@ME: sillyman2013@hotmail.com
2015.9.22 Aslm.
本知识点整理对象《C语言入门经典（第五版）》

C语言 知识整理

- 关键字
- 处理文件
- 基本输入和输出操作
- 字符串
  - 转义序列，eg:打印" 必须为 \"、打印 \ 必须为 \\\
  - 存储字符串时，分配的内存必须比存储的字符多一个字节存储结尾字符 \0
  - 字符串检测函数:strlen()，结尾必须是 \0.【安全版本为strlrn_s()】
  - 字符串复制函数:strcpy()【安全版本为strcpy_s()】
  - 连接字符串函数:strcat()、strncat()【安全版本为:strcat_s()、strncat_s()】；进行字符串操作时，初始化空字符数组时必须赋值 eg:char Str[4] ={"\0"}（空字符串）
    '\0'是必须的，用于程序(C语言)访问字符串数Str[]，而Str[]还没有被赋新赋值新的字符串的情况下，判断字符串是否结束；没有初始化赋值 '\0' 时程序以字符串模式访问时是无法判断空字符数组 Str[]是否结束的！
  - 1、比较字符串函数:strcmp()
    2、搜索字符串函数:strchr()返回地址
    3、字符串中查找子字符串函数:strstr()返回地址
    4、单元化字符串函数:strtok()，将字符串按指定字符分解，调用前后的字符串数组是不同的！
    5、读取执行符函数:fgets()，可避免读取字符串时换行符被省略
    6、分析和转换字符串:包括字符大小写转换、字符串转换数值、控制字符等等，参见P-221
- 数组
  - 数组初始化：只需给第一个元素赋值为0，整个数组初始化就为0.
  - 数组大小的确定：int Temp[5]={0}; sizeof(Temp)为数组所占的总的字节数；sizeof(Temp)/sizeof(Temp[0])为数组元素个数
  - 多维数组：重点在于指针的操作，多维数组的地址！
- 内存
- 指针
- 数据类型
  - 无符号整型
    - unsigned char 1字节
    - unsigned short int OR unsigned short（短整型）2字节
    - unsigned int(长整型) 4字节（后缀:U）
    - unsigned long int OR unsigned long 4字节（后缀:U\UL）
    - unsigned long long int OR unsigned long long 8字节（后缀:U\UL\ULL）
  - 有符号整型
    - signed char 1字节
    - short int（短整型）2字节
    - int(长整型) 4字节
    - long int 4字节（后缀:L）
    - long long int 8字节(后缀:L\LL)
  - 浮点数
    - float 4字节（精确到6到7位小数）(后缀:F)
    - double 8字节（精确到15位小数）(后缀:F
    - long double （精确到18位小数）(后缀:F
  - 其他
    - 枚举（注意，枚举值是可以相同的，也可以赋值，编译器会自动赋递续值，有时候，部分值有明确值，编译器可能会赋相同值 参见P-63）
    - 布尔
- 运算符
  - 算术运算符 +、-、*、....
  - 逻辑运算符 &&、||、！
  - 条件运算符 ？（三目条件运算符）eg: condition?expression1:expressin2(条件为真执行第一个表达式，反之)
- 语句
  - 条件判断
    - if、if-else
    - switch
  - 循环
    - for
    - while、do-while
  - 其他
    - break: 只能跳出当前循环
    - continue ：从该语句开始跳过当前迭代，继续执行下一个迭代
    - goto：从该语句开始跳出外层指定点

- 十六进制 OR 八进制 OR 十进制
  - eg:0xFF
  - eg:1111111h
  - eg: 10U

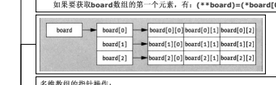
1、后缀：编程时使用，用于编译器识别常量类型，避免溢出
2、常量占内存字节可用 sizeof 运算符计算
3、数据类型的极限值 在c标准中有定义 如，char类型的上下限：CHAR_MAX 、 CHAR_MIN

