

零基础入门 C 语言

勤学如春起之苗，不见其增，日有所长；
辍学如磨刀之石，不见其损，日有所亏。

Author: 王桂林

Mail : guilin_wang@163.com

版本信息：

版本	修订人	审阅人	时间	组织
V1.0	王桂林		2016.02.02	
V1.1	王桂林		2016.11.01	
V1.2	王桂林		2016.04.01	能众软件科技有限公司

更多学习：



1. Hello World	1-1
1.1. 源程序.....	1-1
1.1.1. C 语言版.....	1-1
1.1.2. C++版	1-1
1.1.3. hello word collection.....	1-1
1.2. 注释	1-1
1.3. 从源程序到可执行程序.....	1-2
1.3.1. 集成开发环境:	1-2
1.3.2. 手动编译:	1-2
1.4. 致敬大师.....	1-3
2. Linux 常用基础命令	2-5
2.1. 版本选择.....	2-5
2.1.1. 内核版本.....	2-5
2.1.2. 发行版本.....	2-5
2.2. 目录结构.....	2-6
2.2.1. 系统目录.....	2-6
2.2.2. 分区与目录.....	2-8
2.2.3. 路径.....	2-10
2.3. 常用命令.....	2-10
2.3.1. 命令格式.....	2-10
2.3.2. 目录切换.....	2-10
2.3.3. 文件操作.....	2-10
2.3.4. 用户管理.....	2-11
2.3.5. 网络命令.....	2-11
2.3.6. Ubuntu 下软件安装	2-11
2.4. vim 文本编辑.....	2-11
2.4.1. vim 编辑器中有三种状态模式.....	2-12
2.4.2. vim 编辑器的启动与退出	2-12
2.4.3. 文件操作.....	2-12
2.4.4. 光标移动操作.....	2-12
2.4.5. 编辑操作.....	2-13
2.4.6. 查找与替换操作.....	2-13
2.4.7. 学习工具 vimtutor.....	2-14
2.4.8. vim 最简易配置.....	2-14
2.4.9. vim 与 HHKB.....	2-14
3. C 语言与自然语言	3-17
3.1. 语法规则.....	3-17
3.2. 程序与流程.....	3-17
3.2.1. 程序.....	3-17
3.2.2. 流程和流程图.....	3-18
3.2.3. 常用制图工具.....	3-19
4. 常/变量与数据类型(DataType).....	4-20

4.1. C 语言中的关键字.....	4-20
4.2. 变量(Variable).....	4-20
4.2.1. 变量的定义.....	4-20
4.2.2. 变量的命名规则:	4-20
4.2.3. 交换两个变量的内容.....	4-21
4.2.4. 骆峰命名法.....	4-21
4.3. 内存模型.....	4-22
4.3.1. 物理基础.....	4-22
4.3.2. CPU 读写内存.....	4-23
4.3.3. 读写流程:	4-23
4.3.4. 代码演示.....	4-24
4.4. 计算机的基石补码.....	4-25
4.4.1. 计算机编码基础.....	4-25
4.4.2. 补码的编码规则.....	4-25
4.4.3. 小结.....	4-27
4.5. 数据类型.....	4-27
4.5.1. 类型总揽.....	4-28
4.5.2. 基本类型之数值类型.....	4-28
4.5.3. 基本类型之字符类型.....	4-30
4.6. 常量(Constant).....	4-34
4.6.1. 常量类型.....	4-34
4.6.2. 常量的大小.....	4-35
4.7. 格式输入与输出.....	4-35
4.7.1. printf.....	4-35
4.7.2. scanf	4-39
4.7.3. putchar && getchar.....	4-41
4.7.4. 输入输出缓冲.....	4-41
4.8. 类型转化(Type Cast).....	4-41
4.8.1. 隐式转化.....	4-41
4.8.2. 强制转化.....	4-43
4.9. 练习	4-44
4.9.1. 浮点数跟 0 值比较.....	4-44
4.9.2. printf 返回值有意义吗？	4-44
5. 运算符与表达式(Operator&&Express).....	5-46
5.1. 运符符的优先级/结合性/分类.....	5-46
5.2. 常见运符算符及构成的表达式.....	5-48
5.2.1. 赋值运算符与赋值表达式.....	5-48
5.2.2. 算术运算符与算术表达式.....	5-49
5.2.3. 关系运算符与关系表达式.....	5-50
5.2.4. 逻辑运算符与逻辑表达式.....	5-51
5.2.5. 条件运算符与条件表达式.....	5-53
5.2.6. 逗号运算符与逗号表达式.....	5-54
5.2.7. sizeof 运算符及其表达式.....	5-54
5.2.8. 小结.....	5-55

5.3. 运算符综合练习.....	5-56
5.3.1. if(x==y) 与 if(x=y) 之间的区别。	5-56
5.3.2. 表达式 3==3==3 的值是多少？	5-56
5.3.3. 表达式 100<=a<=300, 能表达[100,300]这样的区间吗？	5-56
5.3.4. 如何判断，我们输入的字符在[a,z]之间。	5-56
5.3.5. 求[100,999]内的水仙花数	5-56
5.3.6. 判断输入的年份，是否为润年。	5-56
5.3.7. 判断数字是否是回文数。	5-56
6. 程序流程设计(Flow of Control)	6-57
6.1. 选择	6-57
6.1.1. If else	6-57
6.1.2. switch	6-60
6.1.3. 小结.....	6-62
6.1.4. 练习判断按键.....	6-62
6.2. 循环	6-63
6.2.1. 循环三要素.....	6-63
6.2.2. while “当”型循环.....	6-64
6.2.3. do while “直到”型循环.....	6-66
6.2.4. for “列表”型循环.....	6-70
6.2.5. 循环的嵌套.....	6-71
6.2.6. 循环建议.....	6-73
6.2.7. 循环小结.....	6-74
6.2.8. 练习.....	6-74
6.3. 跳转	6-77
6.3.1. break	6-77
6.3.2. continue	6-78
6.3.3. return	6-79
6.3.4. goto	6-79
6.3.5. 练习.....	6-80
6.4. 练习	6-80
6.4.1. 打印如下图形.....	6-80
7. 数组(Array)	7-81
7.1. 一维数组.....	7-81
7.1.1. 逻辑与声明.....	7-81
7.1.2. 初始化与访问.....	7-82
7.1.3. 一维数组的存储.....	7-84
7.1.4. 数组三要素.....	7-85
7.1.5. 关于变长数组.....	7-85
7.1.6. 实战应用(一维数据形态)	7-86
7.1.7. 作业.....	7-90
7.2. 二维数组.....	7-90
7.2.1. 声明/定义	7-90
7.2.2. 初始化.....	7-90
7.2.3. 二维数组的逻辑/存储	7-92

7.2.4. 实战应用(二维数据形态)	7-96
7.3. 数组名的二义性.....	7-101
7.3.1. 一维数组名.....	7-101
7.3.2. 二维数组名.....	7-101
7.4. 练习	7-102
7.4.1. 写出冒泡排序的逻辑，并实现冒泡排序。	7-102
7.4.2. 无序数组查找,返回下标（线性查找）	7-102
7.4.3. 有序数组查找（折半查找）	7-102
7.4.4. 求出矩阵两条对角线上的元素之和.....	7-102
7.4.5. 有序数组去重，并返回去重后数组元素新个数。	7-102
7.4.6. 二维数组按列移位.....	7-102
7.4.7. 五子棋判输赢.....	7-103
8. 指针(Pointer)	8-104
8.1. 认识内存.....	8-104
8.1.1. 线性的内存.....	8-104
8.1.2. 变量的地址与大小.....	8-104
8.1.3. 间接访问内存.....	8-106
8.2. 指针常量.....	8-106
8.2.1. 指针是有类型地址常量.....	8-106
8.3. 指针变量.....	8-108
8.3.1. 定义.....	8-108
8.3.2. 解析.....	8-108
8.3.3. 指针变量大小.....	8-108
8.3.4. 初始化及间接访问.....	8-109
8.3.5. 指向/被指向/更改指向.....	8-109
8.3.6. NULL（空即空）	8-111
8.3.7. 课堂实战.....	8-111
8.4. 指针运算.....	8-112
8.4.1. 赋值运算.....	8-112
8.4.2. 算术运算.....	8-112
8.4.3. 关系运算.....	8-113
8.4.4. 小结.....	8-113
8.5. 数组遇上指针.....	8-114
8.5.1. 一维数组的访问方式.....	8-114
8.5.2. 二维数组的访问方式.....	8-116
8.6. 练习	8-119
8.6.1. 用指针法逆序打印一个数组。	8-119
8.6.2. 往指定内存写入数据.....	8-119
8.6.3. 判断是否是回文串.....	8-119
9. 函数(Function)	9-120
9.1. c 标准库及库函数.....	9-120
9.1.1. 库存在的意义.....	9-120
9.1.2. 如何使用库函数.....	9-120
9.1.3. 库函数使用示例.....	9-121

9.1.4. 常用库函数.....	9-126
9.2. 自定义函数.....	9-127
9.2.1. 语法格式.....	9-127
9.2.2. 调用方法.....	9-128
9.2.3. 前向声明.....	9-128
9.2.4. 函数返回值.....	9-129
9.2.5. 实参与形参.....	9-129
9.3. 传值与传址.....	9-129
9.3.1. 传值与传址的比较.....	9-129
9.3.2. 图解传值与传址.....	9-131
9.3.3. 如何来传递一个一维数组.....	9-132
9.3.4. 如何来传递一个二维数组.....	9-134
9.3.5. 小结.....	9-134
9.4. 函数调用.....	9-134
9.4.1. 普通调用.....	9-134
9.4.2. 递归调用.....	9-135
9.5. 递归(Recursive)详解.....	9-136
9.5.1. 递归定义.....	9-136
9.5.2. 递归小结.....	9-138
9.5.3. 递归与循环论述.....	9-138
9.5.4. 递归练习.....	9-139
9.6. 练习	9-140
9.6.1. 如何产生[m,n]以内的随机数。	9-140
9.6.2. 产生 30 个不重复的随机数.....	9-140
9.6.3. 请用函数封装基于数组的冒泡排序，选择排序。	9-140
9.6.4. 用函数的思想来支解选择法排序	9-140
9.6.5. 请用函数封装基于数组的二分查找。	9-140
9.6.6. 求中位数.....	9-140
10. 变量作用域/生命周期/修饰符.....	10-141
10.1. 作用域(Scope)	10-141
10.1.1. 作用域.....	10-141
10.1.2. 作用域叠加.....	10-142
10.1.3. 全局命名污染.....	10-143
10.2. 生命周期(Duration)	10-144
10.2.1. 局部变量.....	10-144
10.2.2. 全局变量.....	10-144
10.3. 修饰符(Storage Description)	10-144
10.3.1. auto(大将军).....	10-144
10.3.2. register(小太监).....	10-145
10.3.3. extern(通关文牒).....	10-146
10.3.4. static(柱国老臣 限离出境).....	10-148
10.4. 小结	10-150
11. 字符串(String).....	11-151
11.1. 引入	11-151

11.2. 字符串常量.....	11-151
11.2.1. 字符串大小.....	11-152
11.2.2. 字符串存储.....	11-152
11.2.3. C 语言是如何处理字符串常量.....	11-153
11.3. 字符串与字符数组.....	11-153
11.3.1. 常量字符串不可更改.....	11-153
11.3.2. 字符串与字符数组比较.....	11-153
11.3.3. 字符数组存储字符串.....	11-154
11.3.4. 小结.....	11-155
11.4. 字符串的输入与输出.....	11-156
11.4.1. 输出.....	11-156
11.4.2. 输入.....	11-156
11.5. 字符串操作函数.....	11-157
11.5.1. 字符数组原生操作.....	11-157
11.5.2. 库函数操作.....	11-158
11.6. 字符串操作函数自实现.....	11-162
11.6.1. myStrlen	11-162
11.6.2. myStrcpy.....	11-162
11.7. 多文件编程.....	11-163
11.7.1. 函数声明(.h).....	11-163
11.7.2. 函数实现(.c)	11-163
11.7.3. 头文件包含#include	11-164
11.8. 指针数组.....	11-165
11.8.1. 定义.....	11-165
11.8.2. 应用.....	11-166
11.8.3. 思考.....	11-168
11.9. 你所追过的那些“零”	11-168
11.9.1. 零所代表的意义.....	11-168
11.9.2. 实例分析.....	11-168
11.10. 作业	11-169
11.10.1. 自实现 myStrcat	11-169
11.10.2. 自实现 mystrcmp	11-169
11.10.3. 以下代码能打印什么？	11-170
12. 内存管理(Memory)	12-171
12.1. 进程空间.....	12-171
12.1.1. 源程序/程序/进程.....	12-171
12.1.2. 进程空间图示.....	12-171
12.2. 栈内存(Stack).....	12-172
12.2.1. 栈存储的特点.....	12-172
12.2.2. 栈大小.....	12-172
12.2.3. 常见栈溢出案例.....	12-172
12.3. 堆内存(Heap).....	12-173
12.3.1. 堆存储的特点.....	12-173
12.3.2. 堆大小.....	12-173

12.3.3. 堆内存的申请与释放.....	12-173
12.3.4. 应用模型.....	12-177
12.3.5. 常见错误案例剖析.....	12-178
12.3.6. VLD 工具使用	12-179
12.4. 开放的地址空间	12-179
12.5. 堆与栈空间的返回.....	12-180
12.5.1. 栈空间不可以返回.....	12-180
12.5.2. 堆空间可以返回.....	12-181
13. 结构体(Struct)	13-182
13.1. 引例	13-182
13.2. 为什么要引入结构体.....	13-183
13.2.1. 开放类型定义.....	13-183
13.2.2. 从单变量->数组->结构体	13-183
13.3. 结构体 类型 定义.....	13-184
13.3.1. 无名构造类型.....	13-185
13.3.2. 有名构造类型.....	13-186
13.3.3. 别名构造体类型.....	13-186
13.3.4. 小结.....	13-186
13.4. 结构体 变量 初始化及成员访问.....	13-186
13.4.1. 初始化及访问.....	13-187
13.4.2. 成员运算符本质.....	13-189
13.4.3. 赋值.....	13-190
13.5. 结构体类型作参数和返回值.....	13-191
13.5.1. 结构体变量作参数和返回值.....	13-191
13.5.2. 结构体指针作参数.....	13-192
13.5.3. 获取当前时间函数的使用.....	13-192
13.6. 结构体数组.....	13-193
13.6.1. 定义及初始化.....	13-193
13.6.2. 内存存储形式.....	13-194
13.6.3. 实战之选举.....	13-195
13.7. 结构体嵌套.....	13-196
13.7.1. 结构体中可以嵌套结构体.....	13-196
13.7.2. 嵌套结构体变量定义和初始化.....	13-197
13.8. 结构体类型的大小.....	13-197
13.8.1. 结构体成员内存分布.....	13-197
13.8.2. 内存对齐.....	13-198
13.8.3. 结构体中嵌套构造类型成员的对齐.....	13-198
13.9. 结构体使用注意事项.....	13-199
13.9.1. 向结构体内未初始化的指针拷贝	13-199
13.9.2. 未释放结构体内指针所指向的空间.....	13-199
13.9.3. 深拷贝与浅拷贝.....	13-200
13.10. 课堂实战.....	13-200
13.10.1. 栈的自实现.....	13-200
13.11. typedef 类型重命名	13-203

13.11.1. <code>typedef</code> 作用	13-203
13.11.2. 定义新类型.....	13-203
13.11.3. <code>size_t/int8</code> 是什么鬼？	13-204
13.11.4. <code>typedef</code> 和 <code>#define</code> 的区别.....	13-204
13.11.5. 小结	13-204
13.12. 类型大总结	13-205
14. 共用(Union)与枚举(Enum).....	14-206
14.1. 共用体	14-206
14.1.1. 类型定义与变量定义.....	14-206
14.1.2. 内存分析.....	14-206
14.1.3. 共用体小结.....	14-209
14.1.4. 应用.....	14-209
14.2. 枚举	14-211
14.2.1. 枚举类型定义.....	14-211
14.2.2. 枚举变量与初始化.....	14-212
14.2.3. 枚举常量.....	14-212
14.2.4. 应用.....	14-212
14.3. 练习	14-214
14.3.1. 输出一个整型数据的字符形式.....	14-214
14.3.2. 实现 <code>short</code> 类型变量高低位互换	14-214
14.3.3. 将下面程序中 <code>case</code> 分支常量用宏和枚举来实现.....	14-214
15. 单向链表(List).....	15-216
15.1. 链表价值.....	15-216
15.2. 静态链表.....	15-216
15.2.1. 链表节点定义.....	15-216
15.2.2. 图示节点.....	15-216
15.2.3. 图示链表结构.....	15-216
15.2.4. 代码实现.....	15-217
15.3. 动态链表.....	15-217
15.3.1. 链表图示.....	15-218
15.3.2. 链表名字解释.....	15-218
15.3.3. 创建(尾插法)	15-218
15.3.4. 创建(头插法)	15-220
15.3.5. 遍历.....	15-220
15.3.6. 求长度.....	15-221
15.3.7. 插入.....	15-221
15.3.8. 查找.....	15-222
15.3.9. 删除.....	15-222
15.3.10. 排序	15-223
15.3.11. 链表反转.....	15-225
15.3.12. 链表销毁.....	15-226
15.3.13. 功能测试.....	15-226
15.4. 环形链表.....	15-227
15.4.1. 图示.....	15-227

15.4.2. 自实现.....	15-227
15.5. 作业	15-228
15.5.1. 实现手机通讯录功能.....	15-228
15.5.2. 输入一字符串，用链表形式储存。	15-229
15.5.3. 用选择法实现单向链表的排序。	15-229
15.5.4. 倒序打印单链表。	15-229
15.5.5. 合并链表.....	15-229
16. 文件(File)	16-230
16.1. 文件流	16-230
16.1.1. 文件流概念.....	16-230
16.1.2. 文件类型.....	16-230
16.1.3. 文件缓冲.....	16-232
16.2. 文件的打开和关闭.....	16-233
16.2.1. FILE 结构体	16-233
16.2.2. fopen	16-234
16.2.3. fclose	16-235
16.3. 一次读写一个字符(文本操作)	16-235
16.3.1. fputc	16-235
16.3.2. fgets.....	16-236
16.3.3. 关于 feof 的问题	16-237
16.3.4. 练习.....	16-238
16.4. 一次读写一行字符(文本操作)	16-240
16.4.1. 什么是行.....	16-240
16.4.2. fputs	16-241
16.4.3. fgets.....	16-242
16.4.4. 关于 feof 的问题	16-244
16.4.5. 注意事项.....	16-245
16.4.6. 练习.....	16-246
16.5. 一次读写一块数据(二进制操作)	16-247
16.5.1. fwrite / fread	16-247
16.5.2. 试读文本文件.....	16-248
16.5.3. 二进制读写才是本质.....	16-250
16.5.4. 读写结构体是长项.....	16-252
16.5.5. 课堂实战-管理系统	16-253
16.6. 文件指针偏移.....	16-259
16.6.1. rewind	16-259
16.6.2. ftell	16-260
16.6.3. fseek	16-260
16.6.4. 空洞文件.....	16-261
16.7. 练习	16-261
16.7.1. 文本文件内容排序(行数不能变)	16-261
16.7.2. 修改文件内容.....	16-263
17. 位操作(Bit Operation)	17-264
17.1. 位操作与逻辑操作.....	17-264

17.2. 数据的二进制表示.....	17-264
17.2.1. 8 位二进制数据的补码.....	17-264
17.2.2. 二进制打印.....	17-265
17.3. 位操作	17-266
17.3.1. 位与(&)	17-266
17.3.2. 位或()	17-266
17.3.3. 位取反(~)	17-267
17.3.4. 位异或(^)	17-267
17.3.5. 左移(<<)	17-267
17.3.6. 右移(>>)	17-268
17.3.7. 优先级.....	17-269
17.4. 应用	17-269
17.4.1. 掩码(mask).....	17-269
17.4.2. 功能.....	17-270
17.4.3. 生成.....	17-270
17.5. 课堂实战.....	17-270
17.5.1. 输出位数.....	17-270
17.5.2. 判断一个数是不是 2 的幂数。	17-271
17.5.3. 实现循环移位.....	17-271
17.6. 提高篇	17-271
17.6.1. 无参交换.....	17-271
17.6.2. 异或加密(文本与二进制)	17-272
17.6.3. 循环移位加密(文本与二进制)	17-274
17.7. 练习	17-275
17.7.1. 打印数据的二进制形式（32 位）	17-275
17.7.2. 反转一个数据的最后 n 位。	17-275
17.7.3. 依数据的符号们判断正负。	17-275
17.7.4. 练习加密二进制文件.....	17-275
18. 预处理(Pre Compile).....	18-277
18.1. 发生时机.....	18-277
18.2. 宏(Macro).....	18-277
18.2.1. 不带参宏	18-277
18.2.2. 带参宏(宏函数)	18-278
18.2.3. 取消宏.....	18-279
18.3. 条件编译(Condition Compile)	18-280
18.3.1. 单双路(#ifdef / #ifndef #else #endif).....	18-280
18.3.2. 单双多路(#if #elif #endif)	18-280
18.3.3. 编译期指定宏 gcc -D.....	18-281
18.4. 头文件包含(#include)	18-282
18.4.1. 包含的意义.....	18-282
18.4.2. 包含的方式.....	18-282
18.4.3. 多文件编程.....	18-283
18.4.4. 定义头文件.....	18-283
18.4.5. 相互包含的避免.....	18-286

18.5. 其它	18-286
18.5.1. #运算符 利用宏创建字符串	18-286
18.5.2. ##运算符 预处理的粘和剂	18-287
18.5.3. 预定义宏	18-288
18.6. 练习	18-288
18.6.1. 宏展开有次序吗?	18-288
19. 项目	19-290
19.1. 项目简介	19-290
19.2. 概要设计	19-290
19.3. 详细设计	19-290
19.4. 实现	19-290
19.4.1. 界面菜单实现	19-290
19.4.2. 链表实现	19-290
19.4.3. 文件读写实现	19-290
19.5. SVN 版本管理控制	19-290
20. 附录(Appendix)	20-291
20.1. 参考书目	20-291
20.2. ascii 码表详解	20-291
20.3. ascii 特殊字符解释	20-292
20.4. 运算符优先级	20-293
20.5. 易错优先级集锦	20-294
20.6. vs2013 中使用技巧	20-294
20.6.1. 禁用_s 版本函数的方法	20-294
20.7. 附练习答案	20-294
20.7.1. 第六章程序设计	20-294
20.7.2. 第七章数组	20-294
20.8. 章节思维导图	20-296
20.8.1. 常变量与数据类型	20-296
20.8.2. 运算符与表达式	20-296
20.8.3. 数组	20-296
20.8.4. 函数	20-296
20.8.5. 作用域与运算符	20-296
20.8.6. 字符串	20-296
20.8.7. 文件	20-296
20.8.8. 位运算	20-296
20.8.9. 预处理	20-296

1. Hello World

这个世界，几乎每个程序员入门的第一段代码都是 Hello world。原因是当年 C 语言的作者，Dennis Ritchie（丹尼斯·里奇）在他的名著《C 程序设计语言（The C Programming Language）》中第一次引入，传为后世经典，其它语言亦争相效仿，以示致敬。



1.1. 源程序

1.1.1. C 语言版

```
#include <stdio.h> //程序中因为用到了 printf 库函数，所以要包含 printf 所在的头文件
int main() //程序的入口，表示计算机从哪里开始执行此程序，有且，只有一个
{
    printf("Hello World!\n"); //调用库函数，向屏幕打印 Hello World!
    return 0; // main 函数的返回值
}
```

1.1.2. C++ 版

```
#include <iostream> //程序中因为用到了 cout 库函数，所以要包含 cout 所在的头文件
using namespace std; //用到的命名空间
int main() //程序的入口，表示计算机从哪里开始执行此程序，有且，只有一个
{
    cout << "Hello World!" << endl; //通过类对象 cout 向屏幕输出 Hello World!
    return 0; //main 函数的返回值
}
```

1.1.3. hello word collection

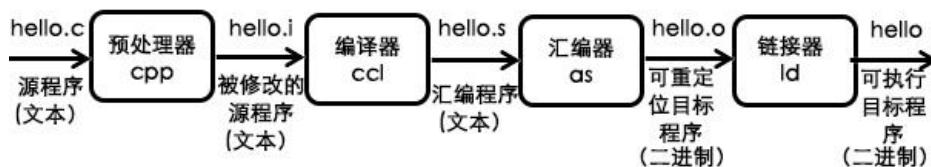
<http://helloworldcollection.de/>

1.2. 注释

上面的两个示例中，//后面的语句，表示程序的注释，用来注解的作用，不参与编译，以便后期的维护和管理

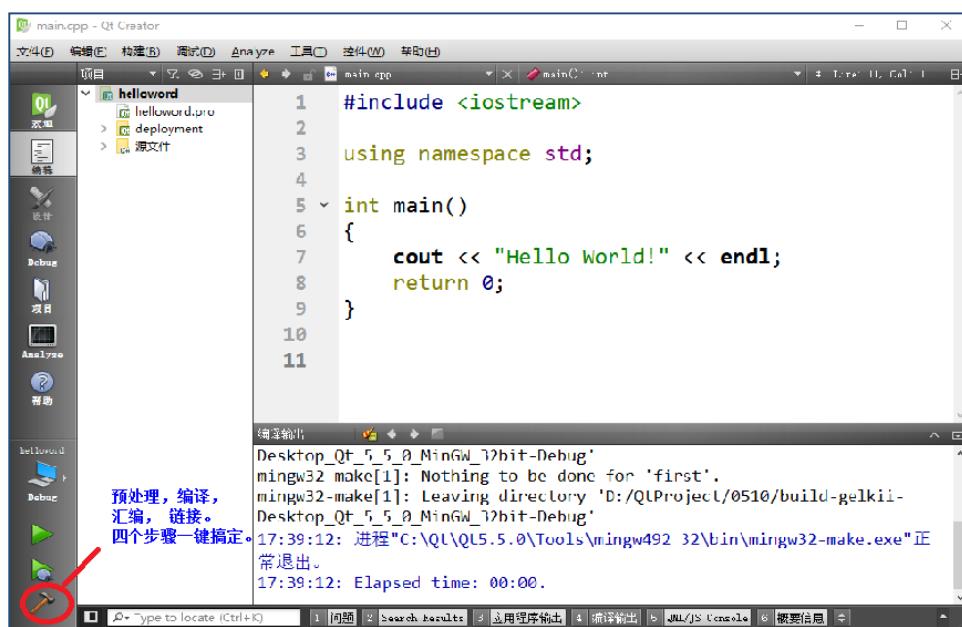
1.3.从源程序到可执行程序

源程序就是一个.txt 的普通文本文件，是经历了哪些过程，变为可执行性文件的呢？大体上分为四个步骤： 预处理 -> 编译 -> 汇编 -> 链接 四个过程。



1.3.1.集成开发环境：

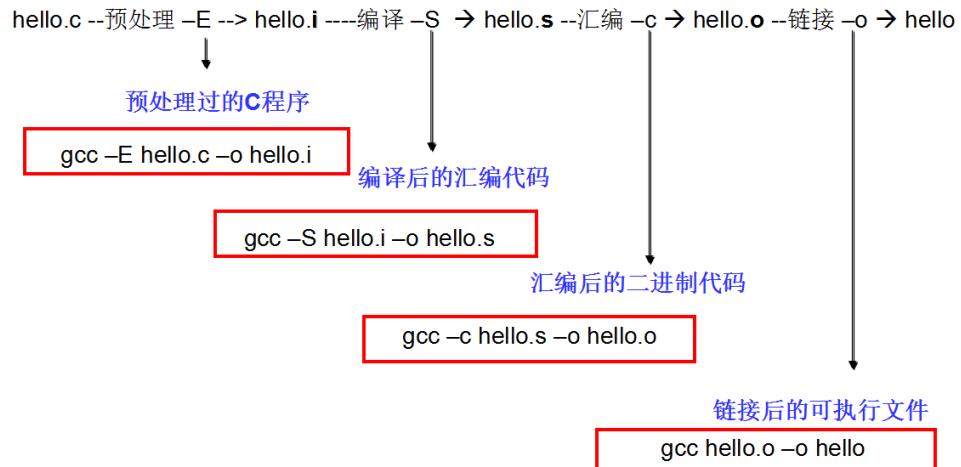
对于集成开发环境（IDE，Integrated Development Environment）而言，减少了环境配置，合并了流程，使其便于快速开发。就 Qt 开发环境而言，只需一步即可：



1.3.2.手动编译：

unix /linux 环境下，通常没有界面，所以少有集成开发环境。所有的开发均是在命令行模式下，开发的。以 vim 为编辑器，以 gcc 为编译器为例，演示。

- ① vim hello.c
- ② gcc -E hello.c -o hello.i //处理文件包含，宏和注释
- ③ gcc -S hello.i -o hello.s //编译为汇编文件
- ④ gcc -c hello.s -o hello.o //经汇编后为二进制的机器指令
- ⑤ gcc hello.o -o hello //链接所用的到库



- ① 预处理:预处理相当于根据预处理命令组装成新的 C 程序，不过常以 **i** 为扩展名。
- ② 编译:将得到的 **i** 文件翻译成汇编代码 **.s** 文件。
- ③ 汇编:将汇编文件翻译成机器指令，并打包成可重定位目标程序的 **o** 文件。该文件是二进制文件，字节编码是机器指令。
- ④ 链接:将引用的其他 **o** 文件并入到我们程序所在的 **o** 文件中，处理得到最终的可执行文件。

Linux 环境虽然简陋，但是向我们呈现出的是本质的东东，一上手，就用集成开发环境可能有些细节，可能很久都搞不明白。

1.4.致敬大师



Dennis Ritchie

KenThompson

“牛顿说他是站在巨人的肩膀上，如今，我们都站在里奇的肩膀上。”-----克尼汉

“他是虔诚而纯粹的计算机天才，侵犯他人电脑是绝不干的。”----- 潘嘉杰。

丹尼斯里奇先生的专业精神令人感动，近 40 年如一日，在他所从事的领域辛勤耕耘，他的多项发明，包括 C 语言，Unix，也包括 Plan9，无论哪一项，在软件发展史上都有着举足轻重的地位，和他的伟大成就形成对照的是他的行事，态度低调，他的表达，像他的软件一样，简洁生动而准确。C++的成功，很大一部分也来自于 C，是 C 语言的普及和深入，才有了后面的凤凰涅槃，从另一个角度，在同另一语言大师 Pascal 之父 Niklaus Wirth 交流时，C++的名字也是源自 C 语言的利器，Wirth 先生不无惋惜地表示，后来他开发的语言可惜没叫 Pascal2。-----著名的计算机科学家 N.Wirth 如是说。

2. Linux 常用基础命令

本章节旨在简略的介绍 linux 的基本使用，可能会在讲解语言的过程中作一些**平台差异对比**。目的只有一个就是混个脸熟，我们后面会有专业课堂来讲解。

2.1. 版本选择

2.1.1. 内核版本

Linux 内核是开源，开源地址：<https://www.kernel.org/> 如此好的东东，都开源了，想想都鸡动。

作者 Linus Torvalds，Linux 最开始是由 linus 完成，然后开源，全世界的程序员都在为它贡献代码。大神级别的人物，无需赘言，只需膜拜。



查看内核版本的命令如下：

```
root@ubuntu:~# uname -r
4.4.0-38-generic
```

版本信息：



2.1.2. 发行版本

内核是不能直接运行在某个具体的机型上，需要适配，于是就有了各种发行版本。当然了，发行版本中，还内置了众多软件服务和人性化的交互界面，常见的发行版本如下：



在众多的版本中，我们为什么会选择 **CentOS/Ubuntu**?

CentOS 是 Community Enterprise Operating System 的简称，我们有很多人叫它社区企业操作系统，不管你叫它什么，它都是 linux 的一个发行版本。CentOS，来自大名鼎鼎的 RedHat 公司的 RHEL(Red Hat Enterprise Linux)产品的克隆版本。RHEL 在系统的稳定性，前瞻性和安全性上有着极大的优势。RHEL 需要向 RedHat 付费才可以使用，并能得到付费用户的服务及技术支持和版本升级。这个 CentOS 可以像 RHEL 一样的构筑 linux 系统环境，但不需要向 RedHat 付任何的费用。

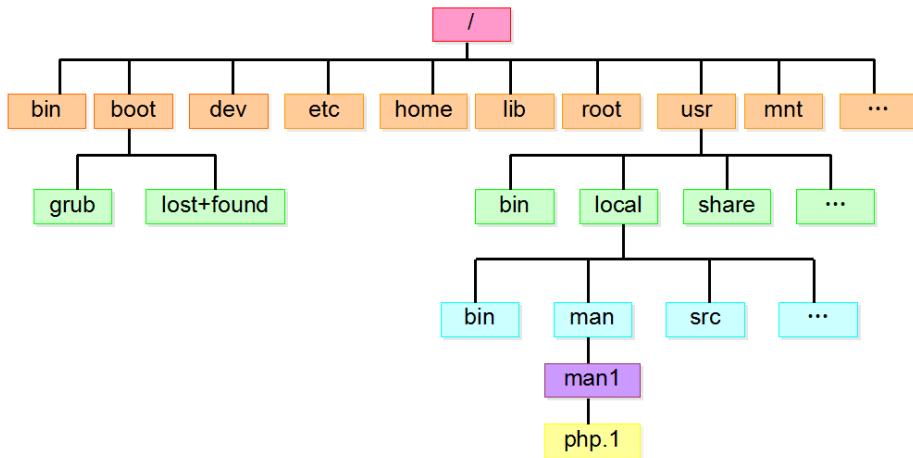
Ubuntu (乌班图) 是一个以桌面应用为主的 Linux 操作系统，其名称来自非洲南部祖鲁语或豪萨语的“ubuntu”一词，意思是“人性”、“我的存在是因为大家的存在”，是非洲传统的一种价值观，类似华人社会的“仁爱”思想。Ubuntu 基于 Debian 发行版和 GNOME 桌面环境，Ubuntu 的目标在于为一般用户提供一个最新的、同时又相当稳定的主要由自由软件构建而成的操作系统。Ubuntu 具有庞大的社区力量，用户可以方便地从社区获得帮助。

2.2. 目录结构

2.2.1. 系统目录

打开我的电脑的一瞬间，大家的第一反应就是，怎么一堆文件夹，我的盘符 (c 盘，d 盘等) 怎么不见了？

Windows 分区在盘符下面，而 linux 分区挂在文件目录下面。可以通过命令查看树结构(tree 命令需要安装) `tree / -L 2`



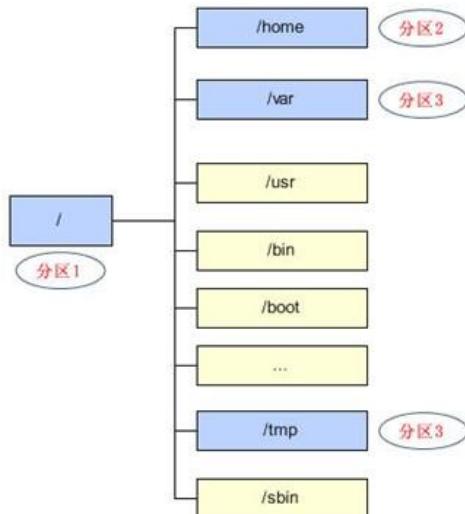
目录	描述
/	根目录
/bin	做为基础系统所需要的最基础的命令就是放在这里。比如 ls、cp、mkdir 等命令；功能和/usr/bin 类似，这个目录中的文件都是可执行的，普通用户都可以使用的命令。
/boot	Linux 的内核及引导系统程序所需要的文件，比如 vmlinuz initrd.img 文件都位于这个目录中。在一般情况下，GRUB 或 LILO 系统引导管理器也位于这个目录；启动装载文件存放位置，如 kernels,initrd,grub。一般是一个独立的分区。
/dev	一些必要的设备,声卡、磁盘等。还有如 /dev/null ./dev/console /dev/zero /dev/full 等。
/etc	系统的配置文件存放地. 一些服务器的配置文件也在这里；比如用户帐号及密码配置文件； /etc/opt:/opt 对应的配置文件 /etc/X11:Xwindows 系统配置文件 /etc/xml:XML 配置文件
/home	用户工作目录，和个人配置文件，如个人环境变量等，所有的账号分配一个工作目录。一般是一个独立的分区。
/lib	库文件存放地。bin 和 sbin 需要的库文件。类似 windows 的 DLL。
/media	可拆卸的媒介挂载点，如 CD-ROMs、移动硬盘、U 盘，系统默认会挂载到这里来。
/mnt	临时挂载文件系统。这个目录一般是用于存放挂载储存设备的挂载目录的，比如有

	cdrom 等目录。可以参看/etc/fstab 的定义。
/opt	可选的应用程序包。
/proc	操作系统运行时，进程（正在运行中的程序）信息及内核信息（比如 cpu、硬盘分区、内存信息等）存放在这里。/proc 目录伪装的文件系统 proc 的挂载目录，proc 并不是真正的文件系统，它的定义可以参见 /etc/fstab 。
/root	Root 用户的工作目录
/sbin	和 bin 类似，是一些可执行文件，不过不是所有用户都需要的，一般是系统管理所需要使用得到的。
/tmp	系统的临时文件，一般系统重启不会被保存。
/usr	包含了系统用户工具和程序。 /usr/bin : 非必须的普通用户可执行命令 /usr/include : 标准头文件 /usr/lib:/usr/bin/ 和 /usr/sbin/ 的库文件 /usr/sbin:非必须的可执行文件 /usr/src:内核源码 /usr/X11R6:X Window System, Version 11, Release 6.
/srv	该目录存放一些服务启动之后需要提取的数据

2.2.2. 分区与目录

Linux 下磁盘分区和目录的关系如下：

- 任何一个分区都必须挂载到某个目录上(同一分区可以同时挂载到不同的目录)。
- 目录是逻辑上的区分,分区是物理上的区分。
- 磁盘 Linux 分区都必须挂载到目录树中的某个具体的目录上才能进行读写操作。
- 根目录是所有 Linux 的文件和目录所在的地方，需要挂载上一个磁盘分区。



例 linux U 盘(FAT32 模式)的使用 (VM usb 3.0 兼容设置) :

1 查看 U 盘是否接入

```
[root@localhost Desktop]# fdisk -l
Device Boot Start End Blocks Id System
/dev/sdb1 * 165 2405 7082112 c W95 FAT32 (LBA)
```

2 建立挂载节点

```
[root@localhost Desktop]# mkdir /mnt/usb
[root@localhost Desktop]# ls /mnt/
hgfs usb
```

3 挂载命令

```
[root@localhost Desktop]# mount /dev/sdb1 /mnt/usb/
[root@localhost Desktop]# cd /mnt/usb/
[root@localhost usb]# ls
160307222636.BMP autorun.inf icon.ico System Volume
Information
360DrvMgrInstaller_net.exe GHO ISO
```

4 卸载命令

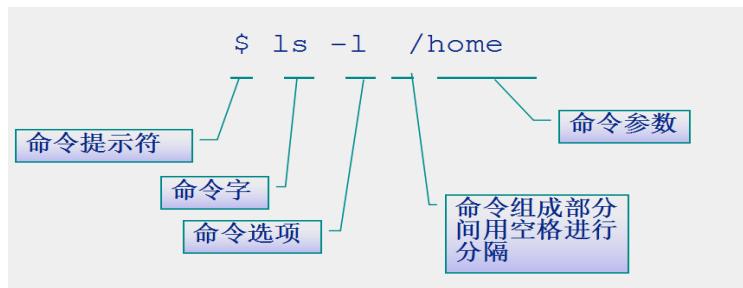
```
[root@localhost mnt]# umount /mnt/usb/
[root@localhost mnt]# ls /mnt/
hgfs
```

2.2.3.路径

linux 中路径分为绝对路径与相对路径，绝对路径是以根起始的路径，相对路径是相对于当前起始的路径。

2.3.常用命令

2.3.1.命令格式



2.3.2.目录切换

命令	参数	释意	示例
ls	-l -a	查看当前目录中的文件	ls /root
pwd		显示当前路径	pwd
cd		切换至不同的目录	cd /opt cd ..
	.	当前目录，即缺省目录	相对路径的起始
	..	上一级目录	
	/	根路径	绝对路径的起始

2.3.3.文件操作

命令	参数	释意	示例
touch		创建空文件	touch aa.c
mkdir		创建空文件夹	mkdir mydir
rm	-r -f	删除文件或目录	rm -rf aa.c
cp	-r	拷贝文件去某个目录，若后面跟不是文件夹，是为复制文件。	cp aa.c /opt cp aa.c aa.c.bak
mv		剪切文件去某个目录，若后面跟的不是目录，则为重命名	mv aa.c /opt mv aa.c bb.c
find		在给定目录下，按名字查找文件或目录。	find /opt -name aa.c find /opt -type l find /opt -size +10000k
cat		在终端中显示文本内容。	cat aa.c
less		可进行翻页的文本内容查看。	less aa.c

cp命令两个主要功能：
1) 复制文件、文件夹(需要-r参数)
2) 备份文件

mv命令两个主要功能：
1) 剪贴文件、文件夹
2) 重命名

head	-n	查看文件开头的 n 行。	cat -n 10 hello.c
tail	-n	查看文件结尾的 n 行。	cat -n 30 hello.c

2.3.4. 用户管理

命令	参数	释意	示例
useradd		添加一个用户	useradd jim
passwd		给新添加用户设密	passwd jim
userdel	-r	删除某个用户(并工作目录)	userdel -r jim
su		切换当前用户为其它用户	su root

2.3.5. 网络命令

命令	参数	释意	示例
ifconfig		查看当前网络状态	ifconfig
ping	-c	查看网络联接状态	ping -c 4 192.168.1.1
setup		setup 设置网络	setup
service	network	重启网络	service network restart
service	iptables	关闭防火墙	service iptables stop
chkconfig	iptables	禁用防火墙	chkconfig iptables off
sestatus		查看 selinux 状态	
setenforce	0	关闭 selinux	setenforce 0

2.3.6. Ubuntu 下软件安装

命令	释意	示例
apt-get update	获取最新软件列表	apt-get update
apt-get upgrade	升级最新列表软件	apt-get upgrade
apt-cache search	查询软件列表	apt-cache search vim
apt-get install	安装软件	apt-get intall qt5-default qtcreator qt5-doc
apt-get remove	卸载软件	apt-get remove vim

2.3.7. 其它命令参考

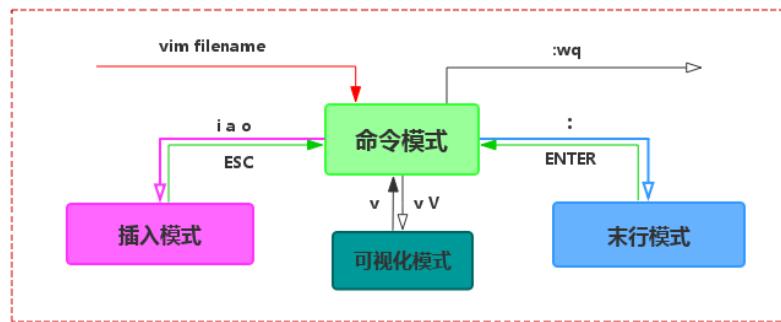
<http://man.linuxde.net/>

2.4. vim 文本编辑

有人曾戏说，世界上只有三种编辑器，VIM、EMACS 和其它。

这里介绍的 vim 就是神一样的编辑器，学习成本有点高，学习路线有点陡峭，命令行界面无可取代，使用是否灵活，当即可鉴别是否具体丰富 linux/unix 的开发经验。

2.4.1.vim 编辑器中有三种状态模式



2.4.2.vim 编辑器的启动与退出

命令	释意
\$ vi	直接进入编辑环境
\$ vi myfile	进入编辑环境并打开（新建）文件
:q	退出 vi 编辑环境

2.4.3.文件操作

命令	释意
:w	保存对 vi 编辑器中已打开文件的修改
:w myfile	将 vi 编辑器中的内容另存为指定文件名
:wq	对 vi 编辑器中的文件进行保存并退出 vi 编辑器
:q!	放弃对文件内容的修改，并退出 vi 编辑器

2.4.4.光标移动操作

命令	光标操作	释意
光标移动	h	向左移动光标
	l	向右移动光标
	k	向上移动光标
	j	向下移动光标
翻页	Ctrl + f	向下翻整页
	Ctrl + b	向上翻整页
	Ctrl + u	向上翻半页
	Ctrl + d	向下翻半页
行内移动光标	^	将光标快速跳转到本行的行首字符
	\$	将光标快速跳转到本行的行尾字符
文件内跳转	:set nu	在编辑器中显示行号
	:set nonu	取消编辑器中的行号显示
	1G(gg)	跳转到文件的首行

	G	跳转到文件的末尾行
	#G	跳转到文件中的第#行

2.4.5. 编辑操作

命令	光标操作	释意
进入输入模式	i	在当前光标处进入插入状态
	a	在当前光标后进入插入状态
	o	在当前行的下面插入新行，光标移动到新行的行首，进入插入状态
	O	在当前行的上面插入新行，光标移动到新行的行首，进入插入状态
	c(n)w	删除当前光标到所在单词尾部的字符，并进入插入状态
	c\$	删除当前光标到行尾的字符，并进入插入状态
	c^	命令删除当前光标之前（不包括光标上的字符）到行首的字符，并进入插入状态
删除操作	x	删除光标处的单个字符
	(n)dd	删除光标所 n 行
	:3,5 d	删除从第 3 行到第 5 行的内容
	dw	删除当前字符到单词尾（包括空格）的所有字符
	de	删除当前字符到单词尾（不包括单词尾部的空格）的所有字符
	d\$	删除当前字符到行尾的所有字符
	d^	删除当前字符到行首的所有字符
	J(大写)	删除光标所在行行尾的换行符，相当于合并当前行和下一行的内容
拷贝和粘贴	(n)yy	复制当前行整行的内容到 vi 缓冲区
	:3,6 co(m) 12	复制（剪切）3-6 行，到第 12 行处
	p	读取 vi 缓冲区中的内容，并粘贴到光标当前位置（不覆盖文件已有的内容）
撤销与恢复	u	取消最近一次的操作，并恢复操作结果 可以多次使用 u 命令恢复已进行的多步操作
	U	取消对当前整行进行的所有操作
	Ctrl + r	对使用 u 命令撤销的操作进行恢复

2.4.6. 查找与替换操作

命令	光标操作	释意
----	------	----

查找	/word	从上而下在文件中查找字符串“word”
	?word	从下而上在文件中查找字符串“word”
	n	定位下一个匹配的被查找字符串
	N	定位上一个匹配的被查找字符串
替换	:s/old/new	将当前行中查找到的第一个字符“old”串替换为“new”
	:s/old/new/g	将当前行中查找到的所有字符串“old”替换为“new”
	:#,##s/old/new/g	在行号“#,#”范围内替换所有的字符串“old”为“new”
	:%s/old/new/g	在整个文件范围内替换所有的字符串“old”为“new”

2.4.7. 学习工具 vimtutor

在命令行中输入 vimtutor 默认是英文版的，可以使用中文版 vimtutor -g zh。

2.4.8.vim 最简易配置

在用户当前的家目录中，输入 vim ~/.vimrc(本用户)或 vim /etc/[vim/]vimrc(全局用户)该文件是一个隐藏文件，如果没有则创建它，该文件中保存一些，vim 的常用配置。

命令	语义	缩写
set number	代码显示行号	set nu
set nonumber		set nonu
syntax on	关键词高亮显示	
set cindent	c 语法自动缩进	set ci
set autoindent	新行自动缩进	set ai
set shiftwidth=4	缩进宽度 4	set sw=4
set tabstop=4	tab 键宽度 4	set ts=4
set softtabstop=4	制表符宽度 4	set st=4
set nobackup	不备份	set nb

vim 应用提高：http://blog.csdn.net/linux_wgl/article/details/51446512

2.4.9.vim 与 HHKB

总有人羡慕大神和黑客，那你愿不愿付出点努力，先学一学大神用的工具呢？“合抱之木，生于毫末；九层之台，起于垒土；千里之行，始于足下。”

大师 & Vim & HHKB：



Richard Matthew Stallman



Bjarne Stroustrup

2.5.ftp 服务

2.5.1. 安装

Centos 服务端：[yum -y install vsftpd](#)

Windows 客户端：[filezilla](#)

2.5.2. 服务端配置

```
[root@localhost vsftpd]# pwd
/etc/vsftpd
[root@localhost vsftpd]# ls
ftptusers user_list vsftpd.conf vsftpd_conf_migrate.sh
[root@localhost vsftpd]#
```

vsftpd.conf 是一个总配置文件，ftptusers 是一个黑名单，user_list 是一个白名单(需要配置文件支持)。

vsftpd.conf 的配置 (grep -v "^#" vsftpd.conf > vsftpd.conf.bak) 如下：

```
1 local_enable=YES
2 write_enable=YES
3 local_umask=022
4 dirmessage_enable=YES
5 xferlog_enable=YES
6 connect_from_port_20=YES
7 xferlog_std_format=YES
8 listen=YES
9
10 pam_service_name=vsftpd
```

