

大计基笔记

PKstu

彭康书院学业辅导与发展中心出品



XJTU彭康学导团编制

编撰：彭康学导团大计基小组

彭小帮2.0 QQ群：397499749

彭小帮2.1 QQ群：612354889

写在前面

本笔记为彭康学导团编写，通过对课本知识的精简与整理，为学习大学计算机基础的同学提供参考与借鉴

本笔记的编写者为材料2201冯俞杰，机械2213林如佳，能动2201徐天珍，在此感谢他们的贡献。

需要注意的是，为帮助更多同学更好的学习与理解这门课程，本笔记不是面向应试的，也就是说，本笔记的内容可能会有许多在考试中不做要求，甚至超出课本的。因此如果你抱着速通考试的心态阅读此笔记，有不少内容可以省略，为此我们也加入了部分真题和易错点，希望可以帮助大家取得理想的成绩。

本文件是由markdown编写。由于编写时间紧张，水平有限，笔误，错漏在所难免，欢迎各位联系QQ2316040921指正。

1.1 引论

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

微机系统组成	硬件系统（外部设备与主机系统的关系是信息交换	主机系统：CPU（核心），存储器，输入输出接口（IO接口），总线 外部设备：键盘鼠标扫描仪/显示器打印机/软驱（被淘汰掉了）、硬盘（盘片+驱动器一体）、光驱（光盘驱动器、（输入/输出/存储设备）
	软件系统	系统软件：（1）操作系统（核心）（2）系统应用程序 应用软件：

1.2 硬件系统

1.2.1 主机：CPU 存储器 接口 内存条 主机板

1.2.2 判断的充要条件：

能否与**处理器（CPU)进行直接信息交流（不通过接口）

注意：主机箱≠主机（引入主机板）

1.2.3 外设=输入 + 输出

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。（注意：一个设备可以分兼两职，磁盘写入数据时为输出设备，读取设备为输入设备）

1.3 软件系统：系统 + 应用

1.3.1 功能：管理、监控、维护计算机软硬件的软件。

1.3.2 分级：

操作系统（OS）：

存储器管理、文件管理、进程管理、设备管理等。运行的一切系统软件与应用程序依赖于OS的支持。

系统应用程序：

语言及处理程序、监控管理程序、支撑软件

1.3.3 总结

软件系统的核心是系统软件，系统软件的核心是操作系统OS。

1.4 主机

1.4.1 CPU

定义：

中央处理器(微处理器)，提供运算和控制功能
(复习：相当于图灵机中的读写头)

结构

寄存器，运算器，控制器，与片内总线

ALU运算器：

算术逻辑单元：完成算术计算与逻辑计算（例如：与或非逻辑）

CU控制器：

指挥控制中心

REGISTERS内部寄存器组:

暂时存储数据的单元。避免 CPU 频繁访问存储器, 缩短运行时间。一个专用的寄存器: PC寄存器, 程序计数器。

1.4.2 存储器

定义: 存放数据, 完成**读&写**两个基本操作

读: 取出数据, 原有数据不挂失

写: 保存数据, 原有数据被覆盖

(复习, 微机系统组成中的外设, 包括输入与输出两个功能)

内存存储器

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 (只读存储器) 在断电时数据不丢失; RAM (随机存储器) 在断电时数据会丢失.

高速缓冲存储器 (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

外存储器

外存: 属于外部设备

1. 记录面=磁头数
2. 磁道: 记录面上的同心圆
3. 扇区: 每个磁道划分为扇区
4. 读写方式: 磁头径向进退找磁道, 盘片旋转读扇区。

存储容量计算

硬盘的存储容量=**磁头数×柱面数×扇区数×扇区容量**

一般每扇区容量是: 512B (字节) (复习, 位是bit)

1.4.3 内存条

以芯片的形式固定在一个条形电路板上。

存储单元

定义：内存按照单元组织；

1. 每个单元大小为一个字节
2. 每个单元都对应一个地址，以实现单元内容的寻址

内存容量

来源：地址码（用二进制表示）地址码长度体现内存容量。

位和字节：

位：

位 (bit)是存储的最小数据单位。

字节：

B，基本的数据单位，一个字母占据1B,一个汉字占据2B。

换算：

$1B=8bit$

$1kB=2^{10}B(1024, \text{全都用}1024!)$

$1MB=2^{10}KB=2^{20}B$

$1GB=2^{10}mb=2^{30}B$

1.5 I/O 接口

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

1.6.1 芯片

典型芯片组：南桥&北桥芯片

北桥芯片：

北桥芯片是芯片组的核心，主要负责处理CPU、内存存储器和显卡三者之间的“交通”。发热量较大，故加装散热片。

南桥芯片：

主要负责硬盘等存储设备与PCI(外围器件互联)之间的数据流通。

BIOS芯片（了解）

是可读写的只读存储器。系统BIOS其实是一种ROM芯片（断电自存），BIOS安装了一个输入输出系统程序，这是计算机出厂时候唯一安装存在的程序，用来完成上电自检，初始化，系统设置等初始功能。

1.6.2 扩展槽

分类

1. **内存插槽**：用于安装内存存储器（内存条）
2. **总线接口插槽**：用来装接口，是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，他们是可插拔的。

1.6.3 总线

分类：

1. **按层次结构**，可分为：CPU总线，系统总线，外设总线。
2. **按照传送信息类型**，分为：地址总线、数据总线、控制总线。

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、控制芯片组，I/O接口芯片以及其他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 可计算性

计算机的计算定义：

计算是对信息的变换

计算是一个系统完成了一次从输入到输出的转换

计算是按照一定的、有限的规则和步骤（算法），将输入转换为输出的过程。

计算的必要条件：

能够被图灵机完成的工作。

图灵机是计算机的理论模型

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
- 状态寄存器：图灵机与计算机都具有内部状态

这就延伸到了计算机的基本结构：微机系统组成

可计算性

算法：使计算机明确按照什么样的方法和步骤完成某项任务。

计算机能不能完成一项任务，取决于能不能设计出完成这项任务的算法。

研究计算的一般性质的数学理论，研究哪些是可计算的，哪些是不可计算的。

由于计算是与算法联系在一起的，因此，可计算性理论也称算法理论。

不可解决性

不可解决性反映在两个方面：

1. 一是不能在有限步骤内被解决；
2. 二是虽然有可能解决，但因过于复杂而不能在可接受的时间内解决

1.8 图灵机

图灵机不是一种具体的机器，而是一种思想模型

组成：

一条无限长的纸带Type，一个读写头Head，一套控制规则Table，一个状态寄存器

工作过程：

1. 读写头从纸带一个单元格读取一个信息

2. 根据当前信息查规则表
3. 由规则表做下一步动作

对应计算机术语：

-内部状态-----指令

-规则-----程序

-程序、指令、输出、输入都用二进制编码表示

五元组

表述

$M = (Q, \Sigma, \delta, B, H)$

Q：有穷个状态集合

Σ ：输入符号的集合

Δ ：控制规则集合

B：初始状态。有 $B \in Q$

H： $H \in Q$ ，停机状态。当控制器内部状态为停机状态时，图灵机结束计算。

理解：

对于状态，为Q，而输入的符号，即为 Σ ，而控制规则则为 Δ ，对于初始状态，则记为B&H。

1.9 习题

1. 计算机系统的主要资源有 (A)

- A. 处理器、存储器、I/O 设备、程序和数据
- B. 处理器、存储器、接口、程序和数据
- C. 处理器、存储器、接口、外部设备
- D. 处理器、存储器、I/O 设备、用户接口

2. 以下哪种说法是错误的 (A)

- 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

3. 微型计算机系统的概念结构由 (A) 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

4.计算机的字长是指 (C)

- A.计算机所能处理的数据的位数
- B.数据总线的位数
- C.一次能处理的二进制数据的位数
- D.存储器地址的位数

5.微处理器主要包含了 (B)等几个功能部件

- A.总线、计算器、控制器、存储器
- B.寄存器、运算器、控制器、片内总线 (内部总线)
- C.程序计数器、指令寄存器、指令译码器、操作控制电路
- D.运算器、移位器、数据暂存器、控制器

6.存储容量为 1KB 的存储器具有 (C)

- A.1000 个存储单元
- B.1000 个二进制元
- C.1024 个存储单元
- D.1024 个二进制元

7.计算机主板上的北桥芯片 (A)

- A.负责 CPU、存储器、显示接口之间的数据交换
- B.负责系统总线和 I/O 接口之间的数据交换
- C.负责 CPU 和南桥之间的数据交换
- D.负责 CPU 和 I/O 接口之间的数据交换

8.I/O 接口是指 (D)

- 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

9.外部设备 (D)

- A.只能通过总线连接到主机
- B.可以通过芯片连接到主机
- C.可以直接与主机的系统总线连接
- D.必须通过输入输出接口连接到主机

10.在 PMA 方式的 I/O 数据传输过程中(A)

- A.控制数据传输过程的是专用的硬件
- B.控制数据传输过程的是 CPU

C.控制数据传输过程的是存储器

D.控制数据传输过程的是总线

2.1 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.

(1) 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)

(2) 算术运算

加法规则: $0+0=0$ $0+1=1$ $1+0=1$ $1+1=10$

乘法规则: $0\times 0=0$ $0\times 1=0$ $1\times 0=0$ $1\times 1=1$

2.1.1 定点与浮点

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 小数部分的数在用这种方式存储时精度很容易受损.

特点: 范围小, 精度低, 节省硬件 (定点数不考)

浮点表示

数中的小数点位置不定, 可采用阶码 E (即指数值, 为带符号的整数) 和尾数 F (通常是纯小数) 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.

溢出现象: 运算的数字超出字长时, 会产生溢出. (不考) $11011B * 2^4$, 阶码 4 的

2.1.2 运算规律

加法运算规则

【例2-10】计算10110110B和01101100B的和。

$$\begin{array}{r} \text{进位} \quad 1\ 1111000 \\ \text{被加数} \quad 10110110 \\ \text{加数 +)} \quad 01101100 \\ \hline 1\ 00100010 \end{array}$$

减法运算规则

运算规则：

0-0=0 1-0=1 1-1=0 0-1=1 (有借位)

【例2-11】计算11000100B和00100101B的差。

解：

$$\begin{array}{r} \text{借位} \quad 01111110 \\ \text{被减数} \quad 11000100 \\ \text{减数 -)} \quad 00100101 \\ \hline 10011111 \end{array}$$

二进制计算 例如 $101001-011010=001111$ ($41-26=15$) 的运算。

$$\begin{array}{r}
 \bullet \quad \bullet \quad \bullet \\
 7 \quad 4 \quad 3 \quad 2 \quad 3 \\
 \hline
 \begin{array}{cccccc}
 & 6 & 3 & 2 & & \\
 & & 13 & 12 & 12 & \\
 \hline
 4 & 7 & 5 & 6 & 2 & \\
 \hline
 2 & 6 & 7 & 6 & 1 &
 \end{array}
 \end{array}$$

二进制乘法

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

求两个二进制数1100B与1001B的乘积

二进制乘法原理：

$$1111B \times 1111B = 11100001$$

1111

11110

111100

+1111000 二进制相加

11100001

规律1：

乘法在二进制中被极端简化，变成了加法与移位运算。

规律2：

每*2, 相当于乘数向左移动1位

例子: $11\ 101100 \times 0100$

二进制除法

$0/0=0$ $0/1=0$ $1/1=1$

不存在 $1/0$

$$\begin{array}{r} 110.1 \\ 110 \overline{) 100111} \\ \underline{110} \\ 111 \\ \underline{110} \\ 110 \\ \underline{110} \\ 0 \end{array}$$

2.2 数制及其转换

2.2.1 不同进制的表示方法:

(1) 括号右下角加进制数

例如 $(1011)_2$ 、 $(56)_8$ 、 $(987)_{10}$ 、 $(1A3)_{16}$

(2) 数字后加字母

H 十六进制 D 十进制 B 二进制 O 八进制

例如, $1011B$ 、 $56O$ 、 $987D$ 、 $1A3H$

2.2.2 十进制转化为N进制

整数部分: 除N取余数

小数部分: 乘N取整数

例1:将十进制整数25转换为二进制数

通过十进制转二进制，领悟：除N取余数

所以：25D=11001B

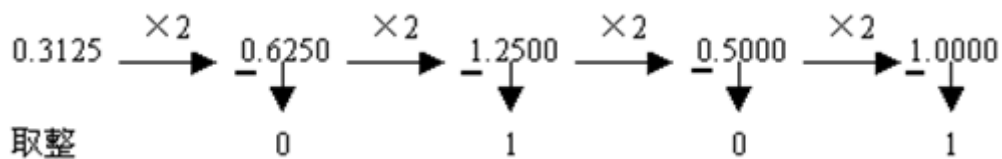
2		25	余数	
2		12	1	—— 最低位
2		6	0	
2		3	0	
2		1	1	
		0	1	—— 最高位

例2: 将十进制小数0.3125转换为二进制数。

采用“乘以2取整”的方法，领悟乘N取整数

用2乘十进制小数，可以得到积，将积的整数部分取出，再用2乘余下的小数部分，又得到一个积，再将积的整数部分取出，如此进行，直到积中的整数部分为零，或者整数部分为1，此时0或1为二进制的最后一位。

所以：0.3125D=0.0101B



具体到题目中所说的(0.787)₁₀（即：0.787D）

0.787*2=1.574==取出整数部分1

0.574*2=1.148==取出整数部分1

0.148*2=0.296==取出整数部分0

0.296*2=0.592==取出整数部分0

0.592*2=1.184==取出整数部分1

0.184*2=0.368==取出整数部分0

$0.368 \times 2 = 0.736$ == 取出整数部分0

$0.736 \times 2 = 1.472$ == 取出整数部分1

故 $(0.787)_{10} = (0.11001001...)_{\text{B}}$

例3: 将十进制整数193转换为十六进制数。

除以“十六取余”

因此, $193_{\text{D}} = \text{C1H}$

N进制转化为十进制数

过程:

==(1) N进制数按权展开

==(2) 展开的算式按十进制运算

运算结果为转换后的十进制数

例 4:将二进制数100110.101转换成十进制数。

$$100110.101_{\text{B}}$$

$$\begin{aligned} &= 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} \\ &= 32 + 0 + 0 + 4 + 2 + 0 + 0.5 + 0 + 0.125 = 38.625_{\text{D}} \end{aligned}$$

例 5: 将十六进制数2BA转换成十进制数。

$$\begin{aligned} 2BA &= 2 \times 16^2 + B \times 16^1 + A \times 16^0 \\ &= 512 + 176 + 10 \\ &= 69 \end{aligned}$$

2.3 机器数

2.3.1 机器数

定义：计算机中存储和处理二进制进制数。

机器数分类

有符号数: 最高位表示数据的正或负

例如一个字节的范围-128~+127

无符号数: 最高位也是绝对值的一部分

例如一个字节的范围0~255

有符号整数的机器数编码

有符号数入门

通过符号位与数值位表示。后面的数值位的意思是**真值**，也即**对应着**绝对值。

+52=0 0110100

-52=1 0110100

在计算机一开始发明的时候，同时处理8位

2.3.2 原码

定义:

最高位符号位，后面是真值。其中，0表示正值，1表示负值。
用 $[X_{\text{原}}]$ 表示原码。

缺点:

这个方法对计算机很麻烦。

1. 在运行程序：判断符号，判断真值，进行运算的过程中，计算机要做的事情太多了。
2. 0的表示存在混淆， ± 0 的表示不同。
原因：00000000/10000000都是0.