编译器给变量分配内存地址时，其地址都是变量字节数的倍数。

运算符有优先级 参见 P-95 (实际运用时采用（）、[]有利于程序阅读)

count++ 与 ++cont 的区别：
1、count++是先引用count值再递增，++count是先递增再引用count值；
2、从程序效率来说，单独的递增是++count的效率更高。因为在++count数值转换中不用新建一个缓存变量（内存）来保存当前值。

指针

- 初始化
  - void *p_Test= NULL 声明了任意类型的数据项地址的引用指针
  - 指针创建时必须初始化 eg: int *p_Test = NULL，不然会成为野指针
  - 1、指针初始化时 右边必须是地址，不是常量！ eg: int *p_Tset = &Num;
    《初始化的操作流程总：声明指针 int *p_Test ，对指针赋值地址 p_Test = &Num）
    2、初始化的赋值与正常操作是不同的，如重新对指针赋地址是 p_Test = &Num1;
    3、要分清声明操作与正常操作的不同。
- 指针类型
  - 指向常量的指针 eg: const int *p_Test = NULL or int const *P_Test = NULL
    - 指针的地址可变，其指向的内容也会跟随地址而变化
    - 指针指向的内容不可以用指针操作来改变 eg: *p_Test = 888，这是错误的。
  - 常量指针 eg: int *const p_Test = NULL
    - 指针指向的内容可以改变
    - 指针的地址不可以改变
- 数组与指针
  - 定义：
    1、数组：相同类型的集合，可用一个名称引用
    2、指针：是一个变量，它的值是给定类型的另一个 变量或是常量的地址
    区别：指针指向的地址是可以操作的，数组的地址是不可操作的
  - 多维数组与指针
    - eg: int board[3][3] = {
      {1,2,3},
      {4,5,6},
      {7,8,9}
      };
      board、board[0]和&board[0][0]在数值上一样的，但是它们代表的是不同的东西：
      1、board是int型二维数组的地址；
      2、board[0]代表的是一维子数组的地址，它是board的一个字数组；
      3、&board[0][0]是int型数组元素的地址，
      如果要获取board数组的第一个元素，有：(**board)=(*board[0])=(board[0][0])
    - 

    - 多维数组的指针操作：
      层面操作：1、在board的层面上，board++是一维子数组的地址递增（可类比一维数组操作，一维数组的元素类比为二维数组的子数组）
      2、在*board的层面上，(*board)++是数组元素的地址递增
      eg: i = 0:1:8;(*(*board + i));可获得数组的元素；
      错误的引用有：
      1、(**board+i)是在第一个元素的值上加 i；
      2、(**(board+i))是在board的层面上进行指针操作，(**(board+0))指向的是board[0][0]，(**(board+1))指向的是board[1][0]，(**(board+2))指向的是board[2][0]，随着i的增大，会造成访问超出数组地址范围的情况！
    - 多维数组的指针引用：
      eg: int *p_board = *board;i = 0:1:8;(*(p_board + i));该操作可遍历数组元素。
      在此操作中 因为board 是int ** 型，(*board)是指针的指针的，故初始化_引用指针_时用 int *。
    - 表 7-1 访问数组元素的指针表达式

board	0	1	2
0	board[0][0] *board[0] **board	board[0][1] *(board[0]+1) *(*board+1)	board[0][2] *(board[0]+2) *(*board+2)
1	board[1][0] *(board[0]+3) *board[1] *(*board+3)	board[1][1] *(board[0]+4) *(board[1]+1) *(*board+4)	board[1][2] *(board[0]+5) *(board[1]+2) *(*board+5)
2	board[2][0] *(board[0]+6) *(board[1]+3) *board[2] *(*board+6)	board[2][1] *(board[0]+7) *(board[1]+4) *(board[2]+1) *(*board+7)	board[2][2] *(board[0]+8) *(board[1]+5) *(board[2]+2) *(*board+8)