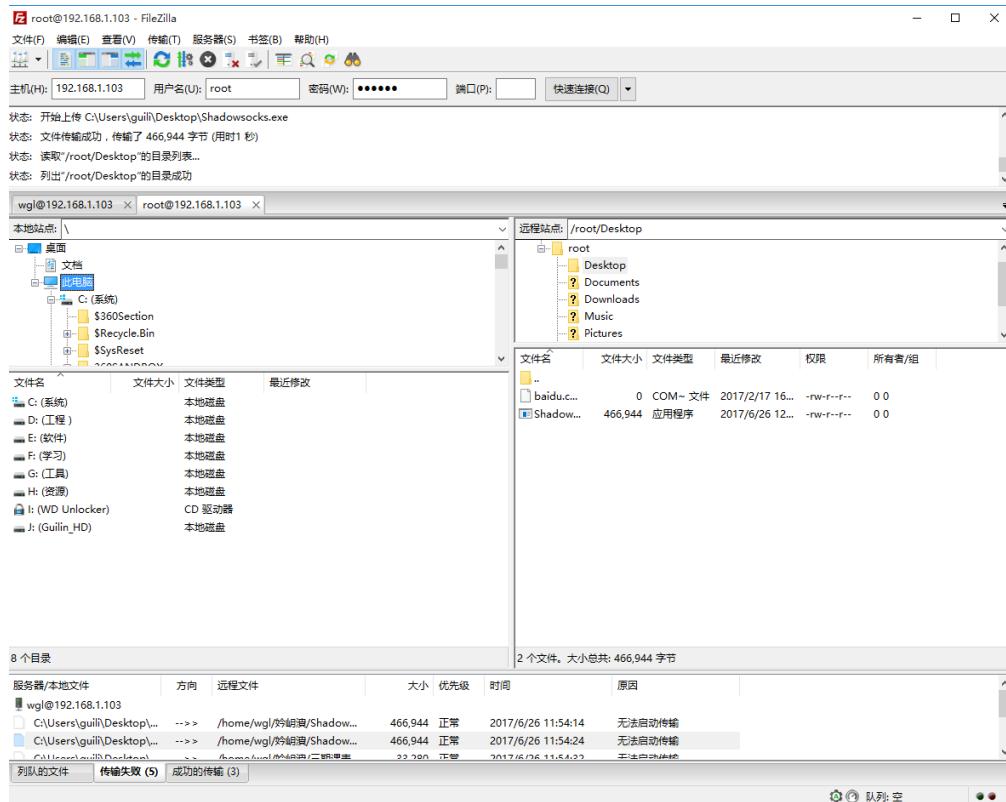
```

11 userlist_enable=YES      //启用 user_list
12 userlist_deny=NO        //设置 user_list 为白名单
13 tcp_wrappers=YES

```

配置完毕，用命令 service vsftpd restart 重启服务

2.5.3. 客户端访问



3. C 语言与自然语言

3.1.语法规则

任何一门计算机语言，其实都是在模仿自然语言(数学语言)，模仿的越像，这门语言更容易，也即更容易上手。

但是计算机语言是一种定义一机器上的一种语言，当然完全模仿是不太可能的，于是就有了规则，规则即语法。只有深谙语法，才能表达自如。

3.2.程序与流程

3.2.1.程序

程序一词来自生活，通常指完成某些事务的一种既定方式和过程在日常生活中，可以将程序看成对一系列动作的执行过程的描述。

生活中：



程序：为了让计算机执行某些操作或解决某个问题而编写的一系列有序指令的集合

计算机中：



3.2.2. 流程和流程图

3.2.2.1. 流程即算法

算法：解决问题的具体方法和步骤。算法是有优劣之分，在计算机中常用时间复杂度和空间复杂度来衡量的，在后序课程讲算法的时候会有详细的讲解。

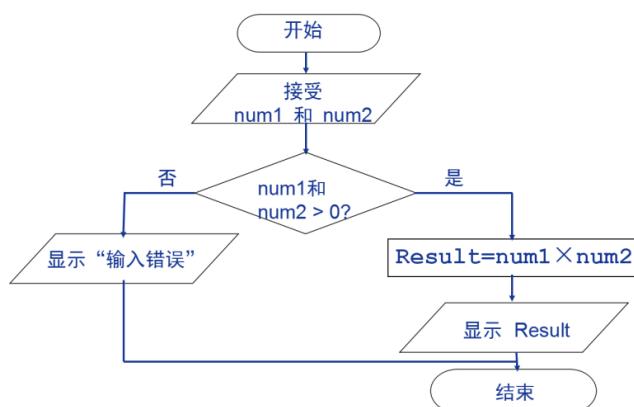
如何来描述算法呢？比如，我现在要计算一个长方形的面积。我就要用自然语言去描述他。

算法如下：

- ① 接收用户输入的长方形长度和宽度两个值；
- ② 判断长度和宽度的值是否大于零；
- ③ 如果大于零，将长度和宽度两个值相乘得到面积，否则显示输入错误；
- ④ 显示面积

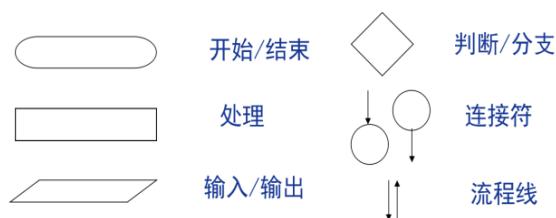
3.2.2.2. 流程图

为了更易于表达清楚，常采用流程图的方式：

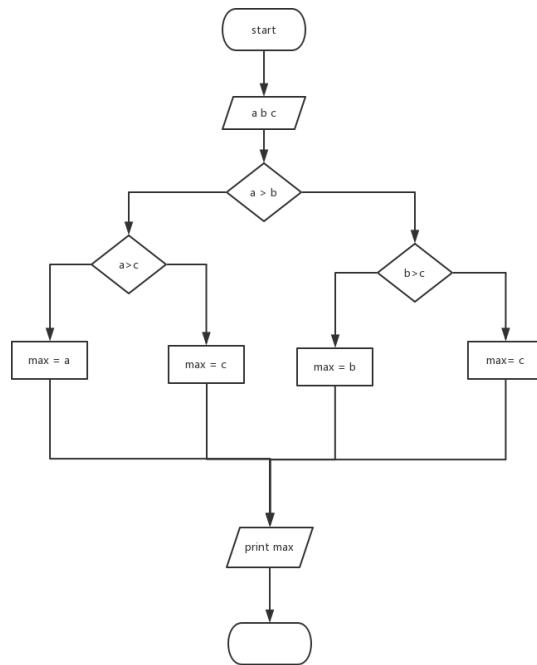


流程图是算法的一种图形化表示方式。流程图直观、清晰，更有利于人们设计与理解算法。

它使用一组预定义的符号来说明如何执行特定任务。



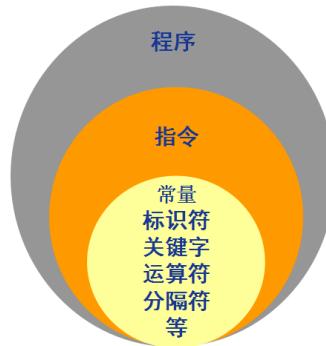
3.2.2.3.请用流程图 将 a,b,c 中最大值放到 max 中并输出。



3.2.3.常用制图工具

visio 2013 processon

4. 常/变量与数据类型(DataType)



当我们用计算机语言来描述世界的时候，比如一个人的性别，身高，体重，收入，就需要变量来存储，变量之间不仅需要**名字**来识别，还需要**类型**来进行限定。

4.1.C 语言中的关键字

关键字，也叫保留字。已经被 C 语言赋予了特殊的意义，所以我们既要研究其特殊的意义，也要跟其不发生冲突。

char	short	int	long	float	double	if	else
return	do	while	for	switch	case	break	continue
default	goto	sizeof	auto	register	static	extern	unsigned
signed	typedef	struct	enum	union	void	const	volatile

其中的 char/short/int /long /float,double , signed/unsigned 均是数值类型关键字。

4.2. 变量(Variable)

4.2.1. 变量的定义

类型 变量名;

4.2.2. 变量的命名规则：

- ❖ 变量名可以由字母、数字和 _ (下划线) 组合而成。
- ❖ 变量名不能包含除 _ 以外的任何特殊字符，如：%、#、逗号、空格等。
- ❖ 变量名必须以字母或 _ (下划线) 开头。
- ❖ 变量名不能包含空白字符 (换行符、空格和制表符称为空白字符)。
- ❖ C 语言中的某些词 (例如 int 和 float 等) 称为保留字，具有特殊意义，不能

用作变量名。

❖C 语言区分大小写，因此变量 `price` 与变量 `PRICE` 是两个不同的变量。

如下名字是合法的

`a_b` `a1b` `_Ab`

如下名字则不合法的

`a@b` `2ab` `a-b`

4.2.3. 交换两个变量的内容

```
int main()
{
    int a = 5;
    int b = 6;
    printf("a = %d b = %d \n",a,b);

//    a = b; b = a;
//    printf("a = %d b = %d \n",a,b);
    int t;
    t = a;
    a = b;
    b = t;
    printf("a = %d b = %d \n",a,b);
    return 0;
}
```

4.2.4. 驼峰命名法

骆驼式命名法就是当变量名或函式名是由一个或多个单词连结在一起，而构成的唯一识别字时，第一个单词以小写字母开始；第二个单词的首字母大写或每一个单词的首字母都采用大写字母，例如：`myFirstName`、`myLastName`，这样的变量名看上去就像骆驼峰一样此起彼伏，故得名。



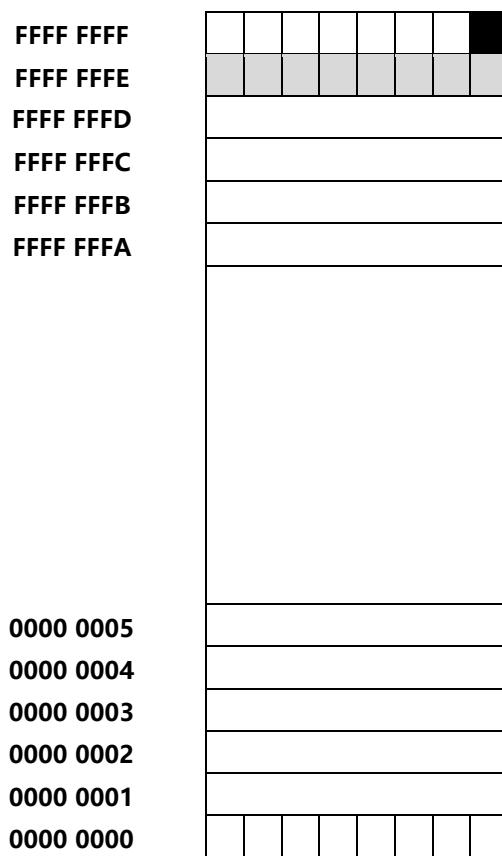
骆驼式命名法的命名规则可视为一种惯例，并无绝对与强制，为的是增加识别和可读性。

4.3.内存模型

4.3.1.物理基础

对于 32 机而言，内存模型线性的，这是硬件基础。左边表示十六进制的访问地址，右边分别表示寻址的最小单位 Byte 和 Byte 的最小单位 bit，1Byte = 8bit。

1B	=	8b
1KB	=	1024B
1MB	=	1024KB
1G	=	1024MB



4.3.2.CPU 读写内存

CPU 在运作时，读取内存数据，首先要指定存储单元的地址。就是要确实读写哪段数据。即要明确三件事。

- 存储单元的地址（地址信息）
- 器件的选择，读 or 写（控制信息）
- 读写的数据（数据信息）

4.3.2.1.地址总线

其中 CPU 通过地址总线要寻址，指定存储单元。可见地址总线上能传送多少个不同的信息，CPU 就可以对多少个存储单元进行寻址。有 10 根地址总线，就能传送 10 位二进制数据，也就是 2^{10} 次方。最小位 0，最大为 1023。也就是 $2^n =$ 最大传输 $n =$ 地址总线数。

CPU 地址总线的宽带决定了 CPU 的寻址能力。

4.3.2.2.数据总线

CPU 与内存或其他器件时行数据传达是通过数据总线来进行的。8 根数据总线一次可以传送 8 位二进制数据。16 根数据总线一次可以传 2 个字节。

数据总线的宽度决定了 CPU 和外界的数据传输速度。

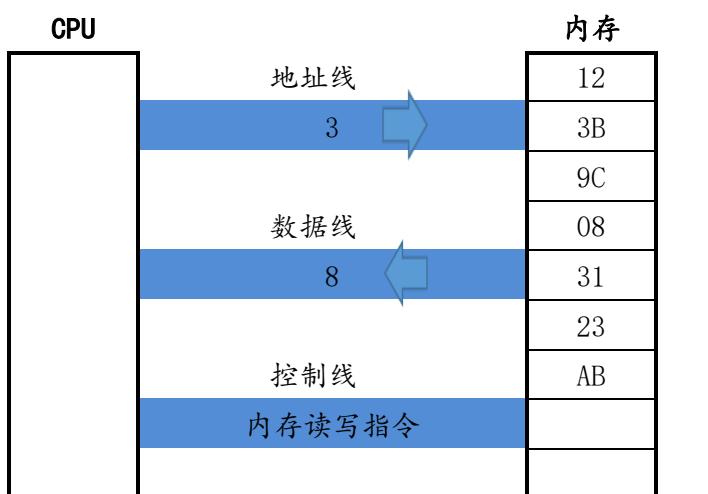
4.3.2.3.控制总线

CPU 对外部部件的控制时通过控制总线来进行的。控制总线是个总称，控制总线是有不同的控制线来集合的。有多少根控制总线，就意味着 CPU 提供了对外部器件的多少种控制。

控制总线的宽带决定了 CPU 对外部部件的控制能力。

4.3.3.读写流程：

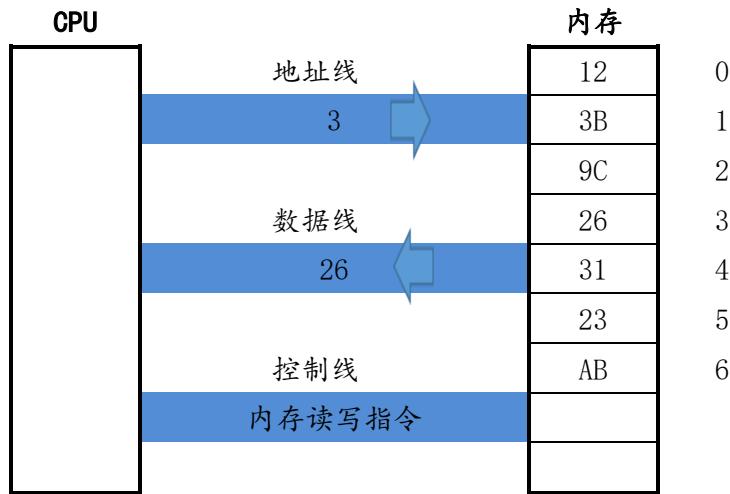
4.3.3.1.读取流程(从 3 单元读取数据)



- ① CPU 通过地址线将地址信息 3 发出。
- ② CPU 通过控制线发出内存读命令，选中存储器芯片，并通知它，将要从中读取数据。

- ③ 存储器将 3 号单元中的数据 8 通过数据线送入 CPU。

4.3.3.2. 写入流程：(将 26 写入单元 3)



- ① CPU 通过地址线将地址信息 3 发出。
- ② CPU 通过地址线发出内存写命令，选中存储器芯片，并通知它，要其写入数据。
- ③ CPU 通过数据线将数据 26 送入内存 3 单元中。

4.3.4. 代码演示

4.3.4.1.c 代码

```
#include <stdio.h>

int main()
{
    int a = 3;
    int b = 4;
    int c = a +b;
    return 0;
}
```

4.3.4.2. 汇编代码

```
6 [1]  int a = 3;
0x40161e  <+0x000e>          c7 44 24 0c 03 00 00 00  movl  $0x3,%eax(%esp)
7 [1]  int b = 4;
0x401626  <+0x0016>          c7 44 24 08 04 00 00 00  movl  $0x4,%eax(%esp)
8 [1]  int c = a +b;
0x40162e  <+0x001e>          8b 54 24 0c           mov    %eax(%esp),%edx
```

0x401632 <+0x0022>	8b 44 24 08	mov 0x8(%esp),%eax
0x401636 <+0x0026>	01 d0	add %edx,%eax
0x401638 <+0x0028>	89 44 24 04	mov %eax,0x4(%esp)
9 [1] return 0;		
0x40163c <+0x002c>	b8 00 00 00 00	mov \$0x0,%eax

4.4.计算机的基石补码

4.4.1.计算机编码基础

对于一个字节（8位），他所能表示的范围有多大呢？所有可能编码如下，共256种。

1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	0
1	1	1	1	1	1	0	1
1	1	1	1	1	1	0	0
<hr/>							
<hr/>							
1	0	0	0	0	0	0	0
0	1	1	1	1	1	1	1
<hr/>							
<hr/>							
0	0	0	0	1	0	1	
0	0	0	0	0	1	0	0
0	0	0	0	0	0	1	1
0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0

4.4.2.补码的编码规则

如何来利用这些硬件的基础来表示我们需要的数据范围呢？就是编码。

在计算机系统中，数值一律用补码来表示和存储。原因在于，使用补码，可以将符号位和数值域统一处理；同时，加法和减法也可以统一处理。此外，补码与原码相互转换，其运算过程是相同的，不需要额外的硬件电路。

4.4.2.1.模的概念

模的概念可以帮助理解补数和补码。

“模”是指一个计量系统的计数范围。如时钟等。计算机也可以看成一个计量机器，它也有一个计量范围，即都存在一个“模”。例如：

时钟的计量范围是 $0 \sim 11$, 模=12。表示 n 位的计算机计量范围是 $0 \sim 2^n - 1$, 模= 2^n 。

“模”实质上是计量器产生“溢出”的量，它的值在计量器上表示不出来，计量器上只能表示出模的余数。任何有模的计量器，均可化减法为加法运算。



例如：假设当前时针指向 10 点，而准确时间是 6 点，调整时间可有以下两种拨法：一种是倒拨 4 小时，即： $10-4=6$ ；另一种是顺拨 8 小时： $10+8=12+6=6$ 。

在以 12 模的系统中，加 8 和减 4 效果是一样的，因此凡是减 4 运算，都可以用加 8 来代替。对“模”而言，8 和 4 互为补数。实际上以 12 模的系统中，11 和 1，10 和 2，9 和 3，7 和 5，6 和 6 都有这个特性。共同的特点是两者相加等于模。

对于计算机，其概念和方法完全一样。 n 位计算机，设 $n=8$ ，所能表示的最大数是 11111111，若再加 1 成为 100000000(9 位)，但因只有 8 位，最高位 1 自然丢失。又回了 00000000，所以 8 位二进制系统的模为 2^8 。在这样的系统中减法问题也可以化成加法问题，只需把减数用相应的补数表示就可以了。

把补数用到计算机对数的处理上，就是补码。

4.4.2.2.补码运算

①正整数的补码是其二进制表示，与原码相同

②负数求补

求负整数的补码，将其对应正数二进制表示所有位取反（包括符号位，0 变 1，1 变 0）后加 1。

【例 1】求-5 的补码。

$-5 \Rightarrow |-5| \Rightarrow 0000\ 0101 \Rightarrow 1111\ 1010 \Rightarrow 1111\ 1011$

【例 2】数 0 的补码表示是唯一的。

$0 \Rightarrow 0000\ 0000 \Rightarrow 1111\ 1111 \Rightarrow 0000\ 0000$

③补码求原

unsi gned	sign ed
255	1 1 1 1 1 1 1
254	1 1 1 1 1 1 0
253	1 1 1 1 1 0 1
252	1 1 1 1 1 0 0
5	1 0 0 0 0 0 0
4	0 1 1 1 1 1 1
3	0 0 0 0 0 1 1
2	0 0 0 0 0 1 0
1	0 0 0 0 0 0 1
0	0 0 0 0 0 0 0

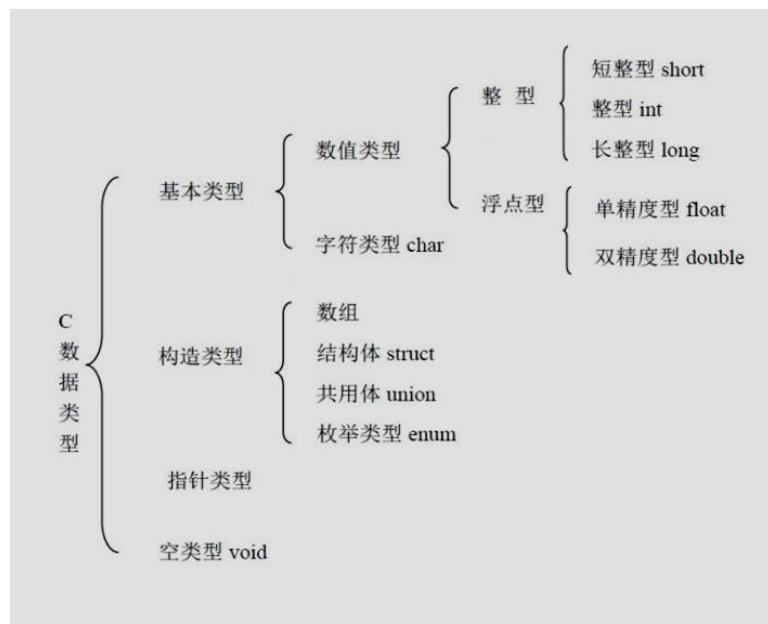
4.4.3. 小结

- 补码解决了，符号位参与运算，不需要单独标识。
- 补码解决了，正负 0 的表示方法。
- 补码实现了，减法变加法，同样也适应于乘法和除法。

4.5. 数据类型

数据类型，打个比方的话，就类似于部队中，前面我们已经给部队中人的起了名字，现在赋予这个权力和管辖范围，那就是军衔。或是通俗的讲，类型相当于模子，变量的定义，就相当于在内存中用模子框出的一块空间。

4.5.1. 类型总揽



4.5.2. 基本类型之数值类型

4.5.2.1. 如何求类型的大小

```

#include <stdio.h>

int main(void)
{
    printf("char      size = %d\n", sizeof(char));
    printf("short     size = %d\n", sizeof(short));
    printf("int       size = %d\n", sizeof(int));
    printf("long      size = %d\n", sizeof(long int));
    printf("long long size = %d\n", sizeof(long long));
    printf("float     size = %d\n", sizeof(float));
    printf("double    size = %d\n", sizeof(double));
    printf("long double size = %d\n", sizeof(long double));
    return 0;
}
  
```

4.5.2.2. 整型数表示范围

类型	位数(32位机)	符号	范围	科学计数
char	8	unsigned	0-255	$0-2^8-1$
		[signed]	-128-+127	-2^7-+2^7-1
short	16	unsigned	0-65535	$0-2^{16}-1$
		[signed]	-32768-+32767	$-2^{15}-+2^{15}-1$
int	32	unsigned		$0-2^{32}-1$

		[signed]		$-2^{31} \sim +2^{31}-1$
long	32	unsigned		$0 \sim 2^{32}-1$
		[signed]		$-2^{31} \sim +2^{31}-1$
long long	64	unsigned		$0 \sim 2^{64}-1$
		[signed]		$-2^{63} \sim +2^{63}-1$

4.5.2.3.浮点数的表示方法和范围

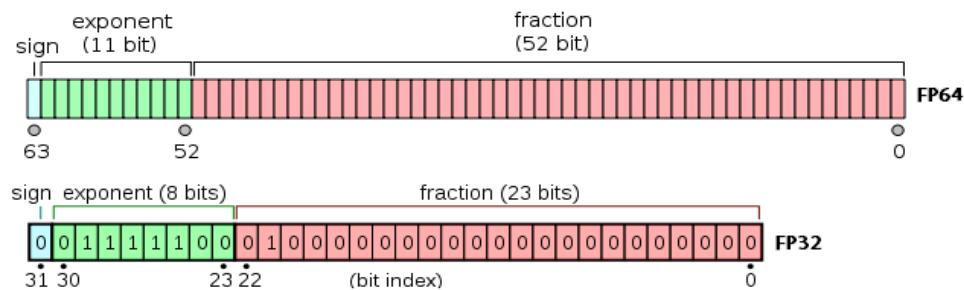
一个浮点数（Floating Point Number）由三个基本成分构成：符号（Sign）、阶码（Exponent）和尾数（Mantissa）。

通常，可以用下面的格式来表示浮点数：



其中 S 是符号位，P 是阶码，M 是尾数。

对于 32/64 位的单/双精度浮点型如下：



类型	位数(32 位机)	范围简算	精算	有效位数
float	32	$[2^{-128}, 2^{127}]$	$[10^{-38}, 10^{38}]$	6-7
		$[-2^{127}, -2^{-128}]$	$[-10^{-38}, -10^{-38}]$	6-7
double	64	$[2^{-2048}, 2^{2047}]$	$[10^{-308}, 10^{308}]$	15-16
		$[-2^{2047}, -2^{-2048}]$	$[-10^{-308}, -10^{-308}]$	15-16

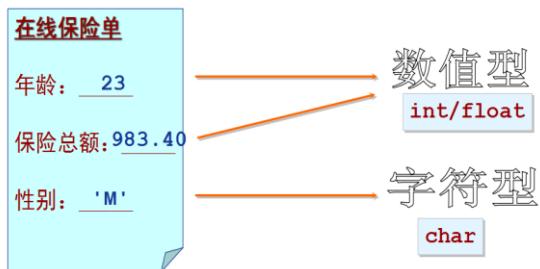
```
#include <stdio.h>

int main(void)
{
    float fvar = 0.1234567654321;
    printf("fvar = %.13f\n", fvar);

    double dvar = 0.1234567654321;
    printf("dvar = %.13f\n", dvar);
    return 0;
}
```

4.5.3. 基本类型之字符类型

4.5.3.1. 引入



但是计算机是不能显示字符的，只能在显示的时候进行转化。这种转化需要一张表，表里记载了转换规则，这张表，就是 ASCII 编码表。该表的本质就是一种对应关系。

4.5.3.2. 表示形式

字符的表现形式，就是用单引号，引起的一个字符。形如下：

```
int main(void)
{
    char val = 'a';
    for(; val<='z'; val++)
    {
        printf("%c\t",val);
    }
    return 0;
}
```

4.5.3.3. ASCII 编码

目前使用最广泛的西文字符集及其编码是 ASCII 字符集和 ASCII 码（ASCII 是 American Standard Code for Information Interchange 的缩写），它同时也被国际标准化组织(International Organization for Standardization, ISO)批准为国际标准。

基本的 ASCII 字符集共有 128 个字符，其中有 96 个可打印字符，包括常用的字母、数字、标点符号等，另外还有 32 个控制字符。下表展示了基本 ASCII 字符集及其编码：

ASCII							
ASCII	字符	ASCII	字符	ASCII	字符	ASCII	字符
0	NUL	32	space	64	@	96	'
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	\$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(72	H	104	h
9	TAB	41)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[123	{
28	FS	60	<	92	\	124	
29	GS	61	=	93]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	

4.5.3.4.自实现打印

```
#include <stdio.h>

int main(void)
{
    unsigned char a; // 用有符号来打印，则容易溢出
    for(a = 0; a < 128; a++)
    {
        printf("%d-->%c\t\t", a,a);
        if(a%8 == 0)
            printf("\n");
    }
    return 0;
}
```

打印结果

0->	1->	lf	2->	nl	3->	ll	4->	u	5->		6->	=	7->		
8->	9->		10->												
14->	15->	*	13->												
16->	17->		18->	+	19->		20->		21->	ll	22->	lf	23->		
24->	25->	lf	26->		27->	←	28->	↑	29->		30->	→	31->	↔	
32->	33->	!	34->		35->	#	36->	\$	37->	%	38->	&	39->	,	
40->	(41->)	42->	*	43->	+	44->	,	45->	-	46->	.	47->	/
48->	0	49->	1	50->	2	51->	3	52->	4	53->	5	54->	6	55->	7
56->	8	57->	9	58->	:	59->	:	60->	<	61->	=	62->	>	63->	?
64->	@	65->	A	66->	B	67->	C	68->	D	69->	E	70->	F	71->	G
72->	H	73->	I	74->	J	75->	K	76->	L	77->	M	78->	N	79->	O
80->	P	81->	Q	82->	R	83->	S	84->	T	85->	U	86->	V	87->	W
88->	X	89->	Y	90->	Z	91->	[92->	\	93->]	94->	^	95->	_
96->	`	97->	a	98->	b	99->	c	100->	d	101->	e	102->	f	103->	g
104->	h	105->	i	106->	j	107->	k	108->	l	109->	m	110->	n	111->	o
112->	p	113->	q	114->	r	115->	s	116->	t	117->	u	118->	v	119->	w
120->	x	121->	y	122->	z	123->	{	124->		125->	}	126->	~	127->	

4.5.3.5.转义字符

为啥叫转义字符呢，就是他本身已有一个字符的意义，我们给他另外一个别的意义。转意字符常起到控制的作用。

转义字符及其意义

字符	意义	ASCII 值	备注
\b	退格(BS)当前位置向后回退一个字符	8	转义后，值发生了改变，通常在 0-32 以内表示控制字符。
\r	回车(CR),将当前位置移至本行开头	13	
\n	换行(LF),将当前位置移至下一行开头	10	
\t	水平制表(HT),跳到下一个 TAB 位置	9	
\0	用于表示字符串的结束标记	0	用作字符串结束标志

\\	代表一个反斜线字符 \	92	
\"	代表一个双引号字符"	34	
\'	代表一个单引号字符'	39	
%%	代表一个百分号%	37	
\ddd	1 到 3 位八进制所代表的任意字符	八进制数制	可表示任意的字符。注意跟后续字符 沾连 的问题。
\xhh	1 到 2 位十六进制所代表的任意字符	十六进制数值	

案例：

```
#include <stdio.h>

int main(void)
{
#if 0
    printf("r = %d n = %d t = %d\n",'r','n','t');
    printf("r = %d n = %d t = %d\n",'\\r','\\n','\\t');
    printf("\r = %d \n = %d \t = %d\n",'\\r','\\n','\\t');
    printf("\\\r = %d \\n = %d \\\t = %d\n",'\\r','\\n','\\t');

    printf("aaaaaaaa\rbbbbbbbbbbbb\n"); //回车 覆盖
    printf("aaaaaaaa\r\nbbbbbbbbbbbb\n");
    printf("aaaaaaaa\nbbbbbbbbbbbb\n");
    printf("mmmmmmmm\tggggggggggggg");

#endif

    printf("i like football\n"); //i like "football"
//    printf("i like "football\"");
    printf("i like \"football\"\n");

    printf("this is liming\'book\n");

    printf("the rate is %d%%\n",95);

    printf("aaaaaaaaaaaa\xd bbbbbbbbbbbb\n");
    printf("aaaaaaaaaaaa\xd\bbbbbbbbbbb\n");

    printf("i like \x22 football\x22 \n"); //i like "football"
    printf("i like \x22 \bfootball\x22 \n");

    return 0;
}
```

4.5.3.6.注意事项

```
char ch = 'abcd'; //编译亦可以通过
```

4.6.常量(Constant)

常量是程序中不可改变的量，常以**字面量(Literal)**，或者**宏(Macro)**的方式出现。主要用于赋值或是参与计算，并且常量也是用类型的。

```
#include <stdio.h>

#define PI 3.1425926
#define PRICE 200

int main(void)
{
    int a = 50;
    return 0;
}
```

4.6.1.常量类型

4.6.1.1.整型常量

整型常量有三种表形式

1.十进制表示：

除表示整数 0 外，不以 0 开头（以 0 开头的数字串会被解释成八进制数）。负数在前面加负号'-'，后缀'L'或'L'表示长整型，'u'或'U'表示无符号数。

例：345 31684 0 -23456 459L 356l 56789u 567LU

2.八进制表示：

以数字 0 开头的一个连续数字序列，序列中只能有 0-7 这八个数字。

例：045 -076 06745l 0177777u

3.十六进制表示：

以 0X 或 0x 开头的连续数字和字母序列，序列中只能有 0-9、A-F 和 a-f 这些数字和字母，字母 a、b、c、d、e、f 分别对应数字 10、11、12、13、14、15，大小写均可。

例：0x10 0X255 0xd4ef 0X6a7bL

4.6.1.2.实型常量

实型常量有两种表示形式：

1 小数形式。

由数字和小数点组成，必须有小数点。

例：4.23、0.15、.56、78.、0.0

2.指数形式：

以幂的形式表示，以字母 e 或 E 后跟一个以 10 为底的幂数。

(1) 字母 e 或 E 之前后必须要有数字。

(2) 字母 e 或 E 后面的指数必须为整数，字母 e 或 E 的前后及数字之间不得有空格。

默认是 double 型，后缀为 “f” 或 “F” 即表示该数为 float 型，后缀 “l” 或 “L” 表示 long double 型。

例：2.3e 500e-2.5 .5E3 4.5e0 34.2f .5F 12.56L 2.5E3L
请问有哪些不合法呢？

4.6.1.3.字符常量

字符常量的表现形式比较简单。以单引号引起的一个字符。

例：'a' 'b' 'c'

4.6.1.4.字符串常量

字符串常量的表现形式比较简单。以单引号引起的一串字符。

例："a" "abcdefg"

4.6.2.常量的大小

默认的整型常量是 int 类型，默认的实型是 double 类型。

```
int main(void)
{
    printf("const sizeof(0.0) = %d\n", sizeof(0.0)); //double
    printf("const sizeof(0.0f) = %d\n", sizeof(0.0f)); //0.0F float
    printf("const sizeof(0.0L) = %d\n", sizeof(0.0L)); //0.0l long double

    printf("const sizeof(0) = %d\n", sizeof(0)); //int
    printf("const sizeof(0l) = %d\n", sizeof(0l)); //long
    printf("const sizeof(0ll) = %d\n", sizeof(0ll)); //long long

    //常量背后的标识，多数是为了节省空间
    printf("const sizeof(0ll) = %d\n", sizeof(2147483648));
    //0xffffffff 不同平台实现略有差异
    printf("const sizeof(0ll) = %d\n", sizeof(2147483648u));
    return 0;
}
```

4.7.格式输入与输出

4.7.1.printf

4.7.1.1.基本使用介绍

printf 函数称为格式输出函数，其关键字最末一个字母 f 即为 “格式” (format) 之意。其功能是按用户指定的格式，把指定的数据显示到显示器屏幕上。

printf 函数调用的一般形式：

```
printf(" 格式控制字符串", [输出表列]);
```

其中格式控制字符串用于指定输出格式。格式控制串可由格式字符串和非格式字符串两种组成。格式字符串是以%开头的字符串，在%后面跟有各种格式字符，以说明输出数据的类型、形式、长度、小数位数等。如：

- "%d" 表示按十进制整型输出。
- "%f" 表示按实型数据输出。
- "%c" 表示按字符型输出等。

非格式字符串原样输出，在显示中起提示作用。输出表列中给出了各个输出项，要求格式字符串和各输出项在数量和类型上应该一一对应。

4.7.1.2.格式串

形式：“% [标志][输出最小宽度][.精度][长度]类型”

类型：

类型字符用以表示输出数据的类型，其格式符和意义如下表所示：

类型	语义	Example
d / i	有符号 10 进制整型	392
u	无符号 10 进制整型	7235
o	无符号 8 进制整型	610
x	无符号 16 进制整型	7fa
X	无符号 16 进制整型(大写)	7FA
f	单/双精度浮点型(默认打印六位小数)	392.650000
e	科学计数 e	3.9265e+2
E	科学计数 E	3.9265E+2
g	%e or %f 的缩短版	392.65
G	%E or %F 的缩短版	392.65
c	字符	a
s	字符串	sample
p	地址	b8000000

宽度：

用十进制整数来表示输出的最少位数。若实际位数多于定义的宽度，则按实际位数输出，若实际位数少于定义的宽度则补以空格或 0。

标志：

标志	意义

-	左对齐；默认右对齐	
+	当一个数为正数时，前面加上一个+号， 默认不显示	
0	右对齐时，用 0 填充左边未使用的列；默认用空格填充	
空格	输出值为正时冠以空格，为负时冠以负号	
#	对 c、s、d、u 类无影响	
	对 o 类，在输出时加前缀 o	
	对 x 类，在输出时加前缀 0x	

精度：

精度格式符以“.”开头，后跟十进制整数。本项的意义是：如果输出数字，则表示小数的位数；如果输出的是字符，则表示输出字符的个数；若实际位数大于所定义的精度数，则截去超过的部分。

长度：

length	d i	u o x X	f F e E g G a	c	s	p
(none)	int	unsigned int	float/double	int	char*	void*
hh	signed char	unsigned char				
h	short int	unsigned short int				
l	long int	unsigned long int		wint_t	wchar_t*	
ll	long long int	unsigned long long int				
L			long double			

注意： hh lf 的平台差异性

示例：

```
#include <stdio.h>

int main(void)
{
    int var = 1234;

    printf("%d\n", var);
    printf("%d%%\n", var);
    printf("%o\n", var);
    printf("%x\n", var);

    printf("=====\\n");

    printf("%d\\n", var);
    printf("%#o\\n", var); // 八进制以 0 开头
}
```

```
printf("%#x\n",var);      // 十六进制以 0x 开头

printf("=====\\n");

printf("%10d\\n",var);
printf("%010d\\n",var);    //前面的空间补以 0
printf("%1d\\n",var);
printf("%-10d\\n",var);   //左对齐

printf("=====\\n");

printf("%+d\\n",var);
var = -var;
printf("%+d\\n",var);     //显示正负号

printf("=====\\n");

var = -1;
printf("%d\\n",var);
printf("%u\\n",var);       //很大很大的数哟

printf("=====\\n");

// //////////////////////

char c = '\\x41';

printf("%c\\n",c);
printf("%d\\n",c);
printf("%x\\n",c);
printf("%#x\\n",c);
printf("=====\\n");

// //////////////////////

float f = 123.4567;

printf("%f\\n",f);
printf("%7.2f\\n",f);
printf("%-7.2f\\n",f);

printf("%e\\n",f);
printf("%g\\n",f);
```

```

printf("=====\\n");
// //////////////////////////////

short a = 32;
long int b =64;
long double cd = 100.001;

printf("%hd\\n",a);
printf("%ld\\n",b);
printf("%Lf\\n",cd); //32 位机下支持并不好， 64 适用
return 0;
}

```

4.7.2.scanf

4.7.2.1.基本使用介绍

scanf 函数的一般形式为：

```
scanf(“格式控制字符串”， 地址表列);
```

其中，格式控制字符串的作用与 printf 函数相同，地址表列中给出各变量的地址。地址是由地址运算符“&”后跟变量名组成的。

例如：&a、&b 分别表示变量 a 和变量 b 的地址。

这个地址就是编译系统在内存中给 a、b 变量分配的地址。在 C 语言中，使用了地址这个概念，这是与其它语言不同的。应该把变量的值和变量的地址这两个不同的概念区别开来。变量的地址是 C 编译系统分配的，用户不必关心具体的地址是多少。

4.7.2.2.格式串

格式字符串的一般形式为：

%[*][输入数据宽度][长度]类型

其中有方括号[]的项为任选项。各项的意义如下。

类型：

格式	字符意义
d	输入十进制整数
o	输入八进制整数
x	输入十六进制整数
u	输入无符号十进制整数
f	输入实型数
c	输入单个字符
s	输入字符串

* :

用以表示该输入项，读入后不赋予相应的变量，即跳过该输入值。

宽度：

长度格式符为 l 和 h，l 表示输入长整型数据（如%ld）和双精度浮点数（如%lf），h 表示输入短整型数据。

注意事项：

- 1，待输入变量，必是以其地址的方式呈现。
- 2，除格式字符，**空格**，回车，Tab 外，其它需要原样输入。
- 3，空格，Tab，回车，均可作为输入间隔，以回车作为结束。
- 4，scanf 函数中没有精度控制，如：scanf("%5.2f",&a); 是非法的。不能企图用此语句输入小数为 2 位的实数。
- 5，在输入字符数据时，若格式控制串中无非格式字符，则认为所有输入的字符均为有效字符。**%d %c %c %c**
- 6，如输入的数据与输出的类型不一致时，虽然编译能够通过，但结果将不正确。

```
#include <stdio.h>

int main(void)
{
//    int a = 1,b = 2,c = 3;

//    scanf("%d%d%d",&a,&b,&c);
//    printf("a = %d b = %d c = %d\n",a,b,c);

//    scanf("%d%*d%d",&a,&b,&c); // 忽略缓冲区中间的输入
//    printf("a = %d b = %d c = %d\n",a,b,c);

//    scanf("%5d",&a);
//    printf("a = %d\n",a);

//    scanf("%4d%4d",&a,&b);
//    printf("a = %d b = %d \n",a,b);

//    scanf("%c%c%c",&a,&b,&c);
//    printf("a = %c b = %c c = %c\n",a,b,c);

//    char a = 1,b = 2,c = 3;

//    scanf("%c, %c, %c",&a,&b,&c);
```

```
//     printf("a = %c  b = %c  c = %c\n",a,b,c);

//     char a = 1,b = 2,c = 3;
//     scanf("a=%c,b=%c,c=%c",&a,&b,&c);
//     printf("a = %c  b = %c  c = %c\n",a,b,c);
float var;
scanf("%d",&var); //编译没有问题，但结果并不正确
printf("%f",var);
return 0;
}
```

4.7.3.putchar && getchar

int putchar (int character);	向屏幕输出一个字符
int getchar (void);	从键盘获得一个字符

```
#include <stdio.h>

int main(void)
{
    char ch;

    printf("pls input a char:");
    putchar(10);
    ch = getchar();
    putchar(ch);

    return 0;
}
```

4.7.4.输入输出缓冲

4.8.类型转化(Type Cast)

4.8.1.隐式转化

不需要人为参与而产生的默认转称为隐式转化。隐式转化，是任何语言层面最复杂的东西。当下，我们需要了解的只是一些规则，不考虑**溢出,截断和有无符号**。后期的c提高，会有更为详细的介绍。

4.8.1.1. 算术转化

■ 整形提升：

char short int 等类型在一起运算时，首先提升到 int，这种现象叫作整形提升。整形提升的原更换是符号扩充。

■ 混合提升：

在进行运算时，以表达式中最长类型为主，将其他类型位据均转换成该类型，如：

- (1)若运算中最大范围为 double，则转化为 double。 (10^{308})
- (2)若运算中最大范围为 float 则转化为 float。 (10^{38})
- (3)若运算中最大范围为 long long 则转化为 long long。 (2^{64})
- (4)若运算中最大范围为 int 则转化为 int。 (2^{32})
- (5)若运算中有 char short 则一并转化为 int。 $(2^8 / 2^{16} / 2^{32})$

```
#include <stdio.h>

int main(void)
{
    char a; short b; int c;
    long long d; float e; double f;

    printf("sizeof(a) = %d\n", sizeof(a));
    printf("sizeof(b) = %d\n", sizeof(b));
    printf("sizeof(c) = %d\n", sizeof(c));

    printf("sizeof(a+b) = %d\n", sizeof(a+b));
    printf("sizeof(a+c) = %d\n", sizeof(a+c));
    printf("sizeof(b+c) = %d\n", sizeof(b+c));

    printf("sizeof(d) = %d\n", sizeof(d));
    printf("sizeof(a+d) = %d\n", sizeof(a+d));

    c = 3; e = 5.6;
    printf("%f\n", c+e); // %d 则会乱了。 int 转化为 float 了。

    e = 10.0;
    printf("%f\n", e/c); // int ->float

    printf("sizeof(a) = %d\n", sizeof(a));
    printf("sizeof(a) = %d\n", sizeof(f));

    printf("sizeof(d+e) = %d\n", sizeof(d+e)); //4
    printf("sizeof(a+f) = %d\n", sizeof(a+f)); //8
```

```

    return 0;
}

```

4.8.1.2.赋值转化

整型和实型之间是可以相互赋值的。赋值的原则是，一个是加零，一个是去小数位。

```

#include <stdio.h>

int main(void)
{
    int a = 5; float b = 10.5;
    int aa = 5; float bb = 10.5;

    a = b;
    printf("a = %d\n",a);
    bb = aa;
    printf("bb = %f\n",bb);
    return 0;
}

```

4.8.1.3.隐式转化规则

高 低		double	
		float	
		long long	
		long	
		int	char short

4.8.2.强制转化

隐式类型转化，是有缺陷的，当隐式类型转化不能满足我们的需求时，就需要强制类型转化。

格式：

(类型) 待转表达式

```

#include <stdio.h>

int main(void)
{
    int a = 3; int b = 10;
    //    float c = b/a;
    //    printf("%f\n",c); //当我们想要的结果是 3.3 的时候
}

```

```

float c = (float)b/a;
printf("%f\n",c); //当我们想要的结果是 3.3 的时候

return 0;
}

```

4.9.练习

4.9.1.浮点数跟 0 值比较

无论是 float 还是 double 类型的变量，都有精度限制。所以一定要避免将浮点变量用“==”或“=”与数字比较，应该设法转化成“>=”或“<=”形式。

假设浮点变量的名字为 x，应当将 if (x == 0.0) // 隐含错误的比较转化为 if ((x>=-EPSINON) && (x<=EPSINON))其中 EPSINON 是允许的误差（即精度）。

```

#include <stdio.h>
int main()
{
//    double tmp = 0.3; //0.3 0.7 坑
//    printf("%d\n", (int)((tmp+0.0000001) *10));

    float a = 0.1;

    if(a*10 >= 1.0 - 0.0000001 && a*10 <= 1.0+0.000001)
    {
        printf("====");
    }
    else
    {
        printf("xxxx");
    }
    return 0;
}

```

4.9.2.printf 返回值有意义吗？

```

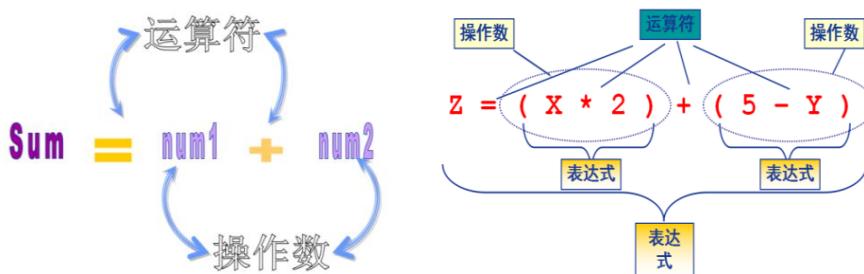
#include<stdio.h>
int main()
{

```

```
int i=43;
printf("%d\n",printf("%d",printf("%d",i)));
return 0;
}
```

5. 运算符与表达式(Operator& Express)

我们前面，已经学习类型，常量与变量，现在就要通过运算符，对他们进行运算。表达式是由操作数和运算符构成，操作数本身也可以是一个表达式。表达式再在其后加一个分号，即构成 C 语言语句。



5.1. 运算符的优先级/结合性/分类

在小学的数学学习中，我想有一句话，大家都很清楚的记得。那就是“**从左往右，先乘除，后加减，有括号的先算括号里面的**”。

上面的这句话，就是蕴含了，**优先级和结合性**的问题，比如，从左往右，表示从左向右结合，这就是数学成的结合性。计算机中除了，从左往右的结合性以外，还有从右往左的结合性。比如赋值运算符。

```
int main(void)
{
    int a , b, c, d;
    a = b = c = d = 5; // (a = (b = (c = (d = 5))));
    return 0;
}
```

还有，乘除的优先级，比加减要高，()的优先级要比乘除要高，计算机中亦是如此。

另外，计算机中还按操作的数的个数将其分为 3 类，单目运算符，双目运算符和三目运算符。

```
a = 4 + 6; // 加号即为双目运算符。
a++; // 自加运算符，即为单目运算符。
a>b ? a:b; // 条件表达式，就三个操作数，即为三目运算符
```

c/c++语全部运算符，预览如下：

Precedence	Operator	Description	Associativity
1	<code>++ --</code> <code>()</code> <code>[]</code> <code>.</code> <code>-></code>	Suffix/postfix increment and decrement Function call and subexpression Array subscripting Structure and union member access Structure and union member access through pointer	Left-to-right
2	<code>++ --</code> <code>+ -</code> <code>! ~</code> <code>(type)</code> <code>*</code> <code>&</code> <code>sizeof</code>	Prefix increment and decrement Unary plus and minus Logical NOT and bitwise NOT Type cast Indirection (dereference) Address-of Size-of	Right-to-left
3	<code>* / %</code>	Multiplication, division, and remainder	
4	<code>+ -</code>	Addition and subtraction	
5	<code><< >></code>	Bitwise left shift and right shift	
6	<code>< <=</code> <code>> >=</code>	For relational operators <code><</code> and <code>≤</code> respectively For relational operators <code>></code> and <code>≥</code> respectively	
7	<code>== !=</code>	For relational <code>=</code> and <code>≠</code> respectively	Left-to-right
8	<code>&</code>	Bitwise AND	
9	<code>^</code>	Bitwise XOR (exclusive or)	
10	<code> </code>	Bitwise OR (inclusive or)	
11	<code>&&</code>	Logical AND	
12	<code> </code>	Logical OR	
13	<code>? :</code>	Ternary conditional	
14	<code>=</code> <code>+= -=</code> <code>*= /= %=</code> <code><<= >>=</code> <code>&= ^= =</code>	Simple assignment Assignment by sum and difference Assignment by product, quotient, and remainder Assignment by bitwise left shift and right shift Assignment by bitwise AND, XOR, and OR	Right-to-Left
15	<code>,</code>	Comma	Left-to-right

