《The C Programming Language》读书笔记总结 .基础篇

写了这么多年的C代码，回过头来再看《The C Programming Language》这本书，作者Brian W. Kernighan和C语言之父Dennis M. Ritchie。感觉里面的知识和书的架构给人非常”合理”的感觉。怎么个合理法呢?

- 首先书中的代码，如else-if中使用binsearch函数介绍二分查找、atoi介绍字符串s转换为整数、计算器逆波兰表达式，都是实际中非常经典且常用的知识；
- 然后书中大部分的程序都是基于stdlib.h、string.h、ctype.h等这些源文件并且简化后的代码，可见作者对C语言的了解程度不是一般，这也是有别于国内的C语言书籍的地方；
- 最后书中的整体逻辑架构非常不错，而且书中的例子是串联起来的，如push和pop函数，而且通俗易懂并结合了Unix的相关知识，毕竟C语言和Unix之父。

总体感觉一句话”还君明珠双泪垂，恨不相逢未嫁时”，如果时光可以倒流，当时还是应该看些经典的书籍啊!
这篇文章主要是我的读书笔记，记录C语言中一些比较难或经典的知识，希望能勾起你的回忆。如果对你也有所帮助，那我就非常满足了；如果有错误或不足之处，还请海涵~

第2章 类型、运算符与表达式

数据类型及长度

int通常代表特定机器中整数的自然长度，short类型通常为16位，long类型通常为32位，int类型可以为16位或32位。各编译器可以根据硬件特性自主选择合适的类型长度，但要遵循下列限制：short与int类型至少为16位，而long类型至少为32位，并且short类型不得长于int类型，而int类型不得长于long类型。

类型限定符signed与unsigned用于限定char类型或任何整型。unsigned类型数总是正值或0，并遵循算术模 2^n 定律，其中n是该类型占用的位数。例如char对象占用8位，那么unsigned char类型变量的取值范围为0~255，而signed char类型变量的取值范围为-128~127（在采用对二的补码的机器上）。

long double类型表示高精度的浮点型。同整型一样，浮点型的长度也取决于具体的实现，float、double与long double类型可以表示相同的长度，也可以表示两种或三种不同的长度。有关这些类型长度定义的符号常量及其它与机器和编译器有关的属性可以在标准头文件< limits.h >与< float.h >中找到，这些内容在附录B中。

格式说明可以忽略宽度与精度，其中%6f表示待打印的浮点数至少有6个字符宽；%.2f指定待打印的浮点数的小数点后有两位小数，但宽度没有限制；%f则仅仅要求按照浮点数打印该数。同时，printf函数还支持下列格式说明：%o表示八进制数；%x表示十六进制数；%c表示字符；%s表示字符串；%%表示百分号（%）本身。

常量 strlen

ANSI C语言中的全部转移字符序列如下所示：

\a	响铃符	\\	反斜杠
\b	回退符	\?	问号
\f	换页符	\'	单引号
\n	换行符	\"	双引号
\r	回车符	\ooo	八进制数
\t	横向制表符	\xhh	十六进制数
\v	纵向制表符		

其中’\ooo’表示任意的字节大小的为模式，代表1~3个八进制数字（0...7）；’\xhh’其中hh是一个或多个十六进制数字(0...9，a...f，A...F)。例如：

```
#define VTAB '\013'   /* ASCII vertical tab */
#define VTAB '\xb'    /* ASCII vertical tab */
```

字符常量’\0’表示值为0的字符，也就是空字符(null)。
常量表达式是仅仅只包含常量的表达式，这种表达式在编译时求值，而不在运行时求值，它可以出现在常量可以出现的任何位置。如下常用语定义数组长度大小：

```
#define MAXLINE 1000
char line[MAXLINE+1]
```

字符串常量是用双括号括起来的0个或多个字符组成的字符序列，如”I am a string”。其中空字符串用”“表示，字符串中使用\”表示双引号字符。其实，字符串常量就是字符数组，使用空字符’\0’作为串的结尾。因此，存储字符串的物理存储单元数比括在双引号中的字符数多一个。
这种表示方法也说明，C语言对字符串的长度没有限制，但程序必须扫描完整个字符串后才能确定字符串的长度。标准库函数strlen(s)可以返回字符串参数s的长度，但长度不包括末尾’\0’。代码如下：

```
/* strlen: return length of s */
int strlen(char s[])
{
    int i;
    while(s[i] != '\0')
        ++i;
    return i;
}
```