2.3.3 反码:

正：反码等于原码

负：符号位不变，然后按位取反。

取-52

反码表示=1 1001011

取0

+0: 00000000

-0: 11111111

但是发现：依旧没有解决0的表示唯一性问题

2.3.4 补码：

正：补码=原码=反码

负：补码=反码+1（复习：反码：符号位不变，数值位按位取反）

取-52

反码：11001011

补码：11001100（复习：二进制的四则运算）

补码的还原：

如果是正的，第一位符号位没的说，后面就是真值，补码=原码=反码

如果是负的，此补码再取补码

0的补码

+0: 00000000

-0: 原码：10000000——反码：11111111——补码100000000

变成了九位，此时1看不到了，0的表示唯一了

补码的作用：

把减法运算转化为加法运算

$$(X - Y)_{\text{补}} = X_{\text{补}} + (-Y)_{\text{补}}$$

例如：66-51

66: 01000010; -51=11001101

相加=100001111，此时头上的1看不到，而00001111=15

例如: -52+116

$X = -52 = -00110100$; $Y = 116 = +1110100$

最后, $X+Y$ 补=01000000, 这也就是 $X+Y$

总结: 取补码运算的程序

首先明确是相减运算, 然分别转化为补码, 进行相加, 如果出现9位溢出, 那么首位放弃。

2.4 编码

2.4.1 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

2.4.2 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.

2.5 计算机网络

按覆盖区域分类

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

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**

TCP/IP 协议及其体系结构

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.

网络应用模式

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

IP地址与端口号

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.

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 Telnet(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)

子网和子网掩码

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

2.6 习题

1. 二进制数 1110111.11 转换成十进制数是(B)

A. 119.125 B. 119.75 C. 119.375 D. 119.3

2. 下列叙述中, 正确的是(D)

A. 汉字的计算机内码就是国标码

B. 存储器具具有记忆能力, 其中的信息在任何时候都不会丢失

C. 所有十进制小数都能准确地转换为有限位二进制小数

D. 所有二进制小数都能准确地转换为十进制小数

3. 十六进制数 FF.1 转换成十进制数是(A)

A. 255.0625 B. 225.125 C. 127.0625 D. 127.125

4. 当二进制数 $X = -10000$ 时, 则有

A. $[X]_{\text{原}} = 100000$ B. $[X]_{\text{反}} = 100001$ C. $[X]_{\text{补}} = 101111$ D. $[X]_{\text{补}} = 110000$

5. 下列二进制运算中, 正确的是(D)

A. $1 * 0 = 1$ B. $0 * 1 = 1$ C. $1 + 0 = 0$ D. $1 + 1 = 10$

6. 为了避免混淆, 八进制数在书写时常在后面加字母(B)

A. H B. O C. H D. B

7. 有一个数值 152, 它与十六进制 6A 数相等, 那么该数值是(B)进制数

A. 二 B. 八 C. 十 D. 十六

8. 数据 111.H 的最左边的 1 相当于 2 的(A)次方

A. 8 B. 9 C. 11 D. 2

9. 用汉语拼音输入 “西安” 两个汉字, 输入 “xi'an” 5 个字符, 那么 “西安” 两字的机内 码所占用的字节数 (B) A. 2 B. 4 C. 8 D. 5

3.1 逻辑运算与逻辑门

3.1.1 “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:

$$a \wedge b = \begin{cases} 1, a = b = 1 \\ 0, ect \end{cases}$$

$$a \vee b = \begin{cases} 0, a = b = 0 \\ 1, etc \end{cases}$$

一位的所有情况:

$$0 \wedge 0 = 0 \quad 1 \wedge 0 = 0 \quad 0 \wedge 1 = 0 \quad 1 \wedge 1 = 1$$

$$0 \vee 0 = 0 \quad 1 \vee 0 = 1 \quad 0 \vee 1 = 1 \quad 1 \vee 1 = 1$$

对于多位的情况, 右对齐按位做与运算即可. 空位补 0.

3.1.2 “非”运算

符号: 数值上面加一条横线, 如 $B = \overline{A}$.

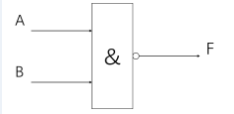
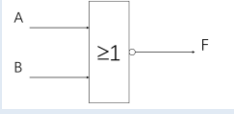

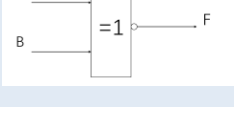
运算法则: 按位取反 (0 转 1, 1 转 0). 一位的情况:

$$\overline{0} = 1$$

$$\overline{1} = 0$$

多位的情况逐位操作即可.

3.1.3 其他逻辑运算

名称	符号	运算规则	真值表	门电路逻辑符号
与非	$\overline{A \wedge B},$	先与 后非	$\overline{0 \wedge 0} = 1, \overline{0 \wedge 1} = 1,$ $\overline{1 \wedge 0} = 1, \overline{1 \wedge 1} = 0$	
或非	$\overline{A \vee B}, \overline{A + B}$	先或 后非	$\overline{0 \vee 0} = 1, \overline{0 \vee 1} = 0,$ $\overline{1 \vee 0} = 0, \overline{1 \vee 1} = 0$	
异或	$A \oplus B = \overline{A} \wedge B + A \wedge \overline{B}$	同0 异1	$0 \oplus 0 = 0, 0 \oplus 1 = 1,$ $1 \oplus 0 = 1, 1 \oplus 1 = 0$	
同或	$\overline{A \oplus B}$	同1 异0	$\overline{0 \oplus 0} = 1, \overline{0 \oplus 1} = 0,$ $\overline{1 \oplus 0} = 0, \overline{1 \oplus 1} = 1$	

3.2 数字电路

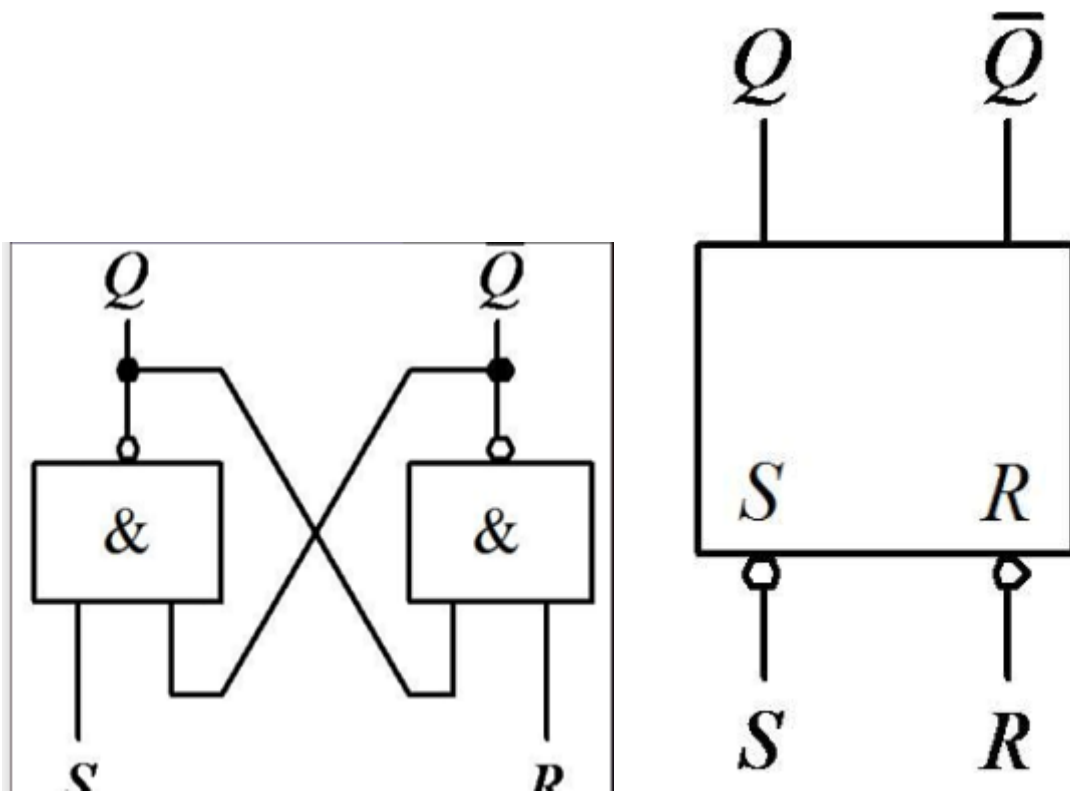
组合逻辑电路特点为任意时刻输出仅仅取决于该时刻输入，与电路原来状态无关
时序逻辑电路任何时刻的稳态输出不仅取决于当前的输入，还与前一刻的输入有

3.2.1 时序逻辑电路

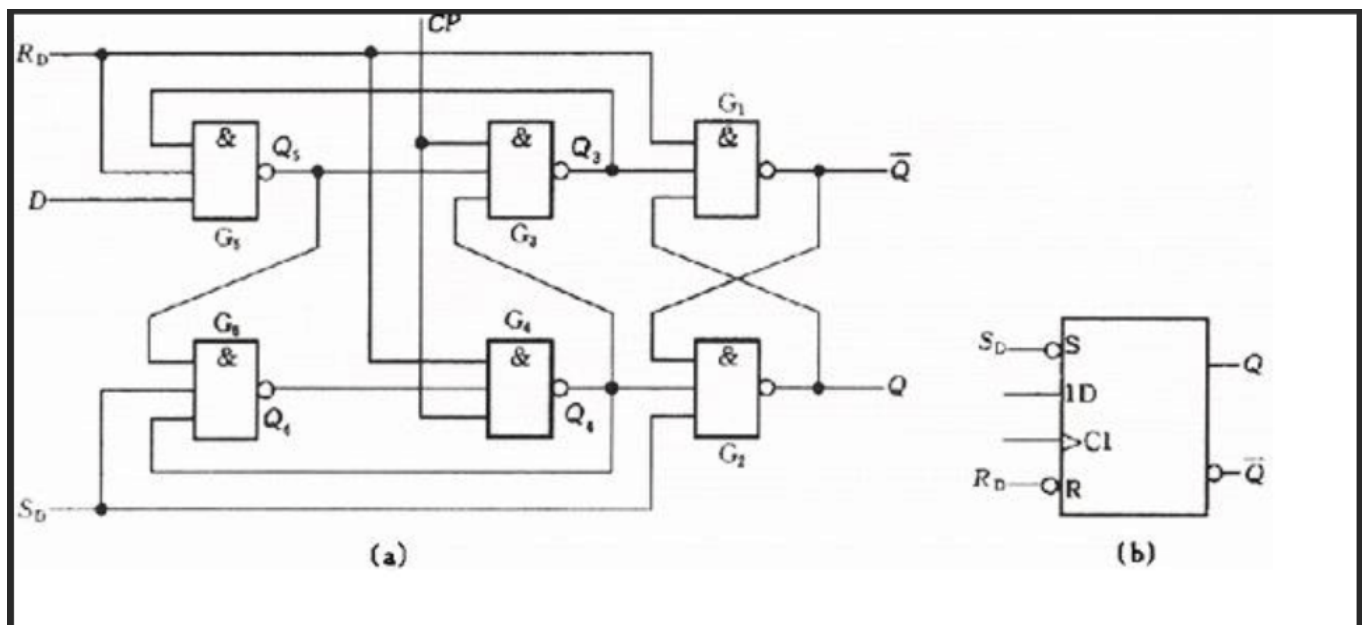
触发器

触发器 (trigger, flip-flop) 也称双稳态多谐振荡器，在外部触发信号的作用下可以改变电压，而当外部触发信号不出现，便可以使输出端状态稳定的保持不变

如图3.5所示的电路中，两个与非门 G_1 ， G_2 门输出端与输入端交叉连接到了一起，构成一种新的功能期间，称为RS触发器



在 RS 触发器基础上增加两个与非门. 下图位 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.



作用

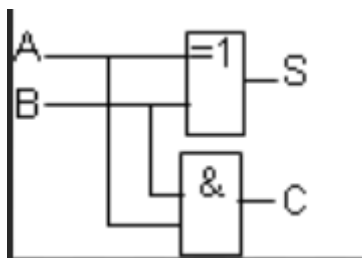
1. 触发器是具有记忆功能的逻辑部件（任何时候输出端都能保持一个确定的稳定状态 (0 或 1)）。
2. 一个触发器能够存储 1 位二进制数。
3. 触发器是构成计算机存储装置的基本单元

真值表：

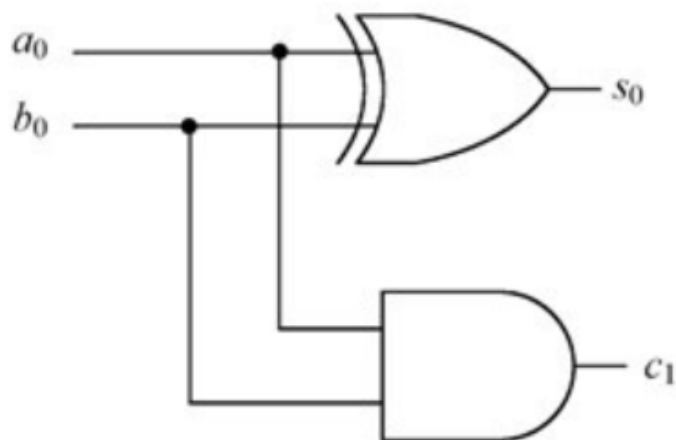
R	S	Q	#Q
0	0	-	-
0	1	0	1
1	0	1	0
1	1	!	!

由触发器引起的思考

一个触发器可以存储一位二进制码，N个触发器可以存储N位二进制码
计算机就是以基本逻辑门为基础，经过层层组合，封装而成



a_0	b_0	s_0	c_1
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



3.2.2数字逻辑电路

加法器

半加器

实现两个1位二进制数相加。有加数和被加数以及“和”与“进位”两个输出，不考虑来自低位进位的加法器

$$S = A \oplus B \quad C = A \wedge B$$

全加器

如果在半加器中添加一个或门来接受低位的进位输出信号，则两个半加器构成一个全加器

如图，CARRY IN是来自低位的进位信号，CARRY OUT是向高位输出的进位信号，SUM是相加的和。



输入			输出	
Ci-1	Ai	Bi	Si	Ci
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
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

http://blog.csdn.net/qq_36148847

多位加法器

全加器只能实现一位的二进制加法，为实现多位二进制数运算，引入多位加法器

连波进位加法器：用 n 个全加器构成，对应低位的全加器将进位信号 C_{out} 连接到高一位全加器的进位端 C_{in} ，并依次像连波一样将信号传递。

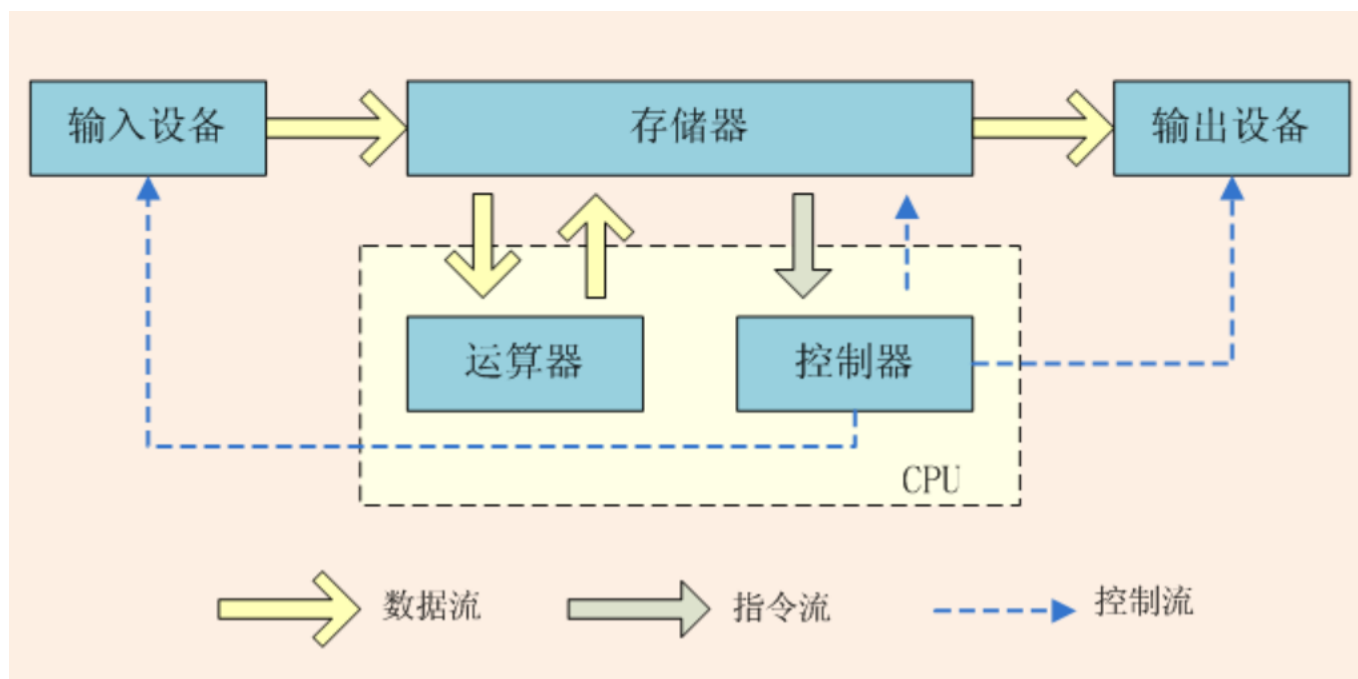
超前进位加法器：目前普遍使用，因篇幅所限，不在进一步描述。

3.3 冯诺依曼结构与原理

3.3.1 冯诺依曼计算机结构

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.