    - 多维数组的指针引用时，要保证指针在同一层面上（术语：同一回级别）。
      有时候有些编译器对不在同一层面上的指针赋值会有警告，但是可以正常编译！这种操作是不建议的！
- 内存与指针 — 具体参见 内存的使用
- 使用指针处理字符串
  - 使用 char* 声明一个char型指针，配合内存分配使用。
    eg:size_t str_size = 10; char* p_Str = malloc(str_size); *p_Str = '\0';
    '\0'是必须的，用于避免程序在访问地址 p_Str 时，*p_Str没有被重新赋值的情况下程序跑偏。具体参见 字符串。
- 指针数组 — eg: char *p_Arry[10] = {NULL};
- 指针与数组记号_的交互
  - eg: int count = 100; double* data = calloc(count,sizeof(double));
    i = 0:1:99; *(data+i) = data[i];数组记号比指针记号更容易理解。
- 函数指针
  - 声明函数指针
    eg:int sum(int a,int b);//声明函数
    int (*p_function(int,int) = sum;//声明函数指针，并赋值
    int result = p_function(45,45);//用函数指针来引用函数，指针函数不需要取消引用运算符(间接引用符)
  - 函数指针的数组
    eg: int （*p_fun[3](int,int);其用法与数组相同，亦可以在声明中初始化指针数组的元素。
  - 作为变元的函数指针
    将函数指针作为变元来传递，这样就可以根据函数指针指向的函数而调用不同的函数。
    eg: int any_function(int (*p_fun(int ,int),int x,int y)
    { return p_fun(x,y);}//可以将_函数的地址_通过指针 p_fun 作为变元传入函数。

内存的使用

堆栈(stack)是很重要的概念，中断切换、OS任务切换(如UCOS系统的多任务切换)等等都会运用到。它与cpu的寄存器密切相关。

- 堆(heap) 与 堆栈(stack) 的区别：
  1、stack由编译器自动分配释放，存放函数的参数值、局部变量的值等。其操作方式类似于数据结构中的栈。
  2、heap一般由程序员分配释放，若程序员不释放，程序结束时可能由OS回收。
  （建议手动回收，并在程序结束后立即释放，因为不是每一个OS都是有足够安全的自动回收机制）。
- 动态内存分配
  - malloc()函数
    1、eg: int *p_Mem = (int*)malloc(25*sizeof(int));
    该程序分配了 25 个 _int型_值(100个字节)，也可以采用 (int*)malloc(100)
    建议使用第一种分配方式，较为直观，为malloc()是一般用途函数，返回的值是(void*)指针，建议加上类型转换，如(int*)malloc(25*sizeof(int))。
    2、如果内存分配不成功malloc()会返回一个NULL指针，建议是使用中加上 if 语句进行判断
    eg: int *p_Mem = (int*)malloc(25*sizeof(int));
    if(!p_Mem){ //code do deal with memory allocation failure....}
  - calloc()函数
    calloc()函数 与 malloc()函数相比有两个优点：
    1、它把内存分配为给定大小的数组2、它初始化了所分配的内存，所有位都是 0。
    eg: int *p_Mem = (int*)calloc(75,sizeof(int));该程序分配了75个_int型_值，
    其他操作，如分配不成功都与malloc()一样。
- 成对出现
- 释放动态内存
  - 内存释放的重要性：
    当动态分配一些内存，巡程序没有保留对他们的引用，就会出现内存泄露，
    如果程序执行的是迭代操作，那么程序会占用越来越多的内存，直至没有内存可以使用，
  - free()函数：
    1、要释放分配的内存，必须能访问引用内存块的地址(如上面的内存地址(指针) p_Mem)
    2、eg: free(p_Mem);p_Mem = NULL;（指针赋值为NULL是很重要的操作，能避免很多错误）
  - 在释放指针指向的_堆内存_时，必须确保它不被另一个地址覆盖。
    1、如果把free()函数传送空指针，该函数不会执行任何操作。
    2、应避免多次释放相同的内存区域(该情形可用指针赋NULL有效避免)，因为在这种情况下，
    free()函数操作是不确定的，会造成很多不可预料的后果。假如在第一次对内存进行释放操作free(p_Mem)，
    但该指针没被赋值NULL，而第一次释放的内存被其他操作重新分配，此时程序_无意识_再一次对p_Mem进行内存释放，会造成不可控制后果。
    3、内存处理要特别小心！！！
- 扩展动态分配的内存
  - realloc()函数
    说明：1、relloc()函数可以_重用_或_扩展_以前用malloc()或是calloc()(或者realloc()分配的内存)。
    2、relloc()函数最重要的特性是保存了原内存区域的内容，且保存的量是新旧内存区域中较小的那一个，
    如果新内存区域大于旧内存，新增的内存就不初始化，而是包含垃圾值(不绝对，视程序需求)。
    3、relloc()函数有两个参数(Mem_add,Mem_size)，当Mem_add为NULL是，函数与malloc()的作用一样，
    重新分配一个新的内存，当Mem_add不是NULL，同时不指向以前分配的内存的时候，或者是指向已经_释放_的内存，
    结果是不确定的。
    eg: size_t Capacity 10; int *p_Mem = relloc(Capacity,sizeof(int));参见P-262 。
- 动态内存分配的基本规则
  - 1、避免分配大量的小内存块，分配堆上的内存有一定的系统开销，分配的许多小的内存比分配一个大的内存系统开销大。
    2、仅在需要的时候分配内存，只要使用完堆上的内存，就释放它，
    3、总是确保释放已分配的内存，在编写的早时候就要规划好分配与释放，
    4、在释放内存的时候，确保不会无意中覆盖堆上已分配的内存地址，否则程序就会出现内存泄露，在 循环中分配内存时，要特别小心。