注意：我们需弄清字符常量和仅包含一个字符的字符串之间的区别：’x’与”x”是不同的。前者是一个整数，其值是字母x在机器字符集中对应的数值；后者是包含一个字符（即字母x）以及一个结束符’\0’的字符数组。

枚举常量 enum

枚举常量是另外一种类型的常量。枚举是一个常量整型值的列表，如：

```
enum boolean { NO, YES};
```

在没有显示说明的情况下，enum类型中第一个枚举名的值为0，第二个为1，依次类推。如果只指定了部分枚举名的值，那么未指定值的枚举名的值将依着最后一个指定值向后递增。
例：其中FEB的值为2，MAR的值为3，依次类推。不同枚举中的名字必须互不相同，同一枚举中不同的名字可以具有相同的值。

```
enum months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC }
```

枚举为建立常量值与名字之间的关联提供了一种便利的方式。**相对于#define语句来说，它的优势在于常量值可以自动生成。**尽管可以声明enum类型的变量，但编译器不检查这种类型的变量中存储的值是否为该枚举的有效值。不过，枚举变量提供这种检查，因此枚举比#define更具优势。

声明 const

所有变量都必须先声明后使用，尽管某些变量可以通常上下文隐式地声明。一个声明指定一种变量类型，后面所带的变量表可以包含一个或多个该类型的变量。

```
int lower, upper, step;
char c, line[1000];
```

还可以在声明的同时对变量进行初始化。常见的如下：

```
int i = 0;
int limit = MAXLINE + 1;
float eps = 1.0e-5;
```

默认情况下，外部变量与静态变量将被初始化为0。未经显示初始化的自动变量的值为未定义值（即无效值）。
任何变量的声明都可以使用const限定符限定。该限定符指定变量的值不能被修改，对数组而言，const限定符指定数组所有元素的值不能被修改。

```
const double e = 2.71828282845905;
const char msg[] = "warning :";
```

const限定符也可配合数组参数使用，它表明函数不能修改数组元素的值。如果试图修改const限定符限定的值，其结果取决于具体的实现。

```
int strlen(const char[]);
```

类型转换 atoi

当一个运算符的几个操作数类型不同时，就需要通过一些规则把它们转换为某种共同的类型。一般来说，自动转换是指把“比较窄的”操作数转换为“比较宽的”操作数，并且不丢失信息的转换。例如在计算表达式f+i时，将整形变量i的值自动转换为浮点型（f此处为浮点型）。
由于char类型就是较小的整形，因此在算术表达式中可以自由使用char类型的变量，这就为实现某些字符转换提供了很大的灵活，如下atoi函数将一串数字转换为相应的数值。

```
/* atoi: convert s to integer */
int atoi(char s[])
{
    int i, n;
    n = 0;
    for (i = 0; s[i] >= '0' && s[i] <= '9'; ++i)
        n = 10 * n + (s[i] - '0');
    return n;
}
```

当然，在任何表达式中都可以使用一个称为强制转换的一元运算符强制进行显示类型转换。
(类型名) 表达式
例如库函数sqrt的参数为double类型，如果处理不当，结果可能会无意义（sqrt在< math.h >中声明）。因此，如果n是整数可以使用sqrt((double) n)，在把n传递给函数前将其转换为double类型。
注意：在通常情况下，参数是通过函数原型声明的。这样，当函数被调用时，声明将对参数进行自动强制转换。例如对于sqrt的函数原型：double sqrt(double) 当调用 root =sqrt(2) 时，不需要使用强制类型转换运算符就可以自动将整数2强制转换为double类型的2.0。

自增和自减运算符 strcat

C语言中提供了两个用于变量递增与递减的特俗运算符。自增运算符++使其操作数递增1，自减运算符--使其操作数递减1。
（1）++n：先将n的值递增1，然后再使用变量n的值
（2）n++：先使用变量n的值，然后再将n的值递增1

```
n = 5;
x = n++;    //x的值置于5 n=6
m = 5;
y = ++m;    //y的值置于6 m=6
```