3.3.2 指令与程序

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

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 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

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:

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

3.4 冯诺依曼计算机基本原理

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.

3.4.1 冯诺依曼系统的局限性

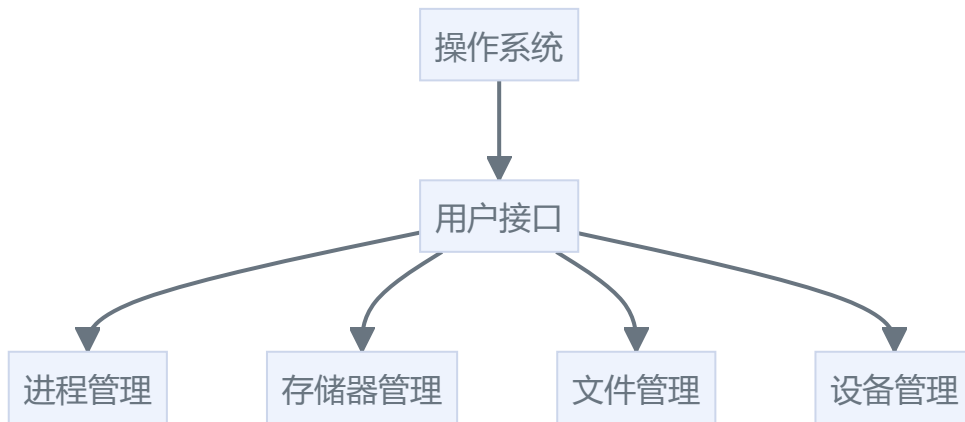
1. CPU 与存储器之间会有大量的数据交互, 造成总线瓶颈.
2. 指令的执行顺序由程序计数器控制, 使得即使有关数据已经准备好, 也必须逐条执 行指令序 列.
3. 指令的执行顺序由程序决定, 对一些大型的, 复杂的任务是比较困难;
4. 以运算器为中心, I/O 设备与存储器间的3数据传送都要经过运算器, 使处理效率, 特 别是对非 数值数据的处理效率比较低.

3.5 操作系统 (OS)

3.5.1 概念

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.

3.5.2 基本功能



3.5.3 进程

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 :

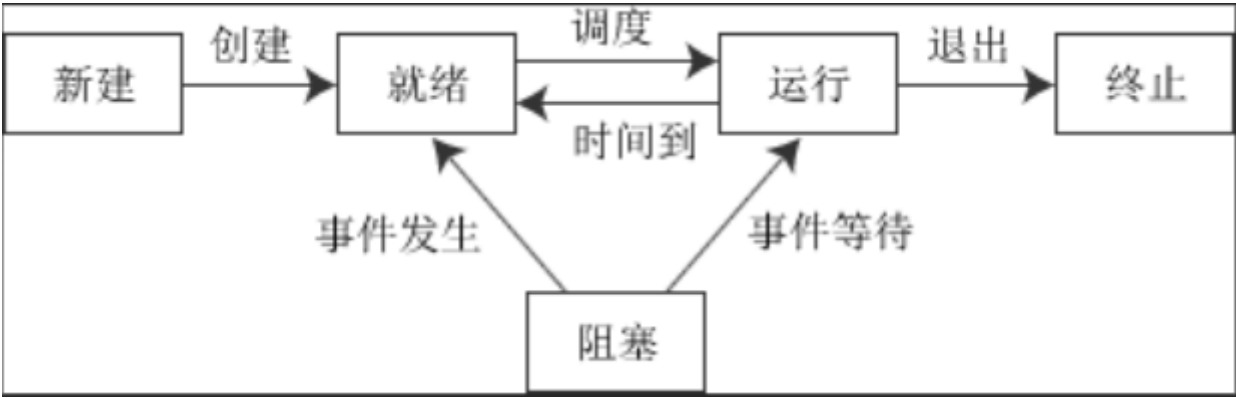
1. 进程是动态的, 程序是静态的: 进程是程序的一次执行, 程序是有序代码的集合
2. 进程是暂时的, 程序是永久的: 进程有生命周期, 会消亡, 程序可长期保存在外存储 器上
3. 进程与程序密切相关: 同一程序的多次运行对应到多个进程; 一个进程可以通过调 用激活多个程序.

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.

进程之间的转换如下图



3.5.4 存储器管理的主要功能

- 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

3.5.5 存储器扩充

- 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
 - ◆ 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.
 - ◆ 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.

4.1 常量与变量

4.1.1 常量

常量的类型	实例	备注
整型常量	10, =20, 0	包括正整数, 负整数和零在内的所有整数
实型常量	3.14, -0.56, 18.0	实型常量既可以称为实数, 也可以称为浮点数
字符型常量	'x','X','?', '0'	用一对单引号括起来的任意字符
字符串常量	"Hello! ", "k88", "9"	用一对双引号括起来的一的或多个字符

注:

1. C程序中的整型常量除了可以用十进制表示之外，还可以用八进制或十六进制表示
2. 科学计数法用e或E来表示以10为底的指数 如 $3.45e-6=0.00000345$

4.1.2 变量

定义变量的一般形式为：

类型关键字 变量名；

【例1.1】 下面先定义整型，实型和字符型三个变量，然后分别为其赋值

```
main()
{
    int a;
    float b;
    char c;
    a=1;
    b=2.5;
    c='A';
}
```

注：任何C程序必须以main（）为开头，它指定了C程序执行的起点，在C程序中只能出现一次。一个C程序必须有且只有一个用main作为名字的函数，这个函数称为主函数。C程序总是从main函数开始执行，与它在程序中的位置无关。main函数的主体部分即程序中的语句用花括号括起来，一般情况下，C语句是以分号结尾的。

变量名的命名规则：

1. 只能由英文字母，数字和下划线开头
2. 标志符必须以字母或下划线开头

注：

1. 变量的定义必须在第一条可执行语句之前完成，否则为一个随机值
2. 注释为`/*和*/`，这种方式可以跨越多行，而`//`只能注释一行
3. 在一条语句中可以定义多个相同类型的变量，多个变量之间用逗号作为分隔符，其书写的先后顺序无关紧要。如：

```
int a,b,c;
```

亦可以为：

```
int a=0,b=0,c=0;
```

但不能写成：

```
int a=b=c=0;
```

4.2 数据类型

由前面所学，每个变量必与一个数据类型相连，下表为一个简单的数据类型分类表

数据类型分类

数据类型分类			关键字	变量声明实例
基本类型	整型	基本整型	int	int a;
		长整型	long	long a;
		短整型	short	short a;
	实型	单精度实型	float	float a;
		双精度实型	double	double a;
	字符型		char	char a;

如何计算数据类型所占内存空间大小

由于同种类型在不同平台所占字节数不尽相同，因此，要想准确计算某种类型数据所占空间的字节数，要用到sizeof（）运算符

【例2.1】

```
printf("int %d\n",sizeof(int));
```

4.3 简单的屏幕输入输出

输入用scanf()
输出用printf()

【例3.1】

```
#include<stdio.h>
main()
{
    int a = 1; float b, c;
    printf("input b\n");
    scanf("%f", &b);
    c = a + b;
    printf("%f", c);
    return 0;
}
```

注：注意到，程序第一行是C的编译预处理命令。这一行将会出现在每一个需要向屏幕输出数据或者从键盘输入数据的程序，尖括号内的文件称为头文件。%f一类为格式字符，%d表示按十进制整型格式输出变量的值，%f表示十进制小数格式，除非特别指明，否则隐含输出6位小数，%c表示输出字符型变量的值。

4.4 运算符

C语言的运算符基本与数学相同，特殊的有以下几项

1. %：求余，即求余数
2. !=：不等号
3. <=：小于等于
4. =：大于等于
5. ==：等于
6. ++：加一
7. --：减一

”

4.5 宏常量与const常量

4.5.1 宏常量

【例5.1】 如下一个求圆的周长和面积的程序

```
#include<stdio.h>
main()
{
    double r;
    double circum;
    double area;
```

```

printf("input r:");
scanf("%lf",&r);
circum=2*3.14159*r;
area=3.14159*r*r;
printf("circumference=%f\n",circum);
printf("area=%f\n",area);
return 0;
}

```

如果用以上的代码来写 π ,不仅麻烦,而且当想要修改精度时十分麻烦,为此我们引入宏常量修改为

【例5.2】

```

#include<stdio.h>
#define PI 3.14159
main()
{
    double r;
    double circum;
    double area;
    printf("input r:");
    scanf("%lf",&r);
    circum=2*PI*r;
    area=PI*r*r;
    printf("circumference=%f\n",circum);
    printf("area=%f\n",area);
    return 0;
}

```

宏常量的一般形式为

```
#define 标识符 字符号
```

为了与源程序中的变量名有所区别,习惯上用字母全是大写的单词来命名宏常量

4.5.2 const常量

宏常量最大的问题是,宏常量没有数据类型,所以极易产生意想不到的错误,为此我们引入了const常量

【例5.3】

```
#include<stdio.h>
main()
{
    const double PI=3.14159
    double r;
    double circum;
    double area;
    printf("input r:");
    scanf("%lf",&r);
    circum=2*PI*r;
    area=PI*r*r;
    printf("circumference=%f\n",circum);
    printf("area=%f\n",area);
    return 0;
}
```

常见错误

实例	描述
(a+b)++	对一个算术表达式用增一运算
#define PI=2.14159	将宏定义当作c语言来使用，在行末加上了分号，或在宏定义后加上了 =
3.5%0.5	对浮点数执行了求余运算
2x^2	将乘法运算符*省略或用^
{a-b}/[a+b]	用放方括号或花括号限定表达式的运算顺序

常见的转义字符

字符	含义
'\n'	换行
'\r'	回车（不换行）
'\"'	双引号
'\''	单引号
'\\'	反斜线
'\?'	问号
'ddd'	ddd表示1~3个八进制数字
'xdd'	dd表示2个十六进制数字

函数printf () 格式转换说明符

说明符	用法
%d	输出带符号的十进制整数，整数的符号省略
%u	无符号的十进制整数输出
%c	输出一个字符
%s	输出字符串
%f	十进制小数
%e	指数形式
%o	八进制输出
%x	十六进制输出、
%%	输出一个百分号

scanf与printf转换说明符相同

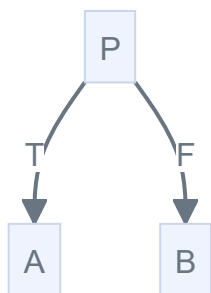
在用scanf () 输入时，遇到以下几种情况都认为输入结束

1. 遇到空格，回车符，制表符 (tab)
2. 达到输出宽域
3. 遇非法字符输入

选择控制结构

if语句

单分支控制的条件语句



结构为

```
if(表达式P) 语句A
```

双分支控制的条件语句

结构为

```
if (表达式B) 语句1  
else 语句2
```

补充：条件运算符

```
表达式1? 表达式2: 表达式3
```

其含义是：若表达式1的值非零，则该条件表达式的值是表达式2的值，否则是表达式3的值
例：

```
#include<stdio.h>  
main()  
{  
    int a,b ,x;  
    printf("Input a,b:");  
    scanf("%d,%d,%d",&a,&b);  
    max = a>b?a:b;  
    printf("max=%d\n",max);  
    return 0;  
}
```

用于多分支控制的条件语句

结构为

```
if (表达式1) 语句1  
else if (表达式2) 语句2  
...  
else if (表达式m) 语句m  
else 语句m+1
```

注意：在if语句中执行的语句如未加花括号只默认未第一行，所以一般推荐无论几行的语句都加上花括号。其次在if后的表达式不能加分号，这点容易忽略。

switch语句

结构为

```
switch(表达式)
{
case 常量1:
    可执行语句序列1
case 常量2:
    可执行语句序列2
...
case 常量n:
    可执行语句序列n
default:
    可执行语句序列n+1
}
```

注意：switch语句相当于一系列if-else语句，注意，常量与case中间至少有一个空格，常量的后面是冒号，常量的类型应与switch后括号内表达式类型一致。表达式只能是int或char型。代码执行完后要使用break来跳出switch语句，如果没有则会以此执行之后的语句。

例：模拟计算器

```
#include<stdio.h>
main()
{
int data1,data2;
char op;
printf("Please enter an expression:");
scanf("%d%c%d",&data,&op,&data2);
switch (op)
{
case '+':
printf("%d+%d=%d\n",data1,data2,data1+data2);
break;
case '-':
printf("%d-%d=%d\n",data1,data2,data1-data2);
break;
case '*':
printf("%d*d=%d\n",data1,data2,data1*data2);
break;
case '/':
if (0==data2)
{
printf("Dividion by zero!\n");
}
else
```

```
    {
        printf("%d+%d=&d\n", data1, data2, data1+data2);
        break;
    }
    break;
default:
    printf("Invalid operator!\n");
}
return 0;
}
```

循环控制结构

while语句

结构为：

```
while (循环控制表达式)
{
    语句序列
}
```

do-while语句

结构为：

```
do
{
    语句序列
}while (循环控制表达式)
```

for语句

结构为：

```
for(初始化表达式; 循环控制表达式; 增值表达式)
{
    语句序列
}
```

例：计算 $\sum 1+2+\dots+n$
用for语句

```

#include<stdio.h>
main()
{
    int i,n,sum;
    printf("Input n:");
    scanf("%d",&n);
    sum=0;
    for(i=1;i<=n;i++)
    {
        sum=sum+i;
    }
    printf("sum=%d\n",sum);
    return 0;
}

```

用while语句

```

#include<stdio.h>
main()
{
    int i,n,sum;
    printf("Input n:");
    scanf("%d",&n);
    sum=0;
    i=1;
    while(i<=n)
    {
        sum=sum+i;
        i++;
    }
    printf("sum=%d\n",sum);
    return 0;
}

```

用do-while语句

```

#include<stdio.h>
main()
{
    int i,n,sum;
    printf("Input n:");
    scanf("%d",&n);
    sum=0;
    i=1;
    do{
        sum=sum+i;
    }
    while(i<=n);
    printf("sum=%d\n",sum);
    return 0;
}

```

```

        i++;
    }while (i<=n);
    printf("sum=%d\n",sum);
    return 0
}