结构化数据

- 结构体
  - struct 定义各种类型的变量组合
    不同的声明方式
    1、eg: typedef struct { int test1;char test2;} test_Typedef;
    test_Typedef 为数据结构的别名，可用它定义相同类型的数据结构，
    用此种方式较为方便，但是不能将_结构指针_用作结构成员。
    2、eg: typedef struct Horse Horse;
    struct Horse{ int age;char name;Horse *next;};
    Horse是数据结构的别名，可用定义相同类型的数据结构，也可将_结构指针_用作结构成员。
    3、eg:struct{ int age;char name} Horse;此种方式为未命名的结构，结构体的其他实例必须在声明的时候定义。
  - 为结构体分配内存:使用指针。
  - 使用sizeof()可以计算结构体的字节数。除了char类型的变量之外，2字节的变量的起始地址是2的倍数，4字节的变量起始地址是4的倍数，依次类推。这称为边界调整，属于硬件需求(与语言无关)，是为了更快地在处理器与内存之间传递数据。
- 链表
  - 链表常用于需要处理数量未知的结构的应用程序中，它的主要优点是内存的使用和处理，存储和处理的链表所占用的内存量量少，但它也有个个缺点:数据处理的速度较慢，尤其是要随机访问数据时。
  - 单向链表、双向链表
- 位字段，参见P-406
- 结构与函数
  - 结构作为函数的变元
  - 结构指针作为函数的变元
  - 作为函数的返回值得结构
- 二叉树 — 二叉树是组织数据的一种非常有效的方式，参见P-414
- 共享内存
  - union 联合
    1、定义在c语言中，允许多个不同变量共享同一内存区，称为联合。联合所占的字节数是最大成员所占的空间。
    2、一个联合体变量，一次只能对其中一个成员赋值。当赋值的成员的字节数小于其他的成员的字节数时，只会覆盖其中部分的数据，

