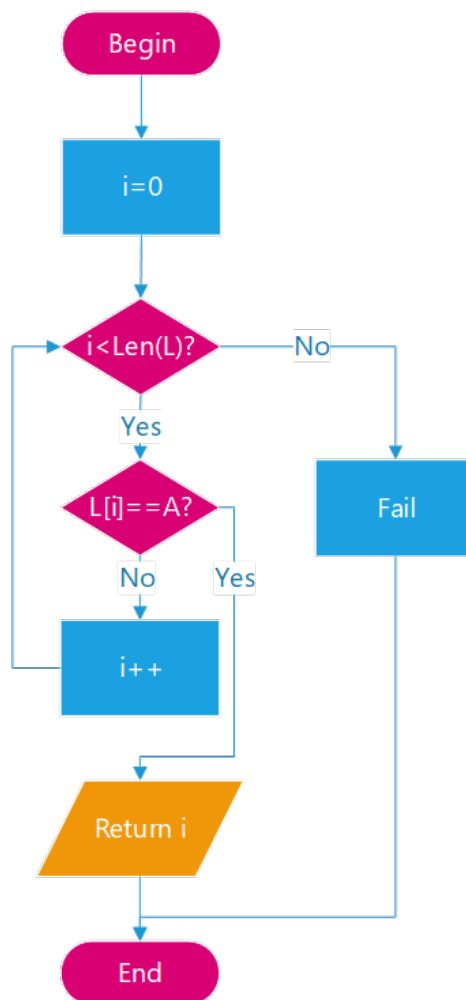


图 6.2: 顺序查找算法流程图

```

1  typedef int DataType;
2  int SeqSearch(DataType *List, int n, DataType A)
3  {    //顺序查找函数,输入待查线性表和长度,输出找到的位置,查找失败
      返回-1
4      for(int i=0; i<n; i++)
5      {
6          if(List[i]==A) return i; //查找成功
7      }
8      return -1; //查找失败
9  }

```

SeqSearch.c

其它需要了解的

顺序查找的平均时间复杂度是 $O(n)$, 最好, 平均, 最差情况下查找次数分别为 $1, \lceil \frac{n+1}{2} \rceil, n$.

顺序查找一般只适用于查找范围有限的情况下, 但不要求其有序. 当查找范围经常发生变化或者不方便排序时特别适合本方法.

6.7.2 折半查找 (应用 ***)

基本思想

如果待查找序列有序 (假设降序), 那么对于范围中的一个元素 a_i , 如果待查找元素 $x > a_i$, 那么 x 只可能存在与 a_i 之前, 否则只可能存在于 a_i 之后. 因此采取一种“跳跃”的方式查找元素 x .

算法流程图与代码实现

折半查找可以使用递归和非递归两种方式实现. 递归方法的思想与实现均非常直接 (直接分左右序列分别查找即可), 故直接使用 C 语言代码表示, 如代码 BinarySearch.c 所示.

```
1  typedef int DataType;
2  int SeqSearchRec(DataType *List, int begin, int end, DataType A)
3  {
4      /* 二分查找函数 (递归), 序列降序, 搜索成功返回第一次出现位置, 搜索失败返回 -1 */
5      if(begin <= end && List[begin] != A)
6          return -1; // 起终点重合 (或反序) 还没找到就搜索失败
7      int k = (begin + end) / 2;
8      if(List[k] == A)
9      {
10         while(List[k] == A) k--;
11         return k + 1;
12     }
13     else if(List[k] < A) // 搜索到的数据太小, 返回前半段
14         return SeqSearchRec(List, begin, k - 1, A);
15     else // 搜索到的数据太大, 返回后半段
16         return SeqSearchRec(List, k + 1, end, A);
```

17 }

BinarySearch.c

非递归方式的流程图如图 6.3 所示, 代码实现如代码 BinarySearch.c 所示. 技巧在于设置上下两个游标 Top 和 Bot, 每次用中位迭代其中一个来折半缩小查找范围.

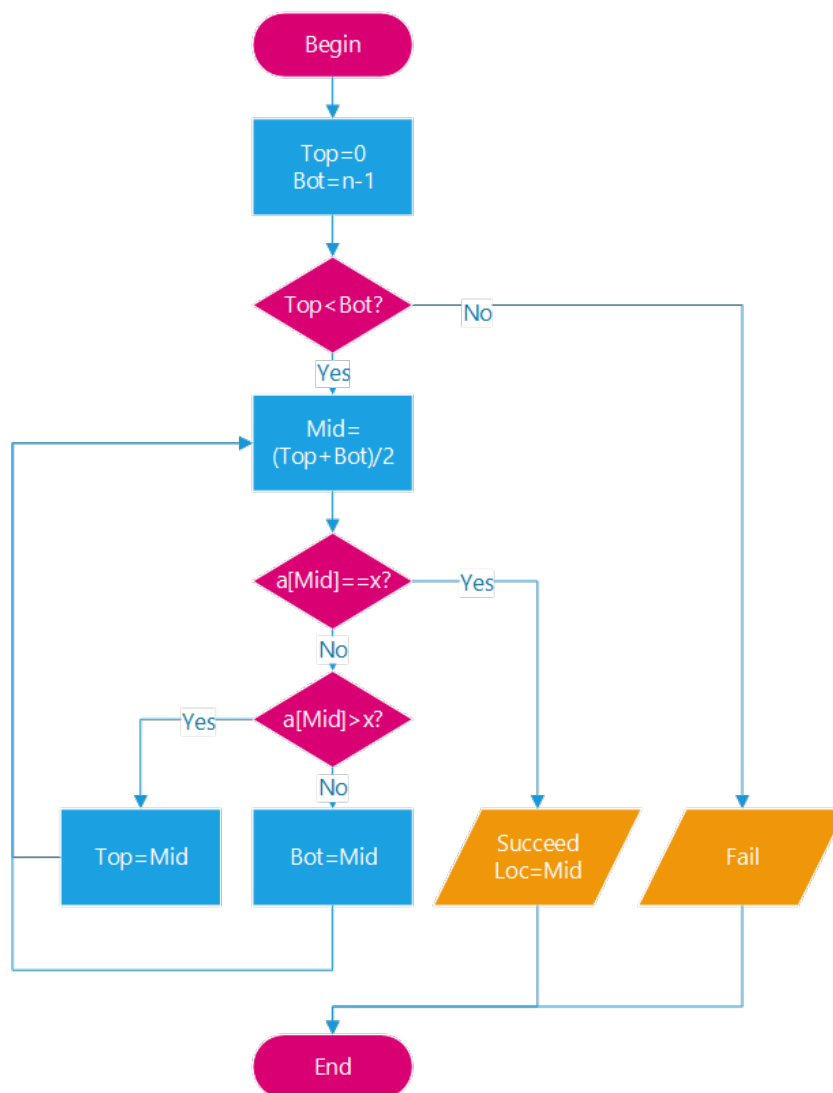


图 6.3: 二分查找 (非递归) 流程图

```

1 typedef int DataType;
2 int SeqSearch(DataType *List, int begin, int end, DataType A)
3 {
4     /* 二分查找函数 (非递归), begin和end分别是搜索起点和终点,
5      序列降序, 搜索成功返回第一次出现位置, 搜索失败返回-1 */
6     int b=begin, e=end, k;
7     while(1)

```

```
7 {
8     if(b<=e&&List[b]!=A) return -1;
9     k=(b+e)/2;
10    if(List[k]==A)
11        { //搜索附近，保证搜索到的是第一次出现位置，不要求可以省略，直接返回k即可.这只是一种处理方法，且会使得最坏时间复杂度上升.读者可以思考为什么，并且自行设计更优的处理方法.
12            while(List[k]==A) k--;
13            return k+1;
14        }
15    else if(List[k]<A) e=k-1;
16    else b=k+1;
17 }
18 }
```

BinarySearch.c

6.7.3 查找算法总结

顺序查找与折半查找的比较

顺序查找与一般查找常用性质的对比如表 6.1 所示.