下面举两个例子。函数squeeze(s, c)表示删除字符串s中出现的所有字符c：

```
/* squeeze: delete all c from s */
void squeeze(char s[], int c)
{
    int i, j;
    for(i = j = 0; s[i] != '\0'; i++)
        if(s[i] != c)
            s[j++] = s[i];
    s[j] = '\0';
}
```

再举个例子标准函数strcat(s, t)，它将字符串t连接到字符串s的尾部。函数strcat假定字符串s中有足够的空间保存这两个字符串连接的结果。（标准库中的该函数会返回一个指向新字符串的指针）。

```
void strcat(char s[], char t[])
{
    int i, j;
    i = j = 0;
    while (s[i] != '\0') /* find end of s */
        i++;
    while ((s[i++] = t[j++]) != '\0') /* copy t */
        ;
}
```

在将t中的字符逐个拷贝到s的尾部时，变量i和j都使用后缀运算符++，从而保证循环过程中i和j均指向下一个位置。

按位运算符

C语言提供了6个位操作运算符。这些运算符只能作用于整形操作数，即只能作用于带符号或无符号char、short、int、long类型。

```
&       按位与（AND）
|       按位或（OR)
^       按位异或（XOR）
<<      左移
>>      右移
~       按位求反（一元运算符）
```

(1) 按位与运算&经常用于屏蔽某些二进制位。
如 n = n & 0177 ，该语句将n中除7个低二进制位外的其他各位均置为0。
(2) 按位或运算|常用于将某些二进制位置为1。
如 x = x | SET_ON 该语句将x中对应于SET_ON中为1的那些二进制位置为1。
(3) 按位异或^表示当两个操作数的对应位不相同时，将该位设置为1，否则为0。
(4) 表达式x<<2表示将x的值左移2位，右边空出的2位用0填补，该表达式等价于左操作数乘以4。而>>表示右移，在对unsigned类型的无符号值进行右移位时，左边空出的部分用0填补，当对signed类型的带符号值进行右移时，某些机器将对左边空出的部分用符号位填补（即算术移位），而另一些机器则对左边空出部分用0进行填补（即逻辑移位）。
例如getbits(x, p, n)函数返回x中从右边数第p位开始向右数n位的字段，假定最右边的一位是第0为，n与p都是正值。getbits(x, 4, 3)返回x中第4、3、2三位的值。

```
/* getbits: get n bits from position p */
unsigned getbits(unsigned x, int p, int n)
{
```

```
    return (x >> (p+1-n)) & ~(~0 << n);
}
```

其中，表达式m >> (p+1-n)将期望获得的字段移位到字的最右端，即7 6 5 **4 3 2** 1 0 右移（4+1-3）=2个字段，得到0 0 7 6 5 **4 3 2**。然后~0的所有位都为1，语句 ~0 << n 将全1向左移动n位，即1 1 1 1 1 0 0 0， 再取反将最右边的n位置为1的屏蔽，即0 0 0 0 0 **1 1 1** 。

赋值运算与条件语句

赋值运算符有时还有助于编译器产生高效代码，下面是最常见的一个：

```
while((c = getchar()) != EOF)
```

EOF：end of file，文件结束。定义在头文件< stdio.h >中，是个整型数，其具体数值是什么并不重要，只要它与任何char类型的值都不相同即可。这里使用符号常量，可以确保程序不需要依赖于其对应的任何特定的数值。
上面面种输入集中化代码，缩短了程序，使整个程序看起来更紧凑，这种风格也会让读者更易阅读。不过，如果过多的使用这种类型的复杂语句，编写程序会很难理解，应尽量避免这种情况。
同时，由于不等式运算符!=的优先级比赋值运算符=的优先级高，故赋值表达式两边的圆括号不能省略。

下面代码是通过函数bitcount统计其整形参数的值为1的二进制位的个数：

```
/* bitcount: count 1 bits in x */
int bitcount(unsigned x)
{
    int b;
    for (b = 0; x != 0; x >>= 1)
        if (x & 01)
          b++;
    return b;
}
```

终于知道LeetCode那道“Number of 1 Bits”题目的出处了，还可以通过x & (x-1)删除最右边为1的二进制。

```
/* bitcount: count 1 bits in x */
int bitcount(unsigned x)
{
    int b;
    while(x != 0) {
        x &= (x - 1);
        b++;
    }
    return b;
}
```