不同优先级的运算符，运算次序按优先级由高到低执行。

同一优先级的运算符，运算次序按结合方向的规定执行。

5.2.常见运符算符及构成的表达式

这些内容，比较多不要求一并识记，而是运用中，自然了解。下面我们针对，常用运算符——解析。

表达式既有类型，也是有值的。

5.2.1.赋值运算符与赋值表达式

5.2.1.1.列表

运算符	释意	结合性	分类	示例
=	赋值运算符	从右往左	双目	a = b
+=	复合赋值运算符	从右往左	双目	a+=b
-=	复合赋值运算符	从右往左	双目	a-=b
=	复合赋值运算符	从右往左	双目	a=b
/=	复合赋值运算符	从右往左	双目	a/=b

5.2.1.2.解析

- ❖ 把右侧表达式的值赋给左侧的变量；表达式的值等于左值。
- ❖ 运算符两侧类型不一致时，要进行类型转换；
- ❖ 赋值表达式可嵌套。
- ❖ 复合赋值运算符是一种简洁的需要。

5.2.1.3.实战

```
#include <stdio.h>

int main(void)
{
    int a,b;
    a=5;           /*表达式的值为 5*/
    a=b=3;         /*a=(b=3)*/
    a=(b=4)+(c=3); /*b=4, c=3, a=?*/ //

    a = a + b;
    a += b;
    return 0;
}
```

注：表达式是有值的

5.2.2. 算术运算符与算术表达式

5.2.2.1. 列表

运算符	释意	结合性	示例
+	加法运算符	从左往右	a+b
-	减法运算符	从左往右	a=b
*	乘法运算符	从左往右	a*=b
/	除法运算符	从左往右	a/b
%	求余运算符	从左往右	a%b
前++	先++后使用	right to left	++a
后++	先使用后++	left to right	a++
前--	先--后使用	right to left	--a
后--	先使用后--	left to right	a--

5.2.2.2. 解析

❖ “/”--除法运算，如果参与运算的两个操作数皆为整数，那么结果也为整数，此处直接取整，没有四舍五入的说法。如果其中有一个是浮点数，那么运算结果一定是浮点数。

❖ “%”--数学语言中，除数除以被除数得商和余数，是一体化的。但是 c 语言中却是分开的。% 表示求余运算，且求余运算的操作数只能是整型数据。通常用求余运算表示是否整除，余数为零则表示整除。还比如判断数值奇偶 a%2。

❖ %求余运算符要求操作数必须为整型，为什么不可以是浮点型呢？因为浮点数不存在求余问题。

❖ ++/--运算独立，则前后无异，若运算不独立则会有先使用和先运算之分。a++++；不需要懂，但是要懂的 a+++b；自增与自减运算也是一种 c 语言简洁的需要。

5.2.2.3. 实战

```
#include <stdio.h>

int main(void)
{
    int a = 10; int b = 3;
    int ires = a/b;
    printf("ires = %d\n", ires);

    float c = 3.0;
    float fres = a/c;
    printf("fres = %f\n", fres);

    int iret = 10/3;
}
```

```

printf("ret = %d\n",iret);

float fret = (float)10/3; //在此实现的是得到 3.33333
printf("fret = %f\n",fret);

//输出以下结果：
// 7/3 -7/3 7/-3
// 7%3 -7%3 7%-3      a%b <==> a - a/b *b

a++; ++a; a--; --a; //独立表达式，效果是一样的
b = a++; b = ++a; //
printf("%d",a++);
if(a++); while(a++);

return 0;
}

```

5.2.2.4.常见应用类型

判断整除问题。分离位数问题，有两种方法：

5.2.3.关系运算符与关系表达式

5.2.3.1.列表

运算符	释意	结合性	示例
>	大于运算符	从左往右	a>b
<	小于运算符	从左往右	a<b
==	等于运算符	从左往右	a==b
!=	不等于运算符	从左往右	a!=b
>=	大于等于运算符	从左往右	a>=b
<=	小于等于运算符	从左往右	a<=b

5.2.3.2.解析

- ❖ 关系表达式值，成立为真（值为 1），不成立为假（值为 0）。C 语言中，没有表示真假的量，非零即表示真，零表示假。
- ❖ 关系运算符，跟数学中的运算符基本一样。只不过 = 用作了赋值运算，而 == 用作了关系等。为什么不像数学中一样，= 表示关系等呢。因为=的使用频率太高了。

5.2.3.3.实战

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

```

int main(void)
{
    int a = 3; int b = 5;

    if(a == b)
    {
        printf("a == b\n");
    }
    else
    {
        printf("a != b\n");
    }

    printf("express a == b value = %d\n",a == b);
    printf("express a != b value = %d\n",a != b);

    return 0;
}

```

5.2.3.4.常见应用类型

表示数值区间，比如，数学语言中的[5,100]，或是(-∞,5] ∪ [100,+∞)，如何用计算机语言来表达呢？

5.2.4.逻辑运算符与逻辑表达式

5.2.4.1.列表

运算符	释意	结合性	示例
&&	与	从左往右	a&&b
	或	从左往右	a b
!	非	从左往右	!a

5.2.4.2.解析

真值表					
&&				!	
1&&1	1	1 1	1	! 1	0
1&&0	0	1 0	1	! 0	1
0&&1	0	0 1	1		
0&&0	0	0 0	0		

- 首先将**运算**数值，按逻辑运算进行处理，零为假，非零为真。然后然真值表进行运算。运算的结果也为逻辑的真与假。

- ❖ &&运算，只要有一个为假，即为假。||运算，只要有一个为真，即为真。
- ❖ ! 运算，用来取反，优先级要高于其它两个同类型运算符。
- ❖ 逻辑运算，存在短路现象。&&当两个运算量都是真时，其结果为真。因此，当左边已是 0，右边的表达式不再求解。|| 当两个运算量有一个为真时，其结果为真。因此，当左边已是 1，右边的表达式不再求解。短路是一种效率的需求。

5.2.4.3.实战

```
#include <stdio.h>

int main(void)
{
    int a = 0; int b = 0; // a = 3 或是 0 结果是不一样的
    if(a && b)
    {
        printf("a && b is true\n");
    }
    else
    {
        printf("a && b is false\n");
    }

    if(a || b)
    {
        printf("a || b is true\n");
    }
    else
    {
        printf("a || b is false\n");
    }

    if(!a)
    {
        printf("!a is true\n");
    }
    else
    {
        printf("!a is false\n");
    }

    printf("a = %d b = %d\n",a,b);

    if( (a < 5) || (b = 3))//注意赋值优先级及短路特点
    //if( (a >5) && (b = 3))//注意赋值优先级及短路特点
```

```

{
    printf("a = %d b = %d\n",a,b);
}
printf("a = %d b = %d\n",a,b);
return 0;
}

```

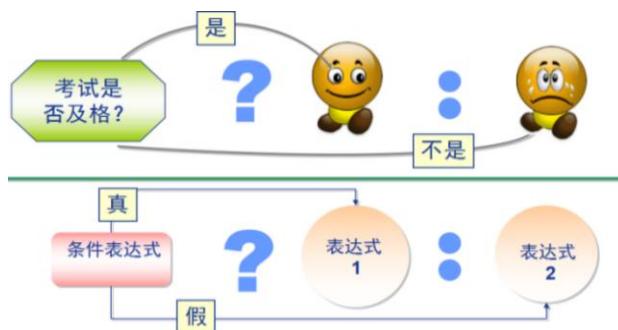
5.2.5. 条件运算符与条件表达式

5.2.5.1. 列表

运算符	释意	结合性	示例
? :	条件运算符	从右往左	a?b:c

5.2.5.2. 解析

条件运算符，实际上是一种 `if else` 结构的简化表达方式。C 语言致于更简洁。并且支持嵌套。



- ◆ 条件运算符的运算优先级低于关系运算符和算术运算符，但高于赋值符。
- ◆ 条件运算符的结合方向是自右至左(类似于赋值运算符 `a= b=c=4`)。

5.2.5.3. 实战

```

#include <stdio.h>

int main(void)
{
    int a, b;
    // 运算符的优先级低于关系和算术，但是高于赋值
    a>b?a:b+1;           /* 相当于 (a>b)?a:(b+1) */
    a*=b+1;                /* a = a*(b+1) */

    int c;
    a>b?a:c>d?c:d;
    // 是等价关系 a>b?a:(c>d?c:d); 探求 if else 嵌套关系?
    return 0;
}

```

```
}
```

5.2.6.逗号运算符与逗号表达式

5.2.6.1.列表

运算符	释意	结合性	示例
,	逗号表达式	从左到右	a+b,c+d

5.2.6.2.解析

逗号运算符，又叫**顺序求值运算符**。依次从左到右依次求取各个表达式的值，最后整个逗号表达式的值等于最后一个表达式的值。逗号表达式支持嵌套。逗号运算符的优先级最低。

逗号表达式常用于循环结构中。 for(, ; ;)

5.2.6.3.实战

```
#include <stdio.h>

int main(void)
{
    int a;
    a = 1,2,3,4,5,6;
    printf("a = %d\n",a);

    a = (1,2,3,4,5,6);
    printf("a = %d\n",a);

    printf("%d %d %d\n",1,2,3); //分隔符与逗号表达式
    printf("%d %d %d\n", (1,2,3),100,200);

    return 0;
}
```

5.2.7.sizeof 运算符及其表达式

5.2.7.1.列表

运算符	释意	结合性	示例
sizeof	类型大小运算符	从右向左	sizeof(int); sizeof(a)

5.2.7.2.解析

使用 `sizeof` 运算符可以获得数据类型占用内存空间的大小。其一般形式为：

`sizeof(type_name)` `sizeof(variable_name)`。 `sizeof` 运算符的结果以字节为单位显示。

5.2.7.3. 实战

```
#include <stdio.h>
int main(void)
{
    int a;
    printf("sizeof(int) = %d sizeof(a) = %d\n", sizeof(int), sizeof(a));
    return 0;
}
```

5.2.7.4. 我本是女娇娥 又不是男儿郎

`sizeof` 本身是一个关键字，但是常常被当作函数看待，主要是后面的跟了一个括号`()`，如果没有后面的括号`()`，又如何呢？

`sizeof (int)` 实际上，是由 `sizeof` 和 `()` 构成的一个复合表达式。

```
#include <stdio.h>

int main(void)
{
    printf("%d\n", sizeof 4);

    int a=1; double b = 100.0;

    int res = sizeof a+b; //sizeof 比+的优先级要高。
    printf("%d\n",res);

    return 0;
}
```

走关键字的路，让人说我是函数去吧。

5.2.8. 小结

经过上面的学习，我们可以得到，如下的结论：

- `()` 的优先级最高，算术>关系>逻辑>条件>赋值>逗号（并非绝对 `(!)`）。
- 单目运算符>双目运算符>三目运算符（并非绝对 `(, =)`）

运算符	类型	目数	优先级
<code>()</code>	括号	单目运算符	
<code>++ --</code>	自增减	单目运算符	
<code>sizeof</code>	求类型大小	单目运算符	

+ - * /	算术	双目运算符	
> < == !=	关系	双目运算符	
&& !	逻辑	双目运算符	
? :	条件	三目运算符	
= += *=	赋值	双目运算符	
,	逗号	双目运算符	

■ **绝招:**如果不清楚其中的优先级，最好加括号的方式加以强化。

5.3.运算符综合练习

5.3.1. `if(x==y)` 与 `if(x=y)` 之间的区别。

5.3.2. 表达式 `3==3==3` 的值是多少？

5.3.3. 表达式 `100<=a<=300`, 能表达`[100,300]`这样的区间吗？

5.3.4. 如何判断，我们输入的字符在`[a,z]`之间。

5.3.5. 求`[100,999]`内的水仙花数

所谓的水仙花数是指，各位上的数据的立方和等于该数本身，比如： $153 = 1^3 + 5^3 + 3^3$

5.3.6. 判断输入的年份，是否为润年。

闰年的判断方法是：1.可以被 4 整除,但不能被 100 整除 2.可以被 400 整除

5.3.7. 判断数字是否是回文数。

“回文”是指正读反读都能读通的句子，它是古今中外都有的一种修辞方式和文字游戏，如“我为人人，人人为我”等。

比如 `:int a = 12321;` 变量 `a` 就是一个回文数。请写程序判断变量 `a` 是否是回文数。

6. 程序流程设计(Flow of Control)

计算机语句执行过程，共分为，顺序，选择，循环三大结构。

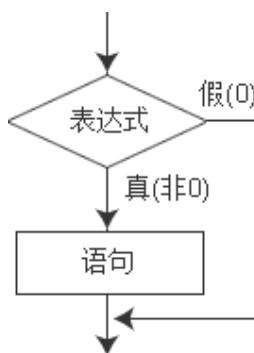
对计算机的进一步流程梳理，可大体总结为[二大选择](#)，[三大循环](#)，[四大跳转](#)，的更为细致的划分。

6.1.选择

选择亦可再次细分，分为单路选择，双路选择，多路选择。

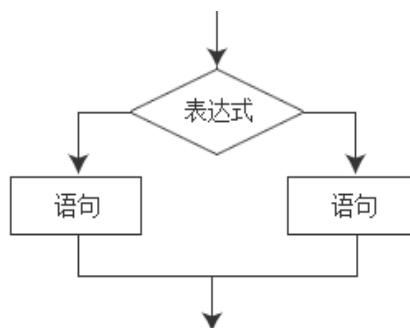
6.1.1.If else

6.1.1.1.if 结构及流程图



```
if(表达式)
{
    语句;
}
```

6.1.1.2.if else 结构及流程图



```
if(表达式)
{
    语句;
}else
```

```
{
    语句;
}
```

6.1.1.3.if else if 结构流程及演变

从 if 语句的嵌套开始讲起，嵌套类型总分两种。嵌套在 if 部分，或是嵌套在 else 部分。

```
if()
    if()
        { 语句; }
else
    { 语句; }
```

```
if()
{ 语句; }
else
    if()
        { 语句; }
```

比如下逻辑就可以利用上述逻辑来解决。有两个变量 a 和 b，如果两个变量想等，则输出相等信息，如果两个 $a > b$ 则输出，否则什么也不输出。

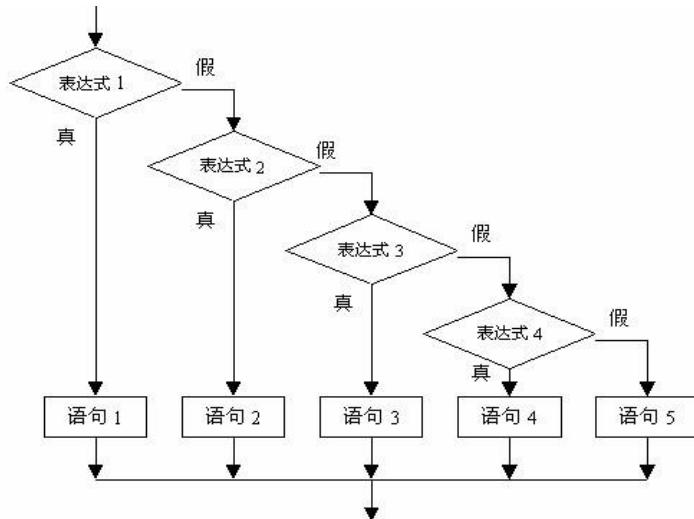
```
int main(void)
{
    int a = 3,b = 4;
    if(a!=b)//注定分枝逻辑在 if
        if(a>b)
            printf("a>b");
    else
        printf("a==b");
    return 0;
}
```

```
int main(void)
{
    int a = 3,b = 4;
    if(a==b)//注定分枝逻辑在 else
        printf("a==b");
    else
        if(a>b)
            printf("a>b");
    return 0;
}
```

我们会发现在嵌套在 if 里的，输出结果是错误的，而嵌入在 else 部分，却是正确的。

原因则于 else 是总是跟前面第一个没有配对的 if 相结合。当然了第二种方法，也是可以通过加大括号的方式补救，但是不如第一种方案，来的自然。

多路选择模型：



```
int main(void)
{
    if(条件 1)
    {
    }
    else if(条件 2)
    {
    }
    else if(条件 3)
    {
    }
    else if(条件 4)
    {
    }
    else if(条件 5)
    {
    }
    else
    {
    }

    return 0;
}
```

练习：输入学生成绩，判断成绩给出评价。

```
#include <stdio.h>
int main()
{
    int score;
    printf("pls input score:\n");
    scanf("%d",&score);
    if(score > 90)
        printf("you xiu\n");
    else if(score>60&&score<=70)
        printf("pass\n");
    else if(score>80&&score<=90)
        printf("liang hao\n");
    else if(score>70&&score<=80)
        printf("zhong deng\n");
    else
        printf("game over\n");
    return 0;
}
```

```
#include <stdio.h>
int main()
{
    int score;
    printf("pls input score:\n");
    scanf("%d",&score);
    if(score > 90)
        printf("you xiu\n");
    else if(score>80)
        printf("liang hao\n");
    else if(score>70)
        printf("zhong deng\n");
    else if(score>60)
        printf("pass\n");
    else
        printf("game over\n");
    return 0;
}
```

第一种情况，随意的打乱分支范围是没有任何问题的。而第二种情况，则需要依赖固有的层次关系。相比之下，我们更喜欢第一种情况，虽然要多写几条语句。

6.1.1.4.注意事项

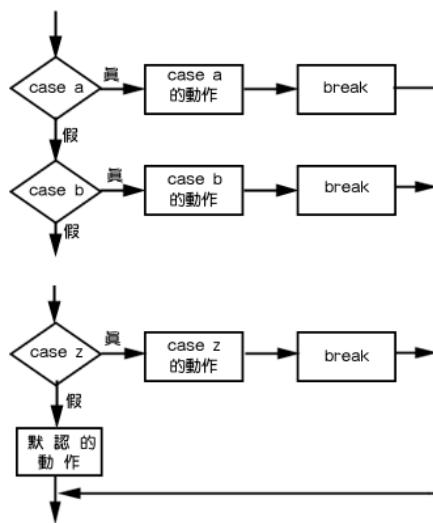
1. `if else` 连用的时候，`if` 和 `else` 之间，不能用其它语句，`if` 和 `else` 之后，只有一条语句属于它，如果有多条语句，就要用{}进行打包，打包的意义就在于，将多条语句并为一条。
2. 多路分支，要么写清分支层次关系，要么，明确各分支(不依赖层次)。
3. 2 中的问题，多存在于**多路范围**的选择，如果是**多路点**的选择，就不存在这些问题了。点的选择，多采用 `switch`。
4. `else` 亦不能单独使用。

6.1.2.switch

6.1.2.1.switch 的意义

由于 `if else if` 还是不够简洁，`switch` 就应运而生了，他跟 `if else if` 互为补充关系。`switch` 提供了点的多路选择。

6.1.2.2.switch 结构



```

switch (表达式)
{
    case 常量 1:
        代码块 1;
        break;
    case 常量 2 :
        代码块 2;
        break;
    default:
        代码块 3;
        break; // 此处的 break 可以省略
}
  
```

6.1.2.3.注意事项

1. switch (表达式), 中的表达式, 必须为整型和字符型。
2. case 只能为常量表达式。
3. case 分支是互斥的。
4. case 分支要同 break 连用, 跳出 switch,不然匹配一次后, 不再匹配。
5. 两 case 块之间, 可以用不大括号。

6.1.2.4.实战项目

```

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

    printf("请输入你的中奖号码:");
    scanf("%d",&choice);
  
```

```

switch(choice)    //可以优化
{
    case 1:
        printf("恭喜你获得一等奖，奖励 HHKB pro2\n");
        //break;
    case 2:
        printf("恭喜你获得，赫曼米勒椅子一把\n");
        //break;
    case 3:
        printf("恭喜你获得，Sony 显示器一台\n");
        //break;
    case 4:
        printf("恭喜你获得，纪念品一份\n");
        //break;
    case 5:
        printf("谢谢惠顾，谢谢惠顾\n");
        //break;
    default:
        printf("谢谢惠顾，谢谢惠顾\n");
}
return 0;
}

```

6.1.3. 小结

if else if 针对于范围的多路选择，而 switch 是针对点的多路选择，亦可以小范围选择。多路选择中，有优化的空间，比如中奖。

6.1.4. 练习判断按键

判断键盘上有哪些键被按下？哪一个，哪几个。

```

void MainWindow::keyPressEvent(QKeyEvent * event)
{
    //判断键是否按下
    //    if(event->key() == Qt::Key_A)
    //    {
    //        printf("a 键被按下了\n"); //window 默认不刷缓冲
    //        fflush(stdout);
    //    }
    //    else
    //    {
}

```

```

//      printf("有其它的按键按下了\n"); //window 默认不刷缓冲
//      fflush(stdout);
// }

switch (event->key()) {
    case Qt::Key_A:
        printf("目标向左移动\n");
        fflush(stdout);
        break;
    case Qt::Key_W:
        printf("目标向上移动\n");
        fflush(stdout);
        break;
    case Qt::Key_D:
        printf("目标向右移动\n");
        fflush(stdout);
        break;
    case Qt::Key_S:
        printf("目标向下移动\n");
        fflush(stdout);
        break;
    default:
        break;
}
}

```

6.2.循环

需要多次重复执行一个或多个任务的问题考虑使用循环来解决。



6.2.1.循环三要素

6.2.1.1.死循环

```

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

```

```

while(1){
    printf("xxxxxxxxxxxxxx\n");
    sleep(1);
}

```

6.2.1.2. 可控循环

```

#include <stdio.h>
int main(){
    int i=10;           //循环变量初始化
    while(i<0){         //循环终止条件
        printf("xxxxxxxxxxxxxx\n");
        sleep(1);
        i--;            //有使循环趋于结束的语句
    }
}

```

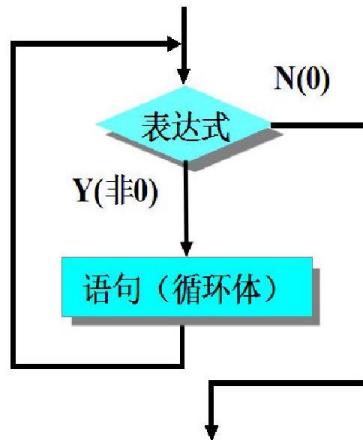
6.2.1.3. 可控循环的三要素

循环三要素，循环变量初始化，循环变量的变化（趋向终止条件），循环终止条件。

要作到可控三个条件缺一不可。

6.2.2. while “当”型循环

6.2.2.1. 语法结构及流程



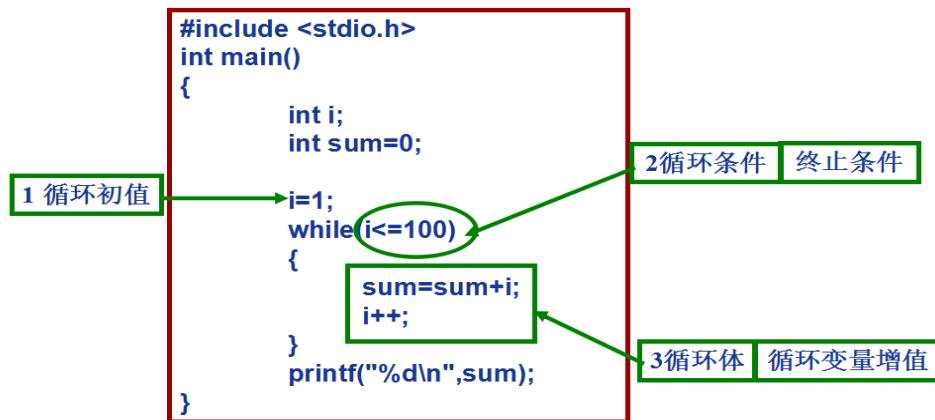
```

while (表达式)
{
    语句;
}

```

6.2.2.2.解析

计算表达式的值，当值为真(非 0)时，执行循环体语句，一旦条件为假，就停止执行循环体。如果条件在开始时就为假，那么不执行循环体语句直接退出循环。



6.2.2.3.实战

用 while 语句求 1~100 所有整数的和。

```

#include <stdio.h>

int main(void)
{
    int i; int sum = 0;

    i = 0;
    while(i <= 100)
    {
        sum += i;
        i++;
    }

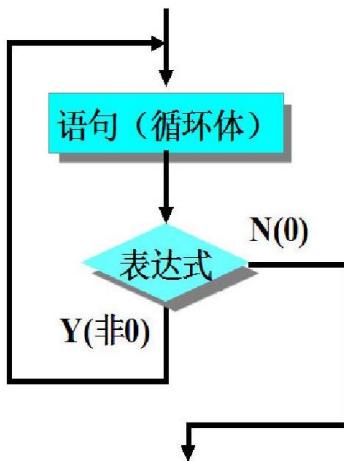
    printf("sum = %d\n",sum);

    return 0;
}

```

6.2.3.do while “直到”型循环

6.2.3.1.语法及流程



```

do
{
    语句;

} while (表达式);
```

6.2.3.2.解析

它先执行循环体中的语句，然后再判断条件是否为真，如果为真则继续循环；如果为假，则终止循环。

```

#include <stdio.h>
int main()
{
    int i,sum=0;
    i=1; //循环变量初始化
    do
    {
        sum+=i;
        i++; //循环体和循环变量增值
    }while(i<=100); //循环终止条件
    printf("%d\n",sum);
}
```

6.2.3.3.实战

用 do-while 语句求 1~100 所有整数的和。

```

#include <stdio.h>

int main(void)
{
```

```
int i; int sum = 0;

i = 0;
do
{
    sum += i;
    i++;
}while(i<=100);

printf("sum = %d\n",sum);

return 0;
}
```

6.2.3.4. while/do while 综合案例

while 与 dowhile 的区别

```
#include <stdio.h>

int main()
{
//    while(1)
//    {
//        printf("xxxxxxxxxx\n");
//        sleep(1);
//    }
//    do
//    {
//        printf("oooooooooo\n");
//        sleep(1);
//    }while(1);

    while(0)
    {
        printf("xxxxxxxxxx\n");
        sleep(1);
    }
    do
    {
        printf("oooooooooo\n");
        sleep(1);
    }while(0);

    return 0;
}
```

```
}
```

如下代码，表示求和,求输入 1 和 11 的比较。

```
#include <stdio.h>
int main()
{
    int sum = 0;
    int i;
    scanf("%d", &i);
    do
    {
        sum += i;
        i++;
    }while(i <= 10);
    printf("sum= %d\n", sum);
    return 0;
}
// /////////////////////////////////
#include <stdio.h>
int main()
{
    int sum = 0;
    int i;
    scanf("%d", &i);
    while(i <= 10)
    {
        sum += i;
        i++;
    }
    printf("sum= %d\n", sum);
    return 0;
}
```

结论是，当输入 1 时两个运算的求和，是相等的，但是当输入 11 是，**while** 版本输出的 **sum = 0**。而 **do while** 版本输出的则是 11。

所以，**do while** 至少执行一次，而 **while** 可能一次都不执行。

6.2.3.5.**do while** 的争议

do while 特点是，循环体至少执行一次，这个是比较可怕的。在没有收到任何限制条件的情况下，就开始执行循环体。很可能会带来不可挽回的后果。比如，先格式化了你的 c 盘，然后再来问要不要格式化。

do while 曾一度提议废除，但是他在输入性检查方面还是有点用的。

比如，输入 Y/y 设置标志位，退出，执行标志位操作，输入 N/n 直接退出，输入其它，循环输入。

```
#include <stdio.h>
int main(void)
{
    char select;
    do
    {
        printf("please input y/n\n");
        scanf("%c",&select); getchar();
        if(select == 'n'||select == 'N')
            exit(-1);
    }while(!(select=='y'||select =='Y'));
    printf("next\n");
    return 0;
}
```

```
#include <stdio.h>

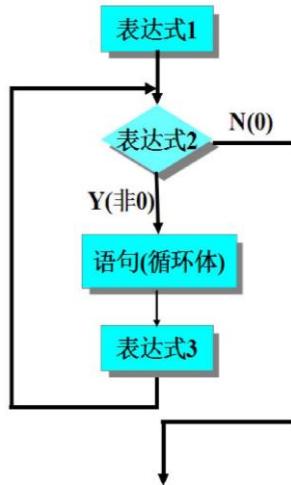
int main()
{
    int name;
    int passwd;

    do
    {
        printf("name:");
        scanf("%d",&name);
        printf("passwd:");
        scanf("%d",&passwd);
    }while!(name == 10 && passwd == 20));

    printf("wellcome\n");
    return 0;
}
```

6.2.4. for “列表”型循环

6.2.4.1. 语法结构及流程



```

for( 表达式 1 ; 表达式 2 ; 表达式 3 )
{
    语句;
}
  
```

6.2.4.2. 解析

- 1、计算表达式 1 的值，通常为循环变量赋初值。
- 2、计算表达式 2 的值，即判断循环条件是否为真，若值为真则执行循环体一次，否则跳出循环。
- 3、计算表达式 3 的值，这里通常写更新循环变量的赋值表达式，然后转回第 2 步重复执行。

6.2.4.3. 实战

用 do-while 语句求 1~100 所有整数的和。

```

#include <stdio.h>

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

求 n 的阶乘， $n! = n * (n-1)!$ ($n > 1$)；

6.2.4.4. 这还是 for 吗？for(;;)

6.2.4.5. for 与逗号表达式

打印 10 的所有加法组和，格式如下：

```
10 = 1 + 9;  
10 = 2 + 8;  
10 = 3 + 7;  
10 = 4 + 6;  
10 = 5 + 5;  
10 = 6 + 4;  
10 = 7 + 3;  
10 = 8 + 2;  
10 = 9 + 1;
```

程序：

```
#include <stdio.h>

int main(void)
{
    int i; int j;

    for(i=1, j=10-i; i<10 && j>0; i++, j--)
    {
        printf(" 10 = %d + %d\n", i, j);
    }

    return 0;
}
```

6.2.5. 循环的嵌套

一重循环解决线性的问题，而二重循环和三重循环就可以解决平面和立体的问题了。比如打印如下的图形：

```
*****
*****
*****
*****          *
*****          **
*****          ***
*****          ****
```

6.2.5.1. 嵌套的语法格式

<code>while ()</code>	<code>do</code>	<code>for (; ;)</code>
<pre>{ while () { }</pre>	<pre>{ do { }</pre>	<pre>{ for (; ;) { }</pre>

三种语法格式，是一种嵌套的结构，不可以交叉。三者都可以实现上述的功能，但是比较倾向于使用第三种，为什么呢，因为第三种，循环的初始化不在循环体外，循环变量的改变不在循环体内。整体呈现会更清晰简洁。

while 版

```
#include <stdio.h>

int main(void)
{
    int i = 0;
    while(i<4)
    {

        int j = 0;
        while(j<6)
        {
            printf("*");
            j++;
        }

        putchar(10);
        i++;
    }
    return 0;
}
```

do while 版

```
int main(void)
{
    int i=0;
    do{
        int j = 0;
        do
        {
            printf("*");
            j++;
        }while(j<6);

        i++;

        putchar(10);
    }while(i<4);
    return 0;
}
```

for 版

```
#include <stdio.h>

int main(void)
{
    for(int i=0; i<4; i++)
    {
        for(int j=0; j<6; j++)
        {
            printf("*");
        }
        putchar(10);
    }
    return 0;
}
```

6.2.6. 循环建议

6.2.6.1. 先外后内的循环嵌套写法

循环的嵌套，先写外重循环，外重循环的循环变量，可能作为内重循环的，循环终止条件。

6.2.6.2.半开半闭区间写法

建议 **for** 语句的循环控制变量的取值采用“半开半闭区间”写法。这样的作法，跟后面的数组下标，是完全吻合的。

但，这并非绝对。

循环变量属于半开半闭区间

```
for ( x=0; x<N; x++)
{
}
```

循环变量属于闭区间

```
for ( x=1; x<=N; x++)
{
}
```

6.2.6.3.嵌套的优化

在多重循环中，如果有可能，应当将最长的循环放在最内层，最短的循环放在最外层，以减少 CPU 跨切循环层的次数。见典型练习，百钱买百鸡的问题。

低效率：长循环在最外层

```
for (row=0; row<100; row++)
{
    for ( col=0; col<5; col++ )
    {
        sum = sum + a[row][col];
    }
}
```

高效率：长循环在最内层

```
for (col=0; col<5; col++ )
{
    for (row=0; row<100; row++)
    {
        sum = sum + a[row][col];
    }
}
```

6.2.7.循环小结

- 一种循环可以解决的问题，使用另外两种同样可行，只是方便程度不同。
- **while** 循环一般用于循环次数不定的情况，**for** 循环一般用于循环次数确定的情况（也可以用于循环次数不定的情况），**do-while** 循环一般用于至少需要执行一次的情况。
- **for** 循环和 **while** 循环是先判断条件是否为真，再执行循环体，因此，可出现循环一次也不执行的情况；**do-while** 循环是先执行循环体，再判断条件是否为真，因此，循环体至少执行一次。

6.2.8.练习

6.2.8.1.打印小九九

格式如下

1*1=1									
1*2=2	2*2=4								
1*3=3	2*3=6	3*3=9							
1*4=4	2*4=8	3*4=12	4*4=16						
1*5=5	2*5=10	3*5=15	4*5=20	5*5=25					
1*6=6	2*6=12	3*6=18	4*6=24	5*6=30	6*6=36				
1*7=7	2*7=14	3*7=21	4*7=28	5*7=35	6*7=42	7*7=49			
1*8=8	2*8=16	3*8=24	4*8=32	5*8=40	6*8=48	7*8=56	8*8=64		
1*9=9	2*9=18	3*9=27	4*9=36	5*9=45	6*9=54	7*9=63	8*9=72	9*9=81	

演化与成形

```
#include <stdio.h>

//int main(void)
//{
//    for(int i = 1; i <= 9; i++)
//    {
//        printf("x * x = xx\n");    打印九行
//    }
//    return 0;
//}

//int main(void)
//{
//    for(int i = 1; i <= 9; i++)
//    {
//        for(int j=1; j<=i; j++)
//            printf("x * x = xx "); 打印九列，且有成三角
//
//        putchar(10);
//
//    }
//    return 0;
//}

int main(void)
```

```
{
    for(int i = 1; i <= 9; i++)
    {
        for(int j=1; j<=i; j++)
            printf("%2d * %2d = %2d ",i,j,i*j); //再套用格式

        putchar(10);

    }
    return 0;
}
```

6.2.8.2.分别用 while 和 do while 实现将输入的整数翻转输出。

比如，输出 12345，输出 54321。

```
#include <stdio.h>
int main()
{
    int value, r_digit;
    value = 0;
    do
    {
        printf("\n 请输入一个数: ");
        scanf("%d", &value);
        if( value <= 0 )
            printf("该数必须为正数\n");
    }while( value <= 0 );
    printf("\n 反转后的数为: ");
    do
    {
        r_digit = value % 10;
        printf("%d", r_digit);
        value = value / 10;
    }while( value != 0 );
    printf("\n");
    return 0;
}
```

6.2.8.3.百钱买百鸡的问题

中国古代数学家张丘建在他的《算经》中提出了著名的“百钱买百鸡问题”：鸡翁一，值钱五，鸡母一，值钱三，鸡雏三，值钱一，百钱买百鸡，问翁、母、雏各几何？

提示：枚(穷)举法
 cock[0,20) / hen[0,33)/ chicken[0,100) chicken 个数必须是 3 的整数倍。

6.2.8.4.女朋友追到手了没有

一个男孩给女孩写了一行代码

```
for(;;) printf("I Love You!\n");
```

女孩回了一行代码

```
for(;;); printf("I Love You Too!\n");
```

6.3.跳转

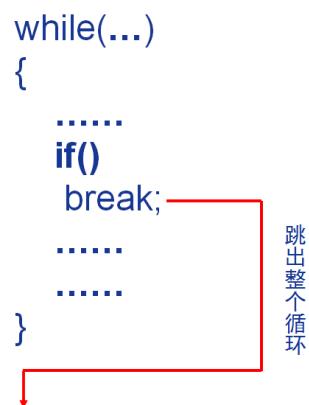
前两种跳转是发生在循环内部的，也就是说 break(switch), continue 只能出现在循环中。后两者则没有限制。

6.3.1.break

6.3.1.1.解析

他存在的意义，就是在提前结束**当前循环**，即跳出当前循环。有点像生活中的“阀”，水阀，电阀一样，到达某一个条件就会自动放水或停电，所以说，break 通常出现在条件表达式的后面，注意，**break 只能跳出一重循环**。

6.3.1.2.语法与流程



6.3.1.3.实战

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

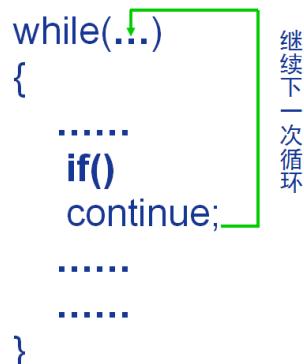
```
#include <unistd.h>
int main(void)
{
    int floodLevel;
    for(floodLevel = 10; floodLevel<100; floodLevel+=5)
    {
        printf("当前水位是%d\n",floodLevel);
        if(floodLevel == 95)
            break;
        sleep(1);
    }
    printf("开阀放水\n");
    return 0;
}sssss
```

6.3.2.continue

6.3.2.1.解析

它存在的意义，在于结束**当前**本轮循环，进入下一轮循环。有点像生活中的“筛子”只留下满足条件的，也像海边捡贝壳的小姑娘，只捡自己喜欢的。所以说，cotinue 通常出现在条件表达式的后面，**特点：只能结束当前循环提前进入下一轮循环。**

6.3.2.2.语法和流程



6.3.2.3.实战

打印[1,100] 以内所有能被 3 整除的数

```
int main()
{
    for(int i=0; i<=100; i++)
    {
```

```

    if(i%3 != 0)
        continue;

    printf("%d\n",i);
}

}

```

6.3.3.return

结束当前函数，返回调用。如果当前函数是 main 函数，则该进程结束，等价于 exit() 函数，因为操作系统发起的对 main() 的调用，main() 函数结束，意味着整个进程/程序的退出。

```

int main()
{
    for(int i=0; i<1; i++)
    {
        for(int j=0; j<1; j++)
        {
            for(int k=0; k<100; k++)
            {

                printf(" for(int k=0; k<100; k++)\n");

                if(k == 5)
                    goto Target; //return ;//break;

            }
            printf("for(int j=0; j<1; j++) \n");
        }

        Target:
        printf("for(int i=0; i<1; i++)\n");
    }
}

```

6.3.4.goto

这是一个不太值得探讨的话题，goto 会破坏结构化程序设计流程，它将使程序层次不清，且不易读，所以慎用。

最早提出，使用 goto 有害的是由荷兰著名计算机科学家 E.W.Dijkstra(艾兹格·迪杰斯特拉)于 1968 年提出的。

<http://ce.sharif.edu/courses/90-91/1/ce364-1/resources/root/GoTo/Dijkstra.pdf>

goto语句，仅能在本函数内实现跳转，不能实现跨函数跳转(短跳转)。但是他在跳出多重循环的时候效率还是蛮高的，再者就是集中错误处理。

6.3.5. 练习

6.3.5.1. 测试 break 能跳出几层结构

```
#include <stdio.h>

int main(void)
{
    while(1)
    {
        switch(1)
        {
            case 1: break;
        }
        printf(" switch(1)\n");
        break;
    }
    printf(" while(1)\n");
}

return 0;
}
```

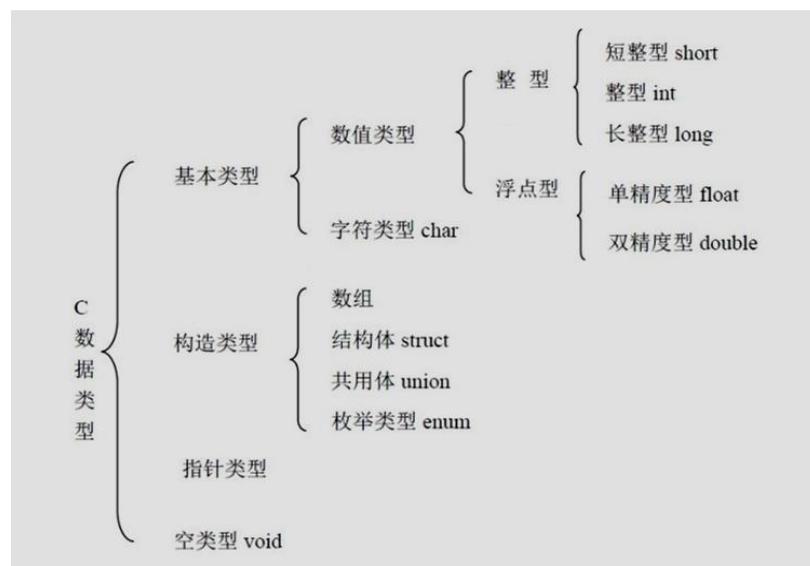
6.4.练习

6.4.1. 打印如下图形



7. 数组(Array)

再次打开我们的数据类型可以看的到，基本数据类型，我们已经搞定了。今天我们来学习第一种构造类型，数组。



为什么说数组是一种构造类型呢，因为他是由基本数据类型构造而成。当我们用一群变量来描述一类相同的东西时，比如有一百个年龄变量，我们可能定义成 `int age1; int age2; int age3; int age4 ;int age5;.....; int age100;` 这样既不方便于书写也不方便于管理，于是数组就出来了，`int age[100]`搞定。

构造类型，带来了书写和管理的方便，那么我们必须研究的初始问题就是**定义，大小，初始化和访问**。

对于任何的构造类型的研究都是从此开始的。

7.1. 一维数组

为了描述一堆由相同数据类型构成的新类型，C 语言提供了**构造类型 数组**，既是构造类型，他的初始化方式和访问方式也同基础数据类型不同。

7.1.1. 逻辑与声明

```
int [10] array; => int array[10];
```

```
#include <stdio.h>

int main(void)
{
    //数组大小
    printf("int[10] = %d \n", sizeof(int[10]));

    //int[10] array;
    int array[10];
    printf("array size = %d\n", sizeof(array));
    printf("array[0] = %d\n", sizeof(array[0]));

    return 0;
}
```

7.1.2. 初始化与访问

7.1.2.1. 初始化

初始化共分五种情况：

- 不初始化 => 成员初始值未知
- 全初始化
- 部分初始化 => 未初始化的部分，自动初始化为零
- 满初始化 => 越界并不潇洒
- 不指定大小初始化 => 经常出没

```
#include <stdio.h>

int main(void)
{
//    int array[10];    //未初始化

//    for(int i=0; i<10; i++)
//    {
//        printf("array[%d] = %d\n", i, array[i]);
//    }

//    int array[10] = {1,2,3,4,5,6,7,8,9,0}; //全初始化

//    for(int i=0; i<10; i++)
//    {
//        printf("array[%d] = %d\n", i, array[i]);
//    }
}
```

```
// int array[10] = {1,2,}; //半初始化，将数组初始化为 0

// for(int i=0; i<10; i++)
// {
//     printf("array[%d] = %d\n",i,array[i]);
// }

// int array[10] = {1,2,3,4,5,6,7,8,9,0,11,22}; //满初始化，c 语言不
// 进行越界检查。

// for(int i=0; i<10; i++)
// {
//     printf("array[%d] = %d\n",i,array[i]);
// }

int array[] = {1,2,3,4,5,6,7,8,9,0,11,22};
for(int i=0; i<sizeof(array)/sizeof(array[0]); i++)
{
    printf("array[%d] = %d\n",i,array[i]);
}

return 0;
}
```

凡是构造类型，要么在初始化的时候初始化， 要么对集合中的每个元素单独初始化。定义以后，不可以再以初始化的方式初始化。凡是基础数据类型，既可以在定义的时候初始化，也可以先定义后初始化。

```
#include <stdio.h>

int main(void)
{
    int array[10];
//    array[10] = {1,2,3,4,5,6,7,8,9,0};

    array[0] = 1;
    array[1] = 2;
    array[2] = 3;
    array[3] = 4;
    array[4] = 5;
    array[5] = 6;
    array[6] = 7;
```

```

for(int i=0; i<10; i++)
{
    printf("array[%d] = %d\n",i,array[i]);
}

return 0;
}

```

7.1.2.2.访问与一重循环

数组的逻辑是一维的，对于其成员的访问，即读写，往往通过一重循环来实现。

7.1.3.一维数组的存储

7.1.3.1.逻辑与存储

一维数组，在内存中是一段**连续**的存储区域。格局如下：

int	0x10024
int	0x10020
int	0x1001c
int	0x10018
int	0x10014
int	0x10010
int	0x1000c
int	0x10008
int	0x10004
int	0x10000

int arr[10]

7.1.3.2.实例验证

```

#include <stdio.h>

int main(void)
{
    int array[10];

    for(int i=0; i<10; i++) //所以 for 循环的书写，常写为左闭右开的形式
    {
        printf("&array[%d] = %p\n",i,&array[i]);
    }
}

```

```

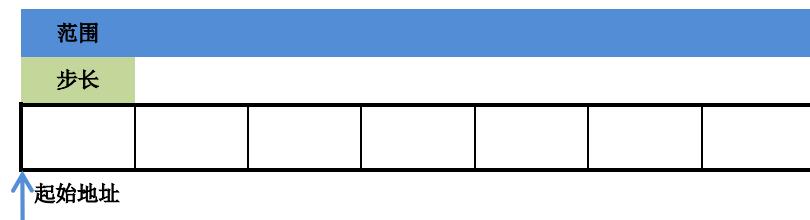
    return 0;
}

```

7.1.4. 数组三要素

7.1.4.1. 三要素图示

数组的声明中，已经把数组访问的三要素，均已表示出来了，三要素就是，**起始位置，移步刻度，终止位置（范围）**。



7.1.4.2. 成员访问

所有学习过，C 语言的人，可能都会有一个疑问，数组的下标为什么从零开始。因为 c 语言，脱胎于汇编语言，稍微懂点汇编的人，就知道 [] 是一种基址变址运算符。

基于起始位置偏移 0 个单位处，开始存放数据，故下标从 0 处开始。

7.1.5. 关于变长数组

前面学到的数组大小，要么是事先给定的，要么通过初始化成员的个数决定的。那么数组的大小，可不可以是变量呢，即大小是可以变化的呢？

C99 为了增加 c 语言的灵活性，提供了，变长数组的概念，(variable-length array)，也简称 VLA。即数组的大小，可以是变量。

```

#include <stdio.h>

int main(void)
{
    int size;
    scanf("%d",&size);

    int array[size]; // 变长数组不能被初始化 = {1,2,3,4,5};

    printf("sizeof(array) = %d\n",sizeof(array));

    size = 15; // 此时再改变 size 的值，数组的大小，不再发生改变

    printf("sizeof(array) = %d\n",sizeof(array));
}

```

```
    return 0;  
}
```

变长数组是指用整型变量或表达式声明或定义的数组，**而不是说数组的长度会随时变化，变长数组在其生存期内的长度同样是固定的。**

注意：变长数组有平台差异性，以 linux 平台为准。

7.1.6. 实战应用(一维数据形态)

前面我们学习了数组的声明/大小，初始化/赋值，访问(一重循环)，及其内存存储，并提出了数组的三要素。

下面，我们要学习的是，数组作为一种数据的集合，它有哪些形态呢？

7.1.6.1.求一个数组的数据和与平均值

7.1.6.2.求一个数组中最大与最小值

```
#include <stdio.h>  
  
int main(void)  
{  
    int array[10] = {100,1,40,29,45,22,98,2,83,75};  
  
    int max = array[0]; // 一种假设  
    int min = array[0];  
    for(int i=1; i<10; i++)  
    {  
        if(max < array[i])  
            max = array[i];  
  
        if(min > array[i])  
            min = array[i];  
    }  
  
    printf("int array max = %d min = %d\n",max,min);  
  
    return 0;  
}
```

■经验：假设其中元素为最大与最小，求其中最大与最小是一种常见策略。

7.1.6.3.对数组元素进行排序

对于一堆集合数据，进行排序也是一种常用的操作。排序是查找的前提，故排序很重要。

下面要讲的一种选择法排序。

```
#include <stdio.h>

int main(void)
{
    int array[10] = {100,1,40,29,45,22,98,2,83,75};

    for(int i=0; i<10-1; i++)
    {
        for(int j = i+1; j<10; j++)
        {
            int tmp;
            if(array[i]>array[j])
            {
                tmp = array[i];
                array[i] = array[j];
                array[j] = tmp;
            }
        }
    }

    for(int i = 0; i<10; i++)
    {
        printf("%d\t",array[i]);
    }

    return 0;
}
```

优化算法【选学章节】：比而不换，只记下标；

分析 1: 1 2 3 4 5

分析 2: 5 2 4 3 1

```
#include <stdio.h>

int main(void)
{
    int array[10] = {100,1,40,29,45,22,98,2,83,75};
    int tmp;
    for(int i=0; i<10-1; i++)
```

```

{
    int idx = i;// 假设当前元素是最小的
    for(int j = i+1; j<10; j++)
    {
        if(array[idx]>array[j])
        {
            idx = j; //有比当前元素小的，更新最小下标，直到循环结束
        }
    }
    if(i != idx) // idx == i 说明假设成立，当前最小 进入下一轮
    {
        tmp = array[i];
        array[i] = array[idx];
        array[idx] = tmp;
    }
}

for(int i = 0; i<10; i++)
{
    printf("%d\t",array[i]);
}

return 0;
}

```

7.1.6.4.线性查找

查找到第一个匹配的元素即刻返回。

7.1.6.5.折半查找

又称为二分查找。这种查找方法要求查找表的数据是线性结构保存，并且还要求查找表中的数据是按关键字由小到大有序排列。