表 6.1: 顺序查找与折半查找比较表		
性质	顺序查找	折半查找
查找方式	顺次	跳跃式
时间复杂度	平均/最差 $O(n)$, 最好 $O(1)$	平均/最差 $O(\log n)$, 最好 $O(1)$
对序列要求	不要求有序 (直接查找)	必须有序 (先排后查)
适合场景	一般序列, 无序, 可经常变动	自然有序/排序后序列, 可排序序列

应用举例

例 6.7.1 计算一个短字符串在一个长字符串中出现的次数.

输入共两行. 第一行为题中的长字符串, 第二行为题中的短字符串. 输入的数据保证两个字符串均不含空格和换行符, 字母区分大小写, 可能出现数字及标点符号. 两个字符串的长度均不超过 1000.

输出共两行.

第一行包含一个整数, 即短字符串在长字符串中出现的次数 n . 如果长字符串中不包含短字符串, 则输出 0.

第二行包含 n 个整数, 分别为短字符串每次在长字符串中出现时, 短字符串第一个字母在长字符串中的位置 (长字符串的第一个字符为第 1 位), 升序排列, 数据之间用空格分隔. 如果长字符串不包含短字符串, 则不输出第二行.

输入示例:

```
sakfjls;akdf.asdfasdfaksdfsadfasdf.asdfasdkhfasdf
asdf
```

输出示例:

```
4
14 18 31 36
```

解析 此问题是一个典型的顺序查找问题. 但由于需要所有出现位置而不只是第一次, 因此不能将之前的顺序查找代码生搬硬套. 很显然, 本题只需要顺次比较长字符串中所有和短字符串长度相等的连续子字符串, 遇到相等的再记下其位置即可. 由于比较字符串比较麻烦, 本题考虑先比较短字符串的首位, 相等再顺次比较其它位. 示例程序如下:

```
1  #include<stdio.h>
2  #include<string.h>
3  int main()
4  {
5      char a[1001],b[1001]; // a为长字符串, b为短字符串
6      scanf("%s%s",a,b);
7      int lena=strlen(a),lenb=strlen(b),nab=0,loc[1000];
8      // lena, lenb分别为两个字符串的长度, nab记录b在a中出现次数,
9      // loc数组记录出现位置
10
11     for(int k=0;k<lena-lenb;k++)
12     { // 先比较短字符串首位, 符合再比较其它位
13         if(a[k]==b[0])
14         {
15             int l;
16             for(l=0;l<lenb;l++)
17             {
18                 if(a[k+l]!=b[l])
19                 {
20                     break;
21                 }
22             }
23             // 利用循环变量l带回的值判断长串子串与短串是否相等
24             if(l==lenb)
```

```

24         {    // 遇到相等就记录出现位置
25             loc[nab]=k+1;
26             nab++;
27         }
28     }
29 }
30
31 printf("%d",nab);
32 if(nab!=0)
33 {
34     printf("\n");
35     for(int i=0;i<nab;i++)
36     {
37         printf("%d",loc[i]);
38         if(i!=nab-1)
39         {
40             printf(" ");
41         }
42     }
43 }
44 return 0;
45 }

```

例 6.7.2 定义“A 氏数”如下：对于一个正整数 m ，如果存在正整数 n ，使得 m 所有数位上的数字的 n 次方之和等于 m ，则称 m 是一个 A 氏数，这个自然数 n 称为数 m 的阶。要求求出小于给定数 a 的所有阿姆斯特朗数以及它的阶。

输入共一行，包含一个正整数，也就是题目中的数 a ($1 \leq a \leq 10^6$)。

输出共 $k+1$ 行。

第一行为一个自然数 k ，为小于 a 的 A 氏数的个数。

随后的 k 行，每行两个正整数，分别为小于 a 的每个 A 氏数及其阶，两个数之间用空格分隔，A 氏数之间升序排列。如果不存在小于 a 的 A 氏数，则不输出第一行之后的行。

输入示例：

500

输出示例：

12

1 1

2 1

3 1


```

4 1
5 1
6 1
7 1
8 1
9 1
370 3
371 3
407 3

```

解析 本例中包含两个查找: 在 $[1, a]$ 中查找 A 氏数, 对于每个数 $n \in [1, a]$ 查找其阶. 第一个查找只能使用顺序查找 (广义阿氏数的出现不是有序的), 而第二个查找顺序和折半都可以用 (注意到幂和随幂单调递增) 且两种排序方式差别不大 (顺序平均 $O(n \log n)$, 二分平均 $O(n \log \log n)$, 当前数据范围时只差常数倍), 都可以通过.

第二个查找需要估计阶的一个上界以缺点查找范围. 注意这个上界不能太粗糙, 否则幂和可能过大而在储存时溢出. 设 n 的十进制表示为 $n = \sum_{k=0}^l a_k \times 10^k$, 其阶为 r . 如果 a_k 全是 1 且 n 不是 1 位数, 容易验证 n 不是 A 氏数. 于是假设 n 有一位不小于 2. 则

$$n = \sum_{k=0}^l r_k \times 10^k = n = \sum_{k=0}^l a_k^r \geq 2^r$$

所以 $r \leq \log_2 n = \frac{\ln n}{\ln 2}$. 这是阶的一个可用的上界.

根据以上分析可用编写示例程序如下:

```

1  #include<stdio.h>
2  #include<math.h>
3  #define MAXN 100000
4  int arn[MAXN],rankarm[MAXN]; // 储存A氏数和阶的数组,由于比较大,
   // 在main外面开出来,以防开不出来
5  int main()
6  {
7      int a,narn=0; // narn为A氏数的个数
8      scanf("%d",&a);
9      int ifarm(int n); //声明用于判断和求阶的函数
10
11     for(int i=1;i<a;i++) //顺序查找搜索A氏数
12     {
13         int j=ifarm(i);
14         if(j)

```

```
15     {
16         arn[narn]=i;
17         rankarm[narn]=j;
18         narn++;
19     }
20 }
21
22 printf("%d",narn);
23 for(int i=0;i<narn;i++)
24 {
25     printf("\n%d %d",arn[i],rankarm[i]);
26 }
27 return 0;
28 }
29
30 int ifarm(int n) // 判断n是不是A氏数,是则返回其阶,不是则返回0
31 {
32     int numdiv[15],numlen=0;
33     for(int m=n;m>0;m=m/10) // 对n数位分解
34     {
35         numdiv[numlen]=m%10;
36         numlen++;
37     }
38
39     int maxr=(int)(log((double)n)/log(2.0))+1,minr=1,midr,sum;
40     while(maxr>=minr) // 使用折半查找搜索阶
41     {
42         midr=(int)(maxr+minr)/2;
43         sum=0;
44         for(int j=0;j<numlen;j++)
45         {
46             sum+=(int)pow((double)numdiv[j],(double)midr);
47         }
48         if(sum==n)
49         {
50             return midr; // 找到则返回当前尝试的阶(就是midr)
51         }
52         else if(sum>n)
53         {
```

```

54         maxr=midr-1;
55     }
56     else
57     {
58         minr=midr+1;
59     }
60 }
61 return 0;    // 能退出循环说明一定查找失败,返回0
62 }
```

例 6.7.3 求开普勒方程

$$x = q \sin x + a$$

的近似解. 其中 $q \in (0, 1), a \in \mathbb{R}$.