```

常见错误表

错误实例	错误描述
if(a>b); mac=a;	在if语句条件表达式圆括号后加了一个分号
if(a>b) max=a; printf("max=%d\n",a);	在if复合语句中，忘记加花括号
if(a=b)	在if条件表达式中，将相等符号写为赋值
if(a=!b)	将关系运算符两个符号写反
if(a< b< c)	错误的表达方式

a++与++a

a++和++a都是自增运算符，区别是对变量a的值进行自增的时机不同。a++是先进行取值后进行自增；++a是先进行自增后进行取值。此区别主要体现在需要输出自增变量时，但在循环语句中，不需要打印自增的循环变量的值时，两者可互换使用。

例如：

```

int i=15;
printf ("%d\n",++i); //16 i先+1，后取值，输出16
printf ("%d\n",i++); //16 i先取值，输出仍为16，后对i+1
printf ("%d\n",i); //17 经过上一条语句，i变为17，输出17

```

数组

数组是一组相同类型的变量，用一个数组名标识，其中每个变量（称为数组元素）通过该变量在数组中的相对位置（称为下标）来引用。数组可以是一维、二维或更高维的。

一维数组

一维数组的定义格式

和变量一样，数组也遵循先定义后使用的原则。一维数组说明语句格式为：

<类型> <数组名> [<常量表达式>];

其中，<数组名>的构成规则同变量名，<常量表达式>必须用方括号括起来，其值给出数组元素的个数，<类型>（如 int、char、double等）指出数组中元素的数据类型。

例如：

```
int array[10]; //说明了一个有十个元素的整型数组
```

注：

1. 数组元素的下标从0开始编号。例如，array[0]是数组array中的第一个元素。
2. 声明数组大小时，数组大小必须为常量，不能为程序运行过程中通过键盘输入或计算出的数值。
3. 数组的存储空间可以不全部使用完

一维数组的初始化

一般形式为：

<类型> <数组名> [<常量表达式>]={<常量1>,<常量2>,...}

1. 如果在声明数组时给数组的每一个元素都提供初值，就可以不必指定数组大小。例如：

```
double x[5]={1,2,3,4,5};
```

等价于

```
double x[ ]={1,2,3,4,5};
```

2. 可以只给一部分元素赋值。例如：

```
int a[10]={0,1,2,3,4}; /*表示只给前五个元素赋初值，后五个元素初值根据系统的不同而不同(linux和mac中是0)*/
```

3. 如果想使所有元素值为0，可以写成

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

4. 对数组元素集体初始化必须在声明数组时同时完成，不能先声明后初始化。否则只能分别对每一个元素进行初始化（此时可以根据数组元素的特征用循环结构——赋值）。
5. 与一般变量不同，C语言不允许对一个数组进行聚集操作，即不能将整个数组作为一个单元操作。例如，假设数组a和b是相同类型和大小的数组，如果想将数组a的值赋给数组b,下面

的语句是错误的：

```
b=a; //不合法的语句
```

要实现这个功能，必须进行对应元素的赋值，一次只能给一个元素赋值。对数组进行操作最常用的方法是通过循环处理。

6. `printf("a");` //输出数组中第一个元素 (`a[0]`) 的地址

7. 当用数组编程时，如需要修改变量的个数，要修改所有的数工作量太大，所有经常引入宏定义
例：给一维数组输入七个整数，找出数组中最大的数

算法分析：假设算法中第一个数最大，赋值给变量big，将其余的数依次与big比较，利用循环找出最大数max

```
#include <stdio.h>
int main()
{
    int array[7];
    printf("Please input an array with seven elements:\n");
    for(int i=0;i<7;i++)
        scanf("%d",&array[i]);
    int big=array[0];
    for(int j=0;j<7;j++)
        if(array[j]>big)
            big=array[j];
    printf("max=%d\n",big);
    return 0;
}
```

二维数组

二维数组定义格式

二维数组用于存放排列成行、成列结构的表格数据，即矩阵形式的数据。与一维数组相比，定义二维数组时，还应给出二维数组的行数和列数。

二维数组说明语句格式如下：

<类型> <数组名> [<常量表达式1>] [<常量表达式2>];

例如：`int matrix[3][4];` //说明了一个3行4列的整型矩阵

二维数组元素的行、列下标值均从0开始，依次加一。上述数组的最左上角元素为`matrix[0][0]`，最右下角元素为`matrix[2][3]`。

二维数组的存储结构

按行存放（行优先）：把第一行按顺序存完再存第二行。

二维数组的初始化

1. 按照二维数组元素的物理存储次序给所有数组元素提供数据值。

例如：`int matrix[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};`

2. 以行结构方式提供各元素数据值。

用花括号按行分组。matrix数组也可表示为：

```
int matrix[3][4]={ {1,2,3,4}, {5,6,7,8}, {9,10,11,12} };
```

允许在为二维数组初始化时省略行下标值，但列下标值不能省略。

例如：`int matrix[][4]={ {1,2,3,4}, {5,6,7,8}, {9,10,11,12} };`

例：将矩阵M置成单位阵

```
include <stdio.h>
int main()
{
    int M[5][5];
    for(int i=0;i<5;i++)
    {
        for(int j=0;j<5;j++)
            M[i][j]=0;
        M[i][i]=1;
    }
    return 0;
}
```

字符串：由N个字符组成的序列

1. 字符串变量由字符数组构成
2. 字符串长度：字符串中包含字符的个数。注意转义字符“\n”只能算作一个字符。字符串结束标识符为'\0'（不表示数字0，而表示ASCII码为0的字符）。
3. 字符串存储时，在内存占用的实际存储字节数比字符串长度值多一个字节，用于存放'\0'。

一维字符数组定义及初始化

字符数组实际上是数组元素类型为char的数组。一下时定义和初始化形式：

```
char name[10];
```

如果省略对数组长度的声明，例如：

```
char str[]={ 'H', 'E', 'L', 'L', 'O', '\0' };
```

那么系统会自动按照初始化列表中提供的初值个数确定数组大小，而：

```
char str[]={ 'H', 'E', 'L', 'L', 'O' };
```

系统将str初始化为一个长度为5的数组，因为存储“hello”至少需要6个单位，而使系统无法将str当作字符串来处理，所以省略对数组长度的声明时，必须人为的在数组的初始化列表中添加'\0'

还可以用如下更简单的方法：

```
char str[6]={ "HELLO" };  
char str[6]="HELLO";  
char str[]="HELLO";
```

二维字符数组定义及初始化

通常，将一个字符串存放在一维字符数组中，将多个字符串存放在二维字符数组中。当用二维字符数组存放多个字符串时，数组第一维的长度代表要存放的字符串的个数，可以省略，但是第二维的长度不能省略，应该按照最长的字符串长度设定数组第二维的长度，例如：

```
char weekday[7][10]=  
{ "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday" };
```

可以写成

```
char weekday[][10]=  
{ "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday" };
```

若字符串太长，无法写在一行中，则可以将其拆分成几个小的片段写在不同行中，例如：

```
char longString[]="This is the first half of the string"  
"and this is the second half";
```

字符型数组输入输出

1. 按c格式符：

```
for (i=0;str[i]!='\0';i++)//借助字符串结束标志'\0'识别字符串结束  
{  
printf("%c",str[i]);  
}
```

2. 按s格式符：

```
scanf("%s",name); //需注意没有&
printf("%s",name);
```

注:

1. scanf 以空格为输入结束标志; gets以回车作为输入结束标志。即若要输入带空格的字符串无法用scanf而要用gets函数。写法为 gets (name)。
2. 输出字符串也可使用puts函数, 如 puts (name), 其相当于 printf("%s\n",name)。注意 puts 函数输出后会自动换行。

例:

```
char name[10];
printf("input your name");
gets(name);
printf("hello\n");
puts(name);
```

字符串处理库函数

C提供了一组用于字符串处理的库函数, 可以完成如下许多常用的字符串操作。

功能	一般形式	功能描述
字符串复制	strcpy(str1,str2)	将str2复制到字符数组str1中
字符串连接	strcat(str1,str2);	将str2添加到str1的末尾
字符串比较	strcmp(str1,str2);	当str1大于str2时, 函数返回值大于0 当str1小于str2时, 函数返回值小于0 当str1等于str2时, 函数返回值等于0
求字符串长度	strlen(str);	返回str实际长度, 即不包括'\0'

因为这些库函数的说明存放在头文件string.h中, 所以如果要在程序中调用这些函数, 应在源程序最前面加上一个文件包含的编译预处理命令:

```
#include<string.h>
```

注: 由于字符串处理函数在学习中属于补充内容, 故仅罗列少数, 读者如有需要, 请自行查找。

利用函数实现模块化程序设计

此笔记参考了《程序设计基础》朱海萍老师的课件

使用函数的目的



1. 使程序清晰、精炼、简单、灵活。
2. 利用函数实现特定的功能。

在简单的程序里，调用函数或许看起来毫无必要；但是，当程序需要实现的功能越来越复杂，调用函数将显示出其优越性。

函数的分类

1. 从用户使用的角度看：
 1. **库函数**：由系统提供的函数，用户可以直接使用。使用前需在程序开头调用库。
 2. **自定义函数**：由用户针对自己的需求编写的实现特定功能的函数。
2. 从函数的形式看：
 1. **无参函数**：在调用函数时，主调函数不向被调用函数传递参数。
 2. **有参函数**：在调用函数时，主调函数通过参数向被调用函数传递数据。

定义函数

定义无参函数

```
//类型名指定函数返还值的类型
类型名 函数名(void)//可以不写void
{
    函数体
}
```

定义有参函数

```
类型名 函数名(形式参数表列)
{
    函数体
}
```

定义空函数

```
类型名 函数名()
{ }
```

调用函数

1. 调用无参函数：
将函数单独作为一个语句，例如：`print_star();`
此时不要求返回值，只要求完成一定的操作。
2. 调用有参函数：
 1. 具有返回值
例如：`c=max(a, b);`
 2. 不具有返回值
例如：`printf("%d", &a);`

返回函数

注意：函数可以有多个return语句，但不表示可以有多个返回值。如若希望从一个函数中得到多个值，可以使用全局变量。

形式参数与实际参数



在定义函数时函数名后面括号中的变量名称为“**形式参数**”（“形参”或“虚拟参数”）。
在调用函数时，函数名后面括号中的变量名称为“**实际参数**”（“实参”）。实参必须要有确定的值。

实参与形参的类型应该相同或者赋值兼容。赋值兼容是指实参与形参类型不同时能按不同类型数值的复制规则进行转换。

例子：

```
#include <stdio.h>

int max(int x, int y);
//声明函数，此函数的目的为返回两数中的较大值
//在函数声明中可以不写变量名: int max(int, int)

int main()
{
    int a, b, c;
    scanf("%d%d", &a, &b);
    c = max(a, b); //a, b为实参
    printf("%d", c);

    return 0;
}

//函数定义
//将函数定义写在主函数之上可以不对函数进行声明
int max(int x, int y) //x, y为形参。不可以写为 int x, y
```

```

{
    if (x > y)
    {
        return x;
    }
    else
    {
        return y; //函数的返回值通过return语句获得
    }
}
//对于不带返回值的函数，应定义函数类型为 void

```

注意!!!

实参向形参的值传递为**单向传递**，形参值的改变**不会**影响实参的值，形参在函数结束后立即被释放。

假如想要在调用函数中改变主调函数中的数据，可以在学习指针后进行相关方面的探索。

函数的递归调用

定义：在调用一个函数的过程中又出现**直接或间接地调用该函数本身**，称为函数的**递归调用**。

假如不能很好的理解递归的意思，我们将通过以下的例子加深对递归函数的理解。

例子：用递归方法求n!

```

#include <stdio.h>

int fac(int n)
{
    int f;
    if (n == 0 || n == 1)
    {
        f = 1; // 递归终止条件。递归函数必须具有终止条件！
    } //0!=1, 1!=1
    else
    {
        f = fac(n - 1) * n; //n>1时, n!=n*(n-1)!
    }

    return f;
}

int main()
{

```

```

int n;
int facn;//用来存储n!
scanf("%d", &n);
facn = fac(n);
printf("%d", facn);
return 0;
}

```

数组名作为函数参数

一维数组名作为函数参数

例子：

计算平均数

```

#include <stdio.h>

float average(float array[10])//10可以不写，即array[]
{
    float aver, sum = array[0];
    for (int i = 1; i < 10; i++)
    {
        sum += array[i];
    }
    aver = sum / 10;

    return aver;
}

int main()
{
    float aver, score[10];
    for (int i = 0; i < 10; i++)
    {
        scanf("%f", &score[i]);
    }
    aver = average(score);//以数组名字作为参数
    printf("%5.2f", aver);//两位小数，右对齐，占5个位置
    return 0;
}

```

在本例的倒数第二行出现了5.2f，在此补充一下printf()的格式修饰符

输出域宽m
(m) 为整数

指定输出项所占的列数
当输出数据的宽度小于m时，在域内向右靠齐，左边多余位补空格；

	当输出数据宽度大于m时，按实际宽度全部输出，若m前有前导符0，则左边多余位补0； 若m为负整数，则输出数据向左靠齐
显示精度.n (n)为大于等于0的整数	对于浮点数，指定输出的小数位数 对于字符串，指定从字符串左侧开始截取的字符串个数

多维数组名作为函数参数

在被调用函数中对形参数组定义时可以也仅可以省略第一维的大小说明。

✓ `int array[3][10]; int array[][10];`

✗ `int array[][]; int array[3][];`

在第二维大小相同的前提下，形参和实参数组的第一维大小可以不同。

例子：

求一个3 * 4 矩阵中元素的最大值。

```
#include <stdio.h>
int max_value(int array[][4])//4与实参第二维大小相同
{
    int max;
    max = array[0][0];
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 4; j++)
        {
            if (array[i][j] > max)
            {
                max = array[i][j];
            }
        }
    }

    return max;
}

int main()
{
    int a[3][4] = {{1, 3, 5, 7}, {2, 4, 6, 8}, {15, 17, 34, 12}};
    printf("Max value is %d\n", max_value(a));

    return 0;
}
```


变量

按照作用域区分

作用域：如果一个变量在某个文件或函数范围内是有效的，就称该范围为该变量的作用域。

变量按照作用域区分，有**局部变量**和**全局变量**。

局部变量

三种情况：

1. 在函数的开头定义
2. 在函数内的复合语句内定义
3. 在函数的外部定义

在一个函数/复合语句内部定义的变量只在本函数/复合语句范围内有效，在函数/复合语句外不可使用。以上这些称为“**局部变量**”。

注意：

1. 不同函数中可以使用同名的变量，他们代表不同的变量，互不干扰。
2. 主函数中定义的变量也只在主函数中有效，主函数也不能使用其他函数中定义的变量。

为了便于理解，我们给出以下例子：

```
#include <stdio.h>
int main()
{
    int a, b;
    ...
    {
        int c;
        c = a + b;
        ...
    } //c在此复合语句内有效
    ...
    return 0;
} //a,b在此范围内有效
```

全局变量

在函数之外定义的变量称为**外部变量**，外部变量是**全局变量**（也称全程变量）。

全局变量的有效范围从**定义变量的位置**开始到本源文件结束。可以被本文件中的函数使用。

注意：

1. 在一个函数中改变了全局变量，就会影响其他函数中全局变量的值。
2. 由于函数的调用只能带回一个函数返回值，因此有时可以利用全局变量使得通过函数调用能得到一个以上的值。
3. 非必要不使用全局变量。