条件语句可以使用三元运算符（“? :”），表达式如下：
expr1 ? expr2 : expr3
它首先计算expr1，如果其值不等于1（为真)则计算expr2的值，并以该值作为条件表达式的值，否则计算expr3的值，并以该值为条件表达式的值。expr2和expr3只能有一个表达式被计算，例如：

```
z = (a > b) ? a: b;     /* z = max(a,b) */
```

第3章 控制流

else-if语句 binsearch

在介绍else-if语句中，书中通过**折半查找**进行讲解。输入值x与数组v的中间元素进行比较：
（1）如果x小于中间元素的值，则在该数组的前半部分查找
（2）否则在数组的后半部分查找
（3）再将x与所选部分的中间元素进行比较，直到找到或查找范围为空
代码如下：

```
/* binsearch: find x in v[0]<=v[1]<=...<=v[n-1] */
int binsearch(int x, int v[], int n)
{
    int low, high, mid;
    low = 0;
    high = n-1;
    while(low <= high) {
        mid = (low+high)/2;
        if (x < v[mid])
            high = mid + 1;
        else if (x > v[mid])
            low = mid + 1;
        else    /* found match */
            return mid;
    }
    return -1; /* no match */
}
```

switch语句

switch语句是一种多路判定语句，如果某个分支与表达式的值匹配，则从该分支开始执行。如果没有哪一个分支能匹配表达式，则执行default分支（**可选**）。
注意：在switch局域中，case的作用知识一个标号，因此某个分支中代码执行完后，程序进入下一分支继续执行。除非程序中显示跳转，通过break或return语句。**建议补全break和default。**
统计数字、空格代码如下：

```
switch (表达式) {
    case 常量表达式: 语句序列
    case 常量表达式: 语句序列
    default: 语句序列
}

while((c = getchar()) != EOF) {
    switch (c) {
    case '0': case '1': ... case '9':
        digit[c-'0']++;
        break;
    case ' ':
    case '\n':
    case '\t':
        white++;
        break;
    default:
        other++;
        break;
    }
}
```

do-while语句 itoa

在程序设计时使用while循环语句还是for循环语句，主要取决于程序设计人员的个人偏爱。
如果没有初始化等操作，使用while循环更自然一些，如while((c=getchar()) != EOF)。

如果语句中需要执行简单初始化和变量递增，使用for语句更适合，它将循环控制语句集中放在循环的开头，结构更紧凑、更清晰，如for(i=0; i < n; i++)。

do-while循环是在循环体执行后测试终止条件，这样循环体至少被执行一次。 do-while循环语句在某些情况下还是很有用的，如下通过函数itoa将数字转换成字符串，数字0也需转换一次。

```
/* itoa: convert n to characters in s */
void itoa(int n, char s[])
{
    int i, sign;
    if ((sign = n) < 0)  /* record sign */
        n = -n;          /* make n positive */
    i = 0;
    do {                 /* generate digits in reverse order */
        s[i++] = n % 10 + '0';
    } while( (n /= 10) > 0);
    if (sign < 0)
        s[i++] = '-';
    s[i] = '\0';
    reverse(s);          /* 字符串翻转 */
}
```

goto语句

C语言提供了可随意滥用的goto语句及标记跳转位置的标号。所有使用了goto语句的程序都能改成不带goto语句的程序，而且大多数情况下，使用goto语句的程序段比不使用goto语句的程序段要难以理解和维护，除少数情况。

建议尽量可能少地使用goto语句。某些场合下goto语句可以终止程序在某些深度嵌套的结构中的处理过程，如下判断a与b数组是否具有相同元素：

```
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            if(a[i] == b[j])
                goto found;
    /* did't find any common element */
    ...
found:
    /* got one: a[i] == b[j] */
    ...
```

第4章 函数与程序结构

函数基础知识 atof

函数的定义形式如下：

返回值类型 函数名 (函数声明表)
{
    声明和语句
}

函数定义中的各构成部分都可以省略，最简单的函数如下所示：

dummy() { }

该函数不执行任何操作也不返回任何值。这种不执行任何操作的函数有时很有用，它可以在程序开发期间用以保留位置（留待以后填充代码）。**如果函数定义中省略了返回值类型，则默认为int类型。** 被调用函数通过return语句向调用者返回值，也可以返回任何表达式。