输入共一行, 包含两个实数 q, a , 意义如题中所述, 保留 7 位小数.

输出共一行, 包含一个实数 x_0 , 为开普勒方程 $x = q \sin x + a$ 的近似解, 保留 4 位小数 (输出时格式控制符使用 `%.4f` 可以保留四位小数).

示例输入:

0.5000000 1.0000000

示例输出:

1.4987

解析 本例选自数学实验课本, 主要希望向读者传达“折半查找是一种思想而不只是一种算法”的想法.

首先将方程转化为 $f(x) = 0$ 的形式, 以便估计出方程的解存在的大致范围. 即定义

$$k(x) = x - q \sin x - a$$

于是只需求 $k(x)$ 的近似零点. 对 $k(x)$ 求导可得

$$k'(x) = 1 - q \cos x \geq 1 - q > 0$$

即 $k(x)$ 单调递增. 又显然

$$\begin{cases} k(a-1) = -1 - q \sin(a-1) < 0 \\ k(a+1) = 1 - q \sin(a+1) < 0 \end{cases}$$

所以 $k(x)$ 一定有唯一零点 $x_0 \in (a-1, a+1)$. 对这个区间折半查找零点即可. 由于精度要求为 4 位小数, 因此跳出的条件为查找区间长度小于 $\varepsilon = 0.5 \times 10^{-4}$ 或区间中点刚好是零点. 根据此编写示例程序:

```
1  #include<stdio.h>
2  #include<math.h>
3  int main()
4  {
5      double q,a;
6      scanf("%lf%lf",&q,&a);
7      double kep(double x, double q, double a);
8      double xLow=a-1,xHigh=a+1,xMid,eps=0.5*pow(10.0,-4.0);
9
10     while(xHigh-xLow>eps)    //二分查找搜索零点
11     {
12         xMid=(xLow+xHigh)/2;
13         if(kep(xMid,q,a)==0)
14         {
15             break;
16         }
17         else if(kep(xMid,q,a)>0)
18         {
19             xHigh=xMid;
20         }
21         else
22         {
23             xLow=xMid;
24         }
25     }
26
27     printf("%.4f",xMid);
28     return 0;
29 }
30
31 double kep(double x,double q,double a)
32 {
33     return x-q*sin(x)-a;
34 }
```

从上面几个例子我们可以得出的启示:

1. 对于搜索问题, 要先明确搜索范围, 再根据搜索范围的性质选择搜索方法.
2. 只要答案所在的范围已知而且有限, 就可以考虑查找算法 (对于范围内的元素依次

比较即可). 这有助于在考场上解决一些“看似很难/没思路”的问题.

3. 二分查找不只是我们在课本中学过的固定的算法, 而是一种思想. 如果答案所在的空间满足“有序性”(a 不满足则 a 之前/之后的都不满足), “可验证性”(虽然可能求阶思路不明确, 但给定数据很容易验证是不是答案) 则可以考虑使用二分查找.

6.8 递归与分治

本部分在考试时一般会出一道填程序代码的大题, 也可能出编程题, 必考且难度不低, 占分值 5-10 分. 希望读者充分理解本部分内容, 达到融会贯通, 学以致用效果.

6.8.1 定义 (背诵 ★)

递归 直接或间接调用自身的算法称为递归.

分治 对一个规模为 n 的问题, 若该问题可以比较容易的解决, 则直接解决; 否则将其分为 k 个规模较小的子问题, 这些子问题互相独立且与原问题形式相同, 于是可以递归的求解这些子问题, 然后将它们的解合并得到原问题的解.

6.8.2 应用举例 (应用 ★★★★★)

前面提到过的很多算法都直接或间接的应用了递归或分治的思想, 而这两种思想经常共同使用, 不分彼此. 如快速排序, 折半查找, 树的先根遍历和后根遍历等都应用了递归和分治的思想. 下面再举几个例子.

例 6.8.1 (快速幂) 编写程序计算 $a^b \bmod c$, 其中 a, b, c 由键盘输入给出.

输入: 共一行, 包含三个正整数, 分别为 a, b, c , 其中 $1 \leq a \leq 10^5, 1 \leq b \leq 10^7, 1 \leq c \leq 10^7$.

输出: 共一行, 一个正整数, 即 $a^b \bmod c$.

输入示例:

2 10 3

输出示例:

1

解析 众所周知, 计算 a^b 的常规方法是将 b 个 a 乘起来, 时间复杂度为 $O(b)$. 本问中 a, b 的值都相当巨大, 按以上方法暴力乘的后果是不仅运算时间太长 (正常的编程题必须在 1s 之内跑出结果), 而且乘出来的数字也没有常规数据结构放得下. 所以必须想办法降时间复杂度和数据规模.

本题的正确做法是利用关系式

$$a^b = \begin{cases} \left(a^{\frac{b}{2}}\right)^2, & b = 2k \\ \left(a^{\frac{b-1}{2}}\right)^2 \cdot a, & b = 2k + 1 \end{cases}$$

构造递归进行分治求解. 只要能想到这个式子, 本题就没有难度. 使用递归方法编程如下:

```

1  #include<stdio.h>
2  int main()
3  {
4      int a,b,c;
5      scanf("%d%d%d",&a,&b,&c);
6      int fastpow(int a,int b,int c);
7      printf("%d",fastpow(a,b,c));
8      return 0;
9  }
10 int fastpow(int a,int b,int c)
11 {
12     if(b==1)
13     {
14         return a%c;
15     }
16     return fastpow(a,b/2,c)^2*(b%2?a:1)%c;
17 }
```

本题也可以使用非递归方式求解, 需要利用 b 的二进制表示, 感兴趣的读者可以自行尝试.

例 6.8.2 以下代码是通过某种方式对整数数组 $List$ 进行排序 (降序) 的函数, 补全 A, B 两部分 (分别位于 20, 21 行的注释处).