按照生存期区分

生存期：如果一个变量值在某一时间而是存在的，则认为这一时刻属于该变量的生存期。

变量按照生存期区分，有**静态存储方式**和**动态存储方式**。

静态存储方式

在程序运行期间由系统分配固定的存储空间的方式。

- ◆ 数据存放在静态存储区中。
- ◆ 全局变量全部存放在静态存储区中，在程序开始执行时给全局变量分配存储区，程序执行完毕就释放。

动态存储方式

在程序运行期间根据需要进行动态的分配存储空间的方式。

- ◆ 数据存放在动态存储区中
- ◆ 在函数调用开始时分配动态存储空间，函数结束时释放这些空间。分配和释放是动态的。

存储类别

在定义和声明变量和函数时，一般应同时指定其数据类型和存储类别，也可以采用默认方式指定（系统默认）。

C的4种存储类别：

- ◆ 自动的（auto）
- ◆ 静态的（static）
- ◆ 寄存器的（register）
- ◆ 外部的（extern）

局部变量的存储类别

自动变量（auto变量）

自动变量：在函数声明时被系统分配存储空间，在函数调用结束时这些存储空间被自动释放。正因如此，在不同的并列语句块内可以定义同名变量，不会相互干扰，因为它们各自占据着不同的内存单元，并且有着不同的作用域。

函数中的**局部变量**，如果不专门声明为static，都是自动变量。

例子：

```
#include<stdio.h>
void Swap(int a,int b);
int main()
{
    int a,b;
    printf("Input a,b:");
    scanf("%d,%d",&a,&b);
    Swap(a,b);
    printf("In main():a=%d,b=%d\n",a,b);
    return 0;
}

void Swap (int a,int b)
{
    int temp;
    temp=a;
    a=b;
    b=temp;
    printf("In Swap():a=%d,b=%d\n",a,b);
}
```

结果如下：

```
Input a,b:15,8
In Swap():a=8,b=15
In main():a=15,b=8
```

可以看出a和b分别为各自语句块中定义的局部变量，虽然同名，但不会互相干扰。

静态局部变量（static变量）

有时希望局部变量的值在函数调用结束后不消失继续保留原值，在下一次函数调用时，该变量为上一次函数调用结束时的值。

这时指定该局部变量为“**静态局部变量**”，用**static**进行声明。

例子：

计算阶乘

```

#include <stdio.h>
int f(int i)
{
    static int f=1; //f保留上次调用结束时的值
    f*=i;
    return f;
}
int main()
{
    for (int i = 1; i <= 5; i++)
    {
        printf("%d!=%d\n", i, f(i));
    }
    return 0;
}

```

输出值：

```

1!=1
2!=2
3!=6
4!=24
5!=120

```

auto变量在定义时不会初始化，所以除非指定初值，否则auto的值为乱码，而static则在第一次进入函数时就被自动初始化为0。

静态变量与全局变量相比，他们的生存周期都是一样的，都是整个程序的运行期间，都被自动初始化为0，但他们们的作用域不同。

注意：虽然静态局部变量一直存在，其他函数仍然不能调用它。

若非必要，不要多用静态局部变量：

用静态存储会多占内存，而且降低程序可读性。当调用次数过多往往会无法确定静态局部变量的当前值。

寄存器变量（register变量）

定义格式为：

```
register 类型名 变量名；
```

存储在CPU的寄存器中，常用于使用频繁的变量，可以避免CPU对存储器的频繁数据访问，使程序更小，执行速度更快

全局变量的存储类别

外部变量（extern变量）

格式为：

```
extern 类型名 变量名
```

extern可以**扩展外部变量的作用域**。比如在某个全局变量声明节点前使用该全局变量；或者将外部变量的作用域从一个文件扩展到程序的其他文件中。

例子：

调用函数，求3个整数中的最大数。

```
#include <stdio.h>

int max();
int main()
{
    extern int A, B, C; //把外部变量的作用域扩展到此
    scanf("%d%d%d", &A, &B, &C);
    printf("max=%d", max());
    return 0;
}
int A, B, C; //定义外部变量A, B, C
int max()
{
    int m;
    m = A > B ? A : B;
    C > m ? m = C : 0;

    return m;
}
```

静态变量（static变量）

将外部变量的作用域限制在本文件中。

以免不同的文件使用相同的变量名产生误用。

地址与指针

地址

可以把计算机的内存储器（简称内存）视作一个巨大的一维数组，每个数组元素就是一个存储单元。就像数组中每个元素都有下标一样，每个内存单元都有一个编号，称为地址，它可以用一个无符号整数来表示。凡是存放在内存中的程序和数据都有一个地址，**可以用它们占用的那片存储单元中的第一个存储单元的地址表示。**

C语言规定：

1. 变量的地址可以用地址运算符&求得。例如，&x 表示变量 x 的地址。
2. 数组的地址，即数组第一个元素的地址，可以直接用数组名表示。
3. 函数的地址用函数名表示。

指针

指针的定义

定义： 某个变量的内存地址称为该变量的指针。因此，用以表示（或存储）不同指针值（亦即地址值）的变量就是指针变量，简称指针。

如上所述，指针是一个变量。因此它也具有变量的几个**要素**：

1. **指针的变量名：**与一般变量的命名规则相同。
2. **指针的变量类型：**它不是指指针自身的类型，而是指针所指向的变量的数据类型。
3. **指针变量的值：**是指针所指向的变量在内存中所处的地址

指针的声明

同样的，指针变量也必须遵循“先声明，后使用”的原则。

指针变量声明形式：

数据类型*指针变量名

例如：

```
int*ptr;
```

定义了一个名为 ptr 的指针，该指针指向一个 int 类型的变量。这里可以将“int *”理解为“* of int”。

指针可以指向任何类型，包括基本类型、数组、函数、对象，甚至也可以指向指针。

指针在定义后必须初始化才能使用，否则结果不确定，会带来严重的错误隐患

指针初始化的一般格式： 指针变量名=数据对象；

举例：

- `int *ptr, i=10; ptr=&i; //指向单个变量`
- `const char*sp="string"; //指向字符串`
- `int a[5], *ap; ap=a; //指向数组`
- `double (*fp)(double); fp=sin; //指向函数`

注意

1. 要使用下面的语句来定义两个具有相同类型的指针变量：`int *pa,*pb;`而不能使用 `int *pa,pb;`。
2. 对于指针变量来说，其运算的基本单位为其指向的数据类型的变量占用的字节数。如指向 `int` 类型变量的指针运算的基本单位为4。

指针运算

指针运算的实质是地址的运算

* 和 & 运算符

“&” 称为取地址运算符，用以返回变量的指针，即变量的地址；

“*” 称为指针运算符，用以返回指针所指向的变量的值。

例如：

```
int a;  
int *p;  
p=&a;  
*p=5; //相当于 int a=5;
```

1. 在上述基础上，若已有 `int *w;`，要使 `w` 也指向 `a`，可以
 1. `w=&a;`
 2. `w=p;` //即指针的赋值运算，将一个指针赋值给另一个指针，结果是两个指针指向一个相同的地址单元。
2. 在上述基础上要使 `a` 的值增1，可以：
 1. `a=a+1;`
 2. `*p=*p+1;`
 3. `*w=*w+1;`

指针变量的运算

1. 指针变量算术运算

指针变量只有加法和减法两种算数运算。

1. 自增++（指针右移一个地址）、自减--运算（指针左移一个地址）。
2. 加、减整型数据（指针右/左移若干个地址）。
3. 指向同一个数组的不同数组元素的指针之间的减法。

指针变量算术运算只进行加减，完成指针移动，实现对不同数据单元的访问操作。对不同的类型，移动的单位长度不同。

对指针变量进行下列算术运算毫无意义：指针间相乘或相除，两个指针相加，指针与浮点型数的加减等。

2. 指针变量比较运算

一般指针的比较常用于两个或两个以上指针变量都指向同一个公共数据对象的情况，如同一个数组中各数组元素的指针之间的比较等。任何指针与空指针（NULL）的比较在程序设计中是必要的，但类型不同的指针之间的比较一般都是没有意义的。

3. 指针变量下标运算

C语言提供了指针变量的下标运算[]，其形式类似于一维数组元素的下标访问形式。例如在声明了指针变量

```
double x,a[100],*ptr=a;
```

之后，也可以使用

```
x=ptr[10];
```

这样的用法，并不表示ptr是一个数组，而只是

```
x=*(ptr+10);
```

的另一种写法。

4. 指针的关系运算

例如：

```
p1==p2
```

指针与数组

指向数组的指针

在C中，指针与数组的关系十分密切，它们都能处理内存中连续存放的一系列数据，数组与指针在访问内存时采用统一的地址计算方法。

由于数组中的元素在内存中是连续排列存放的，所以任何能由数组下标完成的操作都可以由指针来实现。**指向数组地址的指针，称为数组指针。**使用数组指针的主要原因是，程序效率高，执行

速度快。

在数组中，a=数组的首地址，a = &a[0] = &a 只是地址相等 意义不一样，&a是一个数组指针例：

```
int a[]={3,2,1};
int *p = a;
printf("p = %d\n",p);
printf("&a = %d\n",&a);
printf("a = %d\n",a);
printf("&a[0] = %d\n",&a[0]);
printf("&p = %d\n\n",&p);

printf("*p = %d\n",*p);
printf("a[0] = %d\n",a[0]);
```

结果为：

```
p=904090704
&a=904090704
a=904090704
&a[0]=904090704
&p=904090704

*p=3
a[0]=3
```

对数组进行加减运算时 $*(p+i) = p[i] = a[i] = *(a+i)$, $p+i = \&p[i] = a+i = \&a[i]$

```
int a[]={1,2,3};
int *p = a;
printf("*(p+1) = %d\t",*(p+1));
printf("p[1] = %d\n",p[1]);
printf("*(a+1) = %d\t",*(a+1));
printf("a[1] = %d\n\n",a[1]);

printf("p = %d\n",p);
printf("p+1 = %d\n",p+1);
printf("&p[1] = %d\n",&p[1]);
printf("a+1 = %d\n",a+1);
printf("&a[1] = %d\n",&a[1]);
```

结果为：

```
*(p+1)=2    p[1]=2
*(a+1)=2    a[1]=2

p=541035660
p+1=541035664
&p[1]=541035664
a+1=541035664
&a[1]=541035664
```

程序举例 编写一个字符串复制函数 mystrcpy

算法分析

1. 令指针指向字符串1首地址;
2. 将当前地址内容送给字符串2;
3. 字符串1的地址+1;
4. 重复2、3、直到整个字符串复制完毕为止;
5. 用循环语句实现，结束条件是当前值不为\0。

程序

```
#include<stdio.h>
void mystrcpy(char*destin,char*source)
{
    while(*source!=\0)//如果*source==\0则表示原字符串结束
    {
        *destin=*source;//复制字符
        source++;
        destin++;
    }
    *destin=\0;//在新字符串尾部添加一个结束符\0
}
```

这段程序的循环部分也可以写为：

```
while((*destin++=*source++));
```

 新字符串尾部的\0也已经被复制。

注意

在复制字符串时要注意，一定要保证目标数组放得下整个字符串

指向多维数组的指针

以二维数组为例,C语言允许把一个二维数组分解为多个一维数组来处理。设有整型二维数组 a [3][4],则数组a可分解为三个一维数组,即 a[0], a[1], a[2], 每个一维数组含有四个元素。例如a [0] 数组含有a[0][0]、a[0][1]、a[0][2]、a[0][3] 4个元素。

从二维数组角度看，a是二维数组名，所以a代表整个二维数组的首地址，也是二维数组0行的首地址。a+1 表示第一行的首地址，即a[1] 的地址。

借用指针就像拿钥匙开门的比喻，二维数组与指针的结合需要开两扇门，即以往一个*就可以找到内容的形式不再存在

如

```
int a[4][2] = { { 2, 4 }, { 6, 8 }, { 1, 3 }, { 5, 7 } };
printf("a = %p, a + 1 = %p\n", a, a + 1);
printf("a[0] = %p, a[0] + 1 = %p\n", a[0], a[0] + 1);
printf(" *a = %p, *a + 1 = %p\n", *a, *a + 1);
printf("a[0][0] = %d\n", a[0][0]);
printf(" *a[0] = %d\n", *a[0]);
printf(" **a = %d\n", **a);
printf(" a[2][1] = %d\n", a[2][1]);
printf("*(*(a+2) + 1) = %d\n", (*(a + 2) + 1));
```

结果

```
a = 0x0064fd38,
a + 1 = 0x0064fd40 a[0]= 0x0064fd38,
a[0] + 1 = 0x0064fd3c
*a = 0x0064fd38,
*a + 1 = 0x0064fd3c
a[0][0] = 2
*a[0] = 2
**a = 2
a[2][1] = 3
* (* (a+2) + 1) = 3
```

把二维数组a分解为一维数组 a[0]、 a[1]、 a[2] 后，设p为指向二维数组的指针变量，可定义如下：

```
int (*p)[4]; //p是一个指针变量，指向一个包含4个元素的一维数组
```

二维数组指针变量说明的一般形式如下：

类型说明符 (*指针变量名) [长度]

其中

- ◆ “类型说明符”为所指数组的数据类型
- ◆ “长度”为二维数组分解为多个一维数组时，一维数组的长度，也就是二维数组的列数
- ◆ 注意 “(*指针变量名)” 两端的括号不可少，如缺少括号则表示是指针数组，意义完全不同

指针数组

定义： 数组元素是指针的数组

声明： 声明一维指针数组的语法形式与普通数组的声明形式类似，在数组名后加上 **维长** 说明即可：

数据类型* 数组名[常量表达式];

- ◆ 这些指针必须指向同一种数据类型的变量
- ◆ 其中常量表达式指出数组元素的个数
- ◆ 数据类型确定每个元素指针的类型
- ◆ 数组名是指针数组的名称，同时也是这个数组的首地址。

例如

一维指针数组，其中包括10个数组元素：

```
char *ptr[10];    //均为指向字符类型的指针：
```

二维指针数组：

```
int*index[10][2];
```

指针和函数

指针作为函数的参数

当以指针作为形参时，在函数调用过程中实参将值传递给形参，也就是使实参和形参指针变量指向同一内存地址。这样对形参指针所指变量值的改变也同样影响着实参指针所指变量值。通过使用实参和形参指针指向相同的内存空间，达到了参数双向传递的作用。

返回指针的函数

一般来说，函数可以用返回值的形式为调用程序提供一个计算结果，除了 int、double 等简单类型，也可以将一个地址作为函数的返回值。在说明返回值为地址的函数时，要使用指针类型的说明符。

例如

```
char*strchr(char*string,int c);  
char*strstr(char*string1,char*string2);
```

这是两个用于字符串处理的库函数，其返回值均为地址。

前者的功能为在字符串 string 中查找字符 c，如果字符串 string 中有字符 c 出现，则返回字符 c 的地址，否则返回NULL。

后者的功能为在字符串 string1 中查找 string2，如果字符串 string1 中包含子字符串 string2，则返回 string2 在 string1 中的地址（即 string2 中第一个字符的地址，否则返回空指针值NULL。

指向函数的指针

函数本身作为一段程序，其代码也在内存中占有一片存储区域，这些代码中的第一个代码所在的内存地址，称为**首地址**。首地址是函数的入口地址。主函数在调用子函数时，就是让程序转移到函数的入口地址开始执行。

定义 所谓指向函数的指针，就是指针的值为该函数的入口地址。

说明格式 指向函数的指针变量的说明格式为：

<函数返回值类型说明符> (*<指针变量名>)(<参数说明表>);

例如

```
int(*p)();    //p为指向返回值为整型的函数的指针
float(*q)(float,int);    //q为指向返回值为浮点型函数的指针
```

- ◆ 函数名与数组名相似，表示该函数的入口地址，因此可以直接把函数名赋值给指向函数的指针变量。
- ◆ 试比较

```
int*func();    //返回地址的函数
int (*func)();    //指向函数的指针
```

前者说明了一个函数，其返回值为指向整型的指针；
后者说明了一个指向返回值为整型的函数的指针变量。

- ◆ 如果已经将某函数的地址赋给了一个指向函数的指针变量，就可以通过该指针变量调用函数。
- 例如

```
double (*func)(double)=sin;    //说明一个指向函数的指针
double y,x;
x=...;    //计算自变量x的值
y=func(x);    //通过指针调用库函数求x的正弦
```