前面讨论的函数都是不返回任何值（void）或只返回int类型值的函数，现在介绍一个不是高质量的类似于标准库中包含 **atof** 函数，它将字符串s转换为相应的双精度浮点数，在头文件”stdlib.h”中。

```
#include <ctype.h>
```

```c
/* atof: convert string s to double */
double atof(char s[])
{
    double val, power;
    int i, sign;
    for (i=0; isspace(s[i]); i++)  /* skip white space */
        ;
    sign = (s[i] == '-') ? -1 : 1;
    if (s[i] == '+' || s[i] == '-')
        i++;
    for (val = 0.0; isdigit(s[i]); i++)
        val = 10.0 * val + (s[i] - '0');
    if (s[i] == '.')
        i++;
    for (power = 1.0; isdigit(s[i]); i++) {
        val = 10.0 * val + (s[i] - '0');
        power *= 10;
    }
    return sign * val / power;
}
```

代码的核心是通过计算val数字除以小数点后面的power（10的倍数，如12.345为12345除以1000）。同时引用了 ctype.h 中的 isspace（检查ch是否是空格符和跳格符或换行符）和 isdigit （检查ch是否是数字0-9）。

外部变量 逆波兰表示法

1.外部变量定义在函数之外，可以在许多函数中使用。

由于C语言不允许在一个函数中定义其它函数，因此函数本身是”外部的”。由于外部变量可以在全局范围内访问，这就可以替代通过函数参数与返回值这种数据交换方式。

如果函数之间需要共享大量的变量，使用外部变量比使用一个很长的参数表更方便、有效。但是这样做必须谨慎，因为这可能会对程序结构产生不良的影响，而且可能会导致程序中各个函数之间具有太多的数据联系。

2.外部变量的用途还表现在它们与内部变量相比具有更大的作用域和更长的生存期。

自动变量只能在函数内部使用，从函数被调用时存在到函数退出时变量消失。而外部变量是**永久存在**的，它们的值在一次函数调用到下一次函数调用之间保持不变。因此如果两个函数必须共享某些数据，而这两个函数互不调用对方，这种情况最方便的方式就是共享数据定义为外部变量，而不是作为函数参数传递。

书中例子：编写一个具有加减乘除四则运算功能的计算器程序

采用逆波兰表示法替代普通的中缀表示法，如下列中缀表达式：（1 - 2） * （4 + 5）

采用逆波兰表示法为：1 2 - 4 5 + *

基本思路： 每个操作数都被依次压入栈中：当一个运算符到达时，从栈中弹出相应数目的操作数（对二元运算来说是两个操作数），把该运算符作用于弹出的操作数，并把运算结果再压入栈中。

如上1和2入栈，当’ - ‘时作减法，值-1取代它们；然后将4和5压入栈中，再’ +‘作加法得9取代它们；最后从栈中取出栈顶的-1和9，并把它们的乘积-9压入到栈顶，到达输入行末尾时，把栈顶的值弹出并打印。

此时需要在main函数外定义外部变量“int sp = 0;”指向栈顶和数组“double val[MAXVAL];”，再对它们进行push（进栈）和pop（出栈）函数操作。

```
while (下一个运算符或操作数不是文件结束指示符)
    if (是数)
        将该数压入到栈中
    else if (是运算符)
        弹出所需数目的操作数
        执行运算
        将结果压入到栈中
    else if (是换行符)
        弹出并打印栈顶的值
    else
        出错
```

作用域规则

名字的作用域指的是程序中可以使用该名字的部分。
对于在函数开头声明的自动变量来说，其作用域是声明该变量名的函数。不同函数中声明的具有相同名字的各个局部变量之间没有任何关系。函数的参数也是这样，可看作是局部变量。
外部变量或函数的作用域从声明它的地方开始，到其所在的（待编译的）文件的末尾结束。
如下列代码中：push和pop可以访问变量sp，而main函数不行，因为它定义在main函数之后；同样push和pop函数也不能在main函数中使用。

```
main() {...}
int sp = 0;
double val[MAXVAL];
void push(double f) {...}
void pop(void) {...}
```

注意：如果要在外部变量的定义之前使用该变量，或者外部变量的定义与变量的使用不在同一源文件中，则必须在相应的变量声明中强制性地使用关键字extern.