A: _____

B: _____

```

1  void Sort(int *List,int Left,int Right)
2  {
3      // 排序函数,Left,Right是排序范围的左右两端
4      void Sortsup(int*,int,int,int); //声明辅助函数
5      if(Left<Right)
6      {
7          int Mid=(Left+Right)/2;
```

```
8      Sort(List, Left, Mid);
9      Sort(List, Mid+1, Right);
10     Sortsup(List, Left, Mid, Right);
11 }
12 }
13 void Sortsup(int *List, int Left, int Mid, int Right)
14 {
15     // 排序辅助函数
16     int i, j, k;
17     int *SupList = (int*) malloc((Right - Left + 1) * sizeof(int));
18     for(i = Left, j = Mid + 1, k = 0; i <= Mid && j <= Right; k++)
19     {
20         if(List[i] > List[j]) /* A */
21         else /* B */
22     }
23     for( ; i <= Mid; i++, k++) SupList[k] = List[i];
24     for( ; j <= Right; j++, k++) SupList[k] = List[j];
25     for(i = 0; i < Right - Left + 1; i++) List[Left + i] = SupList[i];
26     free(SupList);
27 }
```

解析 本题答案为

A: SupList[k]=List[i++];

B: SupList[k]=List[j++];

本题为书中没有涉及的另一种非常经典的排序方法：归并排序。这是一种非就地（需要辅助空间）递归排序方法，平均时间复杂度 $O(n \log n)$ 。主要思想如下：先将序列等分为两个等长或几乎等长的子序列，对每个序列分别排序（递归调用自身），然后将两个有序的序列合并为一个有序的序列即可。有一定算法基础的读者可以直接看出并给出答案。对于不熟悉归并排序的读者来说，需要分析给出的代码，理解其含义，并分析出空缺位置的作用。

主排序函数只做了 4 件事情：划分左右两个序列，对左序列排序，对右序列排序，某种神奇的操作（辅助函数）。由于已知函数 Sort 的作用是对序列排序，因此可以确定辅助函数的功能就是把两个已经有序的序列合并为一个有序的序列（确定了辅助函数的功能）。

接下来分析辅助函数。16, 17 两行定义了所用的累加变量，申请了所需的辅助空间。接下来一个 for 循环，i, j 分别从左右两个子序列的首位开始，i 到左序列的尾端或 j 到右序列的尾端结束，我们可以猜测这个 for 循环的作用是遍历左右序列，过程中做某种操作。这种操作是：当左序列当前位置的值比较大时 A，否则 B。于是我们可以

猜测是把比较大的那个存下来, 结合下文可以猜测应该存到辅助空间的当前位置, 使用 k 进行遍历, 因此可以填出 $\text{SupList}[k]=\text{List}[i]$; 和 $\text{SupList}[k]=\text{List}[j]$;。检查整个程序发现此处需要让对应游标自增, 于是可以得到答案 $\text{SupList}[k]=\text{List}[i++]$; 和 $\text{SupList}[k]=\text{List}[j++]$;

如果将赋值和自增两个操作分两句写, 加大括号括起来也可以, 但是不加大括号就错了。

6.8.3 不合理递归 (应用 ★★★)

不合理递归是一种非常特殊的递归形式. 这一部分内容极少考察, 但只要考察难度就会相当高. 下面通过一些例子来说明这种递归.

例 6.8.3 编写程序计算斐波那契数列 F_n 的第 k 项. 其中 F_n 的递推式为

$$F_n = F_{n-1} + F_{n-2}, \forall n > 2$$

初始条件为 $F_1 = 0, F_2 = 1$. k 的值由键盘输入给出.

解析 本题有很多种方法可以解决. 这里提供两种方法.

解法 1: 直接利用递推, 递归求解. 程序如下:

```
1  #include<stdio.h>
2  int main()
3  {
4      int k;
5      int Fib(int n); //声明计算F_n的函数
6      scanf("%d",&k);
7      printf("%d",Fib(k));
8      return 0;
9  }
10 int Fib(int n)
11 {
12     if(n<=2)
13     {
14         return 1;
15     }
16     else
17     {
18         return Fib(n-1)+Fib(n-2);
19     }
20 }
```

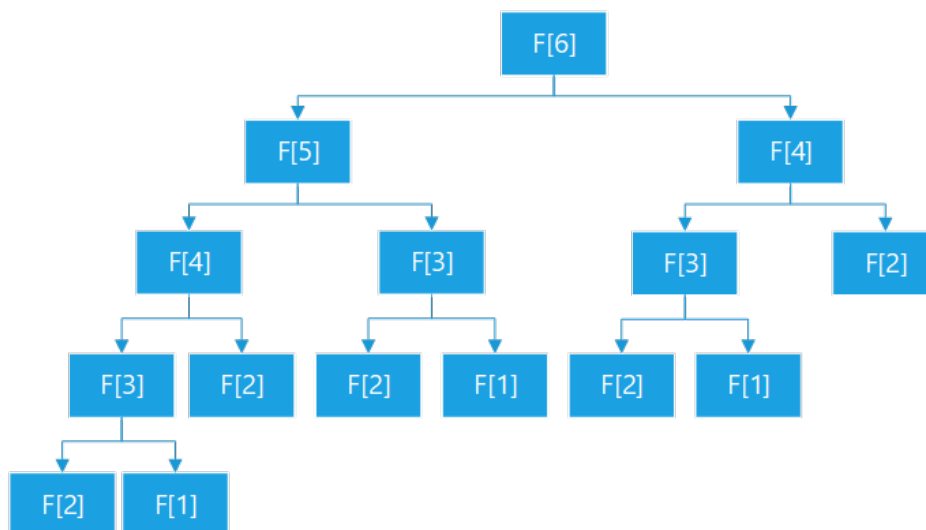

建议读者自行上机编写或验证本程序. 不难发现, 当 k 的值不太大时, 这个程序跑的非常正常; 但是当 k 的值增大到 50 左右的时候程序运行的相当慢. 接下来给出另一种方法.

解法 2: 利用一个数组存储 F_n 的值, 迭代计算, 程序如下 (为尽量避免溢出, 使用储存范围较大的 long long 型数组存储 F_n 的值):

```

1  #include<stdio.h>
2  int main()
3  {
4      int k;
5      scanf("%d",&k);
6      long long Fib[100]={0,1};
7      for (int i = 2; i < k; i++)
8      {
9          Fib[i]=Fib[i-1]+Fib[i-2];
10     }
11     printf("%lld",Fib[k-1]);
12     return 0;
13 }
```

这个程序在 k 较大时会发生溢出 (由于 $F_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^{n-1} - \left(\frac{1-\sqrt{5}}{2} \right)^{n-1} \right]$, 当 n 较大时指数爆炸, 增长极快), 但是运行速度相当快. 下面我们来分析一下这两个程序运行一个快一个慢的原因.



调用, 如图 6.4 所示.

从图 6.4 中可以看出, 计算 F_n 时的递归调用是以指数阶增长的. 事实上, 通过计算可以得知, 递归法 (解法 1) 的时间复杂度为 $O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$. 这种递归中对一些值反复进行计算, 造成算法效率极其低下, 称为不合理递归. 而解法 2 中使用一个数组存储所需的值, 将计算改为了调用, 大大提高了运行效率. 这是避免不合理递归的常用方法.

本题可通过矩阵快速幂将时间复杂度进一步提高到 $O(\log n)$, 读者可自行尝试.