函数

- 变量的两要素: 1、作用域
  2、生存周期
  有关变量的本地变量、全局变量使用相同的名称的例子(本地变量优先)，参见 P-324
- 函数指针，参见 指针
- 函数中的变量
  - 静态变量
    1、作用域：函数内
    2、生存周期：只需初始化一次，即使退出函数，静态变量也不会销毁，
    3、静态变量初始化都为0,除非将它们初始化为其他值。
  - 全局变量
- 调用自己的函数：递归
  - 递归的主要用途是解决复杂的问题。如果递归的层次过多，会很耗内存，而使用循环来处理，不但节省内存，而且速度较快，
    如果一定要使用递归，就一定要有停止处理的机制。
- 变元个数可变的函数
  - 常用函数 printf()
    就是变元个数可变函数，具体的编写规则，参见P-328
- main()函数
- 结束程序
  - 当函数检测到某个灾难性后果使程序不能继续执行时，需要_结束程序_来停止程序执行，
    stdlib.h
    头文件提供的几个函数可以用于终止程序执行:
    1、不正常终止程序函数：abort();
    2、exit()、atexit();
    3、_Exit();
    4、quick_exit()、at_quick_exit();
    以上程序执行参数与退回，以及执行效果参见:P-33。
- 提高性能
  - 内联声明函数
    短函数的每次调用可以用实现函数功能的内联代码替代，以提高执行性能。
    这意味着不用给函数传递值或返回一个值。要让编译器执行这种技术，可以把
    短函数指定为 inline。
    eg: inline double bmi(double kg_wt,double m_height) {
    return
    kg_wt/(m_hight*m_hight); }
  - 使用restrict关键字(一般不使用)
    当指针引用的数据项都在指定范围内引用时，可用restrict关键字修饰告诉编译器进行优化，参见P-335。
  - _Noreturn函数限定符
    有时有些程序是不用退回，如调用终止程序，参见P-336。

支持功能

- 预处理
  - #include
    头文件不宜添加执行代码。可以声明，但不能包含函数定义或初始化的全局数据。
    使用 #include "xxx.c" 与 #include <xxx.c> 的区别只是查找源文件的过程不同，与编译器有关。
  - 外部变量，关键字 extern 。功能是告诉编译器这些变量在外部定义，而编译器也不会创建这些变量
  - 静态函数
    源文件中的函数都是隐式的 extern 。用关键字 static 定义为静态函数。
  - 替换程序源代码 #define
- 宏
  - eg: #define max(x,y) (x>y ? x :y)
    宏是直接插入函数中，所以在使用过程中要特别注意_运算顺序、逻辑顺序_。
  - 字符串作为宏参数
    eg: #define show(var) printf(#var"= %d \r\n",var);
    该函数会打印出变量的名称跟数值，当对于变化的数组元素，只能打印编程时的初态名称。
    如 i = 0 :1 :10;show(test[i]); 只会printf出: test[i]=x
    这是个较为复杂的预处理机制，参见P-509
  - 在宏展开式中结合两个变元
    eg: #define join(a,b) a##b; //##连接符
    用于合成变量名称，或是从多个或多个宏参数中生成一个格式字符串。
    不理解。。。。
  - 预处理指令
    #if define #endif
    #ifdef #endif
    ……
    #if !define #endif
    #ifndef #endif
    ……
    #if #elif #else #endif
  - 取消定义标识符
    #undef
  - #define DEBUG_Printf_Loction() \
    (printf("Error: file %s, function %s(), line %d\r\n", __FILE__, __FUNCTION__,__LINE__))

关键字

- auto 局部变量（自动储存）
- break 无条件退出程序最内层循环
- case switch 语句中选择项
- char 单字节整型数据
- const 定义不可更改的常量值
- continue 中断本次循环，并转向下一次循环
- default switch 语句中的默认选择项
- do 用于构成do....while循环语句
- double 定义双精度浮点型数据
- else构成if.....else选择程序结构
- enum 枚举
- extern 在其它程序模块中说明了全局变量
- float 定义单精度浮点型数据
- for构成for循环语句
- goto构成goto转移结构
- if构成if....else选择结构
- int 基本整型数据
- long 长整型数据
- registerCPU 内部寄存的变量
- return 用于返回函数的返回值
- short 短整型数据
- signed 有符号数
- sizeof 计算表达式或数据类型的占用字节数
- static 定义静态变量
- struct 定义结构类型数据
- switch构成switch选择结构
- typedef 重新定义数据类型
- union 联合类型数据
- unsigned 定义无符号数据
- void 定义未类型数据
- volatile 该变量在程序中执行中可被隐含地改变
- while用于构成do....while或while循环结构
- volatile 告诉编译器，该参数不进行编译优化，在读取该参数时不从寄存器读取，而是从数据地址中读取。