```

#include <stdio.h>

int main(void)
{
    int array[10] = {100,1,40,29,45,22,98,2,83,75};

    for(int i=0; i<10-1; i++)
    {
        for(int j = i+1; j<10; j++)
        {
            int tmp;
            if(array[i]>array[j])

```

```
{  
    tmp = array[i];  
    array[i] = array[j];  
    array[j] = tmp;  
}  
}  
}  
  
for(int i=0; i<10; i++)  
{  
    printf("%d\t",array[i]);  
}  
putchar(10);  
  
  
  
  
int find = 83;  
int idx = -1;  
  
int low =0;  
int high = sizeof(array)/sizeof(array[0])-1;  
  
while(low <= high) // 等号也是一种判断条件  
{  
    int mid = (low+high)/2;  
  
    if(find == array[mid])  
    {  
        idx = mid;  
        break;  
    }  
  
    if (find > array[mid])  
    {  
        low = mid+1;  
    }  
    else  
    {  
        high = mid-1;  
    }  
}  
  
if(idx == -1)  
    printf("find none in array\n");
```

```

    else
        printf("the find idx in array = %d\n ",idx);

    return 0;
}

```

7.1.7.作业

7.1.7.1.求一个数组中第二大的数（用一重循环）

7.2.二维数组

二维数组的本质，也是一维数组，只不过，一维数组中的每个元素，又是一个一维数组而已。

7.2.1.声明/定义

int[4] array[3] => int array[3][4];

```

#include <stdio.h>
int main(void)
{
    int arr[3][4]; // int[4] int array[3];

    //成员的大小
    printf("sizeof(arr[0]) = %d sizeof(int[4])= %d\n",
           sizeof(arr[0]),sizeof(int[4]));

    //数组的大小

    printf("sizeof(arr) = %d\n",sizeof(arr));
    return 0;
}

```

7.2.2.初始化

初始化共分五种情况

- 不初始化 => 成员初始值未知
- 全初始化
- 部分初始化(行，全局) => 未初始化成员，自动清零

- 满初始化 =>这样太过分了
- 不指定行大小初始化 =>新常态

```
#include <stdio.h>

int main(void)
{
//    int a[3][4]; // 未初始化
//    for(int i=0; i<3; i++)
//    {
//        for(int j=0; j<4; j++)
//        {
//            printf("a[%d][%d] = %4d\t",i,j,a[i][j]);
//        }
//        putchar(10);
//    }

//    int a[3][4] = {{1,2,3,4},{1,2,3,4},{1,2,3,4}}; //全初始化
//    //此处大括号中不加大括号也是正确的，但有警告
//    for(int i=0; i<3; i++)
//    {
//        for(int j=0; j<4; j++)
//        {
//            printf("a[%d][%d] = %4d\t",i,j,a[i][j]);
//        }
//        putchar(10);
//    }

//    int a[3][4] = {{1},{1,2},{1,2,3}}; //行部分初始化
//    //未初始化的部分清零
//    for(int i=0; i<3; i++)
//    {
//        for(int j=0; j<4; j++)
//        {
//            printf("a[%d][%d] = %4d\t",i,j,a[i][j]);
//        }
//        putchar(10);
//    }

//    int a[3][4] = {11,22,33,44,55}; //全局部分初始化
//    //未初始化的部分清零
//    for(int i=0; i<3; i++)
//    {
//        for(int j=0; j<4; j++)
```

```

//      {
//          printf("a[%d][%d] = %4d\t", i, j, a[i][j]);
//      }
//      putchar(10);
//  }

// int a[3][4] = {{1,2,3,4},{1,2,3,4},{1,2,3,4},{1,2,3,4}}; //满初始化
// //c 语言不进行越界型检查
// for(int i=0; i<3; i++)
// {
//     for(int j=0; j<4; j++)
//     {
//         printf("a[%d][%d] = %4d\t", i, j, a[i][j]);
//     }
//     putchar(10);
// }

int a[][] = {{1,2,3,4},{1,2,3,4},{1,2,3,4},{1,2,3,4}}; //省行初始化
//行可以省，列不可以省，列里面有类型信息(一维数组的类型)
for(int i=0; i<sizeof(a)/sizeof(a[0]); i++)
{
    for(int j=0; j<4; j++)
    {
        printf("a[%d][%d] = %4d\t", i, j, a[i][j]);
    }
    putchar(10);
}
return 0;

```

7.2.3.二维数组的逻辑/存储

7.2.3.1.逻辑

二维是一种逻辑概念，也就是说逻辑上来说二维的，是行列矩阵，但它在内存中的存储却是一维的，按行顺序存储。以 int a[3][4]为例：

逻辑形式：

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

7.2.3.2.存储

存储形式：

a[2][3]
a[2][2]
a[2][1]
a[2][0]
a[1][3]
a[1][2]
a[1][1]
a[1][0]
a[0][3]
a[0][2]
a[0][1]
a[0][0]

验证：

```
#include <stdio.h>

int main(void)
{
    int a[3][4] = {0};

    for(int i=0; i<3; i++)
    {
        for(int j=0; j<4; j++)
        {
            printf("%2d ",a[i][j]);
        }
        putchar(10);
    }

    for(int i=0; i<3; i++)
    {
        for(int j=0; j<4; j++)
        {
            printf("%p    ",&a[i][j]);
        }
        putchar(10);
    }
}
```

```
    return 0;
}
```

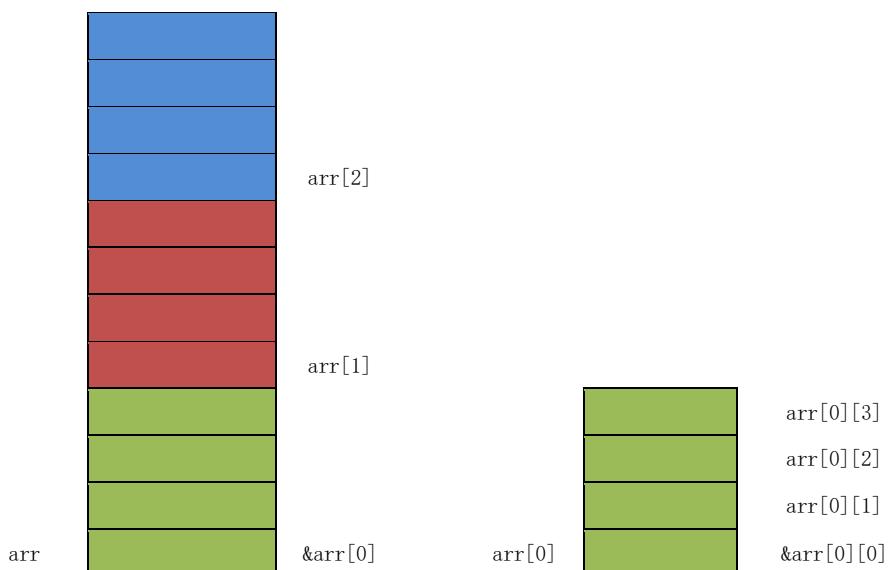
7.2.3.3.数组三要素

范围

步长



起始 地址



```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int arr[3][4] = {1,2,3,4,
                     5,6,7,8,
                     11,22,33,44};

    // int arr[5]; 一维数组的三要素，起始地址 arr，步长 int，范围 5
    // int[4] arr[3]; 二维数组的三要素， arr, int[4] 3;

    // int arr1d[5] = {1,2,3,4,5};
    // printf("arr1d = %p &arr1d[0] = %p\n",arr1d,&arr1d[0]);
    // printf("arr1d+1 = %p &arr1d[0]+1 = %p\n",arr1d+1,&arr1d[0]+1);
    // printf("arr1d+1 = %d &arr1d[0]+1 = %d\n",
                           *(arr1d+1),*(&arr1d[0]+1));
}
```

```
    printf("arr = %p &arr[0] = %p &arr[0][0] = %p\n",
           arr, &arr[0],&arr[0][0]);
    printf("arr+1 = %p &arr[0]+1 = %p &arr[0][0]+1 = %p\n",
           arr+1, &arr[0]+1,&arr[0][0]+1);

    return 0;
}
```

7.2.3.4.数组中数组成员的访问

```
#include <stdio.h>
int main(void)
{
    int a[3][4] = {{1,2,3,4},{1,2,3,4},{1,2,3,4}};

    printf("%d\t%d\t%d\n",a[0][0],a[0][1],a[0][2]);
    printf("%d\t%d\t%d\n",a[1][0],a[1][1],a[1][2]);
    printf("%d\t%d\t%d\n",a[2][0],a[2][1],a[2][2]);

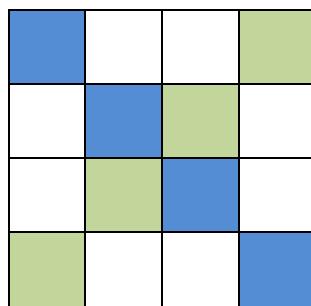
    printf("*****\n");

    for(int i=0; i<3; i++)
    {
        for(int j=0; j<4; j++)
        {
            printf("%d\t",a[i][j]);
        }
        putchar(10);
    }
    return 0;
}
```

7.2.4. 实战应用(二维数据形态)

7.2.4.1. 主对角线与次对角线输出

输入一个 4x4 的二维数组，并输出该数组的主对角线和次对角线上的元素。



```
#include <stdio.h>

int main()
{
    int array[4][4];
    //输入
    for(int i=0; i<4; i++)
    {
        for(int j=0; j<4; j++)
        {
            scanf("%d",&array[i][j]);
        }
    }
    //输出
    for(int i=0; i<4; i++)
    {
        for(int j=0; j<4; j++)
        {
            printf("%d\t",array[i][j]);
        }
        putchar(10);
    }
    // 对角线 二重循环，也可以一重循环
    for(int i=0; i<4; i++)
    {
        for(int j=0; j<4; j++)
        {
            if(i == j)

```

```

        printf("%d\n",array[i][j]);
    }
    putchar(10);
    int t = i+1;
    while(t--)
        putchar('\t');
}

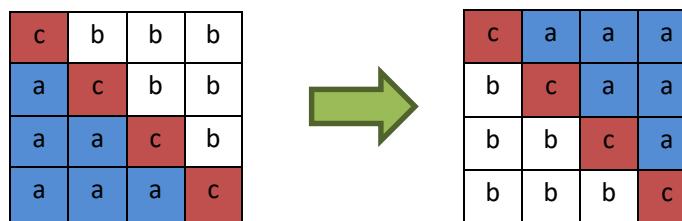
//对角线/ 一重循环
for(int j = 3; j>=0; j--)
{
    int t = j;
    while(t--)
        putchar('\t');
    printf("%d\n",array[3-j][j]);
}

return 0;
}

```

7.2.4.2.逆置一个二维字符数组

将一个矩阵 (4×4) 进行转置处理，要求初始化原始矩阵、输出原矩阵和转置后的矩阵



```

#include <stdio.h>

int main(void)
{
    int a[4][4] = {1,1,1,1,
                   2,1,1,1,
                   2,2,1,1,
                   2,2,2,1};

    for(int i=0; i<4; i++)

```

```
{  
    int tmp;  
    for(int j=0; j<4; j++)  
    {  
        if(i<j)  
        {  
            tmp = a[i][j];  
            a[i][j] = a[j][i];  
            a[j][i] = tmp;  
        }  
    }  
  
    for(int i=0; i<4; i++)  
    {  
        for(int j=0; j<4; j++)  
        {  
            printf("%d\t",a[i][j]);  
        }  
        putchar(10);  
    }  
  
    return 0;  
}
```

7.2.4.3.天生棋局

生成一个 10*10 的棋局，要求，初始化为零。随机置入 10 颗棋子，棋子处置为 1，并打印。

```
#include <stdio.h>  
#include <stdlib.h>  
#include <time.h>  
  
int main(int argc, char *argv[])  
{  
    int chess[10][10] = {0}; // 初始化为零  
  
    int counter = 0;  
    int x, y;  
    srand(time(NULL));  
  
    //    while(1)  
    //    {
```

```
//      x = rand()%10;
//      y = rand()%10;
//      if(chess[x][y] == 0)
//      {
//          chess[x][y] = 1;
//          counter++;
//      }
//      if(counter == 10)
//          break;
//  }

while(counter<20)
{
    x = rand()%10;
    y = rand()%10;
    if(chess[x][y] == 1)
        continue;

    chess[x][y] = 1;
    counter++;
}

for(int i=0; i<10; i++)
{
    for(int j=0; j<10; j++)
    {
        printf("%2d",chess[i][j]);
    }
    putchar(10);
}

int good = 0;
int flag = -1;
for(int i=0; i<10; i++) //同时扫描行列
{
    for(int j=0; j<10; j++)
    {
        if(chess[i][j] == 1)
        {
            good++;
            if(good == 3)      //连续三颗棋子。
            {
                flag = 0;
                break;
            }
        }
    }
}
```

```
        }
    }
else
    good = 0;
}
if(flag == 0)
    break;

good = 0; //行扫描完毕，重置标志位。

for(int j=0; j<10; j++)
{
    if(chess[j][i] == 1)
    {
        good++;
        if(good == 3)
        {
            flag = 0;
            break;
        }
    }
    else
        good = 0;
}
if(flag == 0)
    break;
}

if(flag == 0)
    printf("good Chess\n");
else
    printf("not good\n");

return 0;
}
```

若在棋中出现连续三个棋子在一起，包含横行和竖行，则输出好棋，否则输入臭棋。

7.2.4.4.有序数组归并

合并两个已经有序的数组 A[M], B[N]，到另外一个数组 C[M+N]中去，使用另外一个数组依然有序。其中 M 和 N 均是宏常量。

7.3.数组名的二义性

数组名，是数组的唯一标识符，既表示一种构造数据类型的大小，也表示访问组中的成员的首地址用来访问数据成员使用。

结构体的类型与成员访问，是分开的，为此增加了成员运算符。而数组名，却是一身而兼二任的，这也是简洁的需要。

7.3.1.一维数组名

一维数组名，从总体来看，他是一种构造类型，同时他又承担了，访问每个数组元素的责任。所以数组名，就有两重性。

```
#include <stdio.h>

int main(void)
{
    int array[10];

    printf("sizeof(int[10]) = %d  sizeof(array) = %d\n", //整体
           sizeof(int[10]),sizeof(array));

    // array  int   10

    printf("sizeof(int) = %d      sizeof(array[0])= %d\n", //局部
           sizeof(int),sizeof(array[0]));

    printf("array    = %p    &array[0] = %p\n",           //数组名与首元素地址
           array, &array[0]);

    printf("array+1  = %p    &array[0] +1 = %p\n", //数组名的移动刻度
           array+1, &array[0]+1);

    return 0;
}
```

7.3.2.二维数组名

数组名，是数组的唯一标识符。二维数组也是如此，二维数组本质是一种嵌套关系，其本质是一种一维数组，每个一维数组成员又是一个一维数组。

```

int main()
{
    int array[3][4];

    printf("sizeof(array) = %d sizeof(int[3][4]) = %d\n", //整体
           sizeof(array), sizeof(int[3][4]));

    // int[4] array[3]; type array[3]

    printf("sizeof(int[4]) = %d sizeof(array[0]) = %d\n", //局部
           sizeof(int[4]), sizeof(array[0]));

    //数组名与首元素地址
    printf(" array      = %p &array[0]      = %p\n", array, &array[0]);
    //数组名的刻度
    printf(" array+1   = %p &array[0]+1   = %p\n", array+1, &array[0]+1);

    //二维数组名中的一维成员
    printf("array[0]      = %p &array[0][0]      = %p\n", array[0], &array[0][0]);
    printf("array[0] +1   = %p &array[0][0] +1 = %p\n", array[0]+1, &array[0][0]+1);
}

```

7.4.练习

7.4.1.写出冒泡排序的逻辑，并实现冒泡排序。

7.4.2.无序数组查找，返回下标（线性查找）

比如 array[10] = {100,99,2,200,35,28,22,64,56,1}; 找到返回下标，找不到返回-1。

7.4.3.有序数组查找（折半查找）

比如 array[10] = {1,2,22,28,35,56,64,99,100,200}; 找到返回下标，找不到返回-1。

7.4.4.求出矩阵两条对角线上的元素之和

int a[3][3]={1, 3, 6, 7, 9, 11, 14, 15, 17};

7.4.5.有序数组去重，并返回去重后数组元素新个数。

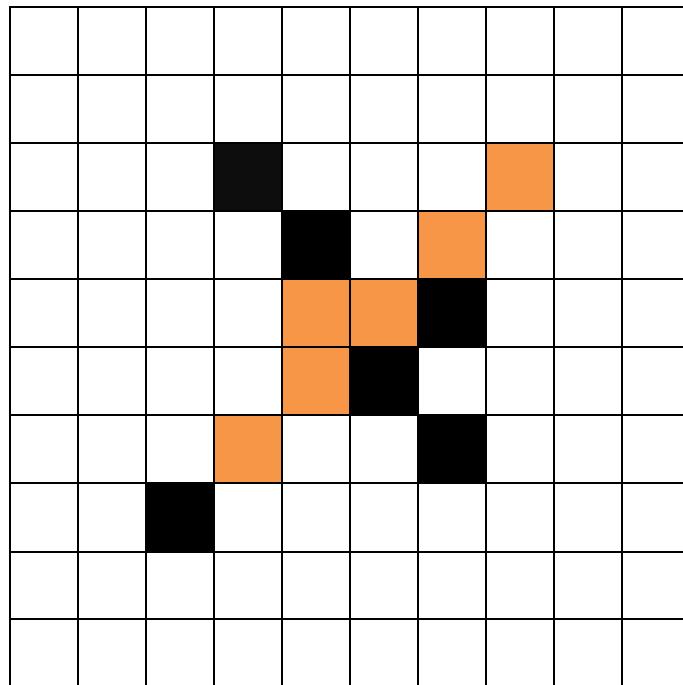
比如：int A[3]={1,1,2}，返回 length = 2, and A is now {1,2}。

7.4.6.二维数组按列移位

每个元素向右移一列，最右一列换到最左一列，如下图所示：

$$\begin{vmatrix} 6 & 4 & 5 \\ 3 & 1 & 2 \end{vmatrix} \Rightarrow \begin{vmatrix} 5 & 6 & 4 \\ 2 & 3 & 1 \end{vmatrix}$$

7.4.7.五子棋判输赢



8. 指针(Pointer)

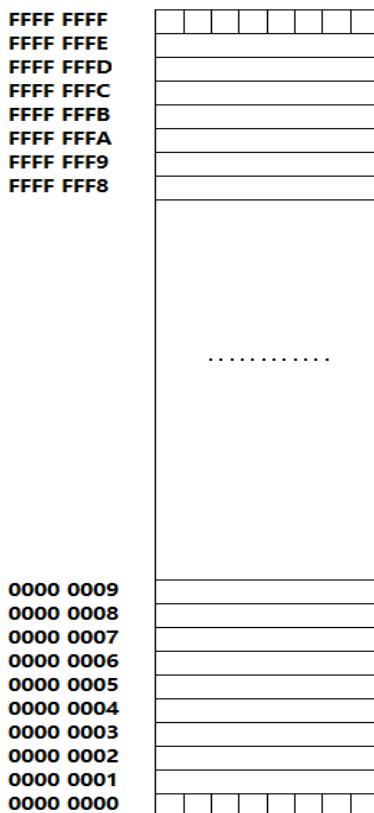
指针，**无疑是 C 语言的精华，没有掌握指针，也就没有掌握 C 语言。**指针让 C 语言更像是结构化的底级语言，所有在内存中的数据结构均可用指针来访问，指针让 C 语言带来了更简便的操作，更优效率，更快的速度。他是天使，也是魔鬼。

Just do it;

8.1. 认识内存

8.1.1. 线性的内存

前面我们学习的一维数组也好，二维数组也好都仅是一种**逻辑上的体现**，最终数据是要保存到内存中的，而**内存是线性的**，内存的线性是**物理基础**。



内存是以**字节为单位进行编址**的，内存中的每个字节都对应一个地址，通过地址才能找到每个字节。

8.1.2. 变量的地址与大小

8.1.2.1. 变量的地址

变量对应内存中的一段存储空间，该段存储空间占用一定的字节数，可能是 1

个字节，也可能是 4 或是 8 个字节，用这段存储空间的第一个字节的地址表示变量的地址，即低位字节的地址。

变量的地址，可以通过 **Reference (&)** 引用运算符取得，在此可以称为取地址运算符。

```
#include <stdio.h>

int main(void)
{
    int a; int b;
    printf("&a = %p\n", &a);
    printf("&b = %p\n", &b);
    return 0;

}

//  &a = 0060FEAC
//  &b = 0060FEA8 // A8+4 = AC
```

8.1.2.2. 地址的大小

```
#include <stdio.h>
int main(void)
{
    char a ; short b; int c; long d;
    float e; double f;
    // 是一种 32 位 16 进制的整数
    printf("&a = %p\n", &a);
    printf("&b = %p\n", &b);
    printf("&c = %p\n", &c);
    printf("&d = %p\n", &d);
    printf("&e = %p\n", &e);
    printf("&f = %p\n", &f);

    //类型不同，大小相同，均为 4 字节
    printf("sizeof(&a) = %d\n", sizeof(&a));
    printf("sizeof(&b) = %d\n", sizeof(&b));
    printf("sizeof(&c) = %d\n", sizeof(&c));
    printf("sizeof(&d) = %d\n", sizeof(&d));
    printf("sizeof(&e) = %d\n", sizeof(&e));
    printf("sizeof(&f) = %d\n", sizeof(&f));
    return 0;
}
```

通过运算的方式，我们可以求得变量的地址。32 位机的情况下，无论是什么类型大小均是 4。而 64 位机大小均是 8。这是由当前机型的地址总线决定的。

8.1.3. 间接访问内存

上一节中，我们拿到的变量的地址，其实，就是指针了。除了变量，我们还可以通过指针的方式间接的访问内存。

dereference (*) 解引用运算符，在此处我们可以称为，**取内容运算符**。用法如下：

```
int main(void)
{
    char a = 1 ; short b=2; int c=3; long d = 4;
    float e= 1.2; double f = 2.3;
    printf("&a = %p\n", &a);
    printf("&b = %p\n", &b);
    printf("&c = %p\n", &c);
    printf("&d = %p\n", &d);
    printf("&e = %p\n", &e);
    printf("&f = %p\n", &f);
    printf("&g = %p\n", &g);

    printf("a = %d\n", *(a));
    printf("b = %d\n", *(b));
    printf("c = %d\n", *(c));
    printf("d = %ld\n", *(d));
    printf("e = %f\n", *(e));
    printf("f = %f\n", *(f));

    return 0;
}
```

8.2. 指针常量

8.2.1. 指针是有类型地址常量

再次对其研究探讨：

```
int main(void)
{
    char a = 1 ; short b=2; int c=3; long d = 4;

    float e= 1.2; double f = 2.3;
```

```

printf("&a = %p\n", &a);
printf("&b = %p\n", &b);
printf("&c = %p\n", &c);
printf("&d = %p\n", &d);
printf("&e = %p\n", &e);
printf("&f = %p\n", &f);

printf("a = %d\n", *((char*)0x0060FEAF));
printf("b = %d\n", *((short*)0x0060FEAC));
printf("c = %d\n", *((int*)0x0060FEA8));
printf("d = %ld\n", *((long*)0x0060FEA4));
printf("e = %f\n", *((float*)0x0060FEA0));
printf("f = %f\n", *((double*)0x0060FE98));

return 0;
}

```

也就是说，这两个是等价的：

```

printf("a = %d\n", *(&a));
printf("b = %d\n", *(&b));
printf("c = %d\n", *(&c));
printf("d = %d\n", *(&d));
printf("e = %f\n", *(&e));
printf("f = %f\n", *(&f));

```

```

printf("a = %d\n", *((char*)0x0060FEAF));
printf("b = %d\n", *((short*)0x0060FEAC));
printf("c = %d\n", *((int*)0x0060FEA8));
printf("d = %ld\n", *((long*)0x0060FEA4));
printf("e = %f\n", *((float*)0x0060FEA0));
printf("f = %f\n", *((double*)0x0060FE98));

```

也就是说，`&a` 进行取地址，取出来的地址是有类型的，这个信息对我们来说很重要的。

通过上面我们的分析，指针其实就是一个有类型的 4 字节整型常量。那么这个类型信息对我们来说有什么意义呢。

```

#include <stdio.h>

int main(void)
{
    int data = 0x12345678;

    printf("%d\n", *(&data));
    printf("%x\n", *(&data));
    printf("%x\n", *((char*)&data));
    printf("%x\n", *((short*)&data));
    printf("%x\n", *((int*)&data));
}

```

```
    return 0;  
}
```

结论：

指针的类型，决定了，该指针的寻址能力。即从指针所代表的地址处的寻址范围。

8.3.指针变量

严格地说，一个指针是一个**有类型地址**，是一个**有类型的常量**。用以存放指针的量，我们叫作，指针变量。一个指针变量却可以被赋予不同的指针值，可以能过指针变量改变指向和间接操作。

严格意义上来说，指针即是指的指针常量，而我们通常意义上所说的指针，多指的指针变量。

8.3.1. 定义

指针变量的定义：

```
type * variable;
```

8.3.2. 解析

- * 表示该变量是一个指针变量。
- type 表示该变量的内存放的地址的寻址能力。

8.3.3. 指针变量大小

```
#include <stdio.h>  
  
int main(void)  
{  
    char    *a;  
    short   *b;  
    int     *c;  
    float   *d;  
    double  *e;  
    printf("sizeof(a) = %d\n", sizeof(a));  
    printf("sizeof(b) = %d\n", sizeof(b));  
    printf("sizeof(c) = %d\n", sizeof(c));  
    printf("sizeof(d) = %d\n", sizeof(d));  
    printf("sizeof(e) = %d\n", sizeof(e));  
  
    return 0;
```

{

8.3.4. 初始化及间接访问

凡是一个有类型的地址，都是可赋给同类型的指针变量。类型不同，编译器则可能会 Complain。

```
int main(void)
{
    char *p = (char*)0x123456; //0x123456 是数值, (char*)0x123456 是指针
    *p = 'a';
    return 0;
}
```

如果直接赋给指针变量一个地址，对其访问是**很危险的**。因为，我们不知道此址处是否一块什么区域，有可能是一段内核区域，访问很可能**会导致系统崩溃**。

所以通常作法是，把一个已经开辟空间的变量的地址赋给指针变量。

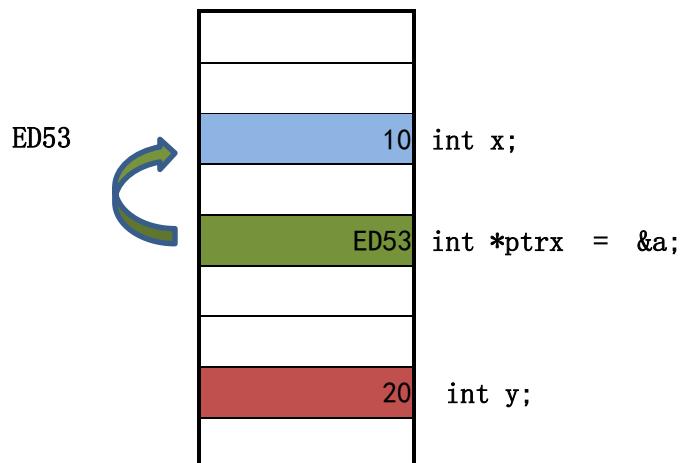
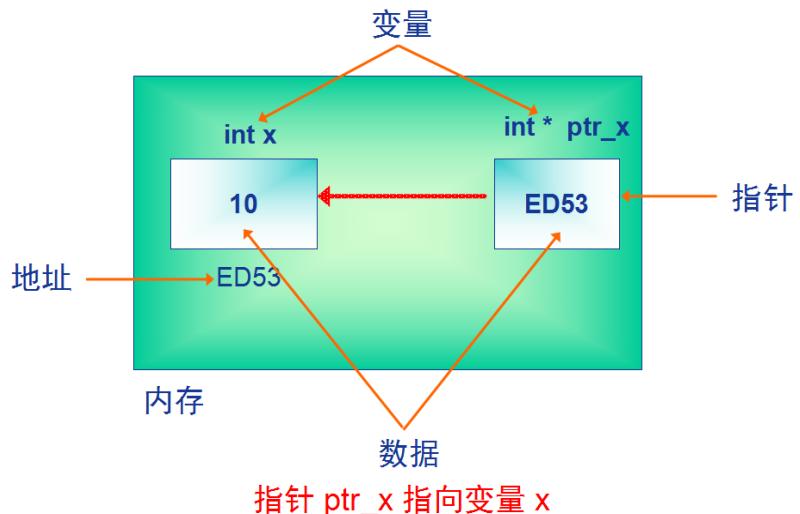
```
#include <stdio.h>

int main(void)
{
    int a = 100;
    int *pa = &a;
    printf("*pa = %d\n", *pa);

    *pa = 200;
    printf("*pa = %d\n", *pa);
    return 0;
}
```

8.3.5. 指向/被指向/更改指向

我们通常进行口述形表达时，**说谁指向了谁**，就是一种描述指针的指向关系。指向谁，即保存了谁的地址。



```
#include <stdio.h>

int main(void)
{
    int x = 10;
    int *p = &x;

    printf("x = %d *p = %d\n", x, *p);

    int y = 20;
    p = &y;
    printf("y = %d *p = %d\n", y, *p);

    return 0;
}
```

8.3.6.NULL (色即空)

8.3.6.1.野指针

一个指针变量，如果，指向一段无效的空间，则该指针称为野指针，是由 `invalid pointer` 翻译过来，直译是**无效指针**。常见情形有两种，一种是未初始化的指针，一种是指向一种已经被释放的空间。

对野指针的读或写崩溃尚可忍受，对野指针的写入成功，造成的后果是不可估量的。对野指针的读写操作，是危险而且是没有意义的。世上十之八九最难调的 bug 皆跟它有关系。

8.3.6.2.NULL 指针(零值无类型指针)

如何避免野指针呢，色即空。

`NULL` 是一个宏，俗称空指针，他等价于指针(`void* 0`)。(`void* 0`) 0 是一个很特别的指针，因为他是一个计算机黑洞，既读不出东西，也不写进东西去。所以被赋值 `NULL` 的指针变量，进行读写操作，是不会有关数据损坏的。

c 标准中是这样定义的：

```
define NULL ((void *)0)
```

故常用 `NULL` 来给临时不需要初初始化的指针变量来进行初始化。或对已经被释放指向内存空间的指针赋值。

可以理解为 C 专门拿出了 `NULL`(零值无类型指针)，用于作标志位使用。

8.3.6.3 void 本质

`void` 即无类型，可以赋给任意类型的指针，本质即代表内存的最小单位，在 32 位机上地位等同于 `char`。

8.3.7.课堂实战

8.3.7.1.打印变量和变量的地址：

```
#include <stdio.h>
int main()
{
    int var = 10;
    int *ptr_var;
    ptr_var = &var;
    printf(" var 的值是: %d", var);
    printf("\n var 的内存地址是: %x", &var);
    printf("\n 指针 ptr_var 的地址是: %x\n", &ptr_var);
    printf("\n var 的值是: %d", *ptr_var);
    printf("\n var 的地址是: %x\n", ptr_var);
    return 0;
}
```

8.3.7.2.如下声明两个变量分别表示什么

```
int *p, q;
```

8.3.7.3.区别指针

```
int *ptr_var;
ptr_var、&ptr_var、*ptr_var 三者的区别？
```

8.4.指针运算

指针运算的本质是指针中存储的地址的运算。指针可参与的运算，并不多，但很特别。常见的有如下：

8.4.1.赋值运算

区别初始化和赋值。

不兼容类型赋值会发生类型丢失。为了避免隐式转化带来可能出现的错误，最好用强制转化显示的区别。

8.4.2.算术运算

指针的算术运算，不是简单的数值运算，而是一种**数值加类型**运算。将指针加上或者减去某个整数值（以 $n * \text{sizeof}(T)$ 为单位进行操作的）。

运算符	+	-	++	--
示例	p+5;	p-5;	p++/++p;	p--/--p;

```
#include <stdio.h>

// T a;    a + 1;
// T* p;   p + 1;  sizeof(T);
//所以说，指针的运算，不是一种单纯的数值运算，而数值+类型一种运算
//p + 2 = q;  p + 2*sizeof(T) = q      (q - p)/ sizeof(T);

int main(int argc, char *argv[])
{
    int a = 0x0001;
    printf("a = %#x  a+1 = %#x\n", a, a+1);

    int *p = (int*)0x0001;
    printf("p= %#x  p+1 = %#x\n", p, p+1);

    int aa = 0x0010;
    printf("aa = %#x  aa+1 = %#x\n", aa, aa+1);

    int *q = (int*)0x0010;
    printf("p= %#x  p-1 = %#x\n", q, q-1);

    int m = 0x11;
    printf("%#x\n", --m);
    double *pm = (double*)0x11;
```

```

printf("%#x\n",pm--);

int arr[10];
int *parr = &arr[0]; int *qarr = &arr[8];
printf("%d\n",parr-qarr);
printf("%d\n", (int)parr-(int)qarr);
return 0;
}

```

注意：只有当指针指向一串连续的存储单元时，指针的移动才有意义。才可以将一个指针变量与一个整数 n 做加减运算。

8.4.3. 关系运算

运算符	==	>	<
示例	p1==p2	p2 > p2	p1 < p2

```

#include<stdio.h>
int main ()
{
    int *ptrnum1, *ptrnum2;
    int value = 1;
    ptrnum1 = &value;
    value += 10;
    ptrnum2 = &value;
    if (ptrnum1 == ptrnum2)
        printf("\n 两个指针指向同一个地址\n");
    else
        printf("\n 两个指针指向不同的地址\n");
    return 0;
}

```

8.4.4. 小结

- 指针的运算只能发生在同类型或整型之间，否则会报错或是警告。
- 指针的运算，除了数值以外，还有类型在里面。

8.5.数组遇上指针

8.5.1.一维数组的访问方式

8.5.1.1.传统方式(下标/偏移法)

数组名是数组的唯一标识符，数组名代表数组首元素的地址。我们可以用**下标**的方式对数组进行访问。

除此之外，还可以用，**本质方法**进行访问。

```
int main(void)
{
    int array[10] = {1,2,3,4,5,6,7,8,9,0};

    for(int i=0; i<10; i++)
    {
        printf("array[%d] = %d\n",i,array[i]);
    }

    printf("+++++++++\n");

    for(int i=0; i<10; i++)
    {
        printf("array[%d] = %d\n",i,*(&array+i));
    }
    return 0;
}
```

8.5.1.2.数组名是常量指针

数组名是常量，才可以唯一的确定数组元素的起始地址。

8.5.1.3.一维数组名跟一级指针的关系

数组除了可以用下标法和本质法访问以外，还可以用指针法访问。能用数组名解决的问题的，都可以用指针来解决，而能用指针来解决的问题，并一定能用数组名来解决。

8.5.1.4.指针访问方式

冲锋枪的年代，是该放弃大刀长矛，小米加步枪，改换AK了，以下是论述，一维数组，在学了指针后如何一步步，改进和更新自己的访问方式的。



```
#include <stdio.h>

int main(void)
{
//大刀，长矛
    int array[10] = {1,2,3,4,5,6,7,8,9,0};
    printf("array      = %p\n",array);
    printf("&array[0] = %p\n",&array[0]);
    //array 代表首元素地址， array[0]就是数组的首元素，
    //类型是 int 类型，其地址类型就是 int *。
    //故可将一维数组跟跟一级指针联系在一起。

    int *p = array;

    for(int i =0; i<10; i++)
    {
        printf("array[%d] = %d\n",i,/*p[i]*/ *(p+i));
    }
// 小米加步枪
//     printf("*****\n");
//     for(int i =0; i<10; i++)
//     {
//         printf("array[%d] = %d\n",i,*p);
//         p++;
//     }
    printf("*****\n");

//冲锋枪
    for(int i =0; i<10; i++)
    {
//        printf("array[%d] = %d\n",i,*p++);
    }
```

```
    printf("array[%d] = %d\n", i, *(p++));
}
return 0;
}
```

8.5.1.5.小结

- 1.数组名是一个常量，不允许重新赋值。
- 2.指针变量是一个变量，可以重新赋值。
- 3. $p+i$ 和 $a+i$ 均表示数组元素 $a[i]$ 的地址，均指向 $a[i]$
- 4. $*(p+i)$ 和 $*(a+i)$ 均表示 $p+i$ 和 $a+i$ 所指对象的内容 $a[i]$ 。
- 5. $*p++$: 等价于 $*(p++)$ 。其作用：先得到 $*p$ ，再使 $p=p+1$ 。
- 6. $(*p)++$: 表示将 p 所指向的变量（元素）的值加 1。即等价于 $a[i]++$ 。
- 7.指向数组元素的指针也可以表示成数组的形式，即允许指针变量带下标，如 $*(p+i)$ 可以表示成 $p[i]$ 。

8.5.2.二维数组的访问方式

8.5.2.1.下标法

数组元素的表示方法是：数组名称[行][列]，对于 m 行 n 列的二维数组， $a[0][0]$ 是数组的第一个元素， $a[m-1][n-1]$ 是最后一个元素。

8.5.2.2.本质偏移法

从 a 到 $a[0]$ 到 $a[0][0]$ 都经历了什么？

a		a[i]	a[i][j]	
		*(a+i)	*(*(a+i)+j)	
		400		
		300		
		200		
	a[2]	100	a[2]	
		40		
		30		
		20		
	a[1]	10	a[1]	
		4		
		3		
		2		
	a[0]	1	a[0]	
a+2	60FE98		a[0]+2	60FE80
a+1	60FE88		a[0]+1	60FE7C
a	60FE78		a[0]	60FE78

求证：

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

#define M 3
#define N 5

int main()
{
    int arr[3][4] = {1,2,3,4,10,20,30,40,100,200,300,400};

    for(int i=2; i>=0; i--)
    {
        for(int j=3; j>=0; j--)
        {
            printf("%#x\n",&arr[i][j]);
        }
    }

    printf("arr      = %#p  arr+1      = %#x      arr+2 = %#x \n",
           arr,arr+1,arr+2);

    printf("&arr[0] = %#p  &arr[0]+1 = %#x  &arr[0]+2 = %#x \n",
           &arr[0],&arr[0]+1,&arr[0]+2);

    printf("*arr      = %#p  *arr+1      = %#x      *arr+2 = %#x \n",
           *arr,*arr+1,*arr+2);

    printf("arr[0] = %#p  arr[0]+1 = %#x  arr[0]+2 = %#x \n",
           arr[0],arr[0]+1,arr[0]+2);

    printf("%d\n",arr[1][1]);
    printf("%d\n",*(arr[1]+1));
    printf("%d\n",*(*(arr+1)+1)); //行，列

    //arr[i] 第 i 行，首元素的地址 一级指针
    //arr+i 第 i 行的地址 数组指针

    return 0;
}
```

8.5.2.3.小结

- **a** 是数组首元素的地址，所以 **a** 的值和 **&a[0]** 的值相同，另一方面，**a[0]** 本身是包含 **4** 个整数的数组，因此，**a[0]** 的值同其首元素的地址 **&a[0][0]** 相同。简单的讲，**a[0]** 是一个整数大小对象的地址，而 **a** 是 **4** 个整数大小对象的地址。因为整数和 **4** 个整数组成的数组开始于同一个地址，因此 **a** 和 **a[0]** 的值是相同的。
- **a** 所指的对象大小是 **4** 个 **int**，而 **a[0]** 所指的对象大小一个 **int**，因此，**a+1** 和 **a[0]+1** 的结果是不同的。

8.5.2.4.推论

- 二维数组名解引用，降维为一维数组名。 ***(a+1) <> a[1]**
- 一维数组名，对其引用，升级为二维数组名。 **&a[1] <> (a+1)**
- **&**引用和*****解引用互为逆向关系。

此上的说法，不是很严谨，但是限于我们的知识面，当下是可以这样理解的。

8.5.2.5.指针法

后续提高课程，再作详细讲解。前文中我们讲到，一维数组名，可以赋给一级指针，也可以说他们是等价的。但是二维数组名，跟二级指针，没有毛关系，不可作赋值或是论等价关系。

二维数组名的本质是，数组指针。

8.6.练习

8.6.1.用指针法逆序打印一个数组。

数组：int arr[10] = {1,2,3,4,5,6,7,8,9,0};

8.6.2.往指定内存写入数据

请写一条语句，往内存地址 0x12345678 中写入整型数据 1234。

8.6.3.判断是否是回文串

有字符数组如下 char name[5] = {'M', 'A', 'D', 'A', 'M'}，判断其是否是回文串。

9. 函数(Function)

前面我们所学的语法，大家觉的已经是 C 语言的大部分或是全部，这样理解其实是有偏颇的。准确的讲 C 语言包含两部分，**c 语法和 c 标准库**。除了标准库里的函数可以用外，我们还可以自定义函数。总结起来，使用函数，可以有如下好处：

- 可以提高程序开发的效率。
- 提高了代码的重用性。
- 使程序变得更简短而清晰。
- 有利于程序维护。

如果听了这些，你不懂，printf 这个函数你估计也用了上百遍了吧，你不会有冲动每次使用前先自己实现一个吧。那你可能会问，我也没有看到过 printf 的实现呀，对，他被封装到了标准库里了，**库**是对函数的再一次升级。

9.1.c 标准库及库函数

9.1.1. 库存在的意义

将常用的函数分类组织到一起，即可制作成函数库。库存在的意义，就是**避免重复造轮子**。像我们使用的 printf 函数，没有必要每个使用此功能的函数，都要重写来实现。**重复现成的、可用的、已经证明很好用的东西就是造轮子。**



9.1.2. 如何使用库函数

由 C 语言系统提供；**用户无须定义，也不必在程序中作类型说明；只需在程序前包含有该函数定义的头文件(/usr/include/stdio.h)，而不关系库在哪里(/usr/lib/libc.so)；**标准库到底提供了哪些函数可以通过查表的方式获得。

函数的三要素，函数名，函数参数，函数返回值，却是我们要研究的。

头文件中函数原型样了如下：

```
returnType funName (arg1Type arg1, arg2Type arg2)
```

解析如下：

函数声明	<code>returnType funName (arg1Type arg1, arg2Type arg2)</code>				
所在文件	<code>xxx.h</code>				
函数功能	介绍函数功能				
参数及返回解析					
参数	<code>arg1</code>		返回	<code>returnType</code>	
	<code>arg2</code>				

重点关心：函数的输入与输出。

9.1.3. 库函数使用示例

9.1.3.1. 随机函数

使用随机函数产生，某一范围内了随机数。比如生成[1,100]以内的随机数。`srand` 和 `rand()`配合使用产生伪随机数序列。`rand` 函数在产生随机数前，需要系统提供的生成伪随机数序列的种子，`rand` 根据这个种子的值产生一系列随机数。如果系统提供的种子没有变化，每次调用 `rand` 函数生成的伪随机数序列都是一样的。



函数声明	<code>int rand (void);</code>	
所在文件	<code>stdlib.h</code>	
函数功能	Generate random number.Returns a pseudo-random integral number in the range between 0 and RAND_MAX.产生一组[0,RAND_MAX]伪随机数。	
参数及返回解析		
参数	<code>void</code>	none
返回值	<code>int</code>	An integer value between 0 and RAND_MAX.