动态存储分配

静态存储分配： 程序中使用的变量和数组的类型、数目和大小是在编写程序时由程序员确定下来的，因此在程序运行时这些数据占用的存储空间数也是一定的。这种存储分配方法被称为静态存储分配。

静态存储分配的缺点是程序无法在运行时根据具体情况(如用户的输入)灵活调整存储分配情况。例如，无法根据用户的输入决定程序能够处理的矩阵的规模。

动态存储分配

动态存储分配机制为克服上述不便提供了手段。

常用的内存管理函数有三个（头文件stdlib.h）

1. 分配内存空间函数malloc

调用形式：(类型说明符*)malloc(size)

功能：在内存的动态存储区中分配一块长度为"size"字节的连续区域。函数的返回值为该区域的首地址。“类型说明符”表示把该区域用于何种数据类型。(类型说明符*)表示把返回值强制转换为该类型指针。“size”是一个无符号数。

举例：`pc=(char *)malloc(100);`

表示分配100个字节的内存空间，并强制转换成字符数组类型，函数返回值为指向该字符数组的指针，把该指针赋予指针变量pc。

2. 分配内存空间函数calloc

调用形式：(类型说明符*)calloc(n,size)

功能：在内存动态存储区中分配n块长度为“size”字节的连续区域。函数的返回值为该区域的首地址。calloc函数与malloc 函数的区别在于一次可以分配n块区域，并且calloc会将分配的内存初始化为0。

举例：`ps=(struct stu*)calloc(2,sizeof(struct stu));`

其中的 sizeof(struct stu) 是求 stu 的结构长度。该语句的意思是，按 stu 的长度分配2块连续区域，强制转换为 stu 类型，并把其首地址赋予指针 ps。

3. 释放内存空间函数free

调用形式：free(void*ptr);

功能：释放ptr所指向的一块内存空间，ptr是一个任意类型的指针变量，它指向被释放区域的首地址。被释放区应是由malloc或calloc函数所分配的区域。

动态存储分配的变量和数组通过指针来访问。例如

```
int x,*ptr=(int*)malloc(5*sizeof(int));
*ptr=5;
x=*ptr;
```

例 利用动态数组求斐波那契数列的前n项。

程序代码如下：

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int n;
    scanf("%d",&n);
    int *p=(int*)malloc(sizeof(n)*(n+1));    /*n+1个长度为sizeof(n)的元素，
    因为需要p[0]=0来计算第二项，所以要n+1个*/
    if(p==NULL || n<=0)//如果没有申请到内存或数据输入有错误则返回
    {
        printf("Error!\n");
        return 1;//return 0代表程序正常退出，return 1代表程序异常退出
    }
    p[0]=0;
    p[1]=1;
    printf("%d\n",p[0]);
    printf("%d\n",p[1]);
    for(int i=2;i<=n;i++)
    {
        p[i]=p[i-2]+p[i-1];
        printf("%d\n",p[i]);
    }
    free(p);    //释放数组空间
    return 0;
}
```

指向指针的指针

指针也是变量,当然也有地址。其地址也可以使用地址运算符“&”求出。能够存放指针地址的变量也是指针,是“指向指针的指针”。

二级指针的说明方法如下：

<数据类型>\$< 指针变量名 >;** 还有更多级指针，几级就有几个*\$（套娃，但多于二级指针的多级指针应用过于复杂，所以应用不多。

指向指针的指针有以下作用：

1. 用于指针数组的处理。详情请见课本253页的例7-3。
2. 作为函数的参数。要修改作为参数的变量的值，必须使用指针；那么要修改作为参数的指针的值，必须使用指针的指针。
3. 作为函数的返回值。

结构体与指针

指向结构体的指针：使用->访问结构体成员

```
typedef struct StudentType
{
    char id[10];
    double score[5];
    double GPA;
}
Student;
Student xjtuStudent;
Student*ptr=& xjtuStudent;
ptr->score[1]=90;
```

指针的初始化

注意!!!

指针的初始化十分重要，建议大家养成定义完指针就进行初始化的习惯

1. 指针变量的初始化
在定义指针时将指针初始化为NULL（空指针，将指针初始化为NULL相当于将指针初始化为0）是一个很好的编程习惯，这样可以防止该指针变量指向某一个未知的内存区域而产生难以预料的错误。
2. 指针数组的初始化
在声明的同时进行初始化

void和const类型的指针

void

void 类型的指针可以指向任何类型的变量。可以直接对 void 型指针赋值或将其与 NULL 做比较，但在求指针的对象变量的内容，或者进行指针运算之前必须对其进行强制类型转换。

例如

```
int x,y;
void *ptr;
ptr=&x;
y=*((int*)ptr);    //通过void指针求值时要用强制类型转换
```


const

用关键字const修饰一个指针时，根据其位置的不同有不同的涵义。

1. `const int p=&a;` (*const和int位置可交换*)

当把 `const` 放最前面的时候，它修饰的就是 `$$ p`，那么 `* p` 就不可变。`* p` 表示的是指针变量 `p` 所指向的内存单元里面的内容，此时这个内容不可变。其他的都可变，如 `p` 中存放的是指向的内存单元的地址，这个地址可变，即 `p` 的指向可变。但指向谁，谁的内容就不可变。

```
int a=10,b=20;
int const * p=&a;
*p=20; //非法
p=&b; //合法
```

2. `int* const p=&a;`

此时 `const` 修饰的是 `p`，所以 `p` 中存放的内存单元的地址不可变，而内存单元中的内容可变。即 `p` 的指向不可变，`p` 所指向的内存单元的内容可变。

```
int a=10,b=20;
int *const p=&a;
*p=20; //合法，相当于a=20
p=&b; //非法
```

基本概念

算法是计算机处理信息的本质，因为计算机本质上是一个算法来告诉计算机确切的步骤来执行一个指定的任务。

算法是独立存在的一种解决问题的方法和思想。

对于算法而言，实现的语言并不重要，重要的是思想。算法可以有不同的语言描述实现版本。(C,C++,C#,python,Java.....)

五大特性

1. **输入**：具有0个或多个输入
2. **输出**：至少有1个输出
3. **有穷性**：在有限的步骤之后会自动结束而不会无限循环，并且每一个步骤可以在**可接受的时间内**完成
4. **确定性**：算法中的每一步都有确定的含义，**不会出现二义性**

5. **可行性**：算法的每一步都是可行的，也就是说每一步都能够执行**有限**的次数完成

描述方法

1. **自然语言**，不严格
2. **伪代码描述**，用自然语言和计算机语言描述算法
3. **流程图描述**

复杂度评价

◆ 为什么引入复杂度评价？

算法的运行需要消耗时间资源和空间（内存）资源，因此为了衡量一个算法的好坏就要从时间和空间两个角度进行评价，即**时间复杂度**与**空间复杂度**。（二者不可兼得）

时间复杂度

时间复杂度又称**计算复杂度**，是算法**有效性**的量度之一。时间复杂度是一个算法运行时间的相对量度。

时间频度与时间复杂度

◆ 时间频度

一个算法花费的时间与算法中语句的执行次数成正比例，哪个算法中语句执行次数多，它花费时间就多。

一个算法中的语句执行次数称为语句频度或时间频度。记为 $T(n)$ 。

◆ 时间复杂度

在 $T(n)$ 中， n 为问题的规模。

一般情况下，算法中基本操作重复执行的次数是问题规模 n 的某个函数，用 $T(n)$ 表示。

若有某个辅助函数 $f(n)$ ，使得当 n 趋近于无穷大时， $T(n)/f(n)$ 的极限值为不等于零的常数，则称 $f(n)$ 是 $T(n)$ 的同数量级函数。记作 $T(n) = O(f(n))$ ，称 $O(f(n))$ 为算法的渐进时间复杂度，简称时间复杂度。

常见的时间复杂度量级

- ◆ 常数阶 $O(1)$
- ◆ 对数阶 $O(\log N)$
- ◆ 线性阶 $O(n)$
- ◆ 线性对数阶 $O(n \log N)$
- ◆ 平方阶 $O(n^2)$
- ◆ 立方阶 $O(n^3)$

- ◆ k 次方阶 $O(n^k)$
- ◆ 指数阶 $O(2^n)$

上面从上至下依次的时间复杂度越来越大，执行的效率越来越低。

为了更好的理解时间复杂度的概念以及了解时间复杂度量级，以下几个例子。

常数阶 $O(1)$

```
int i = 1;    //1
int j = 2;    //1
int m = i + j; //1
//T(n)=3
//f(n)=1
```

像以上例子，语句执行总次数只与语句条数有关，而与变量大小无关。这样的代码无论有多长，其时间复杂度都可以被表示为 $O(1)$ 。

线性阶 $O(n)$

```
for(i=1; i<=n; i++) //1
{
    j = i;           //n
    j++;             //n
}
//T(n)=2n+1
//f(n)=n
```

像以上例子，在循环中，循环里的代码会被执行 n 遍。因此，执行代码消耗时间与 n 有关。这类代码其时间复杂度都可以被表示为 $O(n)$ 。

对数阶 $O(\log N)$

```
int i = 1;    //1
while(i<n)    //1
{
    i = i * 2; //log2(n)
}
//T(n)=2+log2(n)
//f(n)=log2(n)
```

像以上例子，该循环一共循环了 $\log_2 n$ 次，其中的语句执行次数为 $\log_2 n$ 。因此，该代码的时间复杂度为 $O(\log_2 n)$ 。

线性对数阶 $O(n\log N)$

线性对数阶可以理解为将时间复杂度为 $O(\log N)$ 的代码循环n次，如以下例子：

```
for(m=1; m<n; m++) //1
{
    i = 1;          //m
    while(i<n)      //m
    {
        i = i * 2; //m*log2(n)
    }
}
//T(n)=m*log2(n)+2m+1
//f(n)=nlog2(n)
```

平方阶 $O(n^2)$

如果把 $O(n)$ 的代码再嵌套循环一遍，它的时间复杂度就是 $O(n^2)$ 了。

```
for(x=1; i<=n; x++) //1
{
    for(i=1; i<=n; i++) //n
    {
        j = i;          //n*n
        j++;            //n*n
    }
}
//T(n)=2*n*n+n+1
//f(n)=n*n
```

空间复杂度

◆ 空间复杂度

算法在计算机内执行时所需存储空间的度量，也是问题规模n的函数。
空间复杂度不是程序占用了多少bytes的空间，算的是变量的个数。

◆ 常见的空间复杂度

$O(1)$, $O(n)$, $O(\log n)$, $O(n^2)$

排序算法

排序分为：

◆ 内部排序：排序过程仅在内存中进行（数据量少）

- ◆ **外部排序**：排序过程仅在内外存中进行（数据量少）

本节仅讨论内部排序。

排序算法有很多：

- ◆ 冒泡排序
- ◆ 选择排序
- ◆ 快速排序
- ◆ 插入排序
- ◆ 希尔排序
- ◆ 堆排序
- ◆

本节**必须掌握冒泡排序**，**建议掌握选择排序**，**快速排序**仅作参考。

冒泡排序

思路

将相邻的数据进行比较，不断将最值推到某一边。

步骤

举个例子：

有数组 $a[n]$ ，从小到大排序

1. 从**最左边**的数据开始，相邻数据进行比较交换
2. 第一轮结束， $a[n - 1]$ 为最大值
3. 重复步骤1，**已排序数据不参与比较**（一共要进行n轮比较）

$a[0]$	$a[1]$	$a[2]$	$a[3]$
6	0	3	2
0	6	3	2
0	3	6	2
0	3	2	6
0	3	2	6
0	3	2	6
0	2	3	6

$a[0]$	$a[1]$	$a[2]$	$a[3]$
0	2	3	6
0	2	3	6
0	2	3	6
0	2	3	6
0	2	3	6

代码

```
//仅展示冒泡排序的部分，将冒泡排序编写为一个函数
void BubbleSort(int* arr, int n)
{
    for(int end=n; end>0;end--)
    {
        for (int i = 1; i < end; ++i) /*从最左边开始比较，
        i的初始值一定是个常数! */
        {
            if (arr[i - 1] > arr[i]) //相邻数据比较交换
            {
                int tem = arr[i];
                arr[i] = arr[i - 1];
                arr[i - 1] = tem;
            }
        }
    }
}
```

复杂度

- ◆ 时间复杂度
 - ◆ 最好: $O(n)$
 - ◆ 最坏: $O(n^2)$
- ◆ 空间复杂度: $O(1)$

优化

我们可以对冒泡排序进行优化：

当在一轮排序中没有进行数据的交换，说明排序已经完成，此时可以跳出循环不再进行之后的比较。

```

void BubbleSort(int* arr, int n)
{
    for(int end=n; end>0;end--)
    {
        int flag=0;
        for (int i = 1; i < end; ++i)
        {
            if (arr[i - 1] > arr[i])
            {
                int tem = arr[i];
                arr[i] = arr[i - 1];
                arr[i - 1] = tem;
                flag=1;                //标记此循环进行过数据交换
            }
        }
        if(!flag) break;
    }
}

```

选择排序

思路

每次从待排序列中选出一个最小值，然后放在序列的起始位置，直到全部待排数据排完即可。

$a[0]$	$a[1]$	$a[2]$	$a[3]$
6	0	3	2
0	6	3	2
0	2	3	6
0	2	3	6
0	2	3	6

代码

```

void SelectSort(int* arr, int n)
{
    for(int i=0; i<n; i++)
    {
        int mini=i;                //标记最小值的下标
        for(int j=i+1; j<n; j++)
        {
            if(arr[mini]>arr[j])

```

```

        {
            mini=j;
        }
    }
    //最小值与序列开头值交换
    int temp=arr[mini];
    arr[mini]=arr[i];
    arr[i]=temp;
}
}

```

觉得这样子的代码不好看的同学可以考虑添加swap函数（交换）：

```

void swap(int* a, int* b)
{
    int tem = *a;
    *a = *b;
    *b = tem;
}

```

复杂度

- ◆ 时间复杂度
 - ◆ 最好: $O(n^2)$
 - ◆ 最坏: $O(n^2)$
- ◆ 空间复杂度: $O(1)$

优化

我们可以一趟选出两个值，一个最大值一个最小值，然后将其放在序列开头和末尾，这样可以使选择排序的效率快一倍。

```

//选择排序
void swap(int* a, int* b)
{
    int tem = *a;
    *a = *b;
    *b = tem;
}
void SelectSort(int* arr, int n)
{
    //保存参与单趟排序的第一个数和最后一个数的下标
    int begin = 0, end = n - 1;
    while (begin < end)
    {

```



```

        //保存最大值的下标
        int maxi = begin;
        //保存最小值的下标
        int mini = begin;
        //找出最大值和最小值的下标
        for (int i = begin; i <= end; ++i)
        {
            if (arr[i] < arr[mini])
            {
                mini = i;
            }
            if (arr[i] > arr[maxi])
            {
                maxi = i;
            }
        }
        //最小值放在序列开头
        swap(&arr[mini], &arr[begin]);
        //防止最大的数在begin位置被换走
        if (begin == maxi)
        {
            maxi = mini;
        }
        //最大值放在序列结尾
        swap(&arr[maxi], &arr[end]);
        ++begin;
        --end;
    }
}

```

快速排序(了解)

思路

1. 在数组中选一个基准数（通常为数组第一个）；
2. 将数组中小于基准数的数据移到基准数左边，大于基准数的移到右边（具体见以下步骤）；
3. 对于基准数左、右两边的数组，不断重复以上两个过程，直到每个子集只有一个元素，即为全部有序。

步骤

- 1、选出一个key，一般是最左边或是最右边的。
- 2、定义一个begin和一个end，begin从左向右走，end从右向左走。（需要注意的是：若选择最左边的数据作为key，则需要end先走；若选择最右边的数据作为key，则需要begin先走）。

3、在走的过程中，若end遇到小于key的数，则停下，begin开始走，直到begin遇到一个大于key的数时，将begin和right的内容交换，end再次开始走，如此进行下去，直到begin和end最终相遇，此时将相遇点的内容与key交换即可。（选取最左边的值作为key）

4.此时key的左边都是小于key的数，key的右边都是大于key的数

5.将key的左序列和右序列再次进行这种单趟排序，如此反复操作下去，直到左右序列只有一个数据，或是左右序列不存在时，便停止操作，此时此部分已有序。

代码