6.9 练习题

1. 在数据的逻辑结构中, 如果数据元素之间的关系是多对多的关系, 则该结构属于 _____.
 - (a) 线性结构
 - (b) 树
 - (c) 图
 - (d) 集合
2. 下面关于线性表的叙述, 不正确的是 _____.
 - (a) 线性表是 n 个结点的有穷序列
 - (b) 线性表可以为空表
 - (c) 线性表的每一个结点有且仅有一个前趋和后趋
 - (d) 线性表结点间的逻辑关系是一一对应的
3. 线性表采用顺序存储时, 其地址 _____.
 - (a) 必须是连顺的
 - (b) 一定是不连续的
 - (c) 部分地址必须是连续的
 - (d) 连续与否均可以
4. 设顺序表中有 n 个元素, 要删除表中的第 i 个元素, 需要移动 ____ 个元素.
 - (a) $n - i$
 - (b) $n - i + 1$
 - (c) $n + i - 1$
 - (d) i

5. 若进栈序列为 $[1, 2, 3, 4, 5, 6]$, 且进栈与出栈可以穿插进行, 则可能出现的出栈序列为 ____.
- (a) $[3, 2, 6, 1, 4, 5]$
 - (b) $[3, 4, 2, 1, 6, 5]$
 - (c) $[1, 2, 5, 3, 4, 6]$
 - (d) $[5, 6, 4, 2, 3, 1]$
6. 对序列 $[5, 4, 1, 7, 3, 8, 6, 9]$ 进行快速排序 (降序, 选取第一个元素为关键元素), 第一趟排序之后的序列为 ____.
- (a) $[4, 1, 7, 5, 3, 8, 6, 9]$
 - (b) $[4, 1, 3, 5, 8, 6, 9]$
 - (c) $[4, 1, 3, 5, 9, 6, 8, 7]$
 - (d) $[7, 8, 6, 9, 5, 3, 1, 4]$
7. 下列说法不正确的是 ____
- (a) 树和二叉树中同一个结点的孩子结点都是完全等价的.
 - (b) 树是一种“一对多”的结构, 而图是一种“多对多”的结构.
 - (c) 树和图都是非线性结构.
 - (d) 图的邻接矩阵一定是对称阵.
8. 编写程序解决以下问题:
- 输入: 共两行. 第一行为一个正整数 k ($3 \leq k \leq 10^4$), 代表第二行有 k 个数据. 第二行有 k 个正整数.
- 输出: 共一行, 包含 k 个正整数, 依次为上面 k 个正整数中最大的, 最小的, 第二大的, 第二小的, 以此类推.
- 输入示例:
- 1 2 5 3 6 7 4 9 8
- 输出示例:
- 9 1 8 2 7 3 6 4 5
9. 编写程序解决以下问题:
- 输入: 第一行为一个正整数 k , 代表之后有 k 组数据. 接下来的 k 行, 每行一个数学表达式, 含有数字, 加号 (+), 减号 (-), 乘号 (*), 除号 (/) 和括号 ((,), [,], {, }). 输入的表达式保证不会出现除括号匹配之外的任何错误, $k \leq 1000$, 输入每个字符串长度小于 1000.

输出: 共 k 行, 其中如果输入的第 i 个表达式, 输出的第 i 行就为 YES, 否则输出的第 i 行就为 NO.

输入示例:

4 (3+2)/(2+1)+10*(10+2)

1+{(2+3)+(3+4)*[5+6*(7+8)]}

(3*[2+(1+3)*5]+(1*(2+3))

[1+2(3*4))+5

输出示例:

YES

YES

NO

NO

10. 编写程序解决以下问题:

定义阿克曼函数

$$AKM(m, n) = \begin{cases} n + 1, & m = 0 \\ AKM(m - 1, 1), & m > 0 \text{ 且 } n = 0 \\ AKM(m - 1, AKM(m, n - 1)), & \text{etc.} \end{cases}$$

计算阿克曼函数的值. 其中 m, n 的值由键盘给出.

输入: 共一行, 两个正整数 m, n .

输出: 共一行, 一个正整数 $AKM(m, n)$.

输入示例:

1 3

输出示例:

5

如果已经对自己编写程序的正确性确信无疑, 请参看本题提示.

11. 程序填空.

已知某单链表的表头 Head 和结点 Node 定义为

```
1 typedef struct
2 {
3     Node *First;
```

```

4 } Head;
5 typedef struct
6 {
7     Node *Next;
8     double Data;
9 } Node;

```

补全此单链表的删除节点函数 DeleteNode 中的 A, B, C, D 部分 (它们分别位于第 13, 14, 15, 17 行的注释处).

A: _____

B: _____

C: _____

D: _____

```

1 int DeleteNode(Head LstHead, int Loc)
2 { //删除第Loc个结点的函数,删除成功返回0,否则返回其它值
3     if(Loc<=0) return -1;
4     Node *DelFr,*DelLoc;
5     if(Loc==1)
6     {
7         DelLoc=LstHead.First;
8         LstHead.First=DelLoc->First;
9     }
10    else
11    {
12        DelFr=LstHead.First;
13        for(int i=0; /* A */ ; i++) DelFr=DelFr->Next;
14        DelLoc= /* B */ ;
15        DelFr->Next= /* C */ ;
16    }
17    /* D */
18    return 0;
19 }

```

12. 程序填空下面的函数 gcd 是计算正整数 m, n 最大公约数的函数. 补全此程序的 A, B 部分. 它们分别位于第 3, 5 行的注释处.

```

1 int gcd(int m, int n)
2 {

```

```

3   if(n==0) /* A */ ;
4   else if(m<n) return gcd(n,m);
5   else return /* B */ ;
6   }

```

A: _____

B: _____

习题提示

对第 4, 5, 6, 9, 10 题没有思路的读者可以参看以下提示, 尝试不看答案完成这些题目.

4. 试着用图画出这个线性表.
5. 试着手动做出入栈, 拼凑四个选项的出入栈结果.
6. 快速排序的关键元素具有“划界”的重要性质, 也就是将数组划分成左右两个数组, 它们之间有什么宏观的关系?
9. 用栈. 如果括号是匹配的, 一定可以从里到外一个个的提出相邻的左右括号.
10. 试着用编写的程序计算 $AKM(3, 4)$. 如果算不出来证明你做错了.

参考答案

1. (c). 详见课本 P 328 四种逻辑结构的定义.
2. (c). 第一个元素没有前趋, 最后一个元素没有后继.
3. (a). 详见课本 P 331.
4. (a). 如图 6.5 所示, 在第 i 个元素以及之前共有 i 个元素, 第 i 个元素之后有 $n - i$ 个元素. 当第 i 个元素被删除之后, 其后的 $n - i$ 个元素被移动.