函数声明	<code>void srand (unsigned int seed);</code>	
所在文件	<code>stdlib.h</code>	

函数功能	Initialize random number generator.	
参数及返回解析		
参数	seed unsigned int	An integer value to be used as seed by the pseudo-random number generator algorithm.
返回值	void	none

```
#include <stdio.h>

int main(void)
{
    srand(time(NULL));
    for(int i=0; i<10; i++)
    {
        printf("%d \t", rand()%100);
    }
    return 0;
}
```

世界上是没有偶然的，一切随机都只是我们无法解释而已所以计算机的随机数也是假的。

随机数只是用过一个“种子”，也就是一个整数通过某公式得到的，而这个随机数也将作为下一个随机数的“种子”。

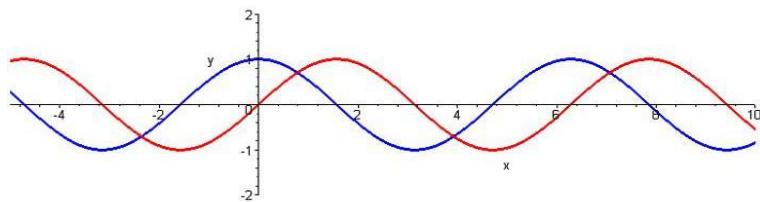
如何生成 30 个不重复的随机数并存储的 int randNum[30]中？如何生成某一范围内的随机数，比如[100,200]内的随机数？

9.1.3.2.正弦函数

应用数学函数，要求在面板上画一个，正弦曲线。在 linux 中使用数学库需要在编译的时候手动加上 -lm 链接参数。

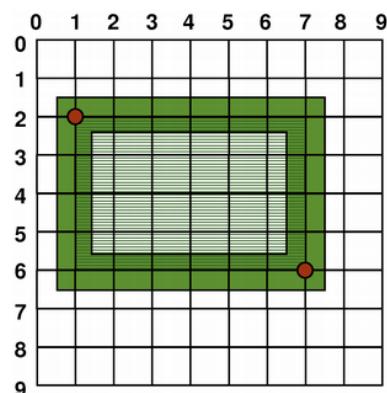
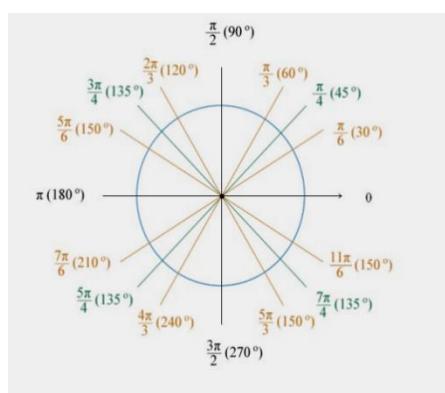
函数声明	<code>double sin(double x);</code>	
所在文件	math.h	
函数功能	Returns the sine of an angle of x radians.	
参数及返回解析		
参数	double	Value representing an angle expressed in radians. One radian is equivalent to 180/PI degrees.
返回值	double	Sine of x radians.

$$y = \sin x \quad y = \cos x$$



角弧变换

窗口坐标系



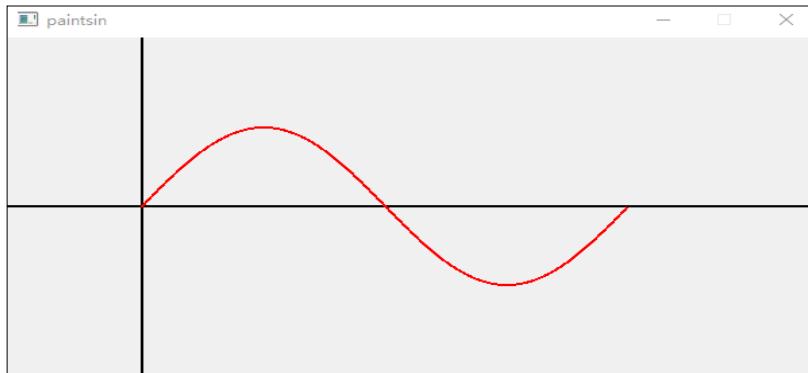
```
void MainWindow::paintEvent(QPaintEvent * event)
{
    //定义画笔
    QPainter painter(this);

    //定义画坐标笔的，宽度，颜色
    QPen pen;
    pen.setWidth(2);
    pen.setColor(Qt::black);
    //使设置生效
    painter.setPen(pen);
    //画坐标线
    painter.drawLine(0,150,600,150);
    painter.drawLine(100,0,100,300);

    //将绘画的坐标原点，转移到(100,150)的位置
    painter.translate(100,150);

    //重新设置画正弦颜色和样式
    pen.setColor(Qt::red);
    pen.setStyle(Qt::SolidLine);
    //使设置生效
    painter.setPen(pen);
    //采集 360 个点用于绘图
    QVector<QPointF> pVector;
    for(int i=0; i<360; i++)
    {
        pVector.push_back(QPointF(i, -sin((3.14/180*i)*70)));
        //y 轴坐标，相反
    }
    // 画 多点线条
    painter.drawPolyline(pVector);
}
```

实验结果：



9.1.3.3. 获取时间函数

函数声明	<code>struct tm * localtime (const time_t * timer);</code>		
所在文件	stdlib.h		
函数功能	Convert time_t to tm as local time		
参数及返回解析			
参数	<code>time_t*</code>	返回从 1990.1.1 00:00:00 累积的秒数	
返回值	<code>struct tm*</code>	详见下表	

返回值详细解析：

Type	variable	Meaning	Range
int	tm_sec	seconds after the minute	0-61*
int	tm_min	minutes after the hour	0-59
int	tm_hour	hours since midnight	0-23
int	tm_mday	day of the month	1-31
int	tm_mon	months since January	0-11
int	tm_year	years since 1900	
int	tm_wday	days since Sunday	0-6
int	tm_yday	days since January 1	0-365
int	tm_isdst;	Daylight Saving Time flag	

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

int main(void)
{
    time_t t = time(NULL);
    struct tm *localt = localtime(&t);

    printf("year    = %d\n", localt->tm_year+1900);
    printf("month   = %d\n", localt->tm_mon);
    printf("day     = %d\n", localt->tm_mday);
```

```

printf("hour    = %d\n", localt->tm_hour);
printf("min     = %d\n", localt->tm_min);
printf("second  = %d\n", localt->tm_sec);
return 0;
}

```

9.1.4. 常用库函数

9.1.4.1. 列表

常用库函数	头文件	功能
<code>double sqrt(double x)</code>	<code>math.h</code>	计算 <code>x</code> 的平方根 (squareroot)
<code>double pow(double x, double y)</code>		计算 <code>x</code> 的 <code>y</code> 次幂 (power)
<code>double ceil(double x)</code>		求不小于 <code>x</code> 的最小整数，并以 <code>double</code> 形式显示
<code>double floor(double x)</code>		求不大于 <code>x</code> 的最大整数，并以 <code>double</code> 形式显示
<code>int toupper(int x)</code>	<code>ctype.h</code>	如果 <code>x</code> 为小写字母，则返回对应的大写字母
<code>int tolower(int x)</code>		如果 <code>x</code> 为大写字母，则返回对应的小写字母
<code>int rand(void)</code>	<code>stdlib.h</code>	产生一个随机数
<code>void srand(unsigned int seedS)</code>		产生一个随机数种子

9.1.4.2. 实例

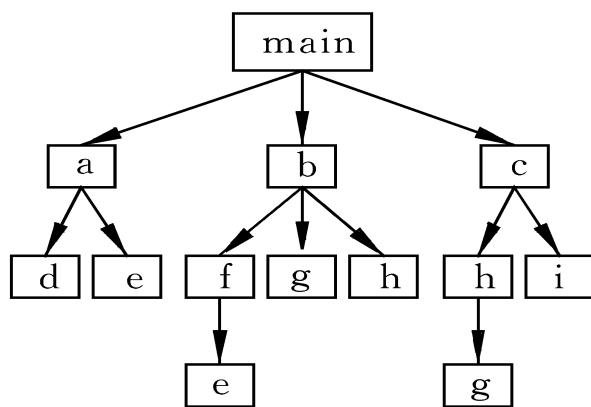
```

#include <stdio.h>
#include <math.h>
int main()
{
    double x=1;
    double squareroot,power;
    while(x <= 10)
    {
        squareroot=sqrt(x);
        power=pow(x,3);
        printf("%.0f 的平方根:%.2f\t%.0f 的立方:%.0f\n",
               x,squareroot,x,power);
        x++;
    }
    return 0;
}

```

9.2.自定义函数

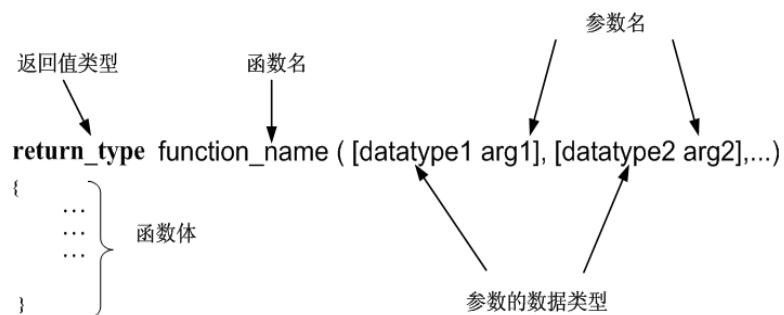
C 语言又称为是**函数语言**，将具体功能函数化，然后组织整个架构逻辑，这样亦是软件的设计原则，**自顶向下，逐步细化**。比如下面的代码结构，就可以看的出来，整个程序实现的功能从 main 开始，分了三个大的模块。每个模块下，又有子模块。



最上面的 main 代表了一个函数，下面的 **a b c d e f g h i g**，都是一个个函数。c 语言就是以函数来组织架构的。

9.2.1.语法规式

自定义函数必须满足如下格式，样式跟我们前面使用的库函数基本一致，只不过多了一个函数体，需要去实现。



例如定义一个函数求两个数的最大值：

```

double max(double x, double y) //函数定义中的参数称为形参
{
    double m;
    m=x>y?x:y;
}
  
```

```

    return m; //return (x>y?x:y);
}

```

该函数名为 max，它有两个 double 类型的参数，返回值为 double 类型。在函数体内有三条语句实现了求两个数中较大的数，并将它返回。

9.2.2. 调用方法

```

#include <stdio.h>

double myMax(double x, double y) //函数定义中的参数称为形参
{
    double m;
    m=x>y?x:y;
    return m; //return (x>y?x:y);
}

int main()
{
    double a = 5.6;
    double b = 7.8;

    double maxDouble = myMax(a,b); // 函数调用时传入的参数称为实参
    printf("两者中的最大值为 = %f\n",maxDouble);
    return 0;
}

```

9.2.3. 前向声明

若是定义在前，调用在后，则会正常。如果定义在后，调用在前，编译器则会 Complain。

```
D:\QtProject\0510\sqrtpow\main.c:9: warning: implicit declaration of function
'myMax' [-Wimplicit-function-declaration] double maxDouble = myMax(a,b);
```

```

#include <stdio.h>

double myMax(double x, double y);

int main()
{
    double a = 5.6;
    double b = 7.8;

    double maxDouble = myMax(a,b);
    printf("两者中的最大值为 = %f\n",maxDouble);
}

```

```

    return 0;
}

double myMax(double x, double y) //函数定义中的参数称为形参
{
    double m;
    m=x>y?x:y;
    return m; //return (x>y?x:y);
}

```

9.2.4. 函数返回值

有调用如下：int * p; p = function(); 此处function该如何声明？

9.2.5. 实参与形参

9.2.5.1. 形参

在定义或声明函数中指定的形参，在未出现函数调用时，它们并不占内存中的存储单元。只有在发生函数调用时，形参才被分配内存单元。在调用结束后，形参所占的内存单元也被释放。如

```
int myMax(int a, int b); int myMax(int a, int b){};
```

9.2.5.2. 实参

实参可以是常量、变量或表达式，但要求它们有确定的值。在调用时将实参的值赋给形参。如：

```
myMax(a,3+b);
```

9.3. 传值与传址

传值与传址，**本质是都是传递一个数值而已。**

9.3.1. 传值与传址的比较

小测试，下面的程序输出几？

```

#include <stdio.h>
void func(int a);
int main()
{
    int a = 0; //初始化 a 值为 0
    func(a); //调用函数 func
    printf("%d",a); //输出 a 的结果
}
//定义函数 func

```

```
void func(int a)
{
    a = 10; //让参数 a 等于 10
//    a++;
}
```

前面我们已经学过，通过第三个变量实现交换两个空间数据。今天我们把它打包成函数，这样就不必每次重写了。

9.3.1.1.mySwap 之传值

```
#include <stdio.h>

int main(void)
{
    int a = 4; int b = 5;
    printf("a = %d b = %d\n",a,b);

    /* 对此段功能进行打包
    int t = a;
    a = b;
    b = t;
    */

    printf("a = %d b = %d\n",a,b);

    return 0;
}
```

```
#include <stdio.h>
void mySwap(int i, int j)
{
    int t = i;
    i = j;
    j = t;
}

int main(void)
{
    int a = 4; int b = 5;
    printf("a = %d b = %d\n",a,b);

    mySwap(a,b);
```

```

    printf("a = %d b = %d\n",a,b);

    return 0;
}

```

9.3.1.2.mySwap 之传址

```

void mySwap(int *pa, int *pb)
{
    int t = *pa;
    *pa = *pb;
    *pb = t;
}

int main(void)
{
    int a = 4; int b = 5;
    printf("a = %d b = %d\n",a,b);

    mySwap(&a,&b);

    printf("a = %d b = %d\n",a,b);

    return 0;
}

```

9.3.2.图解传值与传址

传值				传址		
int t;	int x;	00		00	int *pa	int t;
		00		28		
00		00		fe		00
00		05		bc		00
00	int y;	00		00	int *pb	00
03		00		28		03
		00		fe		
		03		b8		
	int a		00			
			00			
			00			
			03	0028FEB8		

		int b	00			
			00			
			00			
	05		0028FEBC			

传递地址，可能改变地址所对应原数值，是因为，地址对于不同作用域的函数来讲是公开的。

9.3.3.如何来传递一个一维数组

数组的声明中，已经把数组访问的三要素，均已表示出来了，三要素就是，起始位置，移步刻度，终止位置。

那么我们在传递一个数组的时候，能不能完完全的，把这**三要素表示出来呢**？

```
#include <stdio.h>

void selectSort(int *p ,int n)
{
    for(int i=0; i<n-1; i++)
    {
        int tmp;
        for(int j= i+1; j<n; j++)
        {
            if(p[i]>p[j])
            {
                tmp = p[i];
                p[i] = p[j];
                p[j] = tmp;
            }
        }
    }
}

int main()
{
    int array[10] = {99,33,66,22,100,300,600,350,1,2};

    for(int i=0; i<10; i++)
    {
        printf("%d    ",array[i]);
    }
    putchar(10);
}
```

```
selectSort(array,10);

for(int i=0; i<10; i++)
{
    printf("%d    ",array[i]);
}
}
```

支解选择排序，代码重在梳理逻辑：

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

void swap(int *p,int *q)
{
    int t = *p;
    *p = *q;
    *q = t;
}

int getSmallestIdx(int i,int *p, int n)
{
    int idx = i;
    for(int j=i+1; j<n; j++)
    {
        if(p[j] < p[idx])
            idx = j;
    }
    return idx;
}

void selectSort(int *p, int n)
{
    int smallIdx;
    for(int i=0; i<n; i++)
    {
        smallIdx = getSmallestIdx(i,p,n);
        if(smallIdx != i)
            swap(&p[i],&p[smallIdx]);
    }
}

int main(void)
```

```

{
    int a[10] = {1,3,5,7,9,2,4,6,8,0};
    selectSort(a,10);

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

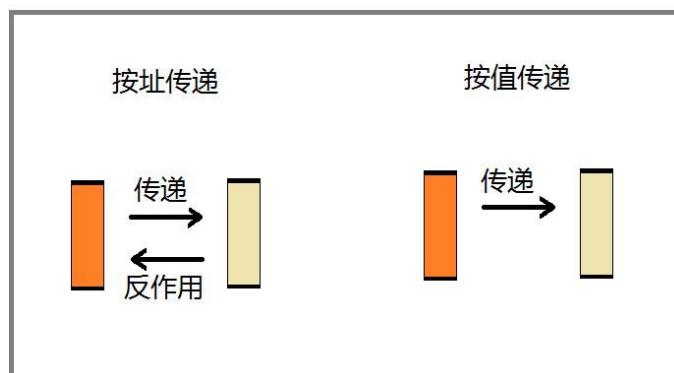
    return 0;
}

```

9.3.4. 如何来传递一个二维数组

留给课下思考，C 提高部分会重点讲解。

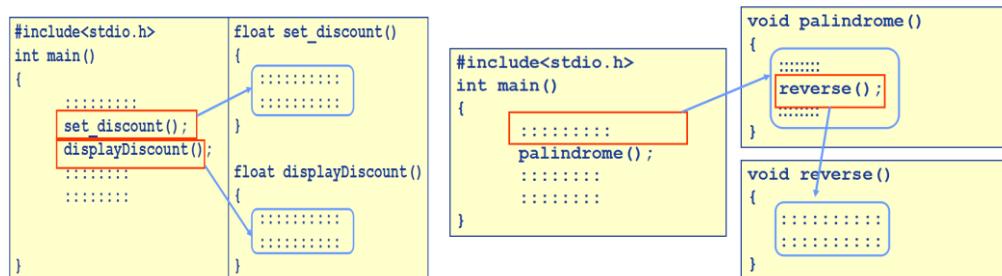
9.3.5. 小结



9.4. 函数调用

9.4.1. 普通调用

所有函数都是平行的，即在定义函数时是分别进行的，是互相独立的。函数间可以互相调用。常见有，平行调用，嵌套调用。



但，一个函数并不从属于另一函数，即函数不能嵌套定义（但是在有的平台，也是允许嵌套定义的，但是为了兼容起见，不建议去这样作）。

```
#include <stdio.h>

void foo(); //前向声明

void func()
{
    printf("void func()\n");
    foo(); //嵌套调用, debug(F10/F11 的区别)
}

void foo()
{
    printf("void foo()\n");
}

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

9.4.2. 递归调用

既然函数，是平行的，可以相互调用，如果自己调用自己，会如何呢？如下代码会输出什么呢？

```
#include <stdio.h>

void func()
{
    printf("void func()\n");
    func();
}

int main()
{
    func();
//    main(); //main 函数能不能调用自己呢？
    return 0;
}
```

9.5.递归(Recursive)详解

9.5.1.递归定义

函数，**自身调用自身**，或是间接调用自身的的现象，称为递归。比如生活中，经常有这样的推理过程。比如，举如下两个精彩的例子。

9.5.1.1.推算年龄

有 5 个人坐在一起，问第 5 个人多少岁？他说比第 4 个人大 2 岁。问第 4 个人岁数，他说比第 3 个人大 2 岁。问第 3 个人，又说比第 2 个人大 2 岁。问第 2 个人，说比第 1 个人大 2 岁。最后问第 1 个人，他说是 10 岁。请问第 5 个人多大？

```
age(5) = age(4)+2
age(4) = age(3)+2
age(3) = age(2)+2
age(2) = age(1)+2
age(1) = 10
```

可以用数学公式表述如下：

$$\text{age}(n) = \begin{cases} \text{age}(1) = 10 & (n=1) \\ \text{age}(n) = \text{age}(n-1)+2 & (n>1) \end{cases}$$

```
#include <stdio.h>

int getAge(int idx)
{
    if(idx == 1)
        return 10;
    else
        return getAge(idx-1)+2;
}

int main(void)
{
    int age = getAge(5);
    printf("第一个人的年龄是 %d\n", Age);
    return 0;
}
```

9.5.1.2.猴子吃桃子的问题

猴子第一天摘下若干个桃子，当即吃了一半，还不过瘾，又多吃了1个。第二天早上又将剩下的桃子吃掉一半，又多吃了1个。以后每天早上都吃了前一天剩下的一半零1个。到第10天早上想再吃时，见只剩下一个桃子了。求第一天共摘了多少？

推导过程：

```

peach( 1) = ( peach( 2) + 1 )*2
peach( 2) = ( peach( 3) + 1 )*2
peach( 3) = ( peach( 4) + 1 )*2
peach( 4) = ( peach( 5) + 1 )*2
peach( 5) = ( peach( 6) + 1 )*2
peach( 6) = ( peach( 7) + 1 )*2
peach( 7) = ( peach( 8) + 1 )*2
peach( 8) = ( peach( 9) + 1 )*2
peach( 9) = ( peach(10) + 1 )*2
peach(10) = 1

```

可以用数学公式表述如下：

$$\text{peach}(n) = \begin{cases} (\text{peach}(n+1)+1) * 2 & (n>1) \\ \text{peach}(n) = 1 & (n=10) \end{cases}$$

代码实现：

```

#include <stdio.h>

int peachCount(int day)
{
    if(day == 10)
        return 1;
    else
        return (peachCount(++day)+1)*2;
}

int main(void)
{
    int count = peachCount(1);
    printf("第一天桃子个数是 %d\n", count);
}

```

```

    return 0;
}

```

9.5.2. 递归小结

9.5.2.1. 公式结构

$$\text{fun}(n) = \begin{cases} 1 & n = 0; \quad // \text{终止条件} \\ \text{fun}(n-1)*2 & n > 0; \quad // \text{递归条件} \end{cases}$$

9.5.2.2. 书写结构

```

递归返回 func(递归条件)
{
    if(递归终止条件)
        终止处理;
    else
        func(趋于递归终结的条件);
}

```

9.5.3. 递归与循环论述

递归和循环，有共同的特点，有起点，有终点，重复作同样的事情。所以很多情况，两者是可以相互转换的。

如果上升一下理论高度，做一个重复而有明确起点和终点的事，有递归和迭代两种选择。循环其实就是一种迭代。

递归的方式，写法比较简洁，符合正常逻辑，但代码理解难度大，内存消耗大(易导致栈溢出)，所以能用迭代(Iterative)解决的问题，不要用递归来完成。

9.5.3.1. 递推年龄，递归改迭代

```

int getAge(int idx)
{
    int age = 10;
    for(int i=1; i<idx; i++)
    {
        age += 2;
    }
    return age;
}

int main(void)

```

```
{  
  
    int age = getAge(5);  
    printf("第一个人的年龄是 %d\n",age);  
    return 0;  
}
```

9.5.3.2.猴子吃桃，递归改迭代

```
int peachCount(int day)  
{  
    int count = 1;  
    for(int i=10; i>day; i--)  
    {  
        count = (count+1)*2;  
    }  
    return count;  
}  
  
int main(void)  
{  
  
    int count = peachCount(1);  
    printf("第一天桃子个数是 %d\n",count);  
    return 0;  
}
```

9.5.4.递归练习

9.5.4.1.下面的代码输出啥

据说，下面的代码，10个人中有9个会出错？

```
void fun(int i)  
{  
    if (i>0)  
    {  
        fun(i/2);  
    }  
    printf("%d\n",i);  
}  
int main()  
{  
    fun(10);  
    return 0;
```

```
    }
```

9.5.4.2.阶乘(factorial)

```
n! = n * (n-1) * (n-2) * ... * 1(n>0); // 0! = 1; n! = n * (n-1)!; 阶乘: factorial
```

9.6.练习

9.6.1.如何产生[m,n]以内的随机数。

写一个程序，实现[30,100]以内的随机数。

9.6.2.产生 30 个不重复的随机数

产生 30 个不重复的随机数放到指数组 int randInt[30]中去。

9.6.3.请用函数封装基于数组的冒泡排序，选择排序。

```
void popSort(int *p, int n);
void selectSort(int *p, int n);
```

9.6.4.用函数的思想来支解选择法排序

```
void selectSort(){
    for(){}
        getSmallestIdx()
        if() swap()
    }
}
```

9.6.5.请用函数封装基于数组的二分查找。

找到返回下标，找不到返回-1。

```
int binarySearch(int *p, int low, int high, int findData);
```

9.6.6.求中位数

给定一个未排序的整数数组，求中位数。

10. 变量作用域/生命周期/修饰符



鹰击长空，鱼翔浅底，万类霜天竞自由。

10.1. 作用域(Scope)

10.1.1. 作用域

10.1.1.1. 划分

某事物起作用或有效的区域，称之为作用域。{}是作用域的限定符。{}以内的区域称为局部作用域，{}以外的称为全局作用域。

```
#include <stdio.h>
void foo(int a){
    a++;
    printf("a = %d\n",a);
    // int a; 不同作用域内可以重名，同一作用域内则会发生重定义
}
int main(void){
    int a = 5;
    printf("a = %d\n",a);
    return 0;
}
```

10.1.1.2. 局部变量(local variable)

凡是大括号{}以内的变量，都是局部变量。附：形式参数也是局部变量。其作用域，起始于定义处，截止于所在的大括号。

若未赋值，其值是随机的。

10.1.1.3. 全局变量(global variable)

凡是大括号{}以外的变量，都是全局变量。其作用域起始于定义处，截止于本文件结束。

若未赋值，系统将其初始化为，零。

全局量，在多文件编程中，可以通 `extern` 声明的方式，将作用域扩展到其它文件中去。

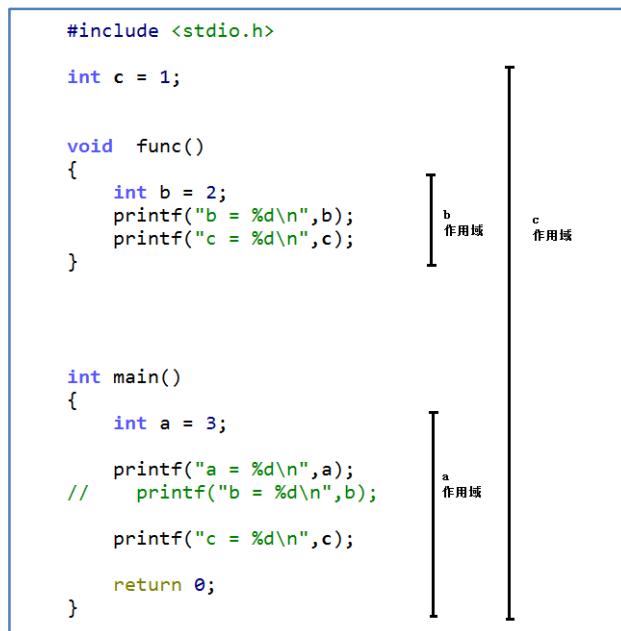
```
#include <stdio.h>

int var = 6;

int main(void)
{
    int var = 16;
    printf("in {} var = %d\n",var);
    return 0;
}
```

10.1.1.4.图示作用域

作用域可以划分为两在类，一类是局部作用域，一类是全局作用域。图示局部变量 `a,b` 和全局变量 `c` 的作用域。



10.1.2.作用域叠加

当小范围内的作用域和大范围的作用域，出现叠加，且有重名变量的时候，小范围作用域内的变量名会覆盖大范围作用域内的变量名。也就是通常所说的，局部变量同全局变量重名的问题。

强龙不压地头蛇。

```
#include <stdio.h>
int a = 5;
int main(void)
```

```
{
    printf("a = %d\n",a);
    int a = 6;
    {
        //作用域限定符，并不会影响执行流程。

        printf("a = %d\n",a);

        int a = 4;
        printf("a = %d\n",a);
    }
    printf("a = %d\n",a);
    return 0;
}
```

10.1.3. 全局命名污染

不同作用域的局部变量，即使重名也是无所谓，不会对程序的编译，造成任何的影响。

全局变量则不然，全局变量是指同一工程内的所有 c 文件中的全局变量，在全局作用域内，如果有重名则会造成重定义。当一个项目，大到由 $n(n>1000)$ 多个文件，有 $m(m>500)$ 多个人参与开发。在进行编译和调试的过程中会带来很多的麻烦，所以尽量少用全局变量。

除非，有人负责对全局变量进行统一管理。

```
main.c
#include <stdio.h>

int a = 8;

int main(void)
{
    printf("a = %d\n",a);

    return 0;
}

other.c
int a = 5;
```

```
error: multiple definition of `a'
```

10.2. 生命周期(Duration)

10.2.1. 局部变量

局部变量的生命周期，同其所在的函数。局部变量随着函数的执行而有生命，随着函数的执行结束而生命截止。



main 函数是一个比较特别的函数，作为进程的第一个调用的函数，进程随着 main 函数的执行而开始，随着 main 函数的结束而结束。所以 main 函数中的局部变量的生命周期同 main 函数或同进程。

10.2.2. 全局变量

全局变量的生命周期同进程，或是 main() 函数或进程。

10.3. 修饰符(Storage Description)

修饰符放在，变量的定义的类型之前，或改变生命周期，或改变存储区域，或者均有所改变。格式如下：

```
auto int a;
```

10.3.1. auto(大将军)

只能修饰局部变量，可以省略，局部变量若无其它的修饰，则默认为 auto。它修饰的变量的特点是，随用随开，用完即消。

结论：忘记。



10.3.2.register(小太监)

只能修饰局部变量，原则上，将内存中的变量升级到 CPU 寄存器中存储，这样访问速度会更快。但由于 CPU 寄存器数量相当有限，通常会在程序优化阶段，被优化为普通的 auto 类型变量。可以通过汇编代码来查看，优化过程(具体优化，与平台和编译相关)。



结论：忘记。

```

register int a = 12;
<+0x0011>      bf 0c 00 00 00      mov    $0xc,%edi
register int b = 12;
<+0x0016>      be 0c 00 00 00      mov    $0xc,%esi
register int c = 12;
<+0x001b>      ba 0c 00 00 00      mov    $0xc,%edx
register int d = 12;
<+0x0020>      b9 0c 00 00 00      mov    $0xc,%ecx
register int e = 12;
<+0x0025>      c7 44 24 0c 0c 00 00 00  movl   $0xc,0xc(%esp)
register int f = 12;
<+0x002d>      c7 44 24 08 0c 00 00 00  movl   $0xc,0x8(%esp)
register int g = 12;
<+0x0035>      c7 44 24 04 0c 00 00 00  movl   $0xc,0x4(%esp)
register int h = 12;
<+0x003d>      c7 04 24 0c 00 00 00      movl   $0xc,(%esp)

```

```

register int i = 12;
<+0x0044>      bb 0c 00 00 00          mov    $0xc,%ebx
int sum = a + b + c + d + e + f + g + h + i;
<+0x0049>      8d 04 37          lea    (%edi,%esi,1),%eax
<+0x004c>      01 d0          add    %edx,%eax
<+0x004e>      01 c8          add    %ecx,%eax
<+0x0050>      03 44 24 0c          add    0xc(%esp),%eax
<+0x0054>      03 44 24 08          add    0x8(%esp),%eax
<+0x0058>      03 44 24 04          add    0x4(%esp),%eax
<+0x005c>      03 04 24          add    (%esp),%eax
<+0x005f>      01 d8          add    %ebx,%eax
<+0x0061>      89 44 24 1c          mov    %eax,0x1c(%esp)
return 0;

```

10.3.3.extern(通关文牒)

只能用来修饰全局变量，**全局变量本身是全局可用的**，但是由于文件是**单个完成编译**，并且编译是**自上而下的**，所以说，对于不是在本范围内定义的全局变量，要想使用必须用 `extern` 进行声明，如果不加上 `extern`，就会造成重定义。

注意，经 `extern` 声明的变量，不可以再初始化。



10.3.3.1.跨文件：

C 语言，是单文件编译的，然后再将编译的.o 文件同库一起链接成可执行文件。正是因为这一点，跨文件使用全局变量，需要声明。

main.c

other.c

```
#include <stdio.h>

//extern int a;

int main(void)
{
    printf("a = %d\n", a);

    return 0;
}
```

```
int a = 5;
```

如果 main.c 中**没有** `extern int a`，但是由于后面又使用了变量 `a`，**则会报错**。如果 main.c 中**有** `extern int a`，则编译通过，如果在链接中，其它文件中**有定义**，则顺利链接通过，否则的话，则会链接失败。

全局变量，本质作用域内通用，即所有的 c 文件中均可使用，但是由于，**单文件编译，则需要声明**。`extern` 有省略用法，不讲，也不推荐。

10.3.3.2.同文件：

```
#include <stdio.h>

extern int a; //extern int a = 5;
//int a;
//int a;
//int a;      //声明

int main(void)
{
    printf("a = %d\n", a);

    return 0;
}

void func()
{
    printf("Hello World!\n");
}

int a = 5;      //定义
```

10.3.3.3.引申：变量的声明与定义

10.3.4.static(柱国老臣 | 限离出境)

10.3.4.1.修饰局部变量

`static` 修饰，局部变量，修改了局部变量的生命周期。使其生命周期同进程或是 `main()` 函数。

`static` 变量若未初始化，则系统初始化为零，并且只进行一次初始化。

```
void func()
{
    int i = 5;
    printf(" i = %d\n", i++);

    static int count=0;           //初始化赋值的不同
    count++;
    printf("count = %d\n", count); //count++;
}

int main()
{
    func();
    printf("+++++\n");
    func();
    printf("+++++\n");
    func();
    return 0;
}
```

10.3.4.2.静态局部变量应用

■ 用于控制流程

```
#include <stdio.h>

void print(int num, int age, char sex, int grade)
{
    static int flag = 1;
    if (flag == 1)
    {
        printf("学号\t年龄\t性别\t年级\n");
        flag = 0;
    }
    static int count = 0;
    printf("%d\t%d\t%c\t%d-->%d\n", num, age, sex, grade, ++count);
```

```
}

int main()
{
    print(1001, 23, 'x', 3);
    return 0;
}
```

- 用于统计函数(功能的)调用次数案例，或是加载资源文件。



```
void MainWindow::on_show_clicked()
{
    static QMovie movie("jzn.gif"); //硬盘资源只需要加载一次。
    ui->picture->setMovie(&movie);
    ui->picture->setVisible(true);
    movie.start();

}

void MainWindow::on_shadow_clicked()
{
    ui->picture->setVisible(false);
}
```

10.3.4.3.修饰全局变量

`static` 修饰，全局变量，限制了他的外延性。使其成为仅在本文件内部使用的全局变量。

我们在前面讲过，全局变量会带来全局空间的命名污染。这样既保留了全局变量的使用便利性，又不会造成全局空间的命名污染。

```

1 #include <stdio.h>
2
3 static int global = 100;
4
5 int main(void)
6 {
7
8     return 0;
9 }
10
11

```

```

1
2
3 static int global = 100;
4

```

10.3.4.4.修饰函数

函数本身就是全局可调用的，在编译时，只需要声明，即可完成编译，在链接的时候才去链接实现体。

`static` 修饰函数的意义，就在于将全局函数变成了，本文件内的全局函数。

10.4.小结

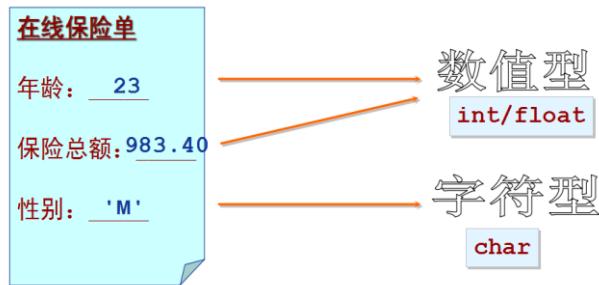
整理表格如下：

变量分类	生命周期	作用域	默认值	修饰符	生命周期	作用域	默认值
局部 变量				auto			
				register			
全局 变量				static			
				extern			

11.字符串(String)

一个普通程序员在企业中 **90%的精力**，都花在了字符串的处理上。足见他的重要性和使用的广泛性。

11.1.引入



前面我们学习了，基本数据类型和构造数据类型数组，那么如果我们想在保单上，表示一个人的名字该如何来描述呢？

依据我们现在所学的知识，可有两种方法：

```
#include <stdio.h>

int main(void)
{
    char a = 'w';
    char b = 'a';
    char c = 'n';
    char d = 'g';
    char e = 'g';
    char f = 'u';
    char g = 'i';
    char h = 'l';
    char i = 'i';
    char j = 'n';

    char name[] = {'w', 'a', 'n', 'g', 'g', 'u', 'i', 'l', 'i', 'n'};

    return 0;
}
```

这两种方法，表现起来显然很麻烦，这当然不是c语言的风格，为此c语言，还提供了字符串。

11.2.字符串常量

c语言提供了字符串，**但是没有提供字符串类型**。那么我们先来看一下，字符串常量，是双引号括起的任意字符序列。

```
"Hello World"
"WangGuLin"
"Please enter your full name:"
"Hello \" guilin \" "
```

11.2.1.字符串大小

```
printf( "%d",sizeof("hello world")); ?
```

结果我们看到的大小，比我们实际字面量要多一个，如何解释呢？直接看其内存：



最后的字符'\0',我们称为字符串结束字符，是系统对双引号引起的字符串**自动加设的**，而非手动干预。

11.2.2.字符串存储

既然 C 语言没有提供字符串变量，那么我们先看一看，他是如何处理字符串常量的，字符串的存储，不像其它普通类型的常量字面量的存储一样，普通类型字面量通常存储在代码段，而字符串，则存储在数据段，且是**只读数据段**。

也就是通常意义上的常量区，但是常量区这种说法，是不太精确的，也不提倡。

对于常量字符串，linux 环境中汇编结果：

```
.LC0:
4     .string "abcdefg"
5     .text
6     .globl  main
7     .type   main, @function
8 main:
9 .LFB0:
10    .cfi_startproc
11    pushq   %rbp
12    .cfi_def_cfa_offset 16
13    .cfi_offset %rbp, -16
14    movq   %rsp, %rbp
15    .cfi_def_cfa_register 6
16    movl   $12, -12(%rbp)
17    movq   $.LC0, -8(%rbp)
18    movl   $0, %eax
19    popq   %rbp
```

```

20     .cfi_def_cfa 7, 8
21     ret
22     .cfi_endproc

```

11.2.3.C 语言是如何处理字符串常量

C 语言将常量字符串，处理为一个指向数据段中一段字符串的字符指针或理解为常量字符数组。我们可以将它付给一个，char * 的指针。

```

#include <stdio.h>

int main(void)
{
    printf("sizeof(\"12345\") = %d\n", sizeof("12345"));
    printf("%s", "12345");

    char * p = "abcdefg"; //等号两侧，通常要等价的
    printf("p = %p, \"abcdefg\" = %p\n", p, "abcdefg");

    printf("\\"abcdef\[0] = %c\n", "abcdef"[0]);
    printf("\\"abcdef\[2] = %c\n", "abcdef"[2]);

    return 0;
}

```

11.3.字符串与字符数组

11.3.1.常量字符串不可更改

在上面的例子中，如果我们试图去改变其内容，则会发现并不能改变。因为，字符串常量存储在数据段的只读数据段。

```

int main(void)
{
    char *p = "abcdef";
    p[0] = 'm';

    return 0;
}

```

11.3.2.字符串与字符数组比较

如果我们想改变其里面的内容，如何办呢？把常量字符串拷贝到字符数组当中去，

字符数组和字符串，是否存在等价关系呢？

```
#include <stdio.h>

int main(void)
{
    char ca[6]; //china;

    printf("sizeof(ca) = %d sizeof(\"china\") = %d\n",
           sizeof(ca),sizeof("china"));

    printf("ca    = %p, \"china\"    = %p\n",ca, "china");
    printf("ca+1 = %p, \"china\"+1 = %p\n",ca+1, "china"+1);

    //char ca[6] == char ca  6      "china" == char "china"  '\0'

    return 0;
}
```

11.3.3.字符数组存储字符串

11.3.3.1.等价条件

经过上面的论证，字符数组跟字符串某些方面是完全等价的。要实现等价，字符数组的大小要比字符串的大小要大。

```
#include <stdio.h>
int main()
{
    char ca[6] = "china"; //    char array[] = "china";

    // "china" 会自动被拷贝到数组中去，重要是包括'\0';
    printf("\\"china\\" = %s\n", "china");
    printf("ca    = %s\n", ca);

    //    "china"[0] = 'a';

    ca[0] = 'a';

    printf("ca = %s\n", ca);

    return 0;
}
```

11.3.3.2.不等价条件

一个没有'\0'结尾的字符串，不能称为一个合格的字符串。所以说，如果字符数组的大小比字符串的大小，要小。此时，字符串会被截断，拷贝到字符数组中去。此时字符串，和被拷贝到数组中的字符串之间不存在等价关系。

```
#include <stdio.h>

int main()
{
    char ca[5] = "china";
    // "china" 会自动被拷贝到数组中去，但是'\0'没有被拷贝进去；
    printf("\\"china\\" = %s\n", "china");
    printf("ca = %s\n", ca); // 会发生越界行为。

    // "china"[0] = 'a';

    ca[0] = 'a';

    printf("ca = %s\n", ca);

    return 0;
}
```

更好的作法：利用数组可以省略大小的特点，依据数组的大小自适应。这样也会避免浪费空间。

```
#include <stdio.h>
int main()
{
    char array[] = "china";
    printf("%s\n", array);
    return 0;
}
```

11.3.4. 小结

类型	意义	表现形式
字符串常量	存储于 rodata 段，编译为指针，'\0'结尾	"abcd"
字符指针	用于指向字符串常量，但不能更改。	char*p = "abcd"
字符数组	用于存储可更改字符串的数组。注意等价关系	char str[]="china"

11.4.字符串的输入与输出

11.4.1.输出

11.4.1.1.printf

printf 的格式字符串”%s”,可以用于输出字符串，特点：遇到字符串结束标记则会停止打印输出。

```
printf("%s\n", "abcdefg");//常见第二个参数: char* /字符数组名/常量字符串
```

11.4.1.2.puts

```
int puts(const char *_Str);
//向屏幕输出，成功返回>0，失败返-1，特点，自动追加换行。
puts("");
```

11.4.2.输入

11.4.2.1.scnf

scanf 遇到空格会截止输入。但是要记得，不要输入的字符长度超过给定的空间大小。在不越界的情况下，scanf 会自动在字符串后面追加'\0'。

```
#include <stdio.h>

int main()
{
    char name[10];
    scanf("%s",name);    //测试空格和越界 "%[^\\n]s" ->fgets()
    printf("%s",name);
    return 0;
}
```

11.4.2.2.gets

gets 直到遇到回车，才停止输入，空格也作为字符输入，但是要记得，不要输入的字符长度超过给定的空间大小。在不越界的情况下，gets 会自动在字符串后面追加'\0'。

```
#include <stdio.h>

int main()
{
    char name[10];
    gets(name);    // 测试空格和越界
    puts(name);
```

```

    return 0;
}

```

11.5.字符串操作函数

前面的字符串的基础语法学习完毕，字符串作为一堆字符的有效组织，后面要学习的就是其数据组织形态。将对**字符串**的处理转化为对**字符数组**或**字符指针**的处理。

11.5.1.字符数组原生操作

当字符串被放到字符数组后，且等价条件成立后，我们就可以通过字符数组名来操作字符串，包括求长度，拷贝，追加等操作。

11.5.1.1.求字符串长度

```

int main()
{
    char array[100] = "china";
    printf("%s\n",array);

    int count = 0;
    for(int i=0; i<100; i++)
    {
        if(array[i] != '\0')
            count++;
    }
    //求字符串的大小的时候，通常不把字符\0计算在内

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

```

11.5.1.2.链接两个字符串

```

#include <stdio.h>
int main()
{
    char firstName[30] = "jim ";
    char lastName[30] = "Green";

    char *p;
    for( p = firstName; *p != '\0';p++ );
    for(char *q = lastName; *p = *q; p++,q++);

    printf("name = %s\n",firstName);
    return 0;
}

```

11.5.2.库函数操作

与字符串有关的内置函数在头文件 string.h 中定义要使用标准库字符串处理函数，程序前应该包含：

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

11.5.2.1.strlen

函数声明	<code>size_t strlen (const char * str);</code>	
所在文件	<code>string.h</code>	
函数功能	Returns the length of the C string str.	
参数及返回解析		
参数	<code>const char*</code>	c string
返回值	<code>size_t</code>	The length of string.

```
#include <stdio.h>
#include <string.h>
int main()
{
    char array[100] = "china";
    printf("%s\n",array);

    int count = strlen(array);
    printf("count = %d\n",count);
    return 0;
}
```

11.5.2.2.strcat (concatenate)

函数声明	<code>char * strcat (char * dest, const char * src);</code>	
所在文件	<code>string.h</code>	
函数功能	追加 src 串到 dest 的末尾， dest 的末尾的'\0'字符，会被 src 的第一个字符所覆盖，追加完成后的新串会被在其末尾自动追加'\0'。	
参数及返回解析		
参数	<code>char*dest</code> <code>char * src</code>	<code>dest</code> 指向被追加的串，并且有足够的空间被追加新串。 <code>src</code> 被追加的字符串。
返回值	<code>char*</code>	<code>destination is returned.</code>

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

int main()
{
    char firstName[30] = "jim ";
    char lastName[30] = "Green";

    strcat(firstName,lastName);

    printf("name = %s\n",firstName);
    return 0;
}
```

11.5.2.3 strcpy

函数声明	<code>char * strcpy (char * dest, const char * src);</code>	
所在文件	<code>string.h</code>	
函数功能	拷贝 <code>src</code> 所指向的字符串，到 <code>dest</code> 所指向空间中去，拷贝到 <code>dest</code> 的内容包含 <code>src</code> 中的结束符'\\0'。	
参数及返回解析		
参数	<code>char*dest</code> <code>char * src</code>	<code>dest</code> 指向被拷贝的空间。为了避免溢出， <code>dest</code> 所指向的空间，要有足够的空间容纳被拷贝的内容。 <code>src</code> 指向待拷贝的空间。
返回值	<code>char*</code>	<code>dest</code> is returned. 可以实现链式拷贝

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

int main(void)
{
    char dest[100] = {0};  char src[] = "china is great";

    printf("dest = %s\n",dest);
    printf("src  = %s\n",src);

    strcpy(dest, src);
}
```

```

    printf("dest = %s\n",dest);
    printf("src  = %s\n",src);

    return 0;
}

```

11.5.2.4strcmp

函数声明	<code>int strcmp (const char * str1, const char * str2);</code>	
所在文件	<code>string.h</code>	
函数功能	比较字符串 str1 和字符串 str2 的大小。该函数从两字符串的第一个字符开始，如果相等，依次往下比较，直到遇到不相同的字符或其中一个遇到'\0's。比较的依据，是两字符对应的 ASCII 值的大小。	
参数及返回解析		
参数	<code>char *str1</code> <code>char * str2</code>	<code>str1</code> 待比较的字符串 <code>str2</code> 待比较的字符串
返回值	<code>int</code>	比较的结果通过返回值来体现： 0 <code>str1 == str2</code> ; >0 <code>str1 > str2</code> ; <0 <code>str1 < str2</code>

```

#include <stdio.h>
#include <string.h>

int main(void)
{
    char str1[] = "1bcdefg";
    char str2[] = "abcdefg";

    int ret = strcmp(str1,str2);

    if(ret == 0)
        printf("str1 = str2");
    else if(ret > 0)
        printf("str1 > str2");
    else
        printf("str1 < str2");

    return 0;
}

```

11.5.2.5.练习

实现登录功能，要求输入用户名和密码，模仿登录，三次输入失败则退出程序。

```
#include <stdio.h>
#include <string.h>
int main()
{
    char username[30],pwd[30];

    int flag = 0;
    int count = 0;
    while(1)
    {
        printf("\n 请输入用户名:  ");
        gets(username);
        printf("\n 请输入密码:  ");
        gets(pwd);
        if(strcmp(username,"王桂林") == 0 &&
           strcmp(pwd,"123456") == 0)
        {
            flag = 1;
            break;
        }
        else
        {
            printf("\n 用户名和/或密码无效 \n ");
        }
        count++;
        if(count == 3)
            break;
    }

    if(flag == 1)
        printf("\n 您已成功登录 \n ");
    else
        printf("\n 请三天以后再次尝试\n");
    return 0;
}
```

11.6.字符串操作函数自实现

11.6.1.myStrlen

```
int myStrlen(const char * src)
{
    int len = 0;
    while(*src != '\0')
    {
        len++;
        src++;
    }
    return len;
}

int main(void)
{
    char str[] = "Great Wall";

    int len = myStrlen(str);
    printf("len of str = %d\n",len);

    return 0;
}
```

11.6.2.myStrcpy

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

int myStrlen(const char * src)
{
    int len = 0;
    while(*src != '\0')
    {
        len++;
        src++;
    }
    return len;
}

char * myStrcpy(char *dest, char * src)
{
    char * retDest = dest;
```

```

while((*dest = *src) != '\0')
{
    dest++;
    src++;
}

//while(*dest++ = *src++); //可以再次优化

return retDest;

}

int main(void)
{
    char dest[100]; char src[100] = "Panda";
    char anotherDest[100];

//    myStrcpy(dest,src);
myStrcpy(anotherDest,myStrcpy(dest,src)); //连续拷贝
printf("dest = %s\n",dest);
printf("anotherDest = %s\n",anotherDest);

return 0;
}

```

11.7.多文件编程

注重实现方法论

11.7.1.函数声明(.h)

```

#ifndef __MYSTR_H__
#define __MYSTR_H__ //避免头文件包含

char * myStrcpy(char *dest, char * src);
int myStrlen(const char * src);

#endif

```

11.7.2.函数实现(.c)

```
#include "mystr.h" //自包含
```

```
int myStrlen(const char * src)
{
    int len = 0;
    while(*src != '\0')
    {
        len++;
        src++;
    }
    return len;
}

char * myStrcpy(char *dest, char * src)
{
    char * retDest = dest;

    while((*dest = *src) != '\0')
    {
        dest++;
        src++;
    }
    return retDest;
}
```

11.7.3.头文件包含#include

```
#include <stdio.h>

#include "mystr.h" //谁用谁包含

int main(void)
{
    char dest[100]; char src[100] = "Panda";
    char anotherDest[100];

//    myStrcpy(dest,src);
    myStrcpy(anotherDest,myStrcpy(dest,src)); //连续拷贝
    printf("dest = %s\n",dest);
    printf("anotherDest = %s\n",anotherDest);

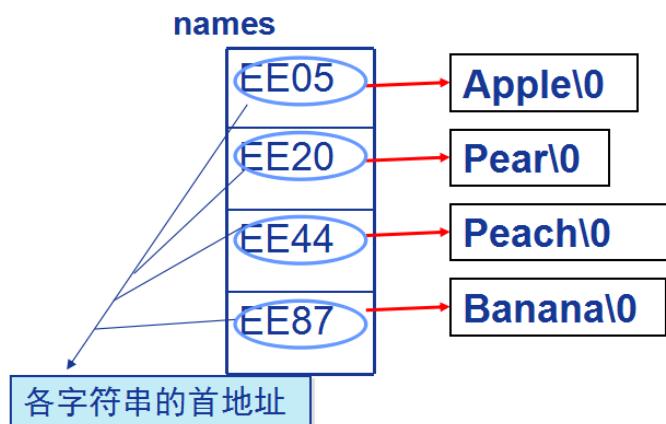
    return 0;
}
```

11.8.指针数组

11.8.1.定义

一个数组中的各个元素都是字符指针，我们称该数组为字符指针数组，或是指针数组。

比如：char *name[] 就是一个指针数组。



```
//指针数组，指的是，数组中的成员，均是指针。通常意义上的指针数组，指的是字符指针数组。
int main()
{
    //int aa, bb, cc, dd;
    //int arr[] = {aa, bb, cc, dd}; // {int,int ,int ,int}

    //char a, b, c, d;
    //char* pArr[4] = {&a, &b, &c, &d}; //{char*, char *, char*, char*}

    char *names [] = {"Apple", "Pear", "Peach", "Banana"};

    for (int i = 0; i < sizeof(names) / sizeof(*names); i++)
    {
        printf("%s\n", names[i]);
    }
    return 0;
}
```

11.8.2.应用

11.8.2.1.输出

```
#include <stdio.h>

int main()
{
    char *names[] = {
        "Apple",
        "Banana",
        "Pineapple",
        "Peach",
        "Strawberry",
        "Grapes"
    };

    for(int i=0; i<sizeof(names)/sizeof(names[0]); i++)
    {
        printf("%s\n", names[i]);
    }
}
```

11.8.2.2.排序

```
#include <stdio.h>

int main()
{
    char *names[] = {
        "Apple",
        "Banana",
        "Pineapple",
        "Peach",
        "Strawberry",
        "Grapes"
    };

    char *temp;
    printf("\n %s %s", names[2], names[3]);
    temp = names[2];
    names[2] = names[3];
    names[3] = temp;
    printf("\n %s %s", names[2], names[3]);
}
```

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

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

int main()
{
    char *names[] = {
        "Pineapple",
        "Peach",
        "Strawberry",
        "Grapes",
        "Apple",
        "Banana"
    };
    int n = (sizeof(names)/sizeof(names[0]));
    for(int i=0; i<n-1; i++)
    {
        char * tmp = NULL;
        for(int j=0; j<n-1-i; j++)
        {
            if(strcmp(names[j],names[j+1])>0)
            {
                tmp = names[j];
                names[j] = names[j+1];
                names[j+1] = tmp;
            }
        }
    }

    for(int i=0; i<sizeof(names)/sizeof(names[0]); i++)
    {
        printf("%s\n",names[i]);
    }
}
```

11.8.2.3.argv

11.8.3.思考

如何让指针数组所指向的空间中的内容可以更改呢？

11.9.你所追过的那些“零”

11.9.1.零所代表的意义

零值	意义	备注
0	整型数据的0	
0.0	实型数据的0	
NULL	指针型数据的0	内存的0地址，用于初始化暂不用指针。或是标识已经被释放堆内存的指针。
'\0'	转义字符型(ASCII值为0)	非可打印字符，用于标识字符串结束标记。
'0'	字符型(ASCII值为48)	可打印字符，用于打印字符0
"0"	字符串型	包含两个ASCII值48和0

11.9.2.实例分析

```

int main()
{
    int a = 0;
    float f = 0.0;

    char *p = NULL;
    char ch = '\0';

    printf("整型的 0      = %d\n", a);
    printf("浮点型的 0.0   = %f\n", f);
    printf("指针的 NULL     = %d\n", p);
    printf("ASCII 零 \'\\0\' = %d\n", ch);

    printf("字符的零\'0\' = %d\n", '0');

    printf("字符串的零\"0\" = %d %d\n", "0"[0], "0"[1]);

    return 0;
}

```

11.10.作业

11.10.1. 自实现 myStrcat

仿标准库字符串函数 strcat,自实现 char *myStrcat(char * dest, char * src);

11.10.2. 自实现 mystrcmp

仿标准库字符串函数 strcmp,自实现 int mystrcmp(char * str1, char * str2);

11.10.2.1. 基础版本

```
int mystrcmp(char * s1, char *s2)
{
    while (*s1 != '\0' && *s2 != '\0')
    {
        if (*s1 == *s2){
            s1++; s2++;
        }
        else if (*s1 > *s2){
            return 1;           //return *s1 -*s2;
        }
        else{
            return -1;         //return *s1 -*s2;
        }
    }
    if (*s1 == '\0'&& *s2 != '\0') {
        return -1;           //return *s1 -*s2;
    }
    else if (*s1 != '\0'&& *s2 == '\0'){
        return 1;           //return *s1 -*s2;
    }
    else{
        return 0;           //return *s1 -*s2;
    }
}
```

11.10.2.2. 优化版本

```
int mystrcmp(char *s1, char *s2)
{
    for(;*s1&&*s2;s1++,s2++)
    {
        if(*s1 != *s2)
            break;
    }
```