```
void QuickSort(int* arr, int begin, int end)
{
    //只有一个数或区间不存在
    if (begin >= end)
        return;
    int left = begin;
    int right = end;
    //选左边为key
    int keyi = begin;
    while (begin < end)
    {
        //右边选小  等号防止和key值相等  防止顺序begin和end越界
        while (arr[end] >= arr[keyi] && begin < end)
        {
            --end;
        }
        //左边选大
        while (arr[begin] <= arr[keyi] && begin < end)
        {
            ++begin;
        }
        //小的换到右边，大的换到左边
        swap(&arr[begin], &arr[end]);
    }
    swap(&arr[keyi], &arr[end]);
    keyi = end;
    //[left,keyi-1]keyi[keyi+1,right]
    QuickSort(arr, left, keyi - 1);
    QuickSort(arr, keyi + 1, right);
}
```

复杂度

时间复杂度： $O(n\log_2 n)$

查找算法

若一组待查找数据中存在给定值，则称查找是**成功**的；否则称查找**不成功**。
待查数据元素的集合称为**查找表**。

常见的查找算法有以下四种：

- ◆ 顺序查找
- ◆ 折半查找/二分查找
- ◆ 插入查找
- ◆ 斐波那契查找

本节，我们只介绍顺序查找和二分查找。

顺序查找

思路

- ◆ 对数据进行逐一对比
- ◆ 检索到指定数据，停止检索，返回下标

代码

```
//顺序查找函数
int SequentialSearch(int * arr, int arrlen, int key)
{
    for(int i=0; i<arrlen ; i++)
    {
        if(arr[i]==key)
        {
            //找到指定数据返回下标
            return i;
        }
    }
    //找不到指定数据，返回-1
    return -1;
}
```

复杂度

时间复杂度： $O(n)$

折半查找（二分查找）

思路

- ◆ 用给定值k先与**中间**结点的关键字比较，中间结点把数组分成两个子集
- ◆ 若相等则查找成功；若不相等，再根据k与该中间结点关键字的比较结果确定下一步查找哪个子集
- ◆ 这样循环进行，直到查找到或查找结束发现数组中没有这样的结点。

注意：二分查找要求数列已排序！

$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[6]$	key
0	2	3	7	9	11	15	9
0	2	3	7	9	11	15	9
0	2	3	7	9	11	15	9

代码

```
//二分查找函数
int binsearch(int * arr, int arrlen, int key)
{
    //记录待查找数组的开头下标和结尾下标
    int begin=0, end=arrlen-1;
    //begin和end最终会相遇
    //循环条件为begin<=end
    while(begin <= end)
    {
        int mid=(begin+end)/2;
        if(arr[mid]==key){
            //假如找到指定数据返回下标
            return mid;
        }else if(arr[mid]<key){
            //为增长数列
            //确定数列的开头下标
            begin=mid+1;
        }else{
            //确定数列的末尾下标
            end=mid-1;
        }
    }
    //找不到指定数据返回-1
    return -1;
}
```

二分查找会有许多微小的变形，要掌握好边界的条件。

复杂度

时间复杂度： $O(\log_2 n)$

关键代码填空

- ◆ 顺序表：
创建表，销毁表，插入，查找
- ◆ 栈：
创建栈，入栈，出栈，判断是否为空栈
- ◆ 队列：
创建队列，入队，出队

”

注：

- ◆ 在以下所有提到的操作中，涉及指针操作均可以用数组进行替代。
- ◆ 使用数组时不需进行malloc操作

数据结构简述

数据结构的定义

1. 数据：是描述客观事物的数和字符的集合
2. 数据项：是具有独立含义的数据最小单位，也称为字段或域
3. 数据对象：是指性质相同的数据元素的集合，它是数据的一个子集。
4. 数据结构：是指所有数据元素以及数据元素之间的关系，可以看作是相互之间存在着某种特定关系的数据元素的集合。

”

数据结构的组成

1. 数据的**逻辑结构**：由数据元素之间的逻辑关系构成
2. 数据的**存储结构**：数据元素及其关系在计算机存储器中的存储表示，也称为数据的物理结构。
3. 数据的运算：施加在该数据上的操作

逻辑结构的类型

1. **集合**：指数据元素之间除了“同属于一个集合”的关系以外别无其他关系
2. **线性结构**：指该数据结构中的数据元素之间存在**一对一**的关系
3. **树形结构**：指该结构中的数据元素之间存在**一对多**的关系
4. **图形结构**：指该结构中的数据元素之间存在**多对多**的关系

存储结构的类型

1. 顺序存储结构

采用一组连续的存储单元存放所有的数据元素。即所有数据元素在存储器中占有一整块存储空间，而且两个逻辑上相邻的元素在存储器中的存储位置也相邻。

优点：存储效率高，通过下标来直接存储

缺点：不便于数据修改，对元素的插入或删除操作可能需要移动一系列的元素

2. 链式存储结构

每个逻辑元素用一个内存结点存储，每个结点是单独分配的，所有的结点的地址不一定是连续的。通过指针域将所有结点链接起来。

优点：便于数据修改，对元素进行插入或删除操作只需修改相应结点的指针域，不必移动结点。

缺点：存储空间的利用率低，不能对元素进行随机存取。

3. 索引存储结构

指在存储数据元素信息的同时还建立附加的索引表。存储所有数据元素信息的表称为主数据表，其中每个数据元素有一个关键字和对应的存储地址。

优点：查找效率高。

缺点：需要建立索引表，增加了空间开销。

4. 哈希（或散列）存储结构

根据元素的关键字通过哈希（或散列）函数直接计算出一个值，并将这个值作为该元素的存储地址

优点：查找速度快

一般适合要求对数据能够进行快速查找和插入的场合

线性表

线性表

线性表：

”

1. 线性表（linear list）是n个具有相同特性的数据元素的有限序列。
2. 线性表在逻辑上是线性结构，但是在物理结构上并不一定是连续的。（这里的物理结构一般指物理地址空间）。

线性表的逻辑特征：

- ◆ 在非空的线性表，有且仅有一个开始结点 a_1 ，它没有直接前趋，而仅有一个直接后继 a_2 ；
- ◆ 有且仅有一个终端结点 a_n ，它没有直接后继，而仅有一个直接前趋 a_{n-1} ；
- ◆ 其余的内部结点都有且仅有一个直接前趋 a_{i-1} 和一个直接后继 a_{i+1} 。
- ◆ 元素个数有限。

- ◆ 线性表是一种逻辑结构，表示元素之间一对一相邻的关系。
- ◆ 线性表顺序存储结构占用一片连续的存储空间。知道某个元素的存储位置就可以计算其他元素的存储位置。

基本操作注意事项：

”

- ◆ 创建线性表时是否创建成功
- ◆ 添加数据时表是否已满
- ◆ 取出/删除数据时表是否为空

顺序表：线性表的顺序存储结构

顺序表：

”

1. 是线性表
2. 物理结构上是连续的
3. 顺序表中任意一个数据元素都可以随机存取

顺序表的特点

- ◆ 利用数据元素的存储位置表示线性表中相邻数据元素之间的前后关系，即线性表的逻辑结构与存储结构一致
- ◆ 在访问线性表时，可以快速计算出任何一个数据元素的存储地址。因此可以粗略地认为，访问每个元素所花时间相等
这种存取元素的方法被称为随机存取法。

顺序表的优缺点

优点：

- ◆ 存储密度大(结点本身所占存储量/结点结构所占存储量)
- ◆ 可以随机存取表中任一元素

缺点：

- ◆ 在插入、删除某一元素时，需要移动大量元素
- ◆ 浪费存储空间
- ◆ 属于静态存储形式，数据元素的个数不能自由扩充

顺序表的分类

1. 静态顺序表：使用定长数组存储
2. 动态顺序表：使用动态开辟的数组存储

顺序表操作

```
SqList * CreatList(int n);           //创建线性表
int InsertToList(SqList * sqlist, int pos, int data); //插入元素
int DeleteFromList(SqList * sqlist, int pos);        //删除元素
int GetPriorElement(SqList * sqlist, int n, int * data); //获取元素前驱
int FindElement(SqList * sqlist, int pos, int data);  //查找元素
void PrintList(SqList * sqlist);                    //输出线性表
void DestroyList(SqList * sqlist);                  //销毁线性表
```

构造

```
typedef struct
{
    int length;    //当前顺序表长度
    int MaxLength; //顺序表最大长度
    int* list;     //指向线性表的指针
}SqList;
```

创建线性表

注意：

返回的是结构体指针！

```
// 创建最大表长度为n的线性顺序表
SqList *CreatList(int n)
{
    SqList *sqlist = (SqList *)malloc(sizeof(SqList));
    // 为线性表分配内存
    if (sqlist != NULL)
    {
        sqlist->list = (int *)malloc(sizeof(int) * n);
        if (sqlist->list == NULL)
        {
            return NULL;
            // 如果分配失败返回NULL
        }
        sqlist->length = 0;
        // 初始化当前顺序表长度为0，即为空表
        sqlist->MaxLength = n;
        // 设置最大表长度
    }
}
```



```
}  
return sqlist;  
// 返回顺序表结构体指针  
}
```

插入元素

在某一位置插入元素后，注意**原该位置及之后**的数据全部要后移一位。

```
// 在线性表的pos位置插入数据data  
int InsertToList(SqlList *sqlist, int pos, int data)  
{  
    if (pos < 0 || pos > sqlist->length || sqlist->length == sqlist->MaxLength)  
        // 假如插入位置不正确或者线性表已满，插入失败  
    {  
        return -1;  
    }  
    for (int n = sqlist->length; n > pos; n--)  
    {  
        sqlist->list[n] = sqlist->list[n - 1];  
        // 自pos起，所有元素后移一位  
    }  
    sqlist->list[pos] = data;  
    // 插入新的元素  
    return ++sqlist->length;  
    // 表长+1，返回表长  
}
```

删除元素

删除某一位置的元素后，注意**原该位置之后**的数据都需要前移一位。

```
//删除线性表位置pos的数据  
int DeleteFromList(SqlList *sqlist, int pos)  
{  
    if (pos < 0 || pos > sqlist->length)  
    {  
        return -1;  
        //假如删除数据位置不正确，删除失败  
    }  
    for (int n = pos; n < sqlist->length - 1; n++)  
    {  
        sqlist->list[n] = sqlist->list[n + 1];  
        //pos+1到末尾的所有元素前移一位  
    }  
    return --sqlist->length;  
}
```

```
//表长-1，返回表长
```

```
}
```

获取元素前驱

即获取该元素前一位的元素。

```
//获取元素前驱
int GetPriorElement(SqlList *sqlist, int n, int *data)
{
    if(n<1 || n>sqlist->length-1)
    {
        return -1;
        //元素位置不正确，获取失败
        //注意第一个元素没有前驱元素
    }
    *data=sqlist->list[n-1];
    //指向前驱元素
    return n-1;
    //返回前驱元素位置
}
```

查找元素

就是普通的顺序查找，只不过换成了结构体、指针和函数来表示。

```
//在线性表中查找下标>=pos中数据data的下标
int FindElement(SqlList *sqlist, int pos, int data)
{
    //顺序查找
    for (int n = pos; n < sqlist->length-1; n++)
    {
        if(data==sqlist->list[n])
        {
            return n;
            //找到数据返回下标
        }
    }
    return -1;
    //查找失败返回-1
}
```

输出线性表

使用函数，按自己需要的格式输出。
以下仅为一个例子。

```
//输出线性表
void PrintList(Sqlist *sqlist)
{
    for(int n=0;n<sqlist->length;n++)
    {
        printf("第%d项:%d\n",n,sqlist->list[n]);
    }
}
```

销毁线性表

释放空间。

```
//销毁线性表
void DestroyList(Sqlist *sqlist)
{
    free(sqlist->list);
    sqlist->length=0;
    //释放空间
}
```

一个完整的例子

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

typedef struct{
    int length;
    int MaxLength;
    int *list;
} Sqlist;

Sqlist *CreatList(int n);
int InsertToList(Sqlist *sqlist, int pos, int data);
int DeleteFromList(Sqlist *sqlist, int pos);
int GetPriorElement(Sqlist *sqlist, int n, int *data);
int FindElement(Sqlist *sqlist, int pos, int data);
void PrintList(Sqlist *sqlist);
void DestroyList(Sqlist *sqlist);

int main(){
    Sqlist *sqlist=CreatList(1000);
```

```

    for(int i=0;i<5;i++){
        InsertToList(sqlist,i,i+1);
    }
    printf("插入值: \n");
    PrintList(sqlist);
    DeleteFromList(sqlist,3);
    printf("删除下标为3的数据: \n");
    PrintList(sqlist);
    int data, sub=GetPriorElement(sqlist,3,&data);
    printf("sqlist中下标3的前驱元素是: %d, 下标是: %d\n",data,sub);
    printf("2在表中的下标是: %d\n",FindElement(sqlist,0,2));
    DestroyList(sqlist);

    return 0;
}

SqlList *CreatList(int n){
    SqlList *sqlist = (SqlList *)malloc(sizeof(SqlList));
    if (sqlist != NULL){
        sqlist->list = (int *)malloc(sizeof(int) * n);
        if (sqlist->list == NULL){
            return NULL;
        }
        sqlist->length = 0;
        sqlist->MaxLength = n;
    }
    return sqlist;
}

int InsertToList(SqlList *sqlist, int pos, int data){
    if (pos < 0 || pos > sqlist->length || sqlist->length == sqlist->MaxLength){
        return -1;
    }
    for (int n = sqlist->length; n > pos; n--){
        sqlist->list[n] = sqlist->list[n - 1];
    }
    sqlist->list[pos] = data;
    return ++sqlist->length;
}

int DeleteFromList(SqlList *sqlist, int pos){
    if (pos < 0 || pos > sqlist->length){
        return -1;
    }
    for (int n = pos; n < sqlist->length - 1; n++){
        sqlist->list[n] = sqlist->list[n + 1];
    }
    return --sqlist->length;
}

```

```

}

int GetPriorElement(SqlList *sqlist, int n, int *data){
    if(n<1 || n>sqlist->length-1){
        return -1;
    }
    *data=sqlist->list[n-1];
    return n-1;
}

int FindElement(SqlList *sqlist, int pos, int data){
    for (int n = pos; n < sqlist->length-1; n++){
        if(data==sqlist->list[n]){
            return n;
        }
    }
    return -1;
}

void PrintList(SqlList *sqlist){
    for(int n=0;n<sqlist->length;n++){
        printf("第%d项:%d\n",n,sqlist->list[n]);
    }
}

void DestroyList(SqlList *sqlist){
    free(sqlist->list);
    sqlist->length=0;
}