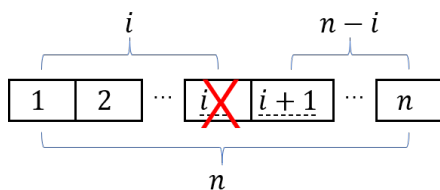


图 6.5: 习题 4 顺序表示意图

5. (b). 本题是考察栈的性质的一种经典题型, 几乎必考, 希望读者掌握分析策略. 对于 (a) 选项, 由于 3 先出栈, 此时栈内为 [1, 2], 接下来 2 出栈, 栈内 [1], 接下来 6 出栈 (之前 4, 5, 6 要进栈), 栈内为 [1, 4, 5], 进栈完毕, 接下来只能 5 出栈, 矛盾, 可知 (a) 错. 同理可以验证 (c) (d) 错, (b) 对.
6. (d). 一趟快速排序中, 关键元素最重要的性质是“划界”. 以降序为例, 划界后比关键元素小的一定在右序列, 比关键元素大的一定在左序列. (a) 没有划界, (b) 元素减少了, (c) 是升序, 所以选 (d).
7. (a). 选项 (a), (b), (c) 具体详见课本对应概念. (d) 图的邻接矩阵不一定是对称阵. 无向图的邻接矩阵一定是对称阵, 而有向图不一定.
8. 分析: 排序裸题, 先排序再按要求输出即可. 按数据范围, 本题选择各种排序方法均可. 这里选择快速排序, 程序如下:

```
1 #include<stdio.h>
2 int main()
3 {
4     int k,a[1001];
5     scanf("%d",&k);
6     for(int i=0;i<k;i++) scanf("%d",a+i);
7     void QuickSort(int *lst,int beginloc,int endloc);
8     QuickSort(a,0,k-1);
9     for(int i=0,j=k-1;i<=j;i++,j--)
10    {
11        if(i-j>=2) printf("%d %d ",a[i],a[j]);
12        else if(i-j==1) printf("%d %d",a[i],a[j]);
13        else printf("%d",a[i]);
14    }
15    return 0;
16 }
17 void QuickSort(int *lst,int beginloc,int endloc)
18 {
19     int QSKeyLoc(int *lst,int beginloc,int endloc);
20     if(beginloc<endloc)
21     {
22         int keyloc=QSKeyLoc(lst,beginloc,endloc);
23         QuickSort(lst,beginloc,keyloc-1);
24         QuickSort(lst,keyloc+1,endloc);
25     }
26 }
```

```

27 int QSKeyLoc(int *lst,int beginloc,int endloc)
28 {
29     int keyloc=beginloc,lastloc=endloc;
30     int keydata=lst[keyloc];
31     while(keyloc<lastloc)
32     {
33         while(keyloc<lastloc&&lst[lastloc]<=keydata)
34             lastloc--;
35         lst[keyloc]=lst[lastloc];
36         while(keyloc<lastloc&&lst[keyloc]>=keydata) keyloc++;
37         lst[lastloc]=lst[keyloc];
38     }
39     lst[keyloc]=keydata;
40     return keyloc;
}

```

考试时不会给数据范围, 最好选择速度尽量快的算法, 一般 $O(n^2)$ 及以下的算法都不会有太大问题, 但 $O(2^n)$ 及以上绝对无法通过. 另外注意, C 语言绝对不允许开动态数组 (数组的长度必须为常数或者常量表达式, 不能现场输入). 如果数据量不确定, 一遍使用 malloc 开动态空间或者开一个相当大的数组而只用一部分.

9. 分析: 本题是典型的栈的应用. 思路是开一个栈, 读到左括号就将其入栈, 读到右括号就将其出栈并检验匹配性. 如果读到右括号而栈空, 或读完而栈不空, 就说明不匹配, 否则说明匹配. 程序如下:

```

1  #include<stdio.h>
2  int main()
3  {
4      char Stack[1001],c; //使用一个字符数组来存储括号栈
5      int k,StackLen,sign;
6      scanf("%d",&k);
7      for(int i=0;i<k;k++)
8      {
9          StackLen=0;sign=0; //重置栈长和游标
10         while((c=getchar())!='\n') //这是经典的字符串逐位读
11             取方法
12             {
13                 switch (c)

```



```

14         case '(':
15         case '[':
16         case '{':
17             Stack[StackLen++]=c;
18             break;
19         case ')': //一定要先验证栈长大于0才能出栈,否则程序可能直接崩溃.这里请不要使用短路逻辑处理,不安全.
20             if(StackLen<=0) {sign=1; break;}
21             if(Stack[--StackLen]!='(') {sign=1; break;}
22         case ']':
23             if(StackLen<=0) {sign=1; break;}
24             if(Stack[--StackLen]!='[') {sign=1; break;}
25         case '}':
26             if(StackLen<=0) {sign=1; break;}
27             if(Stack[--StackLen]!='{') {sign=1; break;}
28         default:
29             break;
30     }
31 }
32 if(StackLen!=0||sign==1) printf("NO");
33 else printf("YES");
34 if(i<k-1) printf("\n");
35 }
36 return 0;
37 }

```

感兴趣的读者可以试着编程自动计算书写正确的表达式的值 (可以用栈来实现, 也可以用二叉树来实现).

10. 分析: 本题是笔者在大计基期末考试上拿到的真题. 在此之前笔者认为大计基考试中的递归就是按题意直接写. 但是写出的程序连 $AKM(3,4)$ 都跑不出来, 这使得笔者意识到事情没有想象中那么简单. 尝试着画了一部分递归树之后笔者意识到这是一道非常困难的不合理递归. 于是笔者利用不合理递归最简单的处理方法 (开数组存值迭代) 在考场上顺利的解决了这道题. 示例程序如下:

```

1 #include<stdio.h>
2 unsigned AKM[5][100000];
3 int main()
4 {
5     for (int m = 0; m < 5; m++)

```

```

6      {
7          for (int n = 0; n < 100000; n++)
8          {
9              if (m==0) AKM[m][n]=n+1;
10             else if (m>0&& n==0) AKM[m][n]=AKM[m-1][1];
11             else AKM[m][n]=AKM[m-1][AKM[m][n-1]];
12         }
13     }
14     int m,n;
15     scanf("%d%d",&m,&n);
16     printf("%u",AKM[m][n]);
17     return 0;
18 }

```

本题的要点有以下 5 点:

- 试出这是一道不合理递归 (示例数据不足以让你意识到这一点, 要自己带数据试出来, 第一个参数取到 3 就足以让你意识到问题的严重性).
- 知道处理不合理递归的最简单方法 (开数组存值迭代, 防止重复计算).
- 通过简单的数学计算知道阿克曼函数关于第一个参数的增长率有多么的巨大, 从而在开数组的时候把第一个参数开的很小, 第二个参数开的很大.
- 通过尝试发现计算 $AKM(m, n)$ 可能调用到第二个参数远大于 n 的函数值, 因此先填整个数组后计算具体值.
- 想到本程序可能出现的严重溢出问题并为此忧心.

如果能不看答案独立完成本题并做到以上 5 点, 建议读者不要再看递归这个模块, 你已经掌握的够好了. 对于本题有更深兴趣的读者可以自行百度“阿克曼函数”.

11. A: $i < \text{Loc} - 1$

B: $\text{DelFr} \rightarrow \text{Next}$

C: $\text{DelLoc} \rightarrow \text{Next}$

D: $\text{free}(\text{DelLoc});$

本题考察链表的基本应用. 要求熟悉单链表的基本操作, 并且能够注意操作中的细节.

删去单链表的结点, 应当先定位到待删除的前一个结点, 然后获取待删除的结点, 将待删除结点的后一个结点关联到前一个结点的 `Next` 位, 然后释放待删除结点. 因此 `DelFr` 是待删除结点的前一个结点, `DelLoc` 是待删除的结点. 因此可知第四行定位 `DelFr` 的循环应该跑 `Loc-1` 遍. 注意到初始条件是 `i=0`, 因此判断条件是

$i \leq \text{Loc}-2$ 或 $i < \text{Loc}-1$. 基于以上分析, 可以很自然的填出 B, C 处的操作. D 处主要考察编程习惯. 被删除的结点所占的空间应该被释放, 如果没有养成这个习惯则很可能对 D 空无从下手.