```
    return *s1-*s2;
}

//是否可以继续优化?
```

11.10.3. 以下代码能打印什么？

```
#include <stdio.h>
int main()
{
    char src[9] = "xxxxoooo"; //越界重叠问题
    char dest[4];

    strcpy(dest,src);

    printf("%s\n",src);
    printf("%s\n",dest);

    return 0;
}
```

12. 内存管理(Memory)

12.1. 进程空间

12.1.1. 源程序/程序/进程

程序，是经源码编译后的可执行文件，可执行文件可以多次被执行，比如我们可以多次打开 office。

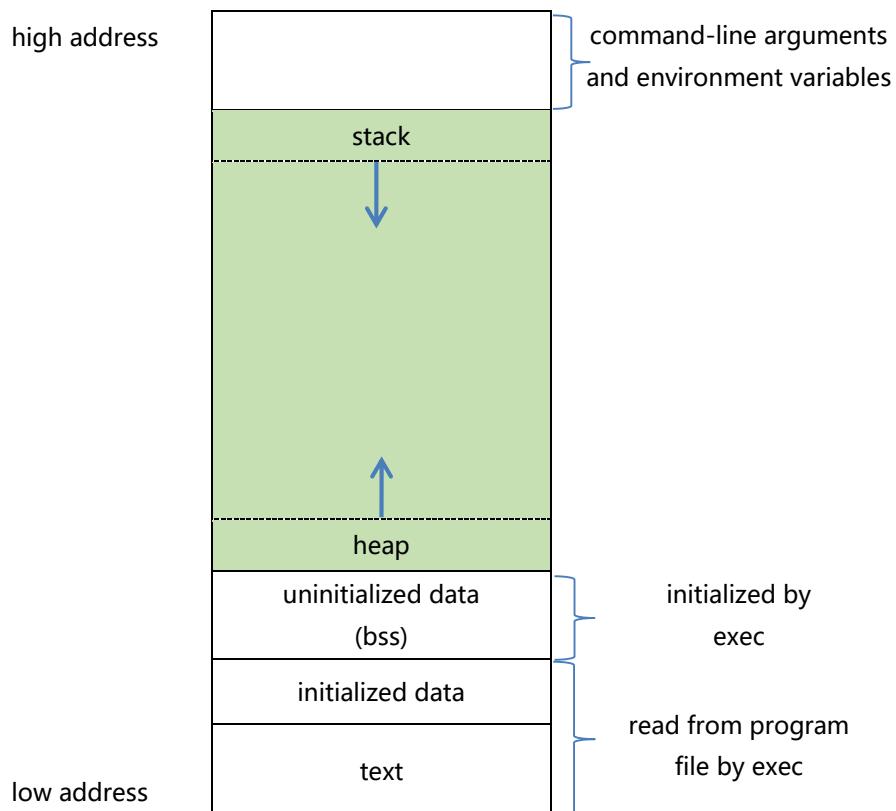
而进程，是程序加载到内存后开始执行，至执行结束，这样一段时间概念，多次打开的 wps，每打开一次都是一个进程，当我们每关闭一个 office，则表示该进程结束。

程序是静态概念，而进程动态/时间概念。

12.1.2. 进程空间图示

有了进程和程序的概念以后，我们再来看一下，程序被加载到内存以后，0-4G 的内存空间。这 0-4G 内存中的布局是什么样子的呢？

如下图：



12.2.栈内存(Stack)

12.2.1.栈存储的特点

栈中存放任意类型的变量，但必须是 auto 类型修饰的，即自动类型的局部变量，
随用随开，用完即消。

内存的分配和销毁系统自动完成，不需要人工干预。

12.2.2.栈大小

栈的大小并不大，他的意义并不在于存储大数据，而在于数据交换。

```
[root@localhost ~]# ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority      (-e) 0
file size                (blocks, -f) unlimited
pending signals          (-i) 16384
max locked memory        (kbytes, -l) 32
max memory size         (kbytes, -m) unlimited
open files               (-n) 1024
pipe size                 (512 bytes, -p) 8
POSIX message queues    (bytes, -q) 819200
real-time priority       (-r) 0
stack size              (kbytes, -s) 10240          //10M
cpu time                  (seconds, -t) unlimited
max user processes        (-u) 16384
virtual memory            (kbytes, -v) unlimited
file locks                (-x) unlimited
```

12.2.3.常见栈溢出案例

局部变量过多，过大 或 递归层数太多。

```
void func(int n)  //递归
{
    if (n > 0)
        func(--n);
    else
        return;
}
int main()
{
    //char array[1024*1024*1024]; //局部变量过多，过大。
    //memcpy(array, "abcd",5);     //centos 避免优化
    func(1000000);             //递归层数太多
```

```
    return 0;  
}
```

12.3.堆内存(Heap)

12.3.1.堆存储的特点

堆内存可以存放任意类型的数据，但需要自己申请与释放。

12.3.2.堆大小

堆大小，想像中的无穷大，对于栈来说，**大空间申请，唯此，无它耳**。但实际使用中，受限于实际内存的大小和内存是否连续性。

测试申请大空间

```
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
  
int main(void)  
{  
    int * p = (int*)malloc(1024*1024*1024); //1G 完全无压力  
    if(p == NULL)  
    {  
        printf("malloc error\n");  
        return -1;  
    }  
  
    int *q = (int*)malloc((unsigned int)-1);  
    //无力归天 42 亿个字节 避免整型溢出  
    if(q == NULL)  
    {  
        printf("malloc error\n");  
        return -1;  
    }  
  
    return 0;  
}
```

12.3.3.堆内存的申请与释放

12.3.3.1.malloc

功能介绍：

函数声明	void * malloc(size_t _Size);	
所在文件	stdlib.h	
函数功能	申请堆内存空间并返回，所申请的空间并未初始化。 常见的初始化方法是 <code>memset</code> 字节初始化。	
参数及返回解析		
参数	<code>size_t</code>	<code>_size</code> 表示要申请的字符数
返回值	<code>void *</code>	成功返回非空指针指向申请的空间， 失败返回 <code>NULL</code>

代码演示：

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

int main(void)
{
    //申请基本数据类型和数组，栈堆空间作对比。
    int a; int *p = &a;
    a = 100; printf("*p = %d\n",a);

    int *pm = (int*)malloc(sizeof(int));
    if(pm == NULL) return -1;

    *pm = 100;
    printf("*pm = %d\n",*pm);

    //申请基本数据类型和数组，栈堆空间作对比。
    int array[10]; int *pa = array;

    pm = (int*)malloc(10*sizeof(int));
    //memset(pm,0,10*sizeof(int));
    //memset(pm,0,10*sizeof(int)); 此时会导致什么样的结果呢？
    for(int i=0; i<10; i++)
    {
        printf("%d\n",pm[i]);
    }

    free(pm);

    return 0;
}
```

12.3.3.2.calloc

函数声明	void *calloc(size_t nmemb, size_t size);	
所在文件	stdlib.h	
函数功能	申请堆内存空间并返回，所申请的空间，自动清零。	
参数及返回解析		
参数	size_t	nmemb 所需内存单元数量
	size_t	size 内存单元字节数量
返回值	void *	成功返回非空指针指向申请的空间，失败返回 NULL

代码演示：

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

int main(void)
{
    int * array = (int*)calloc(10,sizeof(int));

    for(int i=0; i<10; i++)
    {
        printf("%d\n",array[i]); //已被初始化。
    }

    return 0;
}
```

12.3.3.3.realloc

函数声明	void *realloc(void *ptr, size_t size);	
所在文件	stdlib.h	
函数功能	扩容(缩小)原有内存的大小。通常用于扩容，缩小会导致内存缩去的部分数据丢失。	
参数及返回解析		
参数	void *	ptr 表示待扩容(缩小)的指针，ptr 为之前用 malloc 或者 calloc 分配的内存地址。或 ptr==NULL，则该函数等同于 malloc。
	size_t	size 表示扩容(缩小)后内存的大小。
返回值	void*	成功返回非空指针指向申请的空间，失败返回 NULL。 返回的指针，可能与 ptr 的值相同，也有可能不同。 若相同，则说明在原空间后面申请，否则，则可能后续空间不足，重新申请的新的连续空间，原数据拷贝到新空间，原有空间自动释放。

代码演示：

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

int main(void)
{
    int * array = (int*)calloc(10,sizeof(int));
    int * newArray = realloc(array,80);
    //array = realloc(array,80);
    if(newArray == NULL)
    {
        printf("realloc 失败\n");
        return -1;
    }
    for(int i=0; i<20; i++)
    {
        printf("%d\n",newArray[i]);
    }

    return 0;
}
```

12.3.3.4.free

函数声明	void free(void *p);	
所在文件	stdlib.h	
函数功能	释放申请的堆内存	
参数及返回解析		
参数	void*	p 指向手动申请的空间
返回值	void	无返回

代码演示：

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

int main(void)
{
    int * array = (int*)calloc(10,sizeof(int));
    free(array);
    return 0;
}
```

```
}
```

思考：

在释放所申请的空间时，我们只传递了指针到 free 中去，free 是如何知道释放多少个字节的呢？

12.3.4. 应用模型

12.3.4.1. 动态数组

以前我们学过，VLA 变长数组，在运行时，只有一次初始化其长度的机会，且容易发生，栈溢出。现在有了堆空间的动态申请与释放，动态数组，就成了真正意义上的动态数组了。

```
#include <stdio.h>
int main()
{
    int len;
    printf("pls new len:");
    scanf("%d",&len);

    int *p = (int*)realloc(NULL,sizeof(int)*len);
    for(int i=0; i<len; i++)
    {
        printf("%d\n",p[i]);
    }
    //加大
    printf("pls new len:");
    scanf("%d",&len);

    p = (int*)realloc(p,sizeof(int)*len);
    for(int i=0; i<len; i++)
    {
        printf("%d\n",p[i]);
    }
    //变小
    printf("pls new len:");
    scanf("%d",&len);

    p = (int*)realloc(p,sizeof(int)*len);
    for(int i=0; i<len; i++)
    {
        printf("%d\n",p[i]);
    }
    free(p);
    return 0;
```

```
}
```

12.3.5. 常见错误案例剖析

以下三个模型，也是针对，堆内存的使用特点而设计的。即：

- 返回判空。
- 配对使用。
- 自申请，自释放。

12.3.5.1. 置空与判空

堆内存使用的逻辑是这样的，**申请，判空，使用/释放(配对)，置空**。常见错误之一就是释放以后置未置为 NULL 再次作判空使用 或 释放以后继续非法使用。

```

char*p=(char*)malloc(100);
strcpy(p,"hello");
free(p);/*p 所指的内存被释放，但是 p 所指的地址仍然不变*/
//p = NULL;忘了此句，后而又用到了
.....
if(NULL!=p)
{
    /*没有起到防错作用*/
    strcpy(p,"hello");      /*出错*/
}

```

12.3.5.2. 重复申请

在服务器模型中，常用到大循环，在大循环中未释放原有空间，重新申请新空间，造成原有空间，内存泄漏。

```

while (1)
{
    char *p = malloc(1000);
    printf("xxxxxx\n");
    printf("xxxxxx\n");
    printf("xxxxxx\n");
    printf("xxxxxx\n");
    p = malloc(1000);      // 中途可能忘了，重复申请，内存泄漏
    free(p);
    printf("xxxxxx\n");
    Sleep(10);
}

```

12.3.5.3.谁申请谁释放模型（并非绝对）

如果没有协同的原则，则有可能会造成，重复释放。

```
void func(char *p)
{
    strcpy(p, "American");
    printf("%s\n", p);
    free(p);           //此处违反了，，谁申请谁释放的原则。
}
int main()
{
    char * p = malloc(100);
    func(p);
    free(p);
    return 0;
}
```

12.3.5.4.其它常见错误

- free 非 alloc 函数申请赋值的指针
- free(p+i);

12.3.6.VLD 工具使用

12.3.6.1.下载

12.3.6.2.安装

12.3.6.3.使用

12.4.开放的地址空间

传对象的地址到不同的作用域，可依据地址修改地址所指向对象的内容，根本原因是地址空间是开放的。

```
#include <stdio.h>
int foo(){
    int *p =(int *) 0x12345678;
    *p = 200;
}
void func(int *p){
    *p = 400;
}

int main()
{
    int *p =(int *) 0x12345678;
```

```
*p = 200;

int a;
func(&a);
*(&a) = 400;
return 0;
}
```

12.5.堆与栈空间的返回

12.5.1. 栈空间不可以返回

结论正确无疑，虽然有些平台并不报错。

```
#include <stdio.h>

//1 数值是可以返回的
//2 地址也是可以返回
//3 栈上的空间不可以返回， 原因，随用随开，用完即消
//4 堆上的空间，是可以返回的

int func()
{
    int a = 500;
    return a;
}

int* foo()
{
    int a = 500;
    int *pa = &a;
    printf("&a = %p\n", pa);
    return pa;
}

int *func2()
{
    int arr[100];
    return arr;
}

int main(int argc, char *argv[])
{
    int a = func();
    printf("a = %d\n", a);
```

```
int *pa = foo();
printf("pa = %p\n", pa);

printf("%d\n", *pa);
*pa = 300;

return 0;
}
```

12.5.2.堆空间可以返回

```
#include <stdio.h>

//1 数值是可以返回的
//2 地址也是可以返回
//3 栈上的空间不可以返回， 原因，随用随开，用完即消
//4 堆上的空间，是可以返回的


char * getFormatMem(int size,char content)
{
    char *p = (char*)malloc(size *sizeof(char));
    if(NULL == p)
        exit(-1);
    memset(p,content,size *sizeof(char)-1);
    p[size *sizeof(char)-1] = '\0';
    return p;
}
int main()
{
    char *p = getFormatMem(100,'a');
    printf("p = %s\n",p);
    free(p);
    return 0;
}
```

13.结构体(Struct)

从某种程度上来说，会不会用 struct，怎样用 struct 是区别一个开发人员是否具备丰富开发经历的标志！

13.1.引例

问题：存储一个班级中 4 名学员的信息（学号、姓名、性别和成绩）。

方案一：分别建立 4 个数组， int num[4]; char*name[4]; char sex[4]; float score[4];

1001	1002	1003	1004
“zhangsan”	“lisi”	“wangwu”	“zhaoliu”
‘x’	‘m’	‘x’	‘x’
100	89	76	65

点评：功能可实现，但是**建立数组间关联关系**，比较麻烦。

方案二：建立一个二维数组

1001	“zhangsan”	‘x’	100
1002	“lisi”	‘m’	89
1003	“wangwu”	‘x’	76
1004	“zhaoliu”	‘m’	65

点评：想法很好，然并卵，数组中要求元素，类型必须是一致（空间等大）的。

方案三：思考方案二中，一个二维数组，要求每一个元素都是一样的，所以才导致方案不可成行。但是如果，我们把其中一行，看成是一个整体的话。那就是一个一维数组了。同样满足了每个元素都是同一类型的话。

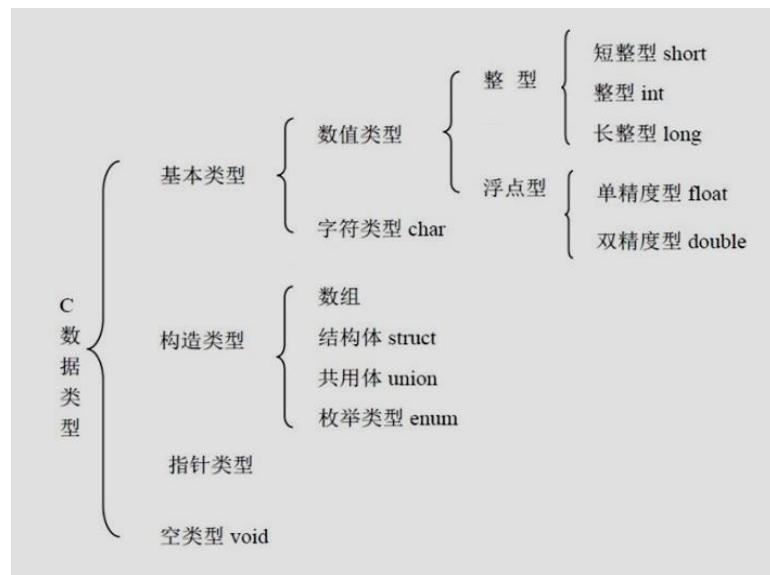
1001	“zhangsan”	‘x’	100
1002	“lisi”	‘m’	89
1003	“wangwu”	‘x’	76
1004	“zhaoliu”	‘m’	65

二维数组，是数组中每个元素又是一维数组，一维数组是一种构造类型，上表格其实也是一种二维的结构，只不过，二维中嵌套的不是一维数组这种构造类型，而是**构造体**这种构造类型而已。

13.2.为什么要引入结构体

《道德经》曰：“一生二，二生三，三生万物”。也就是说先有了一，二，然后有了三，然后三诞生了万物。

13.2.1.开放类型定义



C语言先有了基本类型，当需要表达多个相同变量在一起的类型时(N:1)，c语言提供了数组类型，当需要表达多个不同变量在一起的类型时(N:M)，此时C语言已经无法给出一一定义了。

c语言无法满足你所定义变量的需求时，放开了自定义权限，于是**struct**关键字就应运而生了。

struct 的出现，然后就拥有了几乎所有的自定义类型。

13.2.2.从单变量->数组->结构体

从描述一个人年龄，到描述一堆人的年龄，到描述一个人的状态。从单变量到数组再到结构体的需求转变。

```

#include <stdio.h>

int main(void)
{
    //描述一个人的年龄
    int age;

    //描述一堆人的年龄
    int ageArr[50];    //方便了管理和书写
  
```

```
//描述一个人的状态
struct Student      //生成学生类型    地位等同于 int
{
    char name[100]; //结构体成员名
    char sex;
    int age;
    float score;
};

struct Student st;

printf("%s\n",st.name);
printf("%c\n",st.sex);
printf("%d\n",st.age);

//用 struct 描述 构造类型 数组

struct Array
{
    int a; int b; int c; int d; int e; int f;
};

struct Array arr;
printf("%d %d %d %d %d\n",arr.a,arr.b,arr.c,arr.d,arr.e);

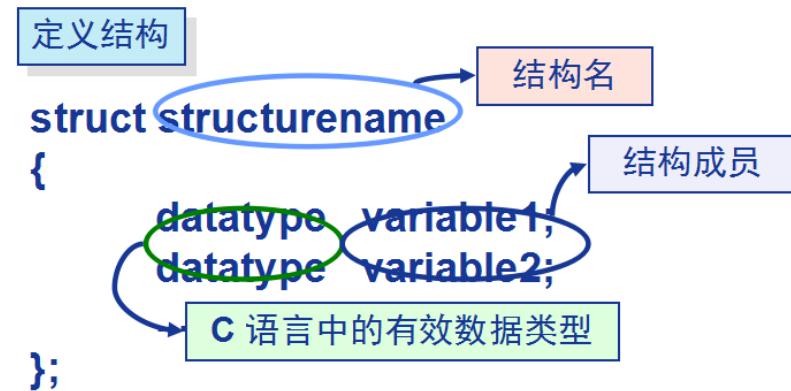
//数组成员不需要区分类型，成员大小一致，所以才用首地址加偏移来表示。
//但是对于结构来说，类型大小均不统一，所以采用的成员运算符来进行访问。
//成员名称，区分不同成员的标识。

return 0;
}
```

13.3.结构体类型定义

如何设计一种新类型，把不同的元素打包在一起就是下面我们要学习的东西了。

struct 是构造新类型的关键字，有了它，就可以构造任意的构造类型了。



13.3.1. 无名构造类型

不带来多余的类型名。若不在定义类型时定义变量，则会带来代码冗余。

```

#include <stdio.h>

struct
{
    char name[30];
    char sex;
    int age;
    float high;
}stu,stu2,stu3;

struct
{
    char name[30];
    char sex;
    int age;
    float high;
}stu4;

int main(void)
{
    struct
    {
        char name[30];
        char sex;
        int age;
        float high;
    }stu6;
    printf("%s\n",stu6.name);
}
  
```

```
    return 0;
}
```

13.3.2.有名构造类型

一处定义，可以多处使用。

```
struct student
{
    char name[30];
    char sex;
    int age;
    float high;

}stu;
struct student stu2;
```

13.3.3.别名构造体类型

更好的使用结构体。

```
typedef struct student
{
    char name[30];
    char sex;
    int age;
    float high;

}STUDENT;
STUDENT stu, stu2;
```

13.3.4.小结

- 我们定义的新类型，它的地位等同于 **int** 类型。还只是个模子，如果没有生成变量的话，是不会占用空间的。
- 结构定义放置在程序的开始部分，位于头文件声明之后。
- 注意{}不表示复合语句，其后有分号。
- 结构体类型名称是 **struct+结构体名**，注意 **struct** 关键字不能省略。

13.4.结构体变量初始化及成员访问

凡是构造类型，要么定义的时候同时初始化。要么先定义成员再分别初始化(赋值)，此不同于基础数据类型，数组亦是如此。

初始化是一种特殊的语法 跟赋值不等价。

13.4.1. 初始化及访问

13.4.1.1. 点成员运算符 (.)

```
#include <stdio.h>
#include <string.h>
typedef struct student
{
    char name[30];
    char sex;
    int age;
    float high;
}STUDENT;

int main(void)
{
    STUDENT stu = {"zhangsa",'x',45,170};
//    stu = {"zhangsa",'x',45,170};

    printf("name = %s, sex = %c ,age = %d ,high = %f\n",
           stu.name,stu.sex,stu.age,stu.high);
    strcpy(stu.name,"zhaosi");
    stu.sex = 'x';
    stu.age = 19;
    stu.high = 165;
    printf("name = %s, sex = %c ,age = %d ,high = %f\n",
           stu.name,stu.sex,stu.age,stu.high);

    printf("请输入姓名:");
    scanf("%s",stu.name);
    getchar();           //%c 前吸收一个回车符
    printf("请输入性别:");
    scanf("%c",&stu.sex);
    printf("请输入年龄:");
    scanf("%d",&stu.age);

    printf("请输入身高:");
    scanf("%f",&stu.high);

    printf("name = %s, sex = %c ,age = %d ,high = %f\n",
           stu.name,stu.sex,stu.age,stu.high);

    return 0;
}
```

13.4.1.2.指向成员运算符(-> (*).)

常见有指向栈内存变量或指向堆内存变量的表示方法。

```
#include <stdio.h>
#include <string.h>
typedef struct student
{
    char name[30];
    char sex;
    int age;
    float high;
}STUDENT;

int main(void)
{
    STUDENT stu = {"zhangsa",'x',45,170};
//    stu = {"zhangsa",'x',45,170};

    STUDENT * ps = &stu;           //栈内存的指针表示法
    printf("name = %s, sex = %c ,age = %d ,high = %f\n",
           ps->name,ps->sex,ps->age,ps->high);
    strcpy(ps->name,"zhaosi");
    ps->sex = 'x';
    ps->age = 19;
    ps->high = 165;
    printf("name = %s, sex = %c ,age = %d ,high = %f\n",
           ps->name,ps->sex,ps->age,ps->high);

    printf("请输入姓名:");
    scanf("%s",ps->name);
    getchar();
    printf("请输入性别:");
    scanf("%c",&ps->sex);
    printf("请输入年龄:");
    scanf("%d",&ps->age);

    printf("请输入身高:");
    scanf("%f",&ps->high);

    printf("name = %s, sex = %c ,age = %d ,high = %f\n",
           ps->name,ps->sex,ps->age,ps->high);

    ps = (STUDENT*)malloc(sizeof(STUDENT)); //堆内存的指针表示法
```

```

printf("请输入姓名:");
scanf("%s", ps->name);
getchar();
printf("请输入性别:");
scanf("%c", &ps->sex);
printf("请输入年龄:");
scanf("%d", &ps->age);

printf("请输入身高:");
scanf("%f", &ps->high);

printf("name = %s, sex = %c ,age = %d ,high = %f\n",
       ps->name,ps->sex,ps->age,ps->high);

printf("name = %s, sex = %c ,age = %d ,high = %f\n",
       (*p).sname,(*p).sex,(*p).age,(*p).shigh);

free(ps);
return 0;
}

```

13.4.1.3.其它初始化方法(重要)

详见博客：http://blog.csdn.net/linux_wgl/article/details/51924191

13.4.2.成员运算符本质

成员运算符的本质，依然是通过**计算偏移**来实现的。此处**不考虑内存对齐**等原因。

```

#include <stdio.h>

struct TYPE
{
    int a;
    int b;
    long long c;
};

int main()
{
    struct TYPE x;
    x.a = 30;
    x.b = 'a';
    x.c = 500;
}

```

```
    return 0;
}
```

汇编如下：

```
13 [1]    x.a = 30;
<+0x000e>    c7 04 24 1e 00 00 00    movl $0x1e,(%esp)
14 [1]    x.b = 'a';
<+0x0015>    c7 44 24 04 61 00 00 00    movl $0x61,0x4(%esp)
15 [1]    x.c = 500;
<+0x001d>    c7 44 24 08 f4 01 00 00    movl $0x1f4,0x8(%esp)
<+0x0025>    c7 44 24 0c 00 00 00 00    movl $0x0,0xc(%esp)
16 [1]    return 0;
0x40163d <+0x002d>      b8 00 00 00 00    mov $0x0,%eax
17 [1]}
```

13.4.3. 赋值

同类变量间，可以赋值。常用于**传参和返回**。

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

typedef struct student
{
    char name[30];
    char sex;
    int age;
    float high;
}STUDENT;

int main(void)
{
    STUDENT stu = {"zhangsan",'x',45,170};

    STUDENT stu2 = stu;

    printf("姓名: %s, 性别: %c, 年龄: %d, 身高 : %f\n",
           stu2.name,stu2.sex,stu2.age,stu2.high);
    return 0;
}
```

实现数组赋值

```
#include <stdio.h>
#include <string.h>
struct ARRAY
{
    int a[10];
};

int main(void)
{
    struct ARRAY array = { {1,2,3,4,5,6,7} }; //内部括号可省
    struct ARRAY array2 = array; //神奇

    for(int i=0; i<10; i++)
    {
        printf("%d\n",array2.a[i]);
    }

    return 0;
}
```

13.5.结构体类型作参数和返回值

13.5.1.结构体变量作参数和返回值

结构体作参数或返回值，会发生同类型复制（本质是赋值）。同类型结构体间，是可以相互赋值的。

```
struct complex
{
    float real;
    float imag;
};

struct complex addComplex(struct complex com1, struct complex com2)
{
    struct complex res;
    res.real = com1.real + com2.real;
    res.imag = com1.imag + com2.imag;
    return res;
}

int main()
{
    struct complex com1 = { 1.2f, 2.3f };
    struct complex com2 = { 2.3f, 2.4f };
```

```

    struct complex res = addComplex(com1, com2);
    printf("res real = %f ,imag = %f\n", res.real, res.imag);
    return 0;
}

```

13.5.2. 结构体指针作参数

传结构体的成本是很高的，用指针作参数有一个好处，就是避免了同类型复制，无论结构体多大，只传 4 个字节的指针。

```

#include <stdio.h>
#include <stdlib.h>
struct complex
{
    float real;
    float imag;
};
struct complex addComplex(struct complex *pcom1, struct complex *pcom2)
{
    struct complex res;
    res.real = pcom1->real + pcom2->real;
    res.imag = pcom1->imag + pcom2->imag;
    return res;
}

int main()
{
    struct complex com1 = { 1.2f, 2.3f };
    struct complex com2 = { 2.3f, 2.4f };
    struct complex res = addComplex(&com1, &com2);
    printf("res real = %f ,imag = %f\n", res.real, res.imag);
    return 0;
}

```

13.5.3. 获取当前时间函数的使用

可以用系统提供的函数 localtime，来求得当前日期的精确值。函数的具体声明可以查阅手册可得。

```

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

int main()
{

```

```

time_t rawTime;
time(&rawTime); //指针作入参

struct tm * st = localtime(&rawTime); //返回了一个静态变量的地址

printf("current year    = %d\n",st->tm_year+1900);
printf("current month   = %d\n",st->tm_mon+1);
printf("current day     = %d\n",st->tm_mday);
printf("current hour    = %d\n",st->tm_hour);
printf("current minute  = %d\n",st->tm_min);
printf("current second  = %d\n",st->tm_sec);

return 0;
}

```

栈上的空间，不能通过，函数返回。因为函数一结束，栈空间已经消失，或已被再分配。再对其空间进行访问，是危险且无意义的。

localtime 只所以可以返回一个指针所指向的空间，是因为 localtime 内部有一个静态的 struct tm 结构变量。

13.6.结构体数组

下面，我们来解决引例中存储一个班级中 4 名学员的信息（学号、姓名、性别和成绩）。

1001	“zhangsan”	‘x’	100
1002	“lisi”	‘m’	89
1003	“wangwu”	‘x’	76
1004	“zhaoliu”	‘m’	65

13.6.1.定义及初始化

```

#include <stdio.h>
#include <string.h>
typedef struct student
{
    int num;
    char name[30];
    char sex;
    float score;
}STUDENT;

```

```

int main(void)
{
    STUDENT stu[4] = {
        {1001,"zhangsan",'x',100},
        {1002,"lisi",'x',89},
        {1003,"wangwu",'m',76},
        {1004,"zhaoliu",'x',65},
    };

    for(int i=0; i<sizeof(stu)/sizeof(stu[0]); i++)
    {
        printf("学号: %d, 姓名:%s, 性别: %c, 成绩: %f\n",
               stu[i].num,stu[i].name,stu[i].sex,stu[i].score);
    }

    printf("*****\n");
//    stu[0] = {1004,"zhaoliu",'x',65};
    stu[0].num = 1008;
    strcpy(stu[0].name,"liuneng");
    stu[0].sex = 'y';
    stu[0].score = 10000;

    for(int i=0; i<sizeof(stu)/sizeof(stu[0]); i++)
    {
        printf("学号: %d, 姓名:%s, 性别: %c, 成绩: %f\n",
               stu[i].num,stu[i].name,stu[i].sex,stu[i].score);
    }
    return 0;
}

```

13.6.2. 内存存储形式



89
‘m’
“lisi”
1002
100
‘x’
“zhangsan”
1001

13.6.3. 实战之选举

现有三位候选人员，候选人包含名字和选票数两项，现在 10 人对其投票，每个人限投票一次，投票完毕，打印投票结果，如有名字打错，算作弃权处理。

```
#include <stdio.h>

typedef struct _candidate
{
    char name[30];
    int voteCount;
}Candidate;

void disCandidateRes( Candidate* c,int n,int count)
{
    for(int i=0; i<n; i++)
    {
        printf("Name:%10s  VoteCount: %2d\n",c[i].name,c[i].voteCount);
    }
    printf("弃权的人数:%d",count);

    int idx = 0;
    for(int i=0;i<n; i++)
    {
        if(c[i].voteCount > c[idx].voteCount)
            idx = i;
    }
    printf("恭喜%s 获得了本次选举\n",c[idx].name);
}
```

```
int main(int argc, char *argv[])
{
    Candidate can[3] =
    {
        {"zhangsan",0},
        {"lisi",0},
        {"wangwu",0}
    };

    char buf[1024];
    //10个人，投10次票
    int count =0;
    for(int i=0; i<10; i++)
    {
        printf("pls input your like:");
        scanf("%s",buf);
        int flag = 1;
        for(int i=0; i<3; i++)
        {
            if(!strcmp(buf,can[i].name))
            {
                can[i].voteCount++;
                flag = 0;
            }
        }
        if(flag != 0)
        {
            count++;
        }
    }
    disCandidateRes(can,3,count);

    return 0;
}
```

13.7.结构体嵌套

13.7.1.结构体中可以嵌套结构体

结构体中，嵌套结构体，称为结构体嵌套。结构体中，既可以嵌套结构体类型变量，
也可以嵌套结构体类型，后一种方式不推荐。

```
struct student
```

```
{
    char name[30];
    char sex;
    int age;
    float high;
    struct birthday
    {
        int year;
        int month;
        int day;
    }birth;      // 如果未声明 birth 成员，则类似只有 int
};
```

13.7.2. 嵌套结构体变量定义和初始化

```
int main()
{
    struct student stu = { "wang", 's', 23, 175.0, { 1988, 8, 18 } };
    printf("name %s birthday year = %d", stu.name, stu.birth.year);
    return 0;
}
```

13.8. 结构体类型的大小

按照以往的惯例，无论是基础类型，还是学习的第一种构造类型数组，我们都是先求其类型的(模子)大小，**为首当其冲**，结构体类型的大小，却放到了最后，因为它涉及到**内存对齐**。

当然了，结构体类型，本身不占有内存空间，只有它生成的变量才占有内存空间的。

13.8.1. 结构体成员内存分布

首成员在低地址，尾成员在高地址。

```
struct Stu
{
    char sex;          //1
    int age;           //4
};

int main()
{
    struct Stu stu;
    printf("struct stu = %d\n", sizeof(struct Stu));
    printf("&stu = %p\n", &stu);
    printf("&stu.sex = %p\n", &stu.sex);
```

```

    printf("&stu.age = %p\n", &stu.age);
    return 0;
}

```

13.8.2. 内存对齐

13.8.2.1. 什么是不对齐

一个成员变量需要多个机器周期去读的现象，称为内存不对齐。为什么要对齐呢？本质是牺牲空间，换取时间的方法。

13.8.2.2. 对齐规则

x86(linux 默认#pragma pack(4), window 默认#pragma pack(8))。linux 最大支持 4 字节对齐。

方法：

- ①取 **pack(n)** 的值（**n=1 2 4 8--**），取结构体中类型最大值 **m**。两者取小即为**对外对齐大小 Y=(m<n?m:n)**。
- ②将每一个结构体的成员大小与 **Y** 比较取小者为 **X**，作为**内对齐大小**。
- ③所谓按 **X** 对齐，即为地址（设起始地址为 **0**）能被 **X** 整除的地方开始存放数据。
- ④外部对齐原则是依据 **Y** 的值（**Y** 的最小整数倍），进行补空操作。

```

#include <stdio.h>
#pragma pack(4)

struct stu
{
    char a;           //1
    short b;          //2
    int c;            //4
    char d;           //1
};

int main()
{
    printf("%d\n", sizeof(struct stu));
    return 0;
}

```

13.8.3. 结构体中嵌套构造类型成员的对齐

13.8.3.1. 数组

13.8.3.2. 结构体成员

13.9.结构体使用注意事项

13.9.1.向结构体内未初始化的指针拷贝

结构体中，包含指针，注意指针的赋值，切不可向未知区域拷贝。

```
struct student
{
    char*name;
    int score;
}stu;

int main()
{
    strcpy(stu.name,"Jimy");
    stu.score=99;
    return 0;
}
```

name 指针并没有指向一个合法的地址，这时候其内部存的只是一些乱码。所以在调用 strcpy 函数时，会将字符串 "Jimy" 往乱码所指的内存上拷贝，内存 name 指针根本就无权访问，导致出错。

```
int main()
{
    struct student *pstu;
    pstu = (struct student*)malloc(sizeof(struct student));
    strcpy(pstu->name,"Jimy");
    pstu->score=99;
    free(pstu);
    return 0;
}
```

为指针变量 pstu 分配了内存，但是同样没有给 name 指针分配内存。错误与上面第一种情况一样，解决的办法也一样。这里用了一个 malloc 给人一种错觉，以为也给 name 指针分配了内存。

13.9.2.未释放结构体内指针所指向的空间

从内向外依次释放空间。

```
#include <stdio.h>

typedef struct student
```

```

{
    char *name;
    int score;
} Stu;
int main()
{
    Stu * p = (Stu*)malloc(sizeof(Stu));
    p->name = malloc(30);
    strcpy(p->name, "wangguilin");
    p->score = 200;

    //free(p->name);      //由内而外的释放空间
    free(p);
    return 0;
}

```

13.9.3.深拷贝与浅拷贝

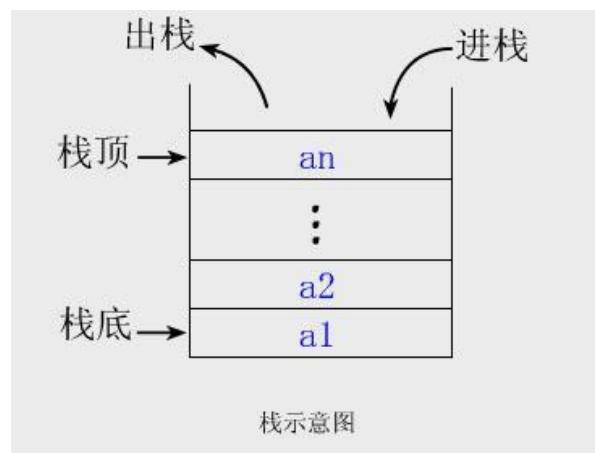
留待高级课程继续研究。

13.10.课堂实战

13.10.1.栈的自实现

13.10.1.1.栈的特点

栈的典型特点就是先进后出(FILO)，或是后进先出(LIFO)。主要接口操作，主要有四类，分别是，判空，判满，压栈，出栈。



13.10.1.2.代码实现

```
//全局版本
```

```

#include <stdio.h>
struct stack
{
    char space[512];
    int top;
};
struct stack stack = {{0},0};           //初始化

int isempty(){                         //判空
    return stack.top==0;
}
int isfull(){                          //判满
    return stack.top==512;
}
void push(char val)                  //压栈
{
    stack.space[stack.top] = val;
    stack.top++;
}
char pop()                           //出栈
{
    stack.top--;
    return stack.space[stack.top];
}
int main()
{
    if(!isfull())
        push('a');
    if(!isfull())
        push('b');
    if(!isfull())
        push('c');
    while(!isempty())
        putchar(pop());
    return 0;
}

```

```

//局部版本
#include <stdio.h>
//8
// top == 0 不能往外弹，已到栈底
// top == 8 不往往里压，已到栈顶

```

```
//top 始终指向一个待插入的位置
//push 操作，1, 写入数据，2, top++ 3, 此两步操作的前提是栈非满
//pop 操作，1, top-- 2,弹出数据 3, 此两步操作的前提是栈非空

typedef struct _Stack
{
    char mem[1024];
    int top;
}Stack;

int isFull(Stack *ps)
{
    return ps->top == 1024;
}

int isEmpty(Stack *ps)
{
    return ps->top == 0;
}

void push(Stack *ps,char ch)
{
    ps->mem[(ps->top)++] = ch;
}
char pop(Stack *ps)
{
    return ps->mem[--(ps->top)];
}

int main(int argc, char *argv[])
{
    Stack s = {{0},0};

    for(char ch='a'; ch<='z'; ch++)
    {
        if(!isFull(&s))
            push(&s,ch);
    }
    while(!isEmpty(&s))
    {
        putchar(pop(&s));
        puts("");
    }
}
```

```
    }
    return 0;
}
```

13.11.typedef 类型重命名

13.11.1.typedef 作用

用自定义名字为**已有数据类型**命名。其实叫 typerename 更合适。

类型定义形式：

```
typedef 现在类型名 新类型名;
```

13.11.2. 定义新类型

使用方法：

- 1 , 先用已有类型定义一个变量。
- 2 , 在定义语句类型前加 `typedef`。
- 3 , 将变量名换成你想要的名字。

```
#include <stdio.h>

int main(void)
{
    typedef int INT32;
    INT32 a;

    printf("sizeof(a) = %d sizeof(INT32) = %d\n",
           sizeof(a),sizeof(INT32));
    typedef int ARRAY10INT[10];

    ARRAY10INT array = {1,2,3,4,5,6,7,8,};

    printf("sizeof(ARRAY10INT) = %d ,sizeof(array) = %d\n",
           sizeof(ARRAY10INT),sizeof(array));

    for(int i=0; i<10; i++)
    {
        printf("%d\n",array[i]);
    }

    return 0;
}
```

13.11.3.size_t/int8 是什么鬼？

13.11.4.typedef 和#define 的区别

typedef 是以;号结尾的 C 语言语句。而#define 则是预处理阶段的文本替换。有时他们是可以互换的，但有时不可以。

```
#include <stdio.h>

int main(void)
{
    #define POINTERD char *
    typedef char * POINTERT;

    POINTERT pta,ptb;
    printf("sizeof(POINTERT) = %d\n", sizeof(POINTERT));
    printf("sizeof(pa) = %d sizeof(pb) = %d\n",
           sizeof(pta),sizeof(ptb));

    POINTERD pda,pdb;
    printf("sizeof(POINTERD) = %d\n", sizeof(POINTERD));

    printf("sizeof(pda) = %d sizeof(pdb) = %d\n",
           sizeof(pda),sizeof(pdb));

    return 0;
}
```

13.11.5.小结

- 新类型名一般用大写表示，以便于区别。
- 用 **typedef** 只能声明新的类型名，不能创造新的类型，只是为已经存在的类型起一个别名，也不能用来定义变量，即只能用其声明的类型来定义变量；
- 有时也可用宏定义来代替 **typedef** 的功能，但是宏定义是由预处理完成的，而 **typedef** 则是在编译时完成的，更为灵活方便。
- **typedef** 可以让类型更见名知意，更便于移植。

13.12.类型大总结

至此，我们完成了 c 语言完成了所有类型的学习。类型对我们的意义是什么呢？内存是以字节为单位进行线性编址的硬件基础，**类型是对内存的格式化**。

14.共用(Union)与枚举(Enum)

14.1.共用体

在 C 语言中，不同的成员使用共同的存储区域的数据构造类型称为共用体，简称共用，又称联合体。共用体在定义、说明和使用形式上与结构体相似。两者本质上的不同仅在于使用内存的方式上。

union 在操作系统底层的代码中用的比较多，因为它在内存共享布局上方便且直观。所以网络编程，协议分析，内核代码上有一些用到 union，比较好懂，简化了设计。

14.1.1.类型定义与变量定义

定义形式：

```
union 共用体名
{
    成员列表;
};
```

类型定义及变量定义：

```
union untest
{
    char c;
    short s;
    int i;
}myun;
```

14.1.2.内存分析

14.1.2.1.大小

共用体占用空间的大小取决于类型长度最大的成员。而结构体变量所占内存长度是各成员占的内存长度之和。每个成员分别占有其自己的内存单元。

```
#include <stdio.h>

union untest
{
    char c;
    short s;
    int i;
};
```

```

struct structtest
{
    char c;
    short s;
    int i;
};

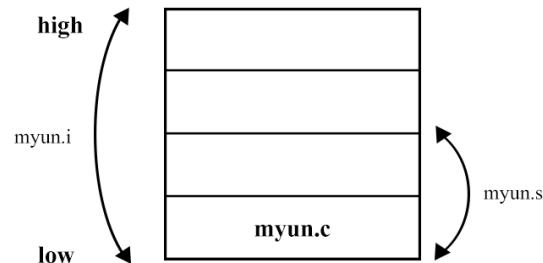
int main()
{
    printf("sizeof(union untest) = %d\n", sizeof(union untest));
    printf("sizeof(struct structtest) = %d\n", sizeof(struct structtest));
}

```

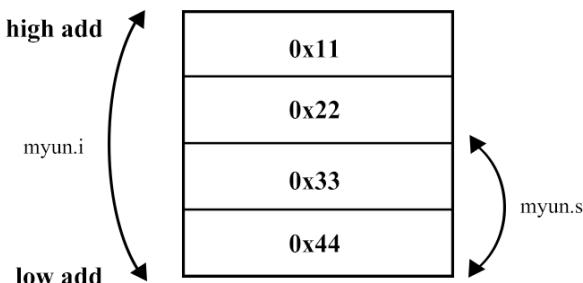
14.1.2.2. 成员访问

共用体变量 myun 的三个成员：myun.c，myun.s 和 myun.i 共用同一块内存（4 个字节大小）。其中，myun.c 只使用第一个字节，myun.s 使用前两个字节而 myun.i 使用全部 4 个字节，三成员享有同一个起始地址。而结构体中每个成员均有自己的地址。

如下图所示：



当我们给其中一个成员赋值时会影响到其他的成员。如 myun.i = 0x11223344，赋值后 myun.c 的值变成 0x44，myun.s 的值变成 0x3344。



```

#include <stdio.h>

union untest

```

```

{
    char c;
    short s;
    int i;
}myun;

struct structtest
{
    //int i;
    char c;
    short s;
    int i;
}myst;
int main()
{
    printf("%p %p %p\n",&myst.c,&myst.s,&myst.i);
    printf("%p %p %p\n",&myun.c,&myun.s,&myun.i); //拥有共同的起始地址

    myun.i = 0x11223344; //union untest myun={0x11223344};

    printf("myun.c = %x\n",myun.c);
    printf("myun.s = %x\n",myun.s);
    printf("myun.i = %x\n",myun.i);// 改变其一，影响其它

    printf("=====\\n");

    return 0;
}

```

14.1.2.3. 成员瞬时有效性

```

#include <stdio.h>

union mix
{
    char name[10];
    int age;
};

int main()
{
    union mix m;
    strcpy(m.name,"china"); //只有最后一次使用的变量是有意义的
}

```

```

printf("%s\n",m.name);
printf("%d\n",m.age);

m.age = 100;
printf("%s\n",m.name);
printf("%d\n",m.age);

return 0;
}

```

14.1.3. 共用体小结

- 共用体变量的地址和它的各成员的地址都是同一地址。
- 同一个内存段可以用来存放几种不同类型的成员，但在每一瞬时只能存放其中一种，而不是同时存放几种。
- 共用体变量中起作用的成员是最后一次存放的成员，在存入一个新的成员后原有的成员就失去作用。
- 共用体类型可以出现在结构体类型定义中，也可以定义共用体数组。反之，结构体也可以出现在共用体类型定义中，数组也可以作为共用体的成员。

14.1.4. 应用

当需要把不同类型的变量存放到同一段内存单元或对同一段内存单元的数据按不同类型处理则需要使用共用体数据结构。

14.1.4.1. 信息存储

设有若干个人员的数据，其中有学生和老师。学生的数据包括：姓名，编号，性别，职业，年级。老师的数据包括：姓名，编号，性别，职业，职务。可以看出，学生和老师所包含的数据是不同的。先要求把他们放在同一个表格中：

name	num	sex	job	Grade Position
Li	1011	f	s	1005
wang	2085	m	t	professor

要求设计程序输入人员信息然后输出。

如果把每个人都看作一个结构体变量的话，可以看出老师和学生的前 4 个成员变量是一样的，并且第五个成员变量可能是 grade 或者 position，当第四个成员变量是 s 的时候，第五个成员变量就是 grade；当第四个成员变量是 t 的时候，第五个成员变量就是 position。

```

struct Staff
{
    char name[30];
    char job;
    union
    {
        int grade;
        char positon[50];
    }gOrp;
};

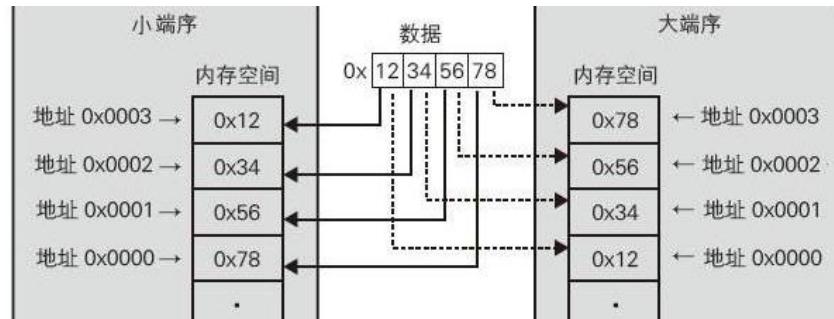
int main()
{
    struct Staff sta;
    printf("pls input name:");
    scanf("%s",sta.name);
    getchar();
    printf("pls input job:");
    scanf("%c", &sta.job);
    if (sta.job == 's') // s student
    {
        printf("pls input grade:");
        scanf("%d", &sta.gOrp.grade);
        printf("name = %s, job = %c, class = %d\n",
               sta.name, sta.job, sta.gOrp.grade);
    }
    else
    {
        printf("pls input positon:");
        scanf("%s", sta.gOrp.position);
        printf("name = %s, job = %c, class = %d\n",
               sta.name, sta.job, sta.gOrp.position);
    }
    return 0;
}

```

14.1.4.2.写程序判断大小端序

所谓的大端模式，是指数据的低位（就是权值较小的后面那几位）保存在内存的高地址中，而数据的高位，保存在内存的低地址中，这样的存储模式有点儿类似于把数据当作字符串顺序处理：地址由小向大增加，而数据从高位往低位放；

所谓的小端模式，是指数据的低位保存在内存的低地址中，而数据的高位保存在内存的高地址中，这种存储模式将地址的高低和数据位权有效地结合起来，高地址部分权值高，低地址部分权值低，和我们的逻辑方法一致。



```
#include <stdio.h>
union
{
    char ch;
    int integer;
}un;
int main(void)
{
    un.integer = 0x12345678;
    if(un.ch == 0x12)
        printf("big endian\n");
    else
        printf("small endian\n");

    return 0;
}
```

14.2.枚举

枚举类型定义了一组整型常量的集合，目的是提高程序的可读性。它的语法跟结构体的语法相同。

14.2.1.枚举类型定义

语法格式：

```
enum 枚举类型
{
    常量列表
};
```

定义：

```
enum DAY
{
```

```
MON,TUE,WED,THU,FRI,SAT,SUN //提供一组可选的常量
};
```

- (1) 枚举型是一个集合，集合中的元素(枚举成员)是一些命名的整型常量，元素之间用逗号(,)隔开。
- (2) 第一个枚举成员的默认值为整型的 0，后续枚举成员的值在前一个成员上加 1。
- (3) 可以人为设定枚举成员的值，从而自定义某个范围内的整数。
- (4) 枚举型是预处理指令#define 的替代。