外部变量定义与外部变量声明 extern

将外部变量的声明与定义严格区分开来很重要。变量声明用于说明变量的属性（主要是变量的类型），而变量定义除此之外还将引起存储器的分配。

1.外部变量定义

如果将下列语句放在所有函数的外部：那么这两条语句将**定义**外部变量sp与val，并为之**分配存储单元**，同时这两条语句还可以作为该源文件中其余部分的声明。

```
int sp;
double val[MAXVAL];
```

2.外部变量声明

而下面的两行语句：为源文件的其余部分**声明**一个int类型的外部变量sp及一个double数组类型的外部变量val（该数组的长度在其它地方确定），但这两个声明并**没有建立变量或为它们分配存储单元。**

```
extern int sp;
extern double val[];
```

在一个源程序的所有源文件中，一个外部变量只能在某个文件中定义一次，而其他文件可以通过extern声明来访问它。外部变量的定义中必须指定数组的长度，但extern声明则不一定要指定数组的长度，如上面的变量val[]。同时，外部变量的初始化只能出现在其定义中。
总之，extern可置于变量或者函数前，以表示变量或者函数的定义在别的文件中或声明之后，提示编译器遇到此变量或函数时，在其它模块中或之后寻找其定义。
举个例子：函数push与pop定义在一个文件file1中，而变量val与sp在file2件中定义并初始化，下面代码将其定义与声明”绑定“在一起。同样，如果要在同一个文件中先使用、后定义sp与val，也需要按照这种方式来组织文件。

```
file1 文件:
    extern int sp;
    extern double val[];
    void push(double f) {...}
```

```
    double pop(void) {...}
file2 文件:
    int sp = 0;
    double val[MAXVAL];
```

静态变量 static

某些变量如文件stack.c中定义的变量sp与val，它们仅供其所在的源文件中的函数使用，其他函数不能访问。用static声明限定外部变量与函数，可以将其后声明的对象的作用域限定为被编译源文件的剩余部分。
通过static限定外部对象，可以达到隐藏外部对象的目的，比如getch-ungetch结构需要共享buf与bufp两个变量，这样buf与bufp必须是**外部变量**，但这两个对象不应该被getch与ungetch函数的调用者所访问。
要将对象指定为静态存储，可以在正常的对象声明之前加上关键字static作为前缀。如下所示放在一个文件中编译：

```
static char buf[BUFSIZE];   /* buffer for ungetch */
static int bufp = 0;        /* next free position in buf */
int getch(void) {...}
void ungetch(int c) {...}
```

那么其他函数就不能访问变量buf与bufp，因此两个名字就不会和同一程序中的其他文件中的相同的名字相冲突。换句话说，外部静态变量和外部变量都是一种公用的全局变量，但外部静态变量的作用域仅仅是在定义它的那个文件中，出了该文件不管是否用extern说明都是不可见的。即：
外部静态变量仅仅作用于定义它的那个文件，而外部变量作用于整个程序。
外部的static声明通常多用于变量，当然，它也可以用于声明函数。通常情况下，函数名字是全局可访问的，对整个程序的各个部分而言都可见。但是如果把函数声明为static类型，则该函数名除了对该函数声明所在的文件可见外，其它文件都无法访问。
static也可用于声明内部变量。static类型的内部变量同自动变量一样，是某个特定函数的局部变量，只能在该函数中使用，但它与自动变量不同的是：
不管其所在函数是否被调用，它一直存在，而不像自动变量那样，随着所在函数的被调用和退出而存在和消失。static会一直占据着存储空间，重复使用值会保留（静态变量存放在内存中的静态存储区）。
例如：静态变量只在第一次进入程序块时被初始化一次。

```
void inc()
{
    static int x = 0;
    x++;
    printf("x = %d\n",x);
}
void main()
{
    inc();      //输出 x = 1
    inc();      //输出 x = 2
    inc();      //输出 x = 3
    return 0;
}
```

寄存器变量 register

register声明告诉编译器，它所声明的变量在程序中使用频率较高。其思想是将register变量放在机器的寄存器中，使程序更小、执行速度更快。但编译器可以忽略此选项。声明如：register int x;
register声明只适用于自动变量以及函数的形式参数，如下：

```
f(register unsigned m, register long n)
{
    register int i;
    ...
}
```