12. A: `return m`

B: `gcd(n,m%n)`

本题的程序只有 3 行有效内容, 要填两个空, 难度看似不大实则不小. A 空为 $n = 0$ 时函数的行为, 注意到 A 被执行后接下来的 `else` 不会被执行, 函数会直接退出, 因而 A 处应该结束函数, 也就是给出返回值. 所以, A 处应该直接给出 $n = 0$ 时的 `gcd(m,n)`, 根据数学知识可得此处应该为 A: `return m`. 而 B 处为 $m \geq n > 0$ 时函数的返回值. 这是一个非常一般的情况, 因而可看出没有直接表达的数学形式, 应该采用递归, 利用 `gcd` 函数本身给出. 结合数学知识 (欧几里得辗转相除法) 可得此处为 `gcd(n,m%n)`. 填 `gcd(n,m-n)` 其实也可以, 因为此处保证 $m \geq n$.

事实上本题的函数可以使用递归方式使用一行写出:

```
1 int gcd(int m, int n)
2 {
3     return n==0? m+n: gcd(n,m%n);
4 }
```

6.10 长图片与长代码

6.10.1 顺序表实现代码 SeqList.c

```
1  /* 借助数组实现顺序表 */
2  #include <stdio.h>
3  #include <stdlib.h>
4  typedef int DataType; //为了方便,数据类型抽象为DataType
5  /*定义顺序表结构SeqList*/
6  typedef struct
7  {
8      DataType *data; //表中的数据
9      int length;      //表长
10     int maxlen;      //表最大长度
11 } SeqList;
12 /*声明线性表的功能*/
13 SeqList *CreateList(int len); //创建最大长度为len的线性表
14 int DelList(SeqList *lst); //删除指定线性表
```

```
15 int ClearList(SeqList *lst); //置空线性表
16 int LenList(SeqList *lst); //求表长
17 int GetElement(SeqList *lst, int n, DataType *data); //获取特定位置元素
18 int GetPrior(SeqList *lst, int n, DataType *data); //求元素的前驱
19 int GetNext(SeqList *lst, int n, DataType *data); //求元素的后继
20 int FindElement(SeqList *lst, DataType data, int pos); //求元素从pos起第一次出现的位置
21 int InsertElement(SeqList *lst, int n, DataType Data); //在第n位插入元素
22 int DelElement(SeqList *lst, int n); //删除第n个元素
23 /*实现线性表的功能*/
24 SeqList *CreateList(int len)
25 {
26     SeqList *pList=(SeqList*)malloc(sizeof(SeqList));
27     if (pList==NULL) return NULL;
28     pList->length=0;
29     pList->maxlen=len;
30     pList->data=(DataType*)malloc(len*sizeof(DataType));
31     return pList;
32 }
33 int DelList(SeqList *lst)
34 { //返回-1表示出现异常,返回0表示操作正常
35     if(lst==NULL) return -1;
36     free(lst->data);
37     return 0;
38 }
39 int ClearList(SeqList *lst)
40 {
41     if(lst==NULL) return -1;
42     lst->length=0;
43     return 0;
44 }
45 int LenList(SeqList *lst)
46 {
47     if(lst==NULL) return -1;
48     return lst->length;
49 }
50 int GetElement(SeqList *lst, int n, DataType *Data)
```

```
51 {
52     if(lst==NULL||n<0||n>=lst->length) return -1;
53     *Data=lst->data[n];
54     return 0;
55 }
56 int GetPrior(SeqList *lst,int n,DataType *Data)
57 {
58     if(lst==NULL||n<=0||n>=lst->length) return -1;
59     *Data=lst->data[n-1];
60     return 0;
61 }
62 int GetNext(SeqList *lst,int n,DataType *Data)
63 {
64     if(lst==NULL||n<0||n>=lst->length-1) return -1;
65     *Data=lst->data[n+1];
66     return 0;
67 }
68 int FindElement(SeqList *lst,DataType Data,int pos)
69 { //操作异常返回-1,查找失败返回-2,其余返回初次出现位置
70     if(lst==NULL||pos<0||pos>=lst->length) return -1;
71     for(int n=pos;n<lst->length;n++)
72     {
73         if(lst->data[n]==Data) return n;
74     }
75     return -2;
76 }
77 int InsertElement(SeqList *lst,int n,DataType Data)
78 { //插入失败返回-1,成功返回当前表长
79     if(lst==NULL||lst->length>=lst->maxlen||n<0||n>lst->length)
80         return -1;
81     DataType Data;
82     for(int loc=lst->length;loc>n;loc--)
83         lst->data[loc]=lst->data[loc-1];
84     lst->data[n]=Data;
85     lst->length++;
86     return lst->length;
87 }
88 int DelElement(SeqList *lst,int n)
89 {
```

```

90     //删除失败返回-1,成功返回当前表长
91     if(lst==NULL||lst->length>=lst->maxlen||n<0||n>=lst->length
    )
92         return -1;
93     for(int loc=n;loc<lst->length-1;loc++)
94         lst->data[loc]=lst->data[loc+1];
95     lst->length--;
96     return lst->length;
97 }

```

SeqList.c

6.10.2 链表实现代码 LinkList.c

```

1  /* 链表的实现 */
2  #include <stdio.h>
3  #include <stdlib.h>
4  typedef int DataType; //为了方便,数据类型抽象为DataType
5  typedef struct /*定义链表结点的结构*/
6  {
7      DataType data;
8      Node *next;
9  } Node;
10 typedef struct /*定义链表的结构*/
11 {
12     Node *head;
13     int length;
14 } LinkList;
15 /*定义链表的功能,简便起见省区声明部分*/
16 LinkList *CreateList()//创建空链表
17 {
18     LinkList *NewList=(LinkList*)malloc(sizeof(LinkList));
19     NewList->head=NULL;
20     NewList->length=0;;
21 }
22 int ClearList(LinkList *lst)//置空链表
23 {
24     //注意不允许使用递归方法或使用栈.原因是递归栈/自己开的栈要求
    //相当于链表长度那么长的连续空间,有这么长的连续空间为什么要

```

使用链表而不直接使用顺序表？

```
25     Node *p=lst->head,*q=lst->head;
26     while(p)
27     {
28         p=p->next;
29         free(q);
30         q=p;
31     }
32     lst->head=NULL;
33     lst->length=0;
34     return 0;
35 }
36 int DelList(LinkList *lst)//删除指定链表
37 {
38     ClearList(lst);
39     free(lst);
40     return 0;
41 }
42 int LenList(LinkList *lst)//求表长
43 {
44     return lst->length;
45 }
46 int GetElement(LinkList *lst,int pos,DataType *data)
47 { //获取特定位置元素
48     if(pos<0||pos>=lst->length)return -1;//方便起见从0开始数
49     Node *p=lst->head;
50     for(int i=0;i<pos;i++)p=p->next;
51     data=&(p->data);
52     return 0;
53 }
54 int GetPrior(LinkList *lst,int pos,DataType *data)
55 { //求元素的前驱
56     if(pos<=0||pos>=lst->length)return -1;
57     return GetElement(lst,pos-1,data);
58 }
59 int GetNext(LinkList *lst,int pos,DataType *data)
60 { //求元素的后继
61     if(pos<0||pos>=lst->length-1)return -1;
62     return GetElement(lst,pos+1,data);
```

```

63 }
64 int FindElement(LinkList *lst,DataType data,int pos)
65 { //求元素从pos起第一次出现的位置
66     if(pos<0||pos>=lst->length) return -1;//输入数据无效返回-1
67     Node *p=lst->head;
68     for(int i=0;i<lst->length;i++)
69     {
70         if(p->data==data||i==pos) return i;//找到返回位置
71     }
72     return -2;//找不到返回-2
73 }
74 int InsertElement(LinkList *lst,int pos,DataType Data)
75 { //在第pos位插入元素
76     if(pos<0||pos>lst->length) return -1;//位置非法,插入失败
77     Node *p=lst->head;
78     for(int i=1;i<lst->length;i++) p=p->next;
79     Node *q=(Node*)malloc(sizeof(Node));
80     q->data=Data;
81     q->next=p->next;
82     p->next=q;
83     return 0;
84 }
85 int DelElement(LinkList *lst,int pos)//删除第n个元素
86 {
87     if(pos<0||pos>=lst->length) return -1;
88     Node *p=lst->head;
89     for(int i=1;i<lst->length;i++) p=p->next;
90     Node *q=p->next;
91     p->next=q->next;
92     free(q);
93     return 0;
94 }