14.2.2. 枚举变量与初始化

```
#include <stdio.h>

enum DAY
{
    MON = 1, TUE, WED, THU, FRI, SAT, SUN
};

int main()
{
    printf("%d, %d, %d, %d, %d, %d, %d\n",
           MON, TUE, WED, THU, FRI, SAT, SUN);

    enum DAY day = MON;

    printf("day = %d\n", day);
    return 0;
}
```

enum DAY 可以定义枚举变量，即可以取枚举过命名值，也可赋于其它整型。C++ 中则不可以，只能取枚举过的命名值。

14.2.3. 枚举常量

在使用过程中，枚举常量，应用更广泛，提高程序的可读性。常用于出错处理的枚举。

14.2.4. 应用

14.2.4.1. 枚举常量在 Qt 中的应用

前面，我们在学习的过程中，我们用过，两个 Qt 的图形用户界面的知识，分别是在判断键盘的按键值和绘制正弦图形使用画笔的颜色。

```
Qt::Key_A;
```

```
Qt::black;
```

还有常用的，返回正确与错误的枚举常量：

```
enum ret
{
    SUCCESS, FAILURE
};
```

14.2.4.2.判断日期

判断一年中第 125 天是工作日，还是休息日？假设一年 365 天，新年第一天是星期一。

```
int main()
{
    unsigned int today;
    printf("Pls input today Num:");
    scanf("%d", &today);

    today = today % 7;
    enum DAY day = today;
    // enum 常用于给 case 语句提供一类方便的标签。
    switch (day)
    {
        case MON:
        case TUE:
        case WED:
        case THU:
        case FRI:
            printf("today is workday\n");
            break;
        case SAT:
        case SUN:
            printf("today is holiday\n");
            break;
    }
    return 0;
}
```

14.2.4.3.增加 bool 类型

c 语方中没有 bool 类型，可以能过 enum 类型来枚举。

```
typedef enum BOOLEAN
{
    false,
    true
```

```
 }Bool;  
  
 Bool a = true;
```

14.3.练习

14.3.1.输出一个整型数据的字符形式

已知一无符号的整数占用了 4 个字节的内存空间，现欲从低位存储地址开始，将其每个字节作为单独的一个 ASCII 码字符输出，试用共同体实现上述转换。

14.3.2.实现 short 类型变量高低位互换

已知一长度为 2 个字节的整数，现欲将其高位字节与低位字节相互交换后输出，试用共同体类型实现这一功能。

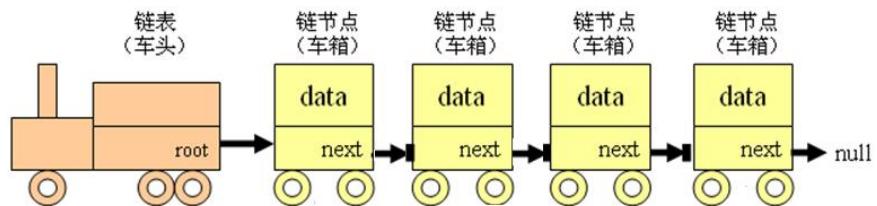
14.3.3.将下面程序中 case 分支常量用宏和枚举来实现

```
#include <stdio.h>  
int main()  
{  
    int choice;  
  
    printf("请输入你的中奖号码:");  
    scanf("%d",&choice);  
  
    switch(choice)      //可以优化  
    {  
        case 1:  
            printf("恭喜你获得一等奖，奖励 HHKB pro2\n");  
            //break;  
        case 2:  
            printf("恭喜你获得，赫曼米勒椅子一把\n");  
            //break;  
        case 3:  
            printf("恭喜你获得，Sony 显示器一台\n");  
            //break;  
        case 4:  
            printf("恭喜你获得，纪念品一份\n");  
            //break;  
        case 5:  
            printf("谢谢惠顾，谢谢惠顾\n");  
            //break;  
        default:    }
```

```
        printf("谢谢惠顾， 谢谢惠顾\n");
    }

    return 0;
}
```

15.单向链表(List)



15.1.链表价值

链表实现了，内存零碎数据的有效组织。比如，当我们用 `malloc` 来进行内存申请的时候，当内存足够，而由于碎片太多，没有连续内存时，只能以申请失败而告终，而用链表这种数据结构来组织数据，就可以解决上类问题。

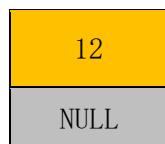


15.2.静态链表

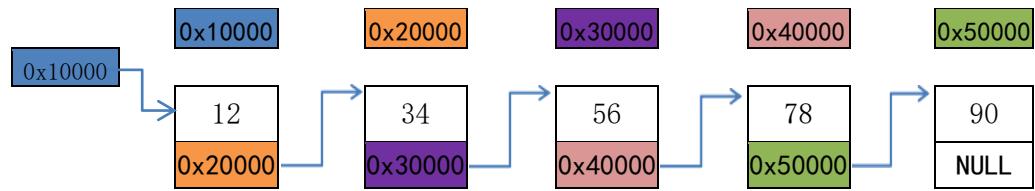
15.2.1.链表节点定义

```
typedef struct node
{
    int data;
    struct node * next;
}Node;
```

15.2.2.图示节点



15.2.3.图示链表结构



15.2.4.代码实现

```

#include <stdio.h>
typedef struct node
{
    int data;
    struct node * next;
}Node;

int main(void)
{
    Node a;
    Node b;
    Node c;
    a.data = 10;
    b.data = 20;
    c.data = 30;
    a.next = &b;
    b.next = &c;
    c.next = NULL;

    Node * head = &a;
    while(head != NULL)
    {
        printf("data = %d\n",head->data);
        head = head->next;
    }
    return 0;
}
  
```

15.3.动态链表

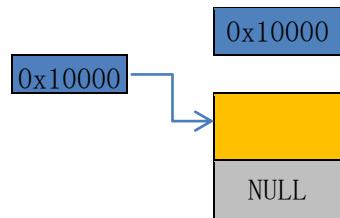
静态链表的意义不是很大，主要原因，数据存储在栈上，栈的存储空间有限，不能动态分配。所空链表要实现存储的自由，要动态的申请堆里的空间。

有一个点要说清楚，我们的实现的链表是**带头节点**。至于，为什么带头节点，需等大

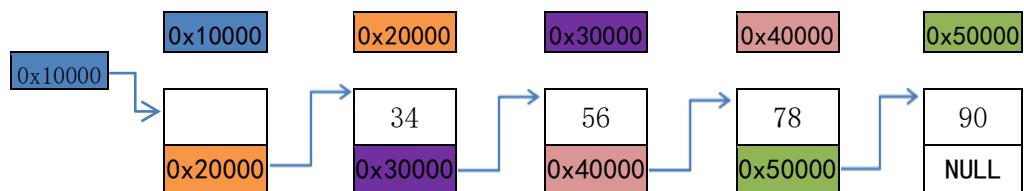
家对链表有个整体的的认知以后，再来体会，会更有意义。

15.3.1.链表图示

15.3.1.1.空链表



15.3.1.2.非空链表



15.3.2.链表名字解释



15.3.3.创建(尾插法)

15.3.3.1.图示



15.3.3.2.代码

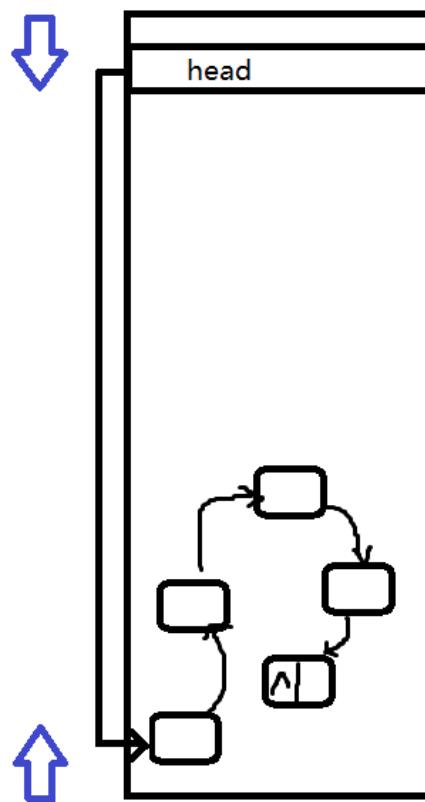
```
//尾插法，在尾节点插入，每插入一个节点，即成尾节点。
```

```
Node * createList()
{
```

```
Node * head = (Node *)malloc(sizeof(Node));
if(NULL == head)
    exit(-1);
head->next = NULL;

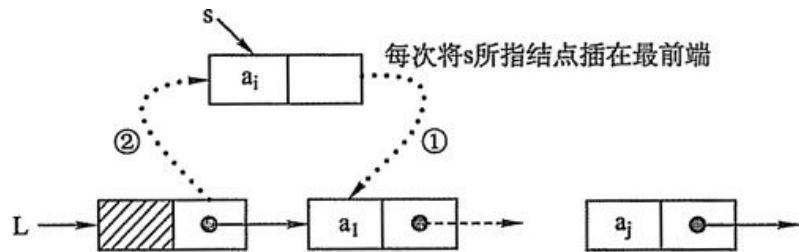
Node *t = head, *cur;
int nodeData;
scanf("%d",&nodeData);
while(nodeData)
{
    cur = (Node *)malloc(sizeof(Node));
    if(NULL == cur )
        exit(-1);
    cur->data = nodeData;
    t->next = cur;
    t = cur;
    scanf("%d",&nodeData);
}
t->next = NULL;
return head;
}
```

15.3.3.3.内存结构



15.3.4. 创建(头插法)

15.3.4.1. 图示



15.3.4.2. 代码

```
// 所谓的头插法，就是在头节点后面插入元素，每插入一下元素，即为首节点
Node * createList()
{
    Node * head = (Node *)malloc(sizeof(Node));
    if(NULL == head)
        exit(-1);
    head->next = NULL;
    Node *cur;

    int nodeData;
    scanf("%d",&nodeData);
    while(nodeData)//在线循环的时候，要注意，一重逻辑到二重逻辑的一致性
    {
        cur = (Node *)malloc(sizeof(Node));
        if(NULL == cur)
            exit(-1);
        cur->data = nodeData;

        cur->next = head->next;
        head->next = cur;

        scanf("%d",&nodeData);
    }
    return head;
}
```

15.3.5. 遍历

```
void traverseList(Node * head)
{
    head = head->next;
    while(head!=NULL)
    {
        printf("%d\n",head->data);
```

```

    head = head->next;
}
}

```

15.3.6.求长度

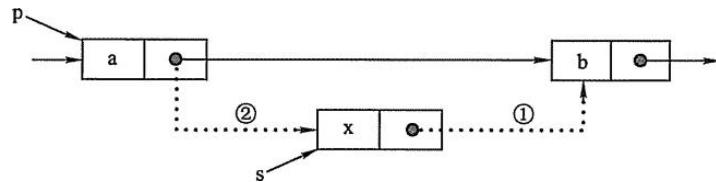
```

int lenList(Node * head)
{
    int len = 0;
    head = head->next;
    while(head)
    {
        len++;
        head = head->next;
    }
    return len;
}

```

15.3.7.插入

15.3.7.1.图示



15.3.7.2.代码

```

void insertList(Node * head,int insertData)
{
    Node * cur = (Node *)malloc(sizeof(Node));
    cur->data = insertData;
    cur->next = head->next;
    head->next = cur;
}

```

15.3.7.3.点评

带头节点的链表，插入元素时，即便新插入的元素成为首元素，不需要更新头节点指针。

头插法的思想，更适合插入操作。如果采用尾插法思想，则效率较低，比头插法每次都多一次遍历。

故而，前面写的创建链表的操作可以改为，**创建空链表 + 插入**，实际生产中亦是这样的。

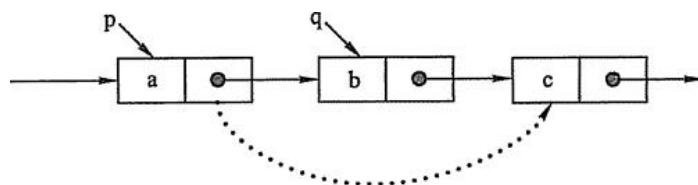
15.3.8.查找

遍历，匹配，返回。

```
Node * searchList(Node *head, int find)
{
    head = head->next;
    while(head)
    {
        if(head->data == find)
            break;
        head = head->next;
    }
    return head;
}
```

15.3.9.删除

15.3.9.1.图示



15.3.9.2.代码

```
void deleteNodeOfList(Node *head, Node *pfind)
{
    if(pfind->next == NULL)
    {
        while(head->next != pfind)
            head = head->next;
        head->next = pfind->next;
        free(pfind);
        pfind = NULL;
    }
    else
    {
        Node *t = pfind->next;
        pfind->data = pfind->next->data;
        pfind->next = pfind->next->next;
        free(t);
    }
}
```

15.3.9.3.点评

带头节点的链表删除元素时，即便删除链表的首元素，也是删除头节点后第一个

元素，不需要更新表头指针。

[从效率上来说，能否不遍历？](#) 提示，让下一个节点数据覆盖本节点，删除下一个节点。注意最后一个节点没有下一个节点。

15.3.10.排序

15.3.10.1.换值排序

```
void popSortList(Node *head)
{
    int len = lenList(head);
    head = head->next;
    Node * p,*q;
    for(int i=0; i<len-1; i++)
    {
        p = head; //每次内重循环从头开始
        q = p->next; //q 总是指向 p 的下一节点，也就是被比较的节点
        for(int j=0; j<len-1-i; j++)
        {
            if(p->data > q->data)
            {
                p->data ^= q->data;
                q->data ^= p->data;
                p->data ^= q->data;
            }
            p = p->next;
            q = p->next;
        }
    }
}
```

15.3.10.2.换指针排序

交换节点数据域的方式，当节点数据域很大时，效率低下。可否有交换指针的方式来提高效率呢？

```
void popSortList(Node *head)
{
    int len = lenList(head);
    Node *prep,*p,*q,*t;
    for(int i=0; i<len-1; i++)
    {
        prep = head;
        p = head->next;
        q = p->next;
        for(int j=0; j<len-1-i; j++)
        {
```

```

    {
        if(p->data > q->data)
        {
            prep->next = q;
            p->next = q->next;
            q->next = p;

            t = p;
            p = q;
            q = t;

        }
        prep = prep->next;
        p = p->next;
        q = p->next;
    }
}
}

```

15.3.10.3.点评

带头节点的链表，在进行排序时即便是首元素位置发生变化，也无需更新头节点指针。

15.3.10.4.补充：冒泡

		j=0	j=1	j=2	j=3	
	i=0	5	4	3	2	1
		4	5	3	2	1
		4	3	5	2	1
		4	3	2	5	1
		4	3	2	1	5
	i=1	4	3	2	1	
		3	4	2	1	
		3	2	4	1	
		3	2	1	4	
	i=2	3	2	1		
		2	3	1		

	2	1	3		
i=3	2	1			
	1	2			

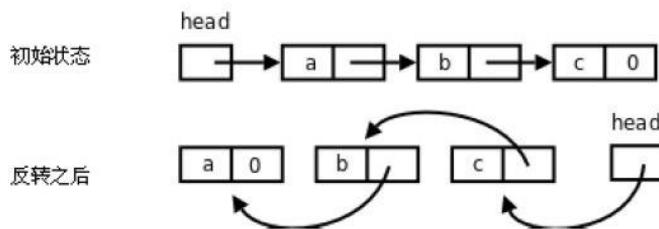
```

for(int i=0; i<N-1; i++)
{
    for(int j=0; j<N-1-i; j++)
    {
        if(arr[i]>arr[j])
        {}
    }
}

```

15.3.11.链表反转

15.3.11.1.图示



15.3.11.2.代码

```

void reverseList(Node * head)
{
    Node *p = head->next,*q; //割裂为两个链表
    head->next = NULL;

    while(p != NULL)
    {
        q = p->next;
        p->next = head->next;
        head->next = p;
        p = q;
    }
}

```

15.3.12.链表销毁

```
void destroyList(Node * head)
{
    Node * p = NULL;
    while(head)      //有多少个 malloc 就有多少个 free
    {
        p = head->next;
        free(head);
        head = p;
    }
}
```

15.3.13.功能测试

```
int main(int argc, char *argv[])
{
    Node *head = createList();

    srand(time(NULL));

    for(int i=0; i<10; i++)
    {
        insertList(head,rand()%100);
    }
    traverseList(head);

    printf("len of list = %d\n",lenList(head));

    Node* pfind = searchList(head,5);
    if(pfind == NULL)
        printf("find none\n");
    else
    {
        pfind->data = 1000;
        traverseList(head);

        printf("your find in list\n");
        deleteNodeOfList(head,pfind);
    }
    traverseList(head);

    popSortList(head);
```

```
printf("after popsort\n");
traverseList(head);

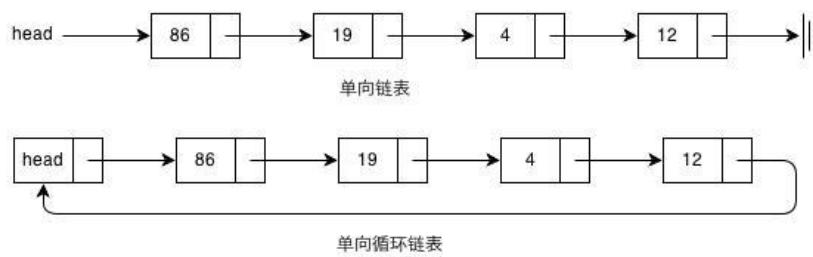
reverseList(head);
printf("after reverse\n");
traverseList(head);

destroyList(head);

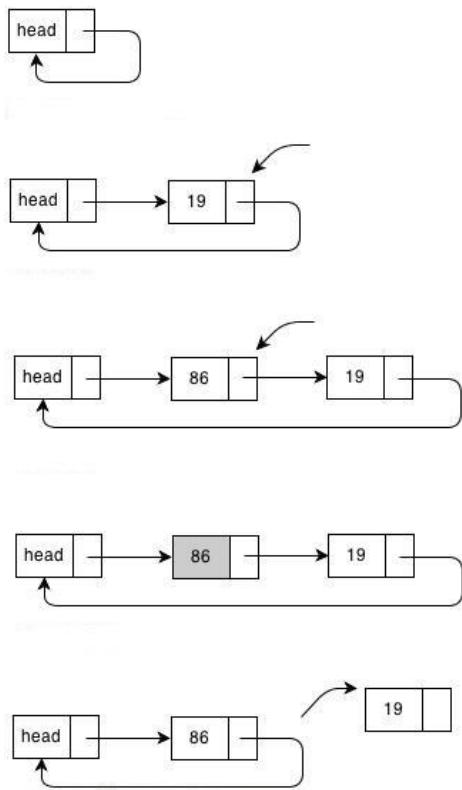
return 0;
}
```

15.4.环形链表

15.4.1.图示



15.4.2.自实现



15.5.作业

15.5.1. 实现手机通讯录功能

手机的通讯录的功能，至少有增查改删的功能。结构节点如下：

```
typedef struct _ContactsData
{
    char name[40];
    char mobileNum[20];
    char fixedNum[20];
    char companyName[50];
}ContactsData;

typedef struct _ContactsNode
{
    ContactsData data;
    struct _ContactsNode *next;
}ContactsNode;
```

15.5.2.输入一字符串，用链表形式储存。

输入一串字符至字符数组 ,遍历字符串生成链表 ,每个结点的数据域存放一个字符 ,最后输出链表中的全部字符。

15.5.3.用选择法实现单向链表的排序。

15.5.4.倒序打印单链表。

提示：可以借助栈或递归来实现。

```
void printReverseOrderList(Node * head)
```

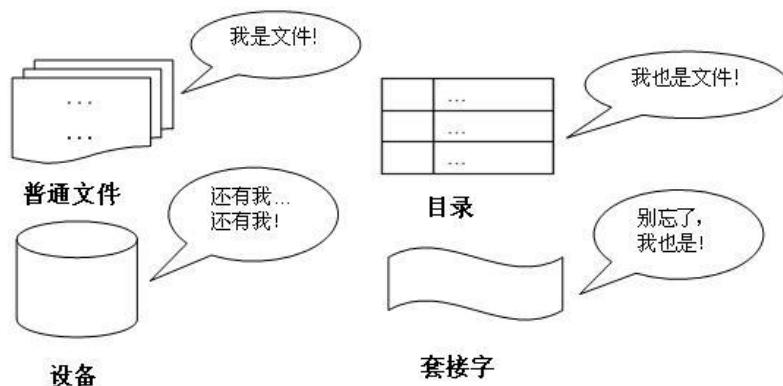
15.5.5.合并链表

合并两个有序单链表，并依然使其有序。

```
Node * mergeSortedList(Node * pHead1, Node * pHead2)
```

16. 文件(File)

Unix 的设计哲学之一，就是 Everything is a file, 可见文件是如此的重要。今天我们学习的是**读写磁盘文件**，为以后读取网络文件和硬件文件打下基础。



16.1. 文件流

16.1.1. 文件流概念

C 语言把文件看作是一个字符的序列，即文件是由一个一个字符组成的字符流，因此 C 语言将文件也称之为文件流。即，当读写一个文件时，可以不必关心文件的**格式或结构**。

文件的形象描述：



16.1.2. 文件类型

16.1.2.1. 文件分类

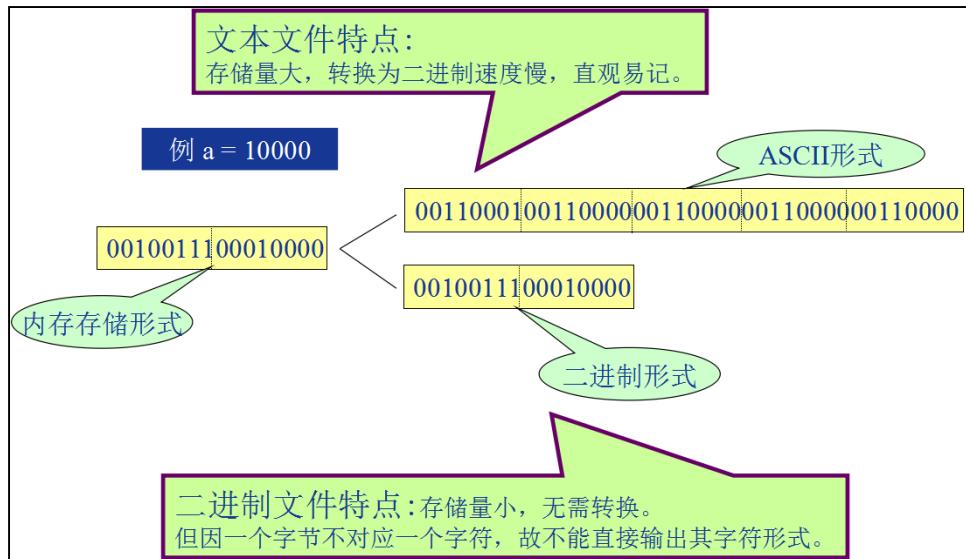
大家都知道计算机的存储，物理上是二进制的，所以文本文件与二进制文件的区别并不是物理上的，而是逻辑上的。这两者只是在编码层次上有差异。简单来说，文本文件是基于字符编码的文件，常见的编码有 ASCII 编码，二进制文件是基于值编码的文件。

文本文件：以 ASCII 码格式存放，一个字节存放一个字符。文本文件的每一个字节存放一个 ASCII 码，代表一个字符。这便于对字符的逐个处理，但占用存储空间较多，而且要花费时间转换。

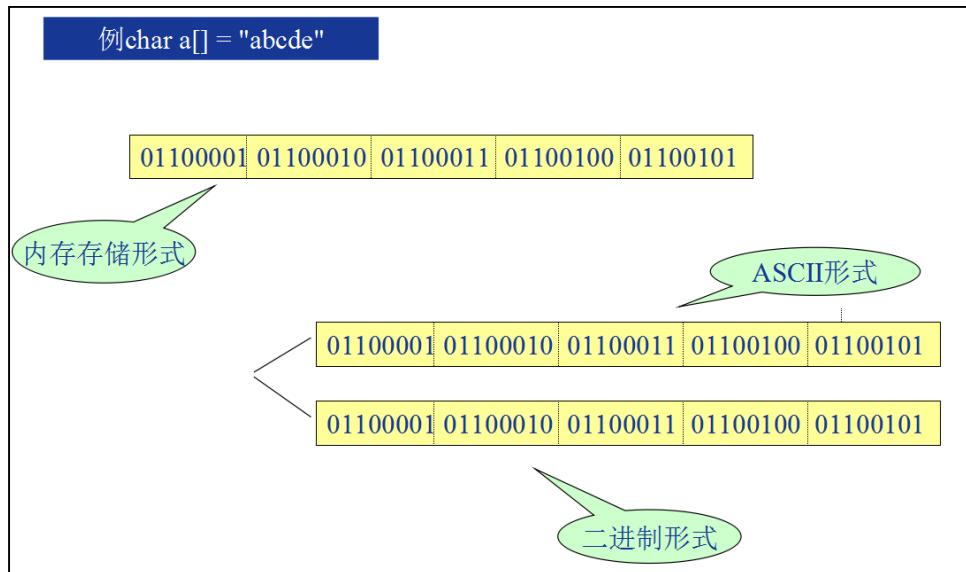
二进制文件：以值（补码）编码格式存放。二进制文件是把数据以二进制数的格式存放在文件中的，其占用存储空间较少。数据按其内存中的存储形式原样存放。

16.1.2.2. 图示代码展示

数值存入文件： `short int a = 10000; //0x2710`



字符存入文件：



```
#include <stdio.h>
int main()
{
    //short a = 10000;
    //FILE * fp = fopen("ascii.txt", "w");
    //fprintf(fp, "%d", a);
    //fclose(fp);
    //FILE *fp2 = fopen("bin.txt", "w");
    //fwrite(&a, 2, 1, fp2);
    //fclose(fp2);

    //花费时间用于转化
    //占用磁盘空间多，磁盘中的文件，易读
}
```

```

char *buf = "abcdefg";
FILE* fp = fopen("ascii2.txt", "w");
fprintf(fp, "%s", buf);
fclose(fp);

//不需要花费时间转化
//相比于文本，占用磁盘空间小，磁盘文件不易读

FILE * fp2 = fopen("bin2.txt", "w");
fwrite(buf, 8, 1, fp2);
fclose(fp2);
return 0;
}

```

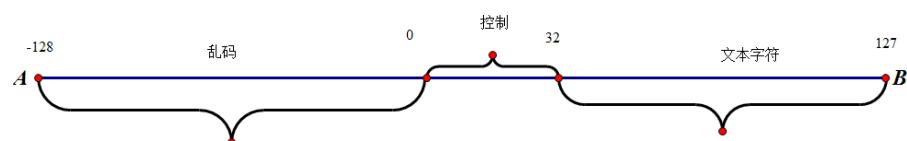
16.1.2.3.乱码原由

举例文本工具，打开文件的过程。拿记事本来说，它首先读取文件物理上所对应的二进制比特流，然后按照你所选择的解码方式来解释这个流，然后将解释结果显示出来。一般来说，你选取的解码方式会是 ASCII 码形式（ASCII 码的一个字符是 8 个比特），接下来，每 8 个比特 8 个比特地来解释这个文件流。

例如文件流 01000000_01000001_01000010_01000011" (下划线 "_"，为了增强可读性手动添加的)，第一个 8 比特"01000000"按 ASCII 码来解码的话，所对应的字符是字符"A"，同理其它 3 个 8 比特可分别解码为"BCD"，即这个文件流可解释成“ABCD”，然后记事本就将这个“ABCD”显示在屏幕上。

记事本无论打开什么文件都按既定的字符编码工作（如 ASCII 码），所以当他打开二进制文件时，出现乱码也是很必然的一件事情了，解码和译码不对应嘛。

例如文件流"00000000_00000000_00000000_00000001"可能在二进制文件中对应的是一个四字节的整数 int 1，在记事本里解释就变成了"NULL_NULL_NULL_SOH"这四个控制符。



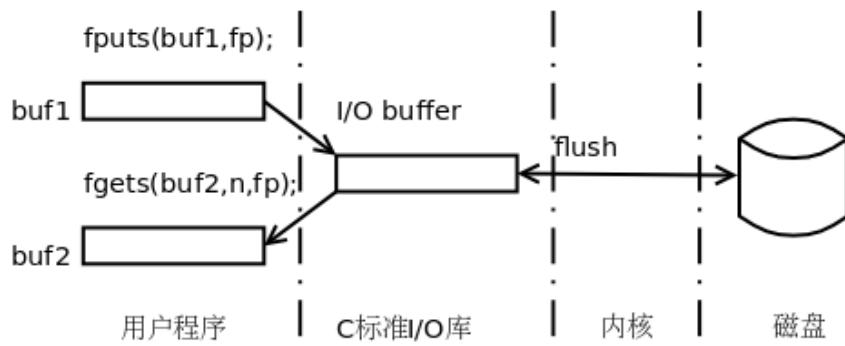
16.1.2.4.二进制与文本转化查看工具

windows	linux	
UltraEdit	hexdump	-c -C -x -o

16.1.3.文件缓冲

为什么要有缓冲区（buffer）原因为多种，有两个重点：

- 从内存中读取数据比从文件中读取数据要快得多。
- 对文件的读写需要用到 **open**、**read**、**write** 等系统底层函数，而用户进程每调用一次系统函数都要从用户态切换到内核态，等执行完毕后再返回用户态，这种切换要花费一定时间成本（对于高并发程序而言，这种状态的切换会影响到程序性能）。



举个例子，如果程序需要处理 10K 个整数（或者 10K 个字符串等等），而这些整数事先存在某个文件中，如果程序每处理一个整数就要从文件中读一个整数（`read` 系统调用），那么每次都要进行硬件 I/O、进程状态切换等操作，这样效率是非常低下的。如果每次从文件中读出 1K 个整数到内存，程序从内存中读取数据并处理，那么程序的性能会明显提高，存储这 1K 个整数的内存区域就是一个缓冲区。

```
#include <stdio.h> // linux code only
int main()
{
    while (1)
    {
        printf("abcdefg"); // 缓冲区满，则会写入文件。
        usleep(10000);
    }
    return 0;           // 也可以通过 fclose 和 fflush 刷缓冲的
}
```

16.2. 文件的打开和关闭

16.2.1. FILE 结构体

`FILE` 结构体是对缓冲区和文件读写状态的记录者，所有对文件的操作，都是通过 `FILE` 结构体完成的。

```

typedef struct {
    short      level;          /* 缓冲区满/空程度 */
    unsigned   flags;          /* 文件状态标志 */
    char       fd;              /* 文件描述符 */
    unsigned char hold;        /* 若无缓冲区不读取字符 */
    short     bsize;           /* 缓冲区大小 */
    unsigned char *buffer;     /* 数据传送缓冲区位置 */
    unsigned char *curp;       /* 当前读写位置 */
    unsigned   istemp;         /* 临时文件指示 */
    short     token;           /* 用作无效检测 */
} FILE;

```

在开始执行程序的时候，将自动打开 3 个文件和相关的流：标准输入流(**stdin**)、标准输出流(**stdout**)和标准错误(**stderr**)，它们都是 **FILE***型的指针。流提供了文件和程序的通信通道。

16.2.2.fopen

函数声明	FILE * fopen (const char * filename, const char * mode);		
所在文件	stdio.h		
函数功能	以 mode 的方式，打开一个 filename 命名的文件，返回一个指向该文件缓冲的 FILE 结构体指针。		
参数及返回解析			
参数	char*filename	filename :要打开，或是创建文件的路径。	
	char *mode	mode :打开文件的方式，内容比较多见下文。	
返回值	FILE*	返回指向文件缓冲区的指针，该指针是后续操作文件的句柄。	

mode 详解：

mode	处理方式	当文件不存在时	当文件存在时	向文件输入	从文件输出
“r”	读取	出错	打开文件	不能	可以
“w”	写入	建立新文件	覆盖原有文件	可以	不能
“a”	追加	建立新文件	在原有文件后追加	可以	不能
“r+”	读取/写入	出错	打开文件	可以	可以
“w+”	写入/读取	建立新文件	覆盖原有文件	可以	可以
“a+”	读取/追加	建立新文件	在原有文件后追加	可以	可以

如果读写的是二进制文件，则还要加 b, 比如 rb, r+b 等。 unix/linux 不区分文本和二进制文件。

16.2.3.fclose

函数声明	<code>int fclose (FILE * stream);</code>	
所在文件	<code>stdio.h</code>	
函数功能	<code>fclose()</code> 用来关闭先前 <code>fopen()</code> 打开的文件。此动作会让缓冲区内的数据写入文件中，并释放系统所提供的文件资源。	
参数及返回解析		
参数	<code>FILE*</code>	<code>stream</code> : 指向文件缓冲的指针。
返回值	<code>int</code>	成功返回 0，失败返回 <code>EOF(-1)</code> 。

```
#include <stdio.h>
int main()
{
    FILE * fp = fopen("data.txt", "w");
    fputs("china is great!!", fp);
    fclose(fp); //fflush(fp)           //设置断点调试
    return 0;
}
```

16.3.一次读写一个字符(文本操作)

16.3.1.fputc

函数声明	<code>int fputc (int ch, FILE * stream);</code>	
所在文件	<code>stdio.h</code>	
函数功能	将 <code>ch</code> 字符，写入文件。	
参数及返回解析		
参数	<code>FILE*</code>	<code>stream</code> : 指向文件缓冲的指针。
	<code>int</code>	: 需要写入的字符。
返回值	<code>int</code>	写入成功，返回写入成功字符，如果失败，返回 <code>EOF</code> 。

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

```

FILE* fp = fopen("ascii.txt","w");
if(fp == NULL)
{
    printf("open error\n");
    return -1;
}

for(char ch = 'a'; ch<='z'; ch++)
{
    printf("%3c",fputc(ch,fp));
}

fclose(fp);
return 0;
}

```

16.3.2.fgetc

函数声明	<code>int fgetc (FILE * stream);</code>	
所在文件	<code>stdio.h</code>	
函数功能	从文件流中读取一个字符并返回。	
参数及返回解析		
参数	<code>FILE*</code>	<code>stream</code> :指向文件缓冲的指针。
返回值	<code>int</code>	正常，返回读取的字符；读到文件尾或出错时，为 EOF。

读取字符，重点是判断，**结束条件是什么？**通常的作法是**依据返回值**。

```

#include <stdio.h>
int main()
{
    FILE* fp = fopen("ascii.txt","r");
    if(fp == NULL)
    {
        printf("open error\n");
        return -1;
    }

//    for(char ch = 'a'; ch<='z'; ch++)
//    {

```

```

//      printf("%3c",fputc(ch,fp));
//  }

char ch;
while((ch = fgetc(fp)) != EOF) //优先级，关系>赋值
    printf("%3c",ch);

fclose(fp);
return 0;
}

```

16.3.3.关于feof的问题

函数声明	<code>int feof(FILE * stream);</code>	
所在文件	<code>stdio.h</code>	
函数功能	判断文件是否读到文件结尾	
参数及返回解析		
参数	<code>FILE*</code>	<code>stream</code> :指向文件缓冲的指针。
返回值	<code>int</code>	0 未读到文件结尾，非零 读到文件结尾。

feof 这个函数，是去读标志位判断文件是否结束的。即在**读到文件结尾的时候再去读一次，标志位才会置位**，此时再来作判断文件处理结束状态，文件到结尾。如果用于打印，则会出现多打一次的现象。

```

#include <stdio.h>
int main()
{
    FILE* fp = fopen("ascii.txt","w+");
    if(fp == NULL)
    {
        printf("open error\n");
        return -1;
    }

    for(char ch = 'a'; ch<='z'; ch++)
    {
        fputc(ch,fp);
    }
    rewind(fp);

    char ch;

```

```
while((ch = fgetc(fp))&& !feof(fp)) //这才是正确的姿势
{
    printf("%3c",ch);
}

fclose(fp);
return 0;
}
```

16.3.4.练习

16.3.4.1.实现 linux 中 cp 命令的功能

```
int main(int argc, char *argv[])
{
    if (argc <3)
    {
        printf("usage: %s src dest\n", argv[0]);
        return 0;
    }
    FILE * fp = fopen(argv[1], "r");
    if (fp == NULL)
    {
        return -1;
    }
    FILE * fp2 = fopen(argv[2], "w");
    if (fp2 == NULL)
    {
        fclose(fp);
        return -1;
    }
    char ch ;
    while ((ch = fgetc(fp)) != EOF)
    {
        fputc(ch, fp2);
        ch = fgetc(fp);
    }

    fclose(fp);
    fclose(fp2);
}
```

16.3.4.2.文件的加密与解密

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

```
#define SEC 10 // #define SEC 150

int main() //解密
{
    FILE * fr = fopen("mainsec.c","r");
    if(fr == NULL)
        return -1;
    FILE *fw = fopen("anothermain.c","w");

    if(fw == NULL)
    {
        fclose(fr); //负责善后事宜
        return -1;
    }
    char ch;
    while(1)
    {
        ch = fgetc(fr);
        if(ch == EOF)
            break;
        ch -= SEC;
        fputc(ch,fw);
    }
    fclose(fr);
    fclose(fw);
}

int mainEncrypt(void) //加密
{
    FILE * fr = fopen("main.c","r");
    if(fr == NULL)
        return -1;
    FILE *fw = fopen("mainsec.c","w");

    if(fw == NULL)
    {
        fclose(fr); //负责善后事宜
        return -1;
    }
    char ch;
    while(1)
    {
        ch = fgetc(fr);
        if(ch == EOF)
```

```

        break;
        ch += SEC;
        fputc(ch, fw);
    }

    fclose(fr);
    fclose(fw);

    return 0;
}

```

评价，此种加密方式，是加密的最初级形式，极易造成溢出，导致加密不可逆。结论呵呵。

16.4.一次读写一行字符(文本操作)

16.4.1.什么是行

16.4.1.1.定义

行是文本编辑器中的概念，文件流中就是一个字符。这个在不同的平台是有差异的。`window` 平台 '`\r\n`'，`linux` 平台是'`\n`'。

16.4.1.2.平台差异

`windows` 平台在写入'`\n`'是会体现为'`\r\n`'，`linux` 平台在写入'`\n`'时会体现为'`\n`'。`windows` 平台在读入'`\r\n`'时，体现为一个字符'`\n`'，`linux` 平台在读入'`\n`'时，体现为一个字符'`\n`'。

`linux` 读 `windows` 中的换行，则会多读一个字符，`windows` 读 `linux` 中的换行，则没有问题。

16.4.1.3.测试

写入

```

#include <stdio.h>
int main()
{
    FILE *fp = fopen("win.txt","w");
    fputc('a',fp);
    fputc('\n',fp);
    fputc('b',fp);
    return 0;
}

```

读出

```

#include <stdio.h>
int main()
{
    FILE * pf = fopen("win.txt","r");

    char ch = fgetc(pf);

```

```

printf("%x-%c",ch,ch);
ch = fgetc(pf);
printf("%x-%c",ch,ch);
ch = fgetc(pf);
printf("%x-%c",ch,ch);

ch = fgetc(pf);
printf("%hx-%c",ch,ch);
ch = fgetc(pf);
printf("%hx-%c",ch,ch);

return 0;
}

```

16.4.1.4.dos2unix 命令

dos2unix 命令用来将 DOS 格式的文本文件转换成 UNIX 格式的(DOS/MAC to UNIX text file format converter)。DOS 下的文本文件是以\r\n 作为断行标志的，表示成十六进制就是 0D 0A。而 Unix 下的文本文件是以\n 作为断行标志的，表示成十六进制就是 0A。

```

yum install dos2unix
dos2unix aa.txt

```

16.4.2.fputs

函数声明	<code>int fputs(char *str,FILE *fp)</code>	
所在文件	<code>stdio.h</code>	
函数功能	把 str 指向的字符串写入 fp 指向的文件中。	
参数及返回解析		
参数	<code>char *</code>	<code>str</code> : 表示指向的字符串的指针。
	<code>FILE *</code>	<code>fp</code> : 指向文件流结构的指针。
返回值	<code>int</code>	正常，返 0；出错返 EOF。

```

#include <stdio.h>
int main()
{
    FILE *fp = fopen("ascci2.txt","w");
    if(fp == NULL)
    {
        printf("fopen error\n");
        return -1;
    }

    printf("%d\n",fputs("abcdefg\n",fp));           // 0 or -1
}

```

```

printf("%d\n", fputs("1234567890\n", fp));      // 0 or -1

char buf[] = "xyz";
printf("%d\n", fputs(buf, fp));                  // 0 or -1
fclose(fp);

return 0;
}

```

16.4.3.fgets

函数声明	<code>char *fgets(char *str,int length,FILE *fp)</code>		
所在文件	<code>stdio.h</code>		
函数功能	从 <code>fp</code> 所指向的文件中，至多读 <code>length-1</code> 个字符，送入字符数组 <code>str</code> 中，如果在读入 <code>length-1</code> 个字符结束前遇 <code>\n</code> 或 <code>EOF</code> ，读入即结束，字符串读入后在最后加一个 ‘\0’ 字符。 <code>fgets</code> 函数返回有三个条件： 1-读 <code>n-1</code> 个字符前遇到 <code>\n</code> ，读取结束 (<code>\n</code> 被读取) + <code>\0</code> 。 2-读 <code>n-1</code> 个字符前遇到 <code>EOF</code> ，读取结束 + <code>\0</code> 。 3-读到 <code>n-1</code> 个符+ <code>\0</code> 。		
参数及返回解析			
参数	<code>char *</code>	<code>str</code>	: 指向需要读入数据的缓冲区。
	<code>int</code>	<code>length</code>	: 每一次读数字符的字数。
	<code>FILE*</code>	<code>fp</code>	: 文件流指针。
返回值	<code>char *</code>	正常，返 <code>str</code> 指针；出错或遇到文件结尾 返空指针 <code>NULL</code> 。	

按行读取，重点是判断，结束条件是什么？通常的作法是依据返回值。

```

#include <stdio.h>

#define N 1024

//N-1 个字符前遇到了 \n EOF
//1->abcd\n+\0
//2->abcd1234567+\0

//还没有遇到\n EOF 读取的数据到了上限
//3->abc+\0


int main()
{

```

```
FILE *pf = fopen("xx.txt","w+");
if(pf == NULL)
    exit(-1);

fputs("abcd\n1234567",pf);
//    fputs("abcd1234567",pf);

rewind(pf);
char buf[N];

while(fgets(buf,N-1,pf))
    printf("%s",buf);
fclose(pf);
}

int main2()
{
FILE *pf = fopen("xx.txt","w+");
if(pf == NULL)
    exit(-1);

fputs("abcd\n1234567",pf);
//    fputs("abcd1234567",pf);

rewind(pf);
char buf[N];

//遇到\n EOF 的时候返回非空，在EOF以后，再去读，NULL
printf("%p\n",fgets(buf,N-1,pf));
printf("%p\n",fgets(buf,N-1,pf));
printf("%p\n",fgets(buf,N-1,pf));
//    fgets(buf,N-1,pf);
//    fgets(buf,4,pf);
puts(buf);
fclose(pf);
}

int main1(int argc, char *argv[])
{
FILE *pf = fopen("xx.txt","w+");
if(pf == NULL)
```

```
exit(-1);

fputs("abcd\n1234567",pf);
//    fputs("abcd1234567",pf);

rewind(pf);
char buf[N];
fgets(buf,N-1,pf);
// fgets(buf,N-1,pf);
// fgets(buf,4,pf);
puts(buf);
fclose(pf);

return 0;
}
```

16.4.4.关于feof的问题

```
#include <stdio.h>
int main()
{
    FILE* fp = fopen("text.txt","w+");
    if(fp == NULL)
        return -1;

    fputs("aaaaaaaaaaaaaaaaaaaaaaaaa\n",fp);
    fputs("bbbbbbbbbbbbbbbbbbbbbbbbb\n",fp);
    fputs("cccccccccccccccccccccccccc\n",fp);
    fputs("dddddddddddddcccccccccccc\n",fp);
    fputs("eeeeeeeeeeeeeeeeeeeeeee\n",fp);
    fputs("ffffffffffffffffff\n",fp);

    rewind(fp);

    char buf[30] = {0};

    while(fgets(buf,30,fp)&&!feof(fp))//取最后一行的时候feof已经认为空了。
    {
        printf("%s",buf);
    }
    fclose(fp);

    return 0;
```

{

如果最后一行，没有行 '\n' 的话则少读一行。

Linux 中无论是 gedit 还是 vim 系统会自动在末行添加\n 标志。Windows 当中要注意系统不会自动添加\n，最好用返回值来判断，是否读到结尾。

另 :window 中可以用 UE 等工具文件的十六进制 ,linux 中可用命令 hexdump -C 查看文件的十六进制。

16.4.5. 注意事项

读到 buf 内的字符串，可含有**空白格式控制字符**，比如 '\n' '\r\n' '\t' 等，如果直接用作**比较**等用途，可能得不到正确的结果。

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

#if 0
xx.ini
wangguilin\t\t
#endif

int main(int argc, char *argv[])
{
    FILE *pf = fopen("xx.ini","r+");
    if(NULL == pf)
        exit(-1);

    char name[1024];
    scanf("%s",name);

    char buf[1024];
    fgets(buf,1024,pf); // \t

    char *p = buf;
    while(*p) p++;
    p--;

    while(*p == '\t')
    {
        *p = '\0';
        p--;
    }
}
```

```

if(strcmp(name, buf) == 0)
{
    printf("欢迎登录\n");
}
else
{
    printf("登录失败\n");
}

return 0;
}

```

16.4.6.练习

16.4.6.1.读以下文件，请问 fgets 共执行了多少次？

`fgets(buf, 10, fp);`

```

1234567890abcdefg
1234567890
abcdefg

```

16.4.6.2.读配置文件，过滤掉以#开头的注释行和空行。

```

# Backend to store user information in. New installations should
# use either tdbsam or ldapsam. smbpasswd is available for backwards
# compatibility. tdbsam requires no further configuration.