实际使用时，底层硬件环境的实际情况对寄存器变量的使用会有一些限制，每个函数中只有很少的变量可以保存在寄存器中，且只允许某些类型的变量。但是，过量的寄存器声明并没有什么害处，这是因为编译器可以忽略过量的或不支持的寄存器变量声明。另外，无论寄存器变量实际上是不是存放在寄存器中，**它的地址都是不能访问的。**

递归 qsort

C语言中的函数可以递归调用，即函数可以直接或间接调用自身。其中举个字符串递归翻转的例子：

```
// 递归实现字符串反转
char *reverse(char *str)
{
    int len;
    char ctemp;
    if( !str ) {
        return NULL;
    }
    len = strlen(str);
    if( len > 1 ) {
        ctemp =str[0];
        str[0] = str[len-1];
        str[len-1] = '/0';    //最后一个字符在下次递归时不再处理
        reverse(str+1);       //递归调用
        str[len-1] = ctemp;
    }
    return str;
}
```

其过程如下，假设现有字符串：a b c d e f
先调换 a 和 f 的位置，然后递归 | b c d e |，这里巧用str+1表示从前移动至b，而str[len-1]=’\0’表示从后移动至e，一次移动两个位置。递归之后再赋值str[len-1]=ctemp，即f位置赋值为a。
另一个比较好说明递归的例子是快速排序。参考我的博客： 彻底搞懂qsort
对于一个给定的数组，从中选择一个元素，该元素为界将余元素划分为两个子集，一个子集中的所有元素都小于该元素，另一个子集中的所有元素都大于或等于该元素。对这两个子集递归执行这一过程，当某个子集中的元素小于2时，这个子集就不再需要再次排序，递归终止。

```
/* qsort: sort v[left]...v[right] into increasing order */
void qsort(int v[], int left, int right)
{
    int i,last;
    void swap(int v[],int i,int j);

    if(left>=right) //若数组包含的元素数少于两个 则推出递归结束
        return;
    swap(v,left,(left+right)/2); //取中间元素作为划分子集的参考数，首先存储参考数于v[0]
    last=left;
    for(i=left+1;i<=right;++i) //划分子集
    {
        if(v[i]<v[left])
            swap(v,++last,i);
    }
    swap(v,left,last); //恢复划分子集的元素
    qsort(v,left,last-1);
    qsort(v,last+1,right);
}
```

这里之所以将数组元素交换操作放在一个单独的函数swap中，是因为它在qsort函数中要使用3次。

```
void swap(int v[], int i, int j)
```

```
{
    int temp;
    temp=v[i];
    v[i]=v[j];
    v[j]=temp;
}
```

注意：递归并不节省存储器的开销，因为递归调用过程中必须在某个地方维护一个存储处理值的栈。递归的执行速度并不快，但递归代码比较紧凑，并且比相应的非递归代码更易于编写与理解。在描述树等递归定义的数据结构时使用递归尤其方便，同时面试过程通常会让你使用栈来模拟二叉树递归遍历的过程。

C预处理器

C语言通过预处理器提供了一些语言功能。从概念上讲，预处理器是编译过程中单独执行第一个步骤。两个常用的预处理器指令时：

#include 指令：用于在编译期间把指定文件的内容包含进当前文件中

#define 指令：用任意字符序列替代一个标记

1.文件包含

文件包含指令（即#include指令）使得处理大量的#define指令以及声明更加方便。在源文件中，任何形如：

#include “文件名”

#include <文件名>

的行都将被替换为由文件名指定的文件的内容。如果文件名用引号引起来，则在源文件所在位置查找该文件；如果在该位置没有找到文件，或者如果文件是用尖括号<与>括起来的，则根据相应的规则查找该文件，这个规则同具体的实现有关。被包含的文件本身也可包含#include指令。

源文件的开始通常会有多个#include指令，它们用以包含常见的#define语句和extern声明，或从头文件中访问库函数的函数原型声明，比如< stdio.h >。

在大的程序中，#include指令是将所有声明绑定在一起的较好的方法。它保证所有的源文件都具有相同的定义与变量声明，这样可以避免出现一些不必要的错误。如果某个包含文件的内容发生了变化，那么所有依赖于该包含文件的源文件都必须重新编译。