```

LinkList.c

6.10.3 数组实现栈的代码 Stack.c

```

1  /* 借助数组实现栈 */
2  #include <stdio.h>

```



```
3  #include <stdlib.h>
4  typedef int DataType; //为了方便,数据类型抽象为DataType
5  typedef struct
6  {
7      DataType* Bottom;
8      DataType* Top;
9      int Length;
10     int MaxLength;
11 } Stack;
12 /* 功能实现,只实现创建,清空,删除,入栈,出栈五个功能 */
13 Stack* CreateStack(int maxlen)//创建栈
14 {
15     Stack *lst=(Stack*)malloc(sizeof(Stack));
16     if(lst==NULL)return NULL;
17     lst->MaxLength=maxlen;
18     lst->Length=0;
19     lst->Bottom=lst->Top=(DataType*)malloc(maxlen*sizeof(
20     DataType));
21     return lst;
22 }
23 int ClearStack(Stack *lst)//清空栈
24 {
25     lst->Top=lst->Bottom;
26     lst->Length=0;
27     return 0;
28 }
29 int DeleteStack(Stack *lst)//删除栈
30 {
31     free(lst->Bottom);
32     free(lst);
33     return 0;
34 }
35 int Push(Stack *lst,DataType data)//入栈
36 {
37     if(lst->Length==lst->MaxLength)return -1;//栈满
38     *(lst->Top)=data;
39     lst->Top++;
40     lst->Length++;
41     return 0;
```

```
41 }
42 int Pop(Stack *lst,int pos)//出栈
43 {
44     if(lst->Length==0)return -1;//栈空
45     lst->Top--;
46     lst->Length--;
47 }
```

Stack.c

6.10.4 数组实现队列的代码 Queue.c

```
1  /* 借助数组实现队列 */
2  #include <stdio.h>
3  #include <stdlib.h>
4  typedef int DataType; //为了方便,数据类型抽象为DataType
5  typedef struct
6  {
7      DataType *First;
8      DataType *Last;
9      DataType *Front;
10     DataType *Back;
11     int Length;
12     int MaxLength;
13 } Queue;
14 /* 功能实现,只实现创建,清空,删除,入队,出队五个功能 */
15 Queue* CreateQueue(int maxlen)//创建队列
16 {
17     Queue *lst=(Queue*)malloc(sizeof(Queue));
18     if(lst==NULL)return NULL;
19     lst->MaxLength=maxlen;
20     lst->First=lst->Front=lst->Back=(DataType*)malloc(maxlen*
sizeof(DataType));
21     lst->Last=lst->First+maxlen;
22     return lst;
23 }
24 int ClearQueue(Queue *lst)//清空队列
25 {
26     lst->Front=lst->Back=lst->First;
```

```
27     lst->Length=0;
28     return 0;
29 }
30 int DeleteQueue(Queue *lst)//删除队列
31 {
32     free(lst->First);
33     free(lst);
34     return 0;
35 }
36 int Push(Queue *lst,DataType data)//入队
37 {
38     if(lst->Length==lst->MaxLength)return -1;//队满
39     *(lst->Back)=data;
40     lst->Back++;
41     lst->Length++;
42     lst->Back=lst->First+(lst->Back-lst->First)%lst->MaxLength;
43     return 0;
44 }
45 int Pop(Queue *lst,int pos)//出队
46 {
47     if(lst->Length==0)return -1;//队空
48     lst->Front++;
49     lst->Length--;
50     lst->Front=lst->First+(lst->Front-lst->First)%lst->
    MaxLength;
51 }
```

6.10.5 冒泡排序流程图

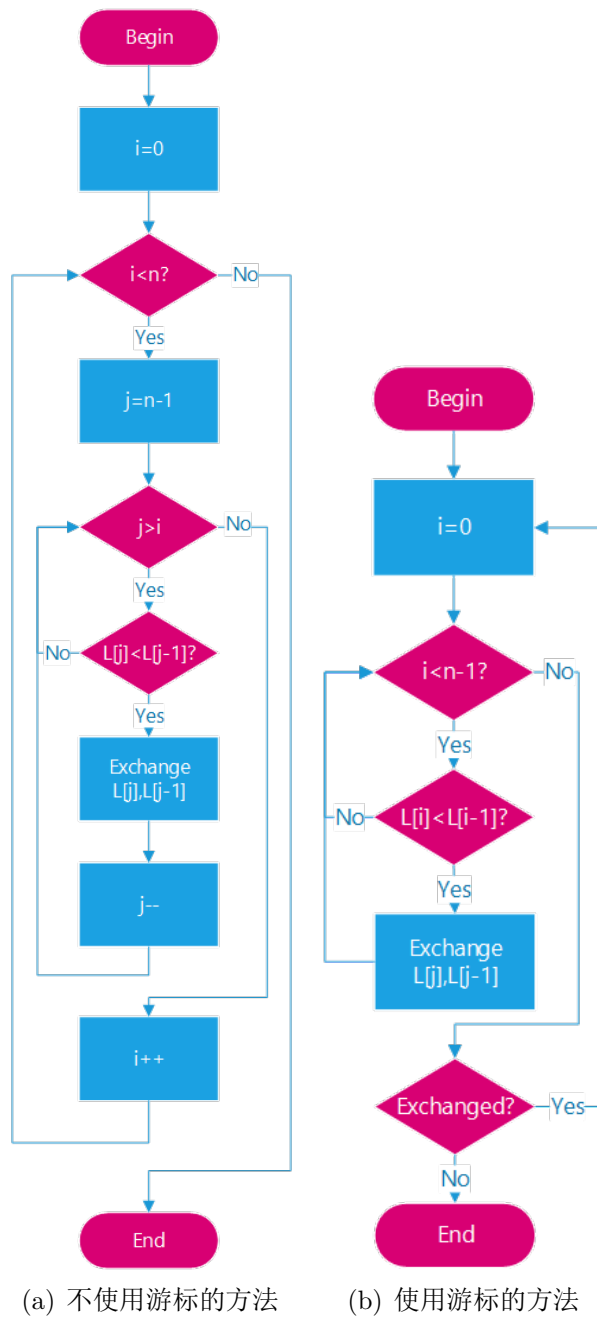


图 6.6: 冒泡排序步骤流程图