# Use password server option only with security = server or if you can't
# use the DNS to locate Domain Controllers
# The argument list may include:
# password server = My_PDC_Name [My_BDC_Name] [My_Next_BDC_Name]
# or to auto-locate the domain controller/s
# password server = *

; security = domains
; passdb backend = tdbsam
; realm = MY_REALM
; password server = <NT-Server-Name>

```

注：空行(?)是以\n开头的行，注释行是以#开头的行。

16.4.6.3.文本等号以后求和

```
a = 3
b = 4
c = 90
d = 20
e = 25
f = 73
g = 89
```

补充：

`atoi(ascii to integer)`

函数原型：`int atoi(const char *nptr);`

函数的功能：将字符串转换成整型数；`atoi()`会扫描参数 `nptr` 字符串，跳过前面的空格字符，直到遇上数字或正负号才开始做转换，而在遇到非数字或字符('0')时才结束转化，并将结果返回（返回转换后的整型数）。

16.5.一次读写一块数据(二进制操作)

C 语言已经从接口的层面区分了，文本的读写方式和二进制的读写方式。前面我们讲的是文本的读写方式。

所有的文件接口函数，要么以 '\0'，表示输入结束，要么以 '\n'，EOF(0xFF)表示读取结束。'\0' '\n' 等都是文本文件的重要标识，而二进制文件，则往往以块的形式，写入或读出。

而所有的二进制接口对于这些标识，是不敏感的。

16.5.1.fwrite / fread

函数声明	<code>int fwrite(void *buffer, int num_bytes, int count, FILE *fp)</code>		
	<code>int fread(void *buffer, int num_bytes, int count, FILE *fp)</code>		
所在文件	<code>stdio.h</code>		
函数功能	把 <code>buffer</code> 指向的数据写入 <code>fp</code> 指向的文件中，或是把 <code>fp</code> 指向的文件中的数据读到 <code>buffer</code> 中。		
参数及返回解析			
参数	<code>char *</code>	<code>buffer</code> : 指向要输入/输出数据存储区的首地址的指针	
	<code>int</code>	<code>num_bytes</code> : 每个要读/写的字段的字节数	
	<code>int</code>	<code>count</code> : 要读/写的字段的个数	
	<code>FILE*</code>	<code>fp</code> : 要读/写的文件指针	
返回值	<code>int</code>	成功，返回读/写的字段数；出错或文件结束，返回 0。	

16.5.2.试读文本文件

当我们试图用 `fread` 去读取文本文件时候，发现文本中的**格式已经没有任何意义**，只是一个普通的字符。

	写入	读出	返回
<code>fgets</code>		<code>\n /EOF /len-1</code>	not NULL/ NULL
<code>fputs</code>	<code>\0</code>		
<code>fread</code>	<code>num_bytes n</code>		not 0 / 0
<code>fwrite</code>		<code>num_bytes n</code>	

16.5.2.1.写特殊字符进文件

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    FILE *fpw = fopen("bin.txt","wb");
    if(fpw == NULL)
        return -1;
    char *p = "china \n is \0 great";
    fwrite(p,1,strlen(p)+6,fpw);
    fclose(fpw);
    return 0;
}
```

16.5.2.2.从文件中读特殊字符

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    FILE * fpr = fopen("bin.txt","rb");
    if(fpr == NULL)
        return -1;

    char buf[1024];
    int n;
    n = fread(buf,1,1024,fpr);
    printf("n = %d\n",n);
    for(int i=0; i<n; i++)
    {
        printf("%#x\n",buf[i]);
    }
}
```

```
fclose(fpr);
}
```

16.5.2.3.返回值的陷阱

没有\n / EOF / len-1 作为读出的结束标识， fread 依靠读出块多少来标识读结果和文件结束标志。

以最小的单元格式进行读，或是以写入的最小单元进行读。

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    FILE *fpw = fopen("bin.txt","wb+");
    if(fpw == NULL)
        return -1;
    char *p = "123456789";
    fwrite(p,1,strlen(p),fpw);
    rewind(fpw);

    char buf[1024];

    int n;
    //step 1
//n = fread((void*)buf,1,1024,fpw);
//printf("n = %d\n",n);

//n = fread((void*)buf,1024,1,fpw);
//printf("n = %d\n",n);

    //step 2
//n =fread((void*)buf,4,1,fpw);
//printf("n = %d\n",n);
//n =fread((void*)buf,4,1,fpw);
//printf("n = %d\n",n);
//n =fread((void*)buf,4,1,fpw);
//printf("n = %d\n",n);

    //step 3
n =fread((void*)buf,1,4,fpw);
printf("n = %d\n",n);
n =fread((void*)buf,1,4,fpw);
printf("n = %d\n",n);
n =fread((void*)buf,1,4,fpw);
printf("n = %d\n",n);
```

```
//step 4 正确姿势

}
return 0;
```

16.5.3.二进制读写才是本质

当我们用 UE 打开一个二进制文件(图片，视频)的时候，发现文件中到处都是文本的标志性字符，但是对于 fread 和 fwrite 来说，一视同仁，都是一个普通的字节而已，所以，二进制文件的读写就要用对文本标记不敏感的 fread 和 fwrite 来进行。

```
#include <stdio.h>
int main(void)
{
    int a[10] = {0xff,0x0,10,2,3,4,5,6,7,8};
    FILE * fp = fopen("xx.txt","wb+"); //xx.txt 文件打开是乱码
    if(fp == NULL)
        return -1;
    fwrite(a,sizeof(int[10]),1,fp);
    rewind(fp);

    int n=0;
    int data;
    while((n = fread(&data,sizeof(int),1,fp)) > 0)
    {
        printf("n = %d data = %x \n",n,data);
    }
    fclose(fp);
    return 0;
}
```

16.5.3.1.文件的加密与解密

加密语法格式

```
file -d src.wmv sec.wmv
file -x sec.wmv anohtersrc.wmv
```

程序：

```
#include <stdio.h>

void encode(char *p,int n)
{
```

```
for(int i=0; i<n ;i++)
    p[i]++;
}

void decode(char *p,int n)
{
    for(int i=0; i<n ;i++)
        p[i]--;
}

int main(int argc, char **argv)
{

    if(argc != 4)
    {
        printf("use: binsec.exe -[d|x] src dest\n");
        return -1;
    }
    if(strcmp(argv[1],"-d")== 0)
    {
        printf("dcode\n");
        FILE *fp = fopen(argv[2],"rb+");
        FILE *fp2 = fopen(argv[3],"wb");

        int res;
        char buf[1024];
        memset(buf,0,1024);
        while((res = fread(buf,1,1024,fp))>0)
        {
            encode(buf,res);
            fwrite(buf,1,res,fp2);
        }
    }
    if(strcmp(argv[1],"-x") == 0)
    {
        printf("xcode\n");
        FILE *fp = fopen(argv[2],"rb+");
        FILE *fp2 = fopen(argv[3],"wb");

        int res;
        char buf[1024];
        memset(buf,0,1024);
        while((res = fread(buf,1,1024,fp))>0)
        {
            decode(buf,res);
        }
    }
}
```

```
        fwrite(buf,1,res,fp2);
    }
}

return 0;
}
```

16.5.4. 读写结构体是长项

结构体中的数据类型不统一，此时最适合用二进制的方式进行读写。二进制的接口可以读文本，而文本的接口不可以读二进制。

```
#include <stdio.h>
typedef struct student
{
    int num;
    char name[30];
    char sex;
    float math;
    float english;
    float chinese;
}Student;

//宏函数，以后会讲
#define OPEN_ERR(fp) \
if(fp == NULL)\
{\
    printf("open error\n");\
    return -1;\
}

int main()
{
    FILE* fp = fopen("stu.data","wb+");
    OPEN_ERR(fp);

    Student stu[] = {
        {1001,"bob",'m',100,30,20},
        {1002,"bob2",'f',100,30,20},
        {1003,"bob3",'m',100,30,20}
    };

    for(int i=0; i<sizeof(stu)/sizeof(stu[0]); i++)
    {
```

```

        fwrite((void*)&stu[i],sizeof(Student),1,fp);
    }

    rewind(fp);

    Student tmp;

    for(int i=0; i<sizeof(stu)/sizeof(stu[0]); i++)
    {
        fread((void*)&tmp,sizeof(Student),1,fp);

        printf("num: %d name:%s sex:%c math: %.2f english: %.2f
chinese: %.2f\n",
        tmp.num,tmp.name,tmp.sex,tmp.math,tmp.english,tmp.chinese);
    }

    fclose(fp);
    return 0;
}

```

16.5.5.课堂实战-管理系统

将链表作为内存数据模型，将文件作为数据库，将终端作为交互界面。读文件生成链表，修改链表写入文件。

结构类型设计如下：

```

struct student
{
    char name[30];
    char sex;
    int age;
    float score;
};

typedef struct node
{
    struct student data;      //数据域，已经不再是基本数据类型了。
    struct node *next;
}Node;

```

实现功能：

- 1，初始化现有数据到文件
- 2，读文件生链表

3 , 操作链表 (增加 , 删除)

4 , 写链表到文件

```
#include <stdio.h>

//1, 初始化数据库, 此时的数据库是文件
//2, 读数据库, 生成内存数据模型, 链表
//3, 增,查, 改, 删, 排序
//4, 更新数据库。

typedef struct student
{
    char name[30];
    char sex;
    int age;
    float score;
}Stu;

typedef struct _StuNode
{
    Stu data;
    struct _StuNode *next;
}StuNode;

void initData2File()
{
    Stu s[4] =
    {
        "liudehua",'x',50,100,
        "zhangxueyou",'x',60,98,
        "liming",'f',50,88,
        "guofucheng",'m',49,90,
    };

    FILE*pf = fopen("stu.data","w+");
    if(NULL == pf)
        exit(-1);

    fwrite((void*)s,sizeof(s),1,pf);
    fclose(pf);
    return ;
}
```

```
StuNode *createListFromFile(char *filePath)
{
    FILE *pf = fopen(filePath,"r+");
    if(NULL == pf)
        exit(-1);

    StuNode *head = (StuNode *)malloc(sizeof(StuNode));
    head->next = NULL;

    StuNode *cur = (StuNode *)malloc(sizeof(StuNode));
    while(fread((void*)&cur->data,sizeof(Stu),1,pf)) // 1 0
    {
        cur->next = head->next;
        head->next = cur;

        cur = (StuNode *)malloc(sizeof(StuNode));
    }
    free(cur);
    return head;
}

void traverseStuList(StuNode * head)
{
    printf("\t\t\t Student Management System\n\n");
    printf("\t\t\t\t\tCopyLeft\n\n");
    printf("\tname\t\tsex\tage\ttscore\n");
    head = head->next;
    while(head)
    {
        printf("\t%-10s\t%c\t%d\t%.2f\n",
               head->data.name,head->data.sex,
               head->data.age,head->data.score);
        head = head->next;
    }
    putchar(10);
}

void addListStu(StuNode *head)
{
    StuNode * cur = (StuNode *)malloc(sizeof(StuNode));
    printf("name :");
    scanf("%s",cur->data.name);

    getchar();
```

```
printf("sex :");
scanf("%c",&cur->data.sex);

printf("age  :");
scanf("%d",&cur->data.age);

printf("score:");
scanf("%f",&cur->data.score);

cur->next = head->next;
head->next = cur;
}

StuNode * searchListStu(StuNode *head)
{
    char name[30];
    printf("pls input your search name:");
    scanf("%s",name);

    head = head->next;
    while(head)
    {
        if(strcmp(head->data.name,name) == 0)
            break;
        head = head->next;
    }
    return head;
}

void deleteListNodeStu(StuNode *head)
{
    StuNode *pfind = searchListStu(head);
    if(pfind == NULL)
    {
        printf("您要删除的人不存在\n");
        getchar();getchar();
        return ;
    }
    while(head->next != pfind)
        head = head->next;
    head->next = pfind->next;
    free(pfind);
    return ;
}
```

```
int lenListStu(StuNode *head)
{
    int len =0;
    head = head->next;
    while(head)
    {
        len++;
        head = head->next;
    }
    return len;
}

void sortListStu(StuNode *head)
{
    int len = lenListStu(head);
    StuNode *prep,*p,*q;
    for(int i=0; i<len-1; i++)
    {
        prep = head;
        p = prep->next;
        q = p->next;
        for(int j=0; j<len-1-i; j++)
        {
            if(strcmp(p->data.name,q->data.name)>0)
            {
                prep->next = q;
                p->next = q->next;
                q->next = p;

                prep = q;
                q = p->next;
                continue;
            }
            prep = prep->next;
            p = p->next;
            q = p->next;
        }
    }
}

void saveListStu2File(StuNode *head,char *filePath)
{
```

```
FILE* pf = fopen(filePath,"w+");
if(NULL == pf)
    exit(-1);
head = head->next;
while(head)
{
    fwrite((void*)&head->data,sizeof(Stu),1,pf);
    head = head->next;
}
fclose(pf);
}

void destroyListStu(StuNode *head)
{
    StuNode *t;
    while(head)
    {
        t = head;
        head = head->next;
        free(t);
    }
}

void destroyListStu(head);
int main(int argc, char *argv[])
{
    //    initData2File();

    StuNode *head =createListFromFile("stu.data");

    while(1)
    {
        system("cls");
        traverseStuList(head);
        printf("1->add\t 2->search 3->delete 4->sort 5->exit\n");
        int choice;
        StuNode *pfind;
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
                addListStu(head);
                break;
            case 2:
```

```

        if(pfind = searchListStu(head))
        {
            printf("您要找的数据在本系统中\n");
            printf("\t%-10s\t%c\t%d\t%.2f\n",
                   pfind->data.name,pfind->data.sex,
                   pfind->data.age,pfind->data.score);
        }
        else
            printf("查无此人\n");
        getchar();getchar();
        break;
    case 3:
        deleteListNodeStu(head);
        break;
    case 4:
        sortListStu(head);
        break;
    case 5:
        saveListStu2File(head, "stu.data");
        destroyListStu(head);
        return 0;
    default:
        printf("您输错了\n");
    }
}

return 0;
}

```

16.6.文件指针偏移

16.6.1.rewind

函数声明	void rewind (FILE * stream);	
所在文件	stdio.h	
函数功能	将文件指针重新指向一个流的开头。	
参数及返回解析		
参数	FILE *	流文件句柄
返回值	void	无返回值

如果一个文件具有读写属性，当我们写完文件，需要读的时候，此时会遇到文件结尾现象。此时就需要 rewind。

16.6.2.ftell

函数声明	<code>long ftell (FILE * stream);</code>	
所在文件	<code>stdio.h</code>	
函数功能	得到流式文件的当前读写位置，其返回值是当前读写位置偏离文件头部的字节数。	
参数及返回解析		
参数	<code>FILE *</code>	流文件句柄
返回值	<code>int</code>	成功，返回当前读写位置偏离文件头部的字节数。失败，返回-1

16.6.3.fseek

函数声明	<code>int fseek (FILE * stream, long offset, int where);</code>		
所在文件	<code>stdio.h</code>		
函数功能	偏移文件指针。		
参数及返回解析			
参 数	<code>FILE *</code> <code>long</code> <code>int</code>	<code>FILE*</code> <code>stream</code> <code>long</code> <code>offset</code> <code>int</code> <code>where</code>	文件句柄 偏移量 偏移起始位置
返回值	<code>int</code>	成功返回 0 ， 失败返回-1	

常见的起始位置有宏定义：

<code>#define SEEK_CUR 1</code>	当前位置
<code>#define SEEK_END 2</code>	文件结尾
<code>#define SEEK_SET 0</code>	文件开头

其中 SEEK_SET,SEEK_CUR 和 SEEK_END 和依次为 0 , 1 和 2. 简言之：

<code>fseek(fp,100L,0);</code>	把 fp 指针移动到离文件开头 100 字节处;
<code>fseek(fp,100L,1);</code>	把 fp 指针移动到离文件当前位置 100 字节处;
<code>fseek(fp,-100L,2);</code>	把 fp 指针退回到离文件结尾 100 字节处。

```
#include <stdio.h>

int main(void)
{
    FILE *fp = fopen("sql.txt","w+");

    fputs("123456789",fp);
}
```

```

//    rewind(fp);
//    fseek(fp,0,SEEK_END);
fseek(fp,-5,SEEK_END);

int len = ftell(fp);
printf("len = %d\n",len);

return 0;
}

```

16.6.4.空洞文件

16.7.练习

16.7.1.文本文件内容排序(行数不能变)

将文件 sort.txt 中的内容排序以后输出 aftersort.txt

sort.txt

```

ba
nm
xyz
q

```

aftersort.txt

```

ab
mn
qxy
z

```

b	a	\n	n	m	\n	x	y	z	\n	q	\n
---	---	----	---	---	----	---	---	---	----	---	----

```

int main()
{
    FILE* fp = fopen("sort.txt","r");
    if(fp == NULL)
    {
        printf("open error\n");
        return -1;
    }
}

```

```
char buf[30];
char sort[100] = {0};
while(fgets(buf,30,fp) != NULL)
{
    strcat(sort,buf);
}
printf("%s",sort);
int len = strlen(sort);
int tmp;
int i,j;
for(i=0; i<len-1;i++)
{
    for(j=0; j<len-1-i; j++)
    {
        if(j+2 > len -1)
            break;
        if(sort[j+1] == '\n')
        {
            if(sort[j]>sort[j+2])
            {
                tmp = sort[j];
                sort[j] = sort[j+2];
                sort[j+2] = tmp;
            }
            j++;
        }
        else
        {
            if(sort[j]>sort[j+1])
            {
                tmp = sort[j];
                sort[j] = sort[j+1];
                sort[j+1] = tmp;
            }
        }
    }
}
printf("%s",sort);
fclose(fp);
return 0;
}
```

16.7.2.修改文件内容

文件中的内容，是以结构的形式写入的，现在要求读第三个和第五个学生成绩，修改学生成绩，并回写。

```
typedef struct student
{
    int num;
    char name[30];
    char sex;
    float math;
    float english;
    float chinese;
}Student;
```

17.位操作(Bit Operation)

17.1.位操作与逻辑操作

位操作不同于逻辑操作，逻辑操作是一种**整体的操作**，而位操作是针对**内部数据位补码**的操作。逻辑操作的世界里，只有真与假（零和非零），而位操作的世界里按位论真假（1和0）。运算符也不同，如下。

位运算符	逻辑运算符
&	&&
~	!
^	
<< >>	
&= = ^= >>= <<=	

17.2.数据的二进制表示

17.2.1. 8位二进制数据的补码

8位数据的排列组合	数值
0000 0000	0
0000 0001	1
0000 0010	2
0000 0011	3
• • • •	
• • • •	
0111 1111	127
1000 0000	-128
• • • •	
• • • •	
1111 1100	-3
1111 1110	-2
1111 1111	-1

17.2.2.二进制打印

```
void bin8bdis(char data)
{
    int i=8;
    while(i--)
    {
        if(data & (1<<i))
            printf("1");
        else
            printf("0");

        if(i%4==0)
        {
            if(i%8==0)
                printf(" ");
            else
                printf("-");
        }
    }
    putchar(10);
}

void bin32bdis(int data)
{
    int i=32;
    while(i--)      //32---1
    {
        if(data & (1<<i))  //31--0
            printf("1");
        else
            printf("0");

        if(i%4==0)
        {
            if(i%8==0)
                printf(" ");
            else
                printf("-");
        }
    }
    putchar(10);
}
```

17.3.位操作

17.3.1.位与(&)

按位与 : &

格式 : $x \& y$

规则 : 对应位均为 1 时才为 1 , 否则为 0。

x	y	$x \& y$
0	0	0
1	0	0
0	1	0
1	1	1

例如 : $3 \& 11 = 3$

$$\begin{array}{r} 0011 \\ \& 1011 \\ \hline 0011 = 3 \end{array}$$

参考用途 :

在某些位保持不变的情况下 , 将其余位置 0。

17.3.2.位或(|)

按位或 : |

格式 : $x | y$

规则 : 对应位均为 0 时才为 0 , 否则为 1。

x	y	$x y$
0	0	0
1	0	1
0	1	1
1	1	1

例如 : $3 | 9 = 11$

$$\begin{array}{r} 0011 \\ | 1001 \\ \hline 1011 = 11 \end{array}$$

参考用途 :

在某些位保持不变的情况下 , 将其余位置 1

17.3.3.位取反(\sim)

按位求反： \sim

格式 : $\sim y$

规则 : 各位翻转，即原来为 1 的位变成 0，原来为 0 的置 1

x	$\sim x$
0	1
1	0

例如 : $\sim 3=12$

~ 0011

1100=12

参考用途：

间接地构造某个数,省却计算的麻烦，以增强程序的可读性。

17.3.4.位异或(\wedge)

按位异或： \wedge

格式 : $x \wedge y$

规则 : 对应相同时 0，不同时则为 1。

x	y	$x \wedge y$
0	0	0
1	0	1
0	1	1
1	1	0

例如 : $3 \wedge 9=10$

$$\begin{array}{r} 0011 \\ \wedge \quad 1001 \\ \hline 1010=10 \end{array}$$

参考用途：

相同者归零，**相异者或**。在某些位保持不变的情况下，将其余位取反。

自身异或，清零。

思考：有没有同与，同或，异与这样运算呢？

17.3.5.左移(<<)

按位左移 : <<

格式 : $x << \text{位数}$

规则 : 使操作数的各位左移，低位补 0，高位溢出。

备注 : 位数为非负整数，且默认对 32 求余

x	y = x<<1	x
0000 0001	0000 0010	0000 0001

例如 : $5 << 2 = 20$: 0101 → 10100

```
#include<stdio.h>

int main()
{
    int a = 1;

    bin32bdis(a);
    bin32bdis(a<<1);

    bin32bdis(a<<31);
    bin32bdis(a<<32);
    bin32bdis(a<<33); //33%32

    //int a = (1<<31) +1;

    //bin32bdis(a);
    //bin32bdis(a<<1);

    //bin32bdis(a<<31);
    //bin32bdis(a<<32);
    //bin32bdis(a<<33); //33%32

    return 0;
}
```

17.3.6. 右移(>>)

按位右移 : >>

格式 : x>>位数

规则 : 使操作数的各位右移，移出的低位舍弃；

高位 : 对无符号数和有符号中的正数补 0；有符号数中的负数，取决于所使用的系统： 补 0 的称为“逻辑右移”，补 1 的称为“算术右移”。

说明 : x、y 和“位数”等操作数，都只能是整型(允许字符型数据)。

备注 : 位数为非负整数，且默认对 32 求余

x	y = x>>1	x
1000 0000	0100 0000	1000 0000

例如 : $5 << 2 = 20$: 0101 → 10100 ,
 $20 >> 2 = 5$: 10100 → 00101。

17.3.7.优先级

17.3.7.1.计算如下值：

```
1 << 3 + 2;
0x5 & 0xa == 0
1 & & 0xa & 0x5
0x5 & 1 << 3
```

17.3.7.2.简记结论

() > 成员运算 > (!) > 算术 > 关系 > 逻辑 > 赋值 ,
() > 成员运算 > (^/!) > 算术 > 关系 > (>> <<) 位逻辑(& | ^) > 逻辑 > 赋值 ,

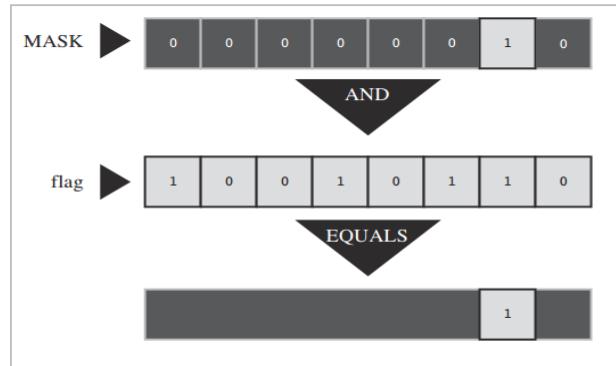
17.4.应用

17.4.1.掩码(mask)



掩盖掉一些东西，然后，留下一些东西，就是掩码存在的意义。

```
unsigned char MASK = 1<<1;
unsigned char flag = 0x96
```



17.4.2. 功能

- ① 打开位（位置 1） **flag |= MASK;**
- ② 关闭位（位置 0） **flag &= ~MASK;**
- ③ 转置位（位反转） **flag ^= MASK;**
- ④ 查看某一位的值 **if((flag&MASK)==MASK)**

17.4.3. 生成

```

int mask = 0;
int mask = (1<<6) |(1<<5)| (1<<4) |(1<<3)| (1<<2);

0100 0000
0010 0000
0001 0000
0000 1000
0000 0100

///////////////////////////////
int mask =0;
for(int i=6; i>=2; i--)
{
    mask |= (1<<i);
}

```

17.5.课堂实战

17.5.1. 输出位数

题目：从键盘上输入 1 个正整数给 int 变量 num，输出由 3~6 位构成的数（从低 0 号开始编号）

基本思路：

1. 截取 3~6 位的数，位移到 0~3 位

- a)构建 3~6 位上为 1 其余为 0 的数
 - b)位与输入数
 - c)得到的结果右移 3 位
- 2.先将 3~6 位移到 0~3 位，截取 0~3 位
- a)输入数右移 3 位
 - b)构建 0~3 位为 1 其余为 0 的数
 - c)位与，得到结果

17.5.2. 判断一个数是不是 2 的幂数。

17.5.3. 实现循环移位

`void circleMove(int *data ,int n);`当 $n > 0$ 的时候左移， $n < 0$ 的时候循环右移。

```
void circleMove(int *pdata, int n) // unsigned int *pdata
{
    int m;                      //用于产生掩码
    m = n>0?n:-n;
    unsigned int mask = 0;
    while(m--)
        mask |= (1<<m);

    if(n>0)                  //左循环
    {
        *pdata = (*pdata<<n) | mask &(*pdata>>(sizeof(*pdata)*8-n));
        //凡右移->清高位
    }
    else                      //右循环
    {
        *pdata = ((*pdata>>-n)) & ~mask<<(32-(-n))) | (*pdata)<<(32-(-n));
        //凡右移->清高位
    }
}
```

提示：我们实现的整体移动 n 位版本，也可以一次移动一位，移动 n 次。

17.6. 提高篇

17.6.1. 无参交换

17.6.1.1. 有参交换

引入变量，增加了空间上的开销。

```
void swap(int *p1,int *p2)
{
    int tmp;
```

```

tmp = *p1;
*p1 = *p2;
*p2 = tmp;
}

```

求和 当要比较的两个数据较大时，求和易发生溢出风险。

```

void swap(int *p1,int *p2)
{
//缺陷，仍然引进了参数
//    int tmp = *p1 + *p2;
//    *p2 = tmp - *p2;
//    *p1 = tmp - *p1;

    *p1 = *p1+*p2;
    *p2 = *p1 -*p2;
    *p1 = *p1 -*p2;
}

```

17.6.1.2.无参交换

x, y 和 $x \wedge y$ ，此三者，可知其中之二，求另外一个数。

x	y	$x \wedge y$
0	0	0
1	0	1
0	1	1
1	1	0

```

void swap(int *p1,int *p2)
{
//    int tmp = *p1 ^*p2;
//    *p2 = tmp ^ *p2;
//    *p1 = tmp ^ *p1;

    *p1 = *p1^*p2;
    *p2 = *p1^*p2;
    *p1 = *p1^*p2;
}

```

无溢出，是交换数据的最高境界。

17.6.2.异或加密(文本与二进制)

1 , key 为单个字符。

注意，加密过程中**相同字符异或**或有'\0'的出现。要滤过以防数据截断，因为我们依然会它当成字符串进行处理。

2 , key 为字符串。

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

void encrypt(char * secret,char *key)
{
    int kn = strlen(key);
    int i = 0;
    while(*secret != '\0')
    {
        if(*secret == key[i])    //注意 key[i++] 会导致什么后果
        {
            secret++;
            i++;
        }
        else
        {
            *secret++ ^= key[i];
            i++;
        }
        if(i%kn ==0)
            i=0;
    }
}
void de_encrypt(char * secret,char *key)
{
    int kn = strlen(key);
    int i = 0;
    while(*secret != '\0')
    {
        if(*secret == key[i])    //注意 key[i++] 会导致什么后果
        {
            secret++;
            i++;
        }
        else
        {
            *secret++ ^= key[i];
            i++;
        }
        if(i%kn ==0)
```

```

        i=0;
    }
}

int main()
{
    char buf[] = "china is great";
    char key[] = "nimeiya";
    encrypt(buf,key);
    printf("%s\n",buf);
    encrypt(buf,key);
    printf("%s\n",buf);
    return 0;
}

```

17.6.3. 循环移位加密(文本与二进制)

位运算的加密应用，这才是真正意义上的加密的开始。解决了，加减法加密溢出的问题：

```

void encode(char *secret)
{
    int n = strlen(secret);
    for(int i=0; i<n; i++)
    {
        unsigned char ch = secret[i];
        ch = 0xff & (((ch &(1<<7))>>7) |(ch<<1));
        secret[i] = ch;
    }
}

void decode(char *secret)
{
    int n = strlen(secret);
    for(int i=0; i<=n; i++)
    {
        unsigned char ch = secret[i];
        ch = 0xff & (((ch&1)<<7) |(ch>>1));
        secret[i] = ch;
    }
}

int main()
{
    char buf[] = "china is great";
    encode(buf);
}

```

```
printf("%s\n",buf);
decode(buf);
printf("%s\n",buf);
return 0;
}
```

17.7.练习

17.7.1.打印数据的二进制形式（32位）

17.7.2.反转一个数据的最后 n 位。

17.7.3.依数据的符号们判断正负。

```
int is_signal(int *p)
```

17.7.4.练习加密二进制文件

对于视频和图片文件，进行加密和解密，可以采用循环移位或是异或加密。注：此时判断条件，应该是什么样了呢？

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

void encrypt(char *buf,int n,char *passwd)
{
    int len = strlen(passwd);
    int j=0;
    for(int i=0; i<n; i++)
    {
        buf[i] ^= passwd[j++];
        if(j%len == 0)
            j=0;
    }
}

int main(void)
{
    FILE *fpr = fopen("1.jpg","rb");

    FILE *fpw = fopen("2.jpg","wb");
```

```
char buf[1024] = {0};

char *passwd = "i love you";

int n = 0;
while((n = fread((void*)buf,1,1024,fpr))>0)
{
    encrypt(buf,n,passwd);
    fwrite((void*)buf,n,1,fpw);
}

fclose(fpr);
fclose(fpw);

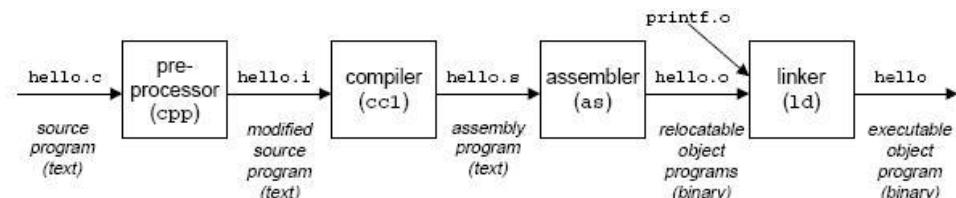
return 0;
}
```

18.预处理(Pre Compile)

18.1.发生时机

预处理操作，**不是c语言语句**，故语句末尾没有分号，在预处理阶段完成，本质是替换操作。

发生时段：



18.2.宏(Macro)

18.2.1.不带参宏

18.2.1.1.宏常量

#define 定义的宏，只能在**一行内**表达(换行符表示结束而非空格)，如果想多行表达，则需要加续行符。

```
#define PI 3.14\
15926
int main(void)
{
    printf("%f",PI);
    return 0;
}
```

宏常量，常被 const/ enum 变量取代，**用于定义文件路径**则被常用。

```
#define FILEPATH "E:\\English\\listen_to_this\\listen_to_this_3"

#define ERR_EXIT(m)\
do\
{\
    printf("Err:%s",m);\
    exit(-1);\
}while(0) //此处的分号，可有可无
```

宏常量的缺陷

```
#define N 2+3 // #define N (2+3)
int main(void)
{
    int num = N*2;
    return 0;
```

```
}
```

解决这一些问题，要不吝惜使用括号。

18.2.1.2.宏类型

宏可以给类型起别名，因其缺点，常被 `typedef` 取代

```
#define CHARP char *
int main(void)
{
    CHARP p,q;
    printf("p = %d q = %d\n",sizeof(p),sizeof(q));
    return 0;
}
```

18.2.2.带参宏(宏函数)

我们常将**短小精悍**的函数进行宏化，这样可以嵌入到代码中，**减少调用的开销**。但是代价就是，编译出的文件可能会变大。

宏函数常常一行表达不完，如多行，为了形式上的方便，多采用续行符进行接续。

18.2.2.1.宏函数

◆例 `#define S(a,b) a*b`
`area=S(3,2);`

宏展开：`area=3*2;`

注：宏名和参数间不能用空格

◆改进版：

例 `#define POWER(x) x*x`
`x=4; y=6;`
`z=POWER(x+y);`

宏展开：`z=x+y*x+y;`

一般写成：`#define POWER(x) ((x)*(x))`

宏展开：`z=((x+y)*(x+y));`

`define` 是个演技高超的**替身**演员。 要搞定它其实很简单，别吝啬括号就行了。

◆常见宏函数

```
#define MAX(a,b) (a>b)?a:b
```

◆宏函数与普通函数的区别

```
#define SQ(y) ((y)*(y))

int main()
{
    int i=1;
    while(i<=5)
        printf("%d\n",SQ(i++));
    return 0;
}

int main()
{
    int i=1;
    while(i<=5)
        printf("%d\n",sq(i++));
    return 0;
}

int sq(int y)
{
    return((y)*(y));
}
```

尽量少用宏函数，能看的懂别人写的宏函数即可。C++中的 inline 函数已经取代了宏函数作用。

18.2.2.2.宏出错处理函数

```
#define ERR_EXIT(m)\n\
do\\
{\\\n\
    printf("Err:%s",m);\\
    exit(-1);\\
}while(0)          //此处的分号，可有可无
```

18.2.3.取消宏

18.2.3.1.取消宏常量

```
#include <stdio.h>\n#define MAX 23\nint main()\n{\n    printf("%d\n",MAX);\n    printf("%d\n",MAX);\n    printf("%d\n",MAX);\n#undef MAX\n    printf("%d\n",MAX);\n    return 0;\n}
```

18.2.3.2.取消宏函数

```
#include <stdio.h>\n\n#define MIN(x,y) x+y\nint main()\n{\n    printf("%d\n",MAX);\n    printf("%d\n",MAX);
```

```

printf("%d\n",MAX);

printf("%d\n",MIN(2,3));
#define MIN //权需要指定宏函数名即可
printf("%d\n",MIN(2,3));
return 0;
}

```

18.3.条件编译(Condition Compile)

依据条件，判断哪些程序段参与编译。

18.3.1.单双路(#ifdef / #ifndef #else #endif)

```

#define X86
void main()
{
#ifdef X86 // #ifndef
    printf("xxxxxxxx\n");
    printf("xxxxxxxx\n");
    printf("xxxxxxxx\n");
    printf("xxxxxxxx\n");
    printf("xxxxxxxx\n");
#else // #else
    printf("oooooooo\n");
#endif
}

```

18.3.2.单双多路(#if #elif #endif)

18.3.2.1.支持不同编译器版本

```
#if __cplusplus >= 201103L
```

18.3.2.2.编译跨平台程序

在写编译跨平台程序时，经常见到这样的语句。

```

#define X86 100
#define MIPS 200
#define POWERPC 300

#define MACHINE POWERPC

void main()
{
#if MACHINE == X86

```

```

printf("xxxxxxxx\n");
printf("xxxxxxxx\n");
printf("xxxxxxxx\n");

#if MACHINE == MIPS
    printf("oooooooo\n");
#elif MACHINE == POWERPC
    printf("xxxxoooo\n");
    printf("xxxxooo\n");
#endif
}

```

skynet 框架中 socket_poll.h

```

#ifndef socket_poll_h
#define socket_poll_h
#include <stdbool.h>
typedef int poll_fd;
struct event {
    void * s;
    bool read;
    bool write;
};
static bool sp_invalid(poll_fd fd);
static poll_fd sp_create();
static void sp_release(poll_fd fd);
static int sp_add(poll_fd fd, int sock, void *ud);
static void sp_del(poll_fd fd, int sock);
static void sp_write(poll_fd fd, int sock, void *ud, bool enable);
static int sp_wait(poll_fd fd, struct event *e, int max);
static void sp_nonblocking(int sock);
#ifdef __linux__
#include "socket_epoll.h"
#endif
#if defined(__APPLE__) || defined(__FreeBSD__) || defined(__OpenBSD__) || defined (__NetBSD__)
#include "socket_kqueue.h"
#endif
#endif

```

18.3.3. 编译期指定宏 gcc -D

-D name, Predefine name as a macro, with definition 1.

```

#include <stdio.h>
#include <stdlib.h>
int main(int argc, char* argv[])

```

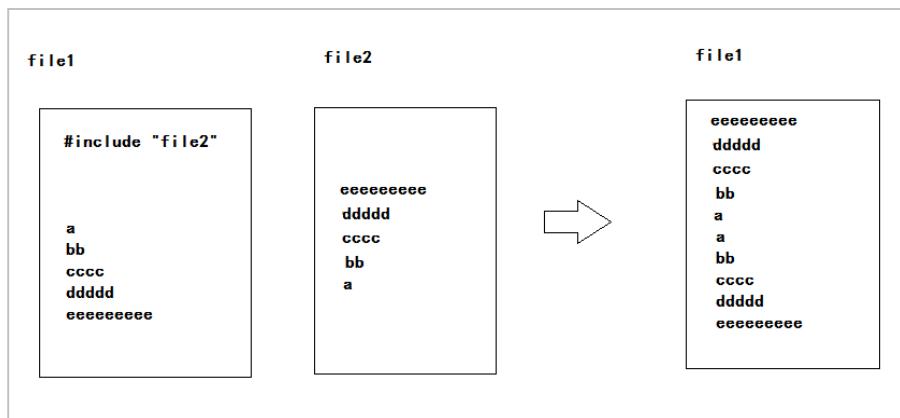
```
{
    #ifdef DEBUG
        printf("gcc 的-D 选项测试\n");
    #endif

    return 0;
}
//QMAKE_CFLAGS += -D DEBUG
```

18.4.头文件包含(#include)

18.4.1.包含的意义

全写入，被包含的文件中。



包含是支持嵌套的。

18.4.2.包含的方式

18.4.2.1.方式<>

#include<stdio.h>, 从系统指定路径中搜索包含头文件, linux 中的系统路径为 (/usr/include)

```
#include <stdio.h> -> #include "stdio.h" ->#include <stdio.h>
```

18.4.2.2.方式" "

#include“myString.h”, 从工程当前路径中搜索包含头文件, 如果当前工程路径下没有的话, 则到系统路径下搜索包含。

```
#include"myString.h" -> #include<myString.h>
```

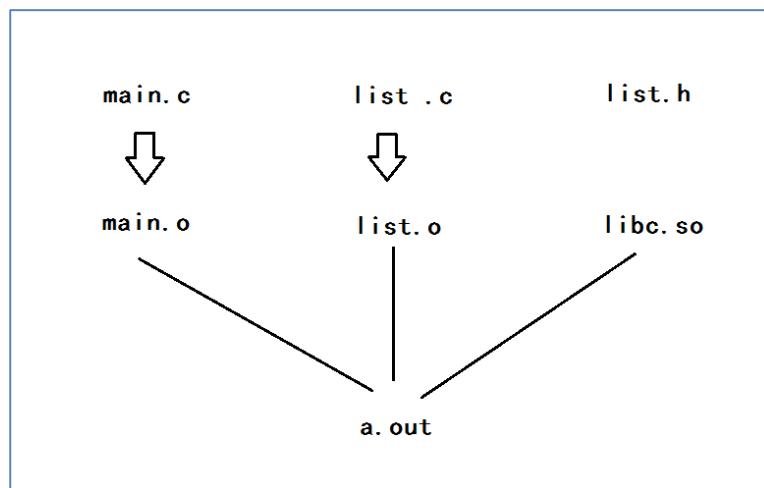
18.4.3.多文件编程

18.4.3.1.多文件编程意义

多文件编程可以至少有两大好处，一是，方便管理，协同开发。二是，便于分享与加密(作成函数库)。

18.4.3.2.多文件编程的前题

c 语言是以文件为单位进行编译的，编译期只需要函数声明即可。链接阶段提供实现就可以完成生成可执行文件。

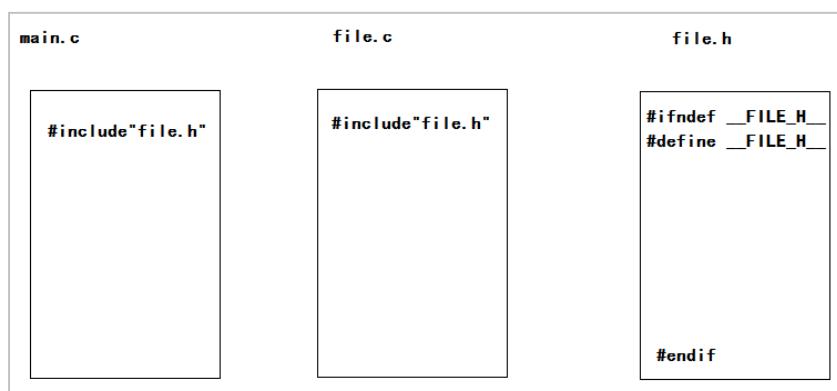


18.4.3.3.实例

将前面学习的链表，改为多文件编程方式。包含三个文件，分别是 main.c list.c list.h。

18.4.4.定义头文件

18.4.4.1.定义头文件



18.4.4.2.头文件自包含

.c 文件中存在相互调用的关系，自包含可以免去了多余的前向声明。

main.c

```
#include <stdio.h>
#include "str.h"

int main()
{
    char buf[] = "zxyaabc";
    mySortStr(buf);
    printf("%s\n",buf);
    return 0;
}
```

str.h

```
#ifndef STR_H
#define STR_H

int myStrlen(char *p);
void mySortStr(char *p);

#endif // STR_H
```

str.c

```
#include "str.h"

void mySortStr(char *p)
{
    int i,j;
    int n = myStrlen(p);
    for(i=0; i<n-1; i++)
    {
        for(j=i+1;j<n; j++)
        {
            if(p[i]>p[j])
            {
                p[i] = p[i]^p[j];
                p[j] = p[i]^p[j];
                p[i] = p[i]^p[j];
            }
        }
    }

    int myStrlen(char *p)
```

```
{  
    int n=0;  
    while(*p++!='\0')  
        n++;  
    return n;  
}
```

18.4.4.3.避免头文件重复包含

假设文件名叫 xx.h

```
#ifndef __XX_H__  
#define __XX_H__  
  
//数据类型声明  
//函数声明  
  
#endif
```

案例：

stu.h

```
#ifndef STU_H  
#define STU_H  
  
struct Stu  
{  
    char name[30];  
    float score;  
};  
  
void display(struct Stu s);  
  
#endif // STU_H
```

sort.h

```
#ifndef SORT_H  
#define SORT_H  
  
  
#include "stu.h"  
  
void sort(struct Stu *s);
```

```
#endif // SORT_H
```

main.c

```
#include <stdio.h>
#include "stu.h"
#include "sort.h"

int main(void)
{
    struct Stu s[5];
    display(s[0]);
    sort(s);
    return 0;
}
```

18.4.5.相互包含的避免

18.5.其它

18.5.1.#运算符 利用宏创建字符串

将替换符 **字符串化**，解决字符串中，不可被替换的参数问题。字符串如下的书写也是合理的。

```
char buf[] = "china ""is ""great";
```

```
//#define str(x) #x
//#define str(x) "aaaaaaaaaxaaaaaaaa"
#define str(x) "aaaaaaaa"#x"aaaaaaaa"
int main()
{
    printf("%s\n",str(100));
    return 0;
}
```

```
#include <stdio.h>
#define PSQR(x) printf("the sqr of "#x"= %d\n", (x)*(x))

int main()
{
```

```

int y =5;
PSQR(y);
PSQR(2+4);
return 0;
}

```

引号中的字符串中的 `x` 被看作普通的文本，而不是被看作一个可被替换的语言符号。`#`号用作一个预处理运算符，它可以把待替换符转化为字符串。

18.5.2.##运算符 预处理的粘和剂

解决了，参数变量与宏展开，无法一一对应的问题。

```

#define sum(a,b) (aa+bb)
#define sum(a,b) (a##a+b##b)
int main()
{
    printf("%d\n",sum(2,3));
    return 0;
}

```

```

#define XNAME(n) x##n
int main()
{
    int XNAME(1) = 14;
    int XNAME(2) = 20;
    printf("x1 = %d,x2 = %d\n",x1,x2);
    return 0;
}

```

综合：

`#`解决了双引号中无法替换问题，`##`解决了非双引号中粘连无法替换的问题。

```

#include <stdio.h>
#define XNAME(n) x##n

#define PRINT_XN(n) printf("x"#n" = %d\n", x##n)

int main()
{
    int XNAME(1) = 14;
    int XNAME(2) = 20;
    printf("x1 = %d,x2 = %d\n",x1,x2);
    PRINT_XN(1); //隐藏了变量的声明和使用 点赞
    PRINT_XN(2);
}

```

```
    return 0;
}
```

18.5.3. 预定义宏

`_DATE_` 进行预处理的日期（“MMmm dd yyyy” 形式的字符串文字）
`_FILE_` 代表当前源代码文件名的字符串文字
`_LINE_` 代表当前源代码中的行号的整数常量
`_TIME_` 源文件编译时间，格式 “hh : mm : ss”
`_func_` 当前所在函数名

在打印调试信息时打印这两个宏 `_FILE_` `_LINE_` 可以给开发者非常有用的提示

```
#include <stdio.h>
#include <stdlib.h>
void why_me();
int main()
{
    printf( "The file is %s.\n", __FILE__ ); // --
    printf( "The date is %s.\n", __DATE__ );
    printf( "The time is %s.\n", __TIME__ );
    printf( "This is line %d.\n", __LINE__ );
    printf( "This function is %s.\n", __func__ );
    why_me();
    return 0;
}

void why_me()
{
    printf( "This function is %s\n", __func__ );
    printf( "The file is %s.\n", __FILE__ );
    printf( "This is line %d.\n", __LINE__ );
}
```

18.6. 练习

18.6.1. 宏展开有次序吗？

```
#include <stdio.h>
#define f(a,b) a##b
#define g(a) #a
#define h(a) g(a)
```

```
int main()
{
    printf("%s\n", h(f(1,2)));
    printf("%s\n", g(f(1,2)));
    return 0;
}
```

19.项目

19.1.项目简介

19.2.概要设计

19.3.详细设计

19.4.实现

 19.4.1.界面菜单实现

 19.4.2.链表实现

 19.4.3.文件读写实现

19.5.SVN 版本管理控制

20.附录(Appendix)

20.1.参考书目

20.2.ascii 码表详解

ASCII 值	控制字符	ASCII 值	控制字符	ASCII 值	控制字符	ASCII 值	控制字符
0	NUT	32	(space)	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	\$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	,	71	G	103	g
8	BS	40	(72	H	104	h
9	HT	41)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	TB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[123	{
28	FS	60	<	92	/	124	
29	GS	61	=	93]	125	}
30	RS	62	>	94	^	126	`
31	US	63	?	95	_	127	DEL

20.3.ascii 特殊字符解释

NUL 空	VT 垂直制表	SYN 空转同步
STX 正文开始	CR 回车	CAN 作废
ETX 正文结束	SO 移位输出	EM 纸尽
EOY 传输结束	SI 移位输入	SUB 换置
ENQ 询问字符	DLE 空格	ESC 换码
ACK 承认	DC1 设备控制 1	FS 文字分隔符
BEL 报警	DC2 设备控制 2	GS 组分隔符
BS 退一格	DC3 设备控制 3	RS 记录分隔符
HT 横向列表	DC4 设备控制 4	US 单元分隔符
LF 换行	NAK 否定	DEL 删除

20.4. 运算符优先级

Precedence	Operator	Description	Associativity
1	<code>++ --</code> <code>()</code> <code>[]</code> <code>.</code> <code>-></code> <code>(type){list}</code>	Suffix/postfix increment and decrement Function call Array subscripting Structure and union member access Structure and union member access through pointer Compound literal(C99)	Left-to-right
2	<code>++ --</code> <code>+ -</code> <code>! ~</code> <code>(type)</code> <code>*</code> <code>&</code> <code>sizeof</code> <code>_Alignof</code>	Prefix increment and decrement Unary plus and minus Logical NOT and bitwise NOT Type cast Indirection (dereference) Address-of Size-of Alignment requirement(C11)	Right-to-left
3	<code>* / %</code>	Multiplication, division, and remainder	Left-to-right
4	<code>+ -</code>	Addition and subtraction	
5	<code><< >></code>	Bitwise left shift and right shift	
6	<code>< <=</code> <code>> >=</code>	For relational operators <code><</code> and <code>≤</code> respectively For relational operators <code>></code> and <code>≥</code> respectively	
7	<code>== !=</code>	For relational <code>=</code> and <code>≠</code> respectively	
8	<code>&</code>	Bitwise AND	
9	<code>^</code>	Bitwise XOR (exclusive or)	
10	<code> </code>	Bitwise OR (inclusive or)	
11	<code>&&</code>	Logical AND	
12	<code> </code>	Logical OR	
13	<code>? :</code>	Ternary conditional	Right-to-Left
14	<code>=</code> <code>+= -=</code> <code>*= /= %=</code> <code><=>=</code> <code>&= ^= =</code>	Simple assignment Assignment by sum and difference Assignment by product, quotient, and remainder Assignment by bitwise left shift and right shift Assignment by bitwise AND, XOR, and OR	
15	<code>,</code>	Comma	Left-to-right

20.5.易错优先级集锦

优先级问题	表达式	经常误认为的结果	实际结果
. 的优先级高于* ->操作符用于消除这个问题	*p. f	p 所指对象的字段 f (*p). f	对 p 取 f 偏移，作为指针，然后进行解除引用操作。*(p. f)
[] 高于*	int *ap[]	ap 是个指向 int 数组的指针 int (*ap) []	ap 是个元素为 int 指针的数组 int *(ap [])
函数() 高于*	int *fp()	fp 是个函数指针，所指函数返回 int。 int (*fp) ()	fp 是个函数，返回 int * int *(fp ())
== 和 != 高于位操作	(val & mask != 0)	(val & mask) != 0	val & (mask != 0)
== 和 != 高于赋值符	c = getchar() != EOF	(c = getchar()) != EOF	c = (getchar()) != EOF
算术运算符高于位移运算符	msb << 4 + lsb	(msb << 4) + lsb	msb << (4 + lsb)
逗号运算符在所有运算符中优先级最低	i = 1, 2	i = (1, 2)	(i = 1), 2

20.6.vs2013 中使用技巧

20.6.1.禁用_s 版本函数的方法

- 在所有的包含头文件之前加入 `#define _CRT_SECURE_NO_WARNINGS` 以禁用不安全警告
- 在项目属性里设置 C/C++ 高级项目中编辑禁用特定的警告：**4996**
- 写预处理语句禁用 **4996** 警告 `#pragma warning(disable:4996)`

20.7.附练习答案

20.7.1.第六章程序设计

20.7.1.1.百钱买百鸡的问题

20.7.2.第七章数组

20.7.2.1.数组去重

```

int main()
{
    int a[10] = {1,1,1,1,1,1,1,1,2,2};

    int idx =10;

    for(int i=0; i<idx-1; i++)
    {

```

```

if(a[i] == a[i+1])
{
    for(int j=i+1; j<10; j++)
    {
        a[j-1] = a[j];
    }
    idx--;
    i--;
}
}

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

return 0;
}

```

20.7.2.2.数组合并有序

```

int main()
{
    int a[M] = {1,3,5,7,9};
    int b[N] = {4,6,10};

    int c[M+N];
    int i=0,j=0,ck =0;

    while( i < M && j < N)
    {
        if(a[i] < b[j])
            c[ck++] =a[i++];
        else
            c[ck++] = b[j++];
    }

    if(i == M)          //可优化
    {
        for(;j<N; j++) // 这样的 for 应该换 while
            c[ck++] = b[j];
    }
    if(j == N)          //可优化
    {
        for(;i<M; i++)
    }
}

```

```
    c[ck++] = a[i];  
}  
  
for(int i=0; i<ck; i++)  
    printf("%d\n",c[i]);  
  
return 0;  
}
```

20.8.章节思维导图

20.8.1.常变量与数据类型

20.8.2.运算符与表达式

20.8.3.数组

20.8.4.函数

20.8.5.作用域与运算符

20.8.6.字符串

20.8.7.文件

20.8.8.位运算

20.8.9.预处理