2.宏替换

宏定义形式如下：

```
#define 名字 替换文本
```

这是一种最简单的宏替换——后续所有出现名字记号的地方都将被替换为替换文本。#define指令中的名字与变量名的命名方式相同，替换文本可以是任意字符串。

宏定义#define通常占一行，若干行时需要在待续的行末尾加上一个反斜杠符\。宏定义定义的名字的作用域从其定义点开始，到被编译的源文件的末尾处结束。宏定义可以使用前面出现的宏定义，替换只对记号进行，对括号中的字符串不起作用。例如宏定义YES在printf(“YES”)或YESMAN中将不执行替换。

宏定义也可以带参数，这样可以对不同的宏调用使用不同的替换文本。如：

```
#define max(A, B) ((A) > (B) ? (A) : (B))
```

使用宏max看起来像是函数词用，但宏调用直接将替换文本插入到代码中。形式参数的每次出现都将被替换成对应的实际参数。如：

```
x = max(p+q, r+s);
```

将替换为：

```
x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

如果对各种类型的参数的处理是一致的，则可以将同一个宏定义应用于任何数据类型，而无需针对不同的数据类型需要定义不同的max函数。

仔细考虑下max展开式，就会发现它存在一些缺陷。其中作为参数的表达式需要重复计算两次，如果含有自增或自减会出现不正确的情况。

同时宏定义尤其需要注意圆括号以保证计算次序的正确性。如：

```
#define square(x) x * x
```

当执行square(5+1)时会替换为”5 + 1 * 1 +5”，其结果为11而不是25。

但是，宏定义还是很有价值的。< stdio.h >头文件中有一个很实用的例子： getchar 与 putchar 函数在实际中常常定义为宏，这样可以避免处理字符时调用函数所需的运行时开销。

< ctype.h >头文件中定义的函数也常常是通过宏实现的。

可以通过#undef指令取消名字的宏定义，这样可以保证后续的调用是函数调用而不是宏调用。

```
#undef getchar
int getchar(void) { ... }
```

预处理器运算符##为宏扩展提供了一种连接实际参数的手段，如果替换文本中的参数与##相邻，则参数将被实际参数替换，##与前后的空白符都将被删除，并对替换后的结果重新扫描。如：宏paste用于连接两个参数

```
#define paste(front, back) front ## back
```

宏调用paste(name, 1)的结果将建立记号name1，关于##详细参考附录A。

3.条件包含

还可以使用条件语句对预处理本身进行控制，这种条件语句的值是预处理执行的过程中进行计算。这种方式为在编译过程中根据计算所得的条件值选择性地包含不同代码提供了一种手段。

语句#if对其中的常量整型表达式（其中不能包含sizeof、类型转换运算符或enum常量）进行**求值**，若该表达式的值不等于0，则包含其后的各行，直到遇到遇到#endif、#elif或#else语句位置（预处理器语句#elif类似于else if）。在#if语句中可以使用表达式defined（名字），该表达式的值遵循下来规则：当名字已经定义时，其值为1；否则其值为0。

例如，为了保证hdr.h文件的内容只被包含一次，可以将该文件的内容包含在下列形式的条件语句中：

```
#if !defined(HDR)
#define HDR
/* hdr.h文件的内容放在这里*/
#endif
```

第一次包含头文件hdr.h时，将定义名字HDR；此后再次包含该头文件时，会发现名字已经定义，这样就直接跳转到#endif处，类似的方法也可以用来避免多次重复包含统一文件。其中#if !defined(HDR)等价于#ifndef HDR。

下面这段代码首先测试系统变量SYSTEM，然后根据该变量的值确定包含哪个版本的头文件：

```
#if SYSTEM == SYSV
    #define HDR "sysv.h"
#elif SYSTEM == BSD
    #define HDR "bsd.h"
#elif SYSTEM == MSDOS
    #define HDR "msdos.h"
#else
    #define HDR "default.h"
#endlif
#include HDR
```

PS: 最后还是希望文章对你有所帮助，强烈推荐大家阅读这本书，这篇文章主要是讲解了一些大家可能会忽略的C语言基础知识和经典知识。

（By：Eastmount 2015-10-20 深夜3点 http://blog.csdn.net/eastmount/**）**