```

链表：线性表的链式存储结构

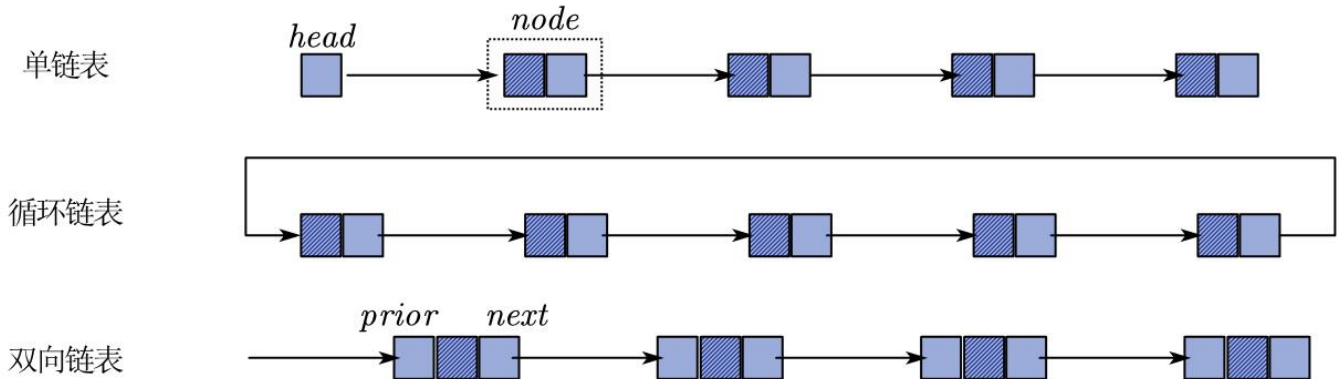
链表特点

- ◆ 结点在存储器中的位置是任意的，即逻辑上相邻的数据元素，在物理上不一定相邻；
- ◆ 线性表的链式表示又称为非顺序映像或链式映像。
- ◆ 用一组物理位置任意的存储单元来存放线性表的数据元素。
- ◆ 这组存储单元既可以是连续的，也可以是不连续的，甚至是零散分布在内存中的任意位置上的。
- ◆ 链表中元素的逻辑次序和物理次序不一定相同。

链表词汇

- ◆ 结点: 数据元素的存储映像。由数据域和指针域两部分组成。

- ◆ 链表: n 个结点由指针链组成一个链表。
它是线性表的链式存储映像, 称为线性表的链式存储结构。
- ◆ 头指针: 是指向链表中第一个结点的指针
- ◆ 首元结点: 是指链表中存储第一个数据元素 a_1 的结点
- ◆ 头结点: 是在链表的首元结点之前附设的一个结点



单链表

单链表是线性表的链式存储。**只能从头节点依次向后遍历。**

单链表是由表头唯一确定, 因此单链表可以用头指针的名字来命名。若头指针名是 L , 则把链表称为表 L 。

在单链表中插入数据需将插入位置前一个数的下一个结点设置为插入数据, 插入数据的下一个结点设为原数据。

队列：操作受限的线性表

限制：仅允许在表的一端进行插入操作, 而在表的另一端进行删除操作。



1. 把进行**插入**的一端成为队的**队尾 (rear)**, 把进行**删除**的一端称为**队头或队首 (front)**。
2. 向队列中插入新元素称为**进队或入队 (enqueue)**, 新元素进队后就成为新的队尾元素
3. 从队列中删除元素称为**出队或离队 (dequeue)**, 元素出队后, 其**直接后继元素**就成为队首元素。(不像顺序表需要将后面所有元素向前移一位)

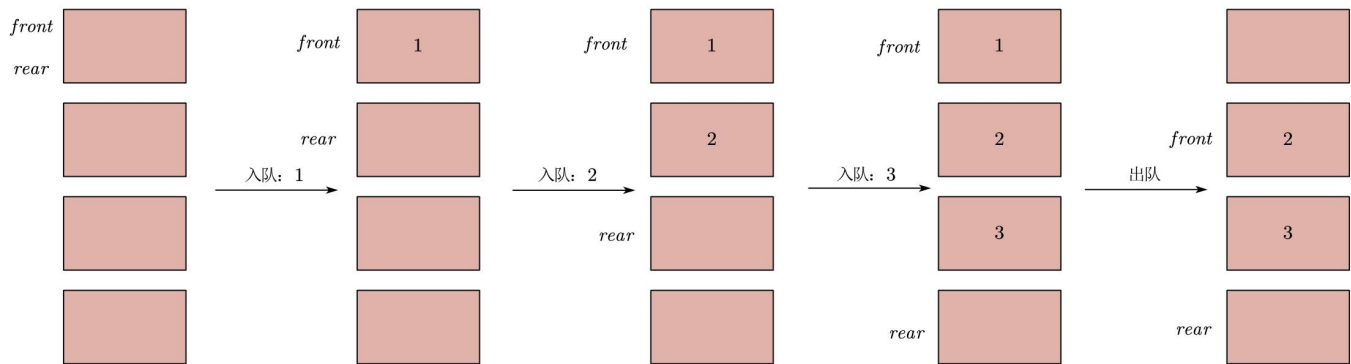
基本操作

```
void InitQueue(Queue *Q);    //初始化
void EnQueue(Queue *Q, int x); //入队
void DeQueue(Queue *Q);      //出队
int IsQueueEm(Queue *Q);     //判断是否为空
void showQueue(Queue *Q);    //打印队列
```

```

int QueueLength(Queue *Q);    //获取队列长度
void ClearQueue(Queue *Q);    //清空队列
void DestroyQueue(Queue *Q);  //销毁队列

```



构造

```

typedef struct Queue{
    int *base;    //队列的存储空间
    int front;    //队头
    int rear;     //队尾
}Queue;

```

初始化

```

//初始化
void InitQueue(Queue *Q)
{
    Q->base=(int *)malloc(sizeof(int)*Maxsize); //开辟存储空间
    if(Q->base==NULL)
    {
        //开辟失败
        return NULL;
    }
    //初始化时，队头和队尾都在0位置
    Q->front=0;
    Q->rear=0;
}

```

判断是否为空

```

//判断是否为空
int IsQueueEm(Queue *Q)
{
    if(Q->front==Q->rear)
    {

```

```
        //为空，返回0
        return 0;
    }
    //不为空，返回1
    return 1;
}
```

入队

```
//入队
void EnQueue(Queue *Q, int x)
{
    //判断是否还有存储空间
    if(Q->rear>=Maxsize)
    {
        return;
    }
    //假如还有存储空间，数据入队
    //队尾下标+1
    Q->base[Q->rear++]=x;
}
```

出队

```
//出队
void DeQueue(Queue *Q)
{
    //判断是否为空
    if(Q->front==Q->rear)
    {
        return;
    }
    //不为空，出队
    Q->front++;
}
```

打印队列

```
//打印队列
void showQueue(Queue *Q)
{
    //遍历
    for(int i=Q->front;i<Q->rear;i++)
    {
        printf("%d ",Q->base[i]);
    }
}
```



```
    }  
    printf("\n");  
}
```

获取队列长度

```
//获取队列长度  
int QueueLength(Queue *Q)  
{  
    return Q->rear-Q->front;  
}
```

清空队列

```
//清空队列  
void ClearQueue(Queue *Q)  
{  
    //队头队尾下标置为0  
    Q->front=0;  
    Q->rear=0;  
}
```

销毁队列

```
//销毁队列  
void DestroyQueue(Queue *Q)  
{  
    //释放队列的存储空间  
    free(Q->base);  
    //将队列空间的位置指针置空  
    Q->base=NULL;  
}
```

一个完整的例子

```
#include <stdio.h>  
#include <stdlib.h>  
  
#define Maxsize 8  
  
typedef struct Queue{  
    int *base;  
    int front;  
    int rear;
```

```

}Queue;

void InitQueue(Queue *Q);
void EnQueue(Queue *Q, int x);
void DeQueue(Queue *Q);
int IsQueueEm(Queue *Q);
void showQueue(Queue *Q);
int QueueLength(Queue *Q);
void ClearQueue(Queue *Q);
void DestroyQueue(Queue *Q);

int main(){
    Queue *Q;
    InitQueue(Q);
    printf("入队: \n");
    for(int i=0;i<Maxsize;i++){
        EnQueue(Q, i+1);
    }
    showQueue(Q);
    printf("队列长度: %d\n",QueueLength(Q));
    DeQueue(Q);
    printf("出队: \n");
    showQueue(Q);
    printf("此时front=%d, rear=%d",Q->front,Q->rear);
    ClearQueue(Q);
    DestroyQueue(Q);
    return 0;
}

void InitQueue(Queue *Q){
    Q->base=(int *)malloc(sizeof(int)*Maxsize);
    if(Q->base==NULL){
        return NULL;
    }
    Q->front=0;
    Q->rear=0;
}

void EnQueue(Queue *Q, int x){
    if(Q->rear>=Maxsize){
        return;
    }
    Q->base[Q->rear++]=x;
}

void DeQueue(Queue *Q){
    if(Q->front==Q->rear){
        return;
    }
}

```

```

    }
    Q->front++;
}

int IsQueueEm(Queue *Q){
    if(Q->front==Q->rear){
        return 0;
    }
    return 1;
}

void showQueue(Queue *Q){
    for(int i=Q->front;i<Q->rear;i++){
        printf("%d ",Q->base[i]);
    }
    printf("\n");
}

int QueueLength(Queue *Q){
    return Q->rear-Q->front;
}

void ClearQueue(Queue *Q){
    Q->front=0;
    Q->rear=0;
}

void DestroyQueue(Queue *Q){
    free(Q->base);
    Q->base=NULL;
}

```

运行结果：

```

入队：
1 2 3 4 5 6 7 8
队列长度： 8
出队：
2 3 4 5 6 7 8
此时front=1, rear=8

```

栈：操作受限的线性表

限制：仅允许在表的一端进行插入删除操作。

”

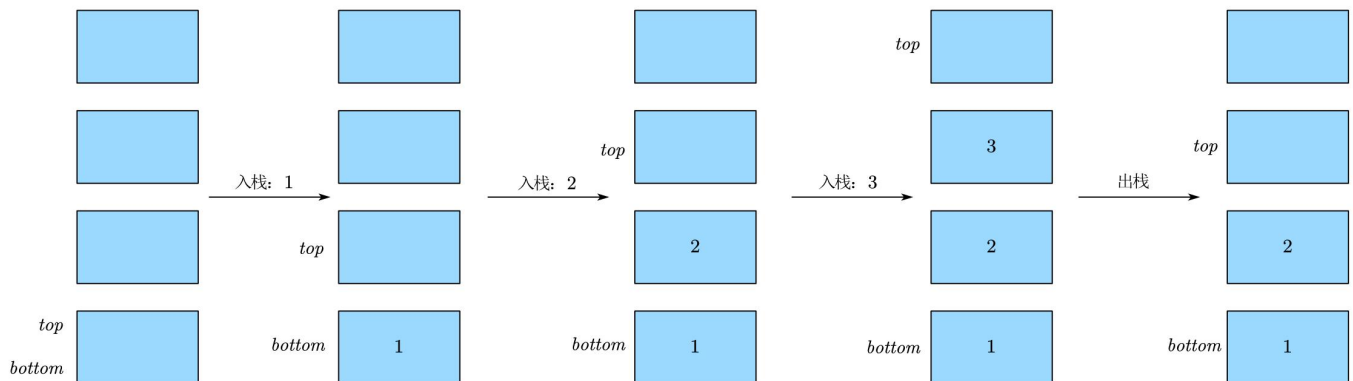
◆ 进行数据插入和删除操作的一端为**栈顶**，另一端为**栈底**。

- ◆ 栈中的数据元素遵守**后进先出**的原则。
- ◆ 压栈：栈的插入操作叫做进栈/压栈/入栈，即从栈顶插入数据。
- ◆ 出栈：栈的删除操作叫做出栈，即从栈顶删除数据。

栈的实现可以使用**数组**和**链表**，相对而言数组的结构实现更优，因为数组在尾上插入数据的代价较小，数组的特征也更符合栈的特性。

栈的基本操作

```
Stack * CreatStack(int length);    //创建栈
int IsStackEm(Stack * stack);      //判断是否为空栈
void Push(Stack * stack, int data); //入栈
int Pop(Stack * stack);             //出栈
void PrintStack(Stack * stack);     //打印栈
void DestroyStack(Stack * stack);   //销毁栈
```



栈的构造

```
typedef struct Stack{
    int *array; //栈的存储空间，用数组也可以
    int top;    //栈顶
    int bottom; //栈底
    int max;    //最大容量
}Stack;
```

创建栈

```
//创建栈
Stack * CreatStack(int length)
{
    Stack * stack=(Stack *)malloc(sizeof(Stack)); //开辟空间
    if(stack)
    {
```

```

        //非空表示申请成功
        stack->array=(int *)malloc(sizeof(int)*length);
        //假如使用数组无需进行这一步
    }
    //开辟空间失败
    if(stack->array==NULL)
    {
        return NULL;
    }
    //初始化
    stack->bottom=0;
    stack->top=0;
    stack->max=length;
}

```

判断是否为空栈

```

//判定是否为空栈
int IsStackEm(Stack * stack)
{
    if(stack->bottom==stack->top)
    {
        //空栈返回0
        return 0;
    }
    //非空栈返回1
    return 1;
}

```

入栈

```

//入栈
void Push(Stack * stack, int data)
{
    if(stack->top==stack->max)
    {
        //判断栈是否已满
        return 0;
    }
    //数据入栈
    stack->array[stack->top]=data;
    //栈顶上移
    stack->top++;
}

```

出栈

```
//出栈
int Pop(Stack * stack)
{
    if(stack->top>stack->bottom)
    {
        //栈不为空
        //栈顶下移
        stack->top--;
        //返回被取出的栈顶数据
        return stack->array[stack->top];
    }else{
        printf("栈空，出栈失败");
        return -1;
    }
}
```

打印栈

```
//打印栈
void PrintStack(Stack * stack)
{
    printf("当前栈中的数据是: ");
    for(int i=0;i<stack->top;i++)
    {
        printf("%d ",stack->array[i]);
    }
    printf("\n");
}
```

销毁栈

```
//销毁栈
void DestroyStack(Stack * stack)
{
    if(stack==NULL)
    {
        return;
    }
    free(stack->array);
    free(stack);
}
```

一个完整的例子

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

typedef struct Stack{
    int *array;
    int top;
    int bottom;
    int max;
}Stack;

Stack * CreatStack(int length);
int IsStackEm(Stack * stack);
void Push(Stack * stack, int data);
int Pop(Stack * stack);
void PrintStack(Stack * stack);
void DestroyStack(Stack * stack);

int main(){
    int maxsize=5;
    Stack * stack=CreatStack(maxsize);
    for(int i=0;i<maxsize;i++){
        Push(stack, i+1);
        PrintStack(stack);
    }
    Pop(stack);
    printf("出栈: \n");
    PrintStack(stack);
    return 0;
}

Stack * CreatStack(int length){
    Stack * stack=(Stack *)malloc(sizeof(Stack));
    if(stack){
        stack->array=(int *)malloc(sizeof(int)*length);
    }
    if(stack->array==NULL){
        return NULL;
    }
    stack->bottom=0;
    stack->top=0;
    stack->max=length;
}

int IsStackEm(Stack * stack){
    if(stack->bottom==stack->top){
        return 0;
    }
}
```

```

        return 1;
    }

    void Push(Stack * stack, int data){
        if(stack->top==stack->max){
            return 0;
        }
        stack->array[stack->top]=data;
        stack->top++;
    }

    int Pop(Stack * stack){
        if(stack->top>stack->bottom){
            stack->top--;
            return stack->array[stack->top];
        }else{
            printf("栈空, 出栈失败");
            return -1;
        }
    }
}

void PrintStack(Stack * stack){
    printf("当前栈中的数据是: ");
    for(int i=0;i<stack->top;i++){
        printf("%d ",stack->array[i]);
    }
    printf("\n");
}

void DestroyStack(Stack * stack){
    if(stack==NULL){
        return;
    }
    free(stack->array);
    free(stack);
}

```

运行结果:

```

当前栈中的数据是: 1
当前栈中的数据是: 1 2
当前栈中的数据是: 1 2 3
当前栈中的数据是: 1 2 3 4
当前栈中的数据是: 1 2 3 4 5

```


出栈:

当前栈中的数据是: 1 2 3 4