## 关键字const

修饰变量时，表示该变量在赋初值之后不可修改；

如：const int limit = 10;

上面的代码等同于 int const limit = 10;

const 修饰它左侧的类型，左侧如果不存在就是右侧的类型。

如下面的两个定义是相同的，都表示指向只读整数类型的只读指针：

int const \* const p;

const int \* const p;

## 指针相容

指针相容，指的是指针A的类型和指针B的类型是相容的，那么可以A = B，A的类型和B是相同的，且A的类型的限定符包含B的类型的限定符；

如下面是相容的（compatible）：

char \*cp;

const char \*ccp;

ccp = cp;

cp和ccp都是相同类型的指针，指向char，ccp包含了const限定符和cp的限定符（无限定符）；

但下面的不相容（incompatible）：

char \*\*cpp;

const char \*\* ccpp;

ccpp = cpp;

cpp和ccpp都不含有限定符，const修饰的是char，两者是不同的类型，不相容。

## 类型转换

自动类型转换：当执行算数运算时，操作数的类型如果不同，就会发生转换，数据类型一般朝着浮点精度更高，长度更长的方向转换，整形数如果转换为signed，不会丢失信息，就转换为signed，否则转换为unsigned。

ANSI标准采用值保留原则（value preserving），K&R采用的是无符号保留原则（unsigned preserving），所以下面代码-1分别被解释为负数和正数。

如下面代码：

#include <stdio.h>

void main()

{

if( -1 < (unsigned char)1)

{

printf("-1 is less than (unsigned char)1: ANSI semantics.\n");

}

else

{

printf("-1 is NOT less than (unsigned char)1: K&R semantics.\n");

}

}

在使用位域或者掩码时才使用无符号数，一般情况下使用有符号数，或者强制转换所有变量为相同的类型，避免将负数解释为超大的数。

## NUL和NULL

NULL是一个宏，它在几个标准头文件中定义，0是一个整型常量，'\0'是一个字符常量，而NUL是一个字符常量的名字。这几个术语都不可互换。

1、NULL用于表示什么也不指向，也就是空指针((void \*)0)

2、0可以被用于任何地方，它是表示各种类型零值的符号并且编译器会挑出它

3、'\0'应该只被用于结束字符串

4、NUL没有被定义于C和C++，它不应该被使用除非你自己定义它，像：#define nul '\0

NULL是在<stddef.h>头文件中专门为空指针定义的一个宏。NUL是ASCII字符集中第一个字符的名称，它对应于一个零值。C语言中没有NUL这样的预定义宏。注意：在ASCII字符集中，数字0对应于十进制值48，不要把数字0和'\0'(NUL)的值混同起来。

NULL可以被定义为(void \*)0，而NUL可以被定义为'\0'。NULL和NUL都可以被简单地定义为0，这时它们是等价的，可以互换使用，但这是一种不可取的方式。为了使程序读起来更清晰，维护起来更容易，你在程序中应该明确地将NULL定义为指针类型，而将NUL定义为字符类型。

对指针进行解引用操作可以获得它的值。从定义来看，NULL指针并未指向任何东西。因此，对一个NULL指针进行解引用操作是非法的。在对指针进行解引用操作之前，必须确保它并非NULL指针。

例如：

#include <stdio.h>

#include <stdlib.h>  
    #define NUL '\0'

int \*ptr = NULL;  
    char ch = NUL;  
\r和\n都是一个字符，分别表示回车和换行，在ASC码里值分别为0x0D、0x0A，windows中文本一行的结尾都是有两个字符\r\n，但是在unix、linux里面只有一个字符\n，Mac里面则是只有字符\r，但是你是用printf("\r");或printf("\n");大部分程序（某些控制台程序除外）都是没有太大区别的。  
至于字符为什么写成\加上0、r和n等，是因为在C语言中表示字符或字符串时遇到\（转义字符）则不会把下一个字符识别为普通字符，比如说'\n'，表示ASC码0x0A，而不是值为0x6E的字母n。

## 字符串连接

相邻的字符串字面值会自动连接为一个字面值，有两种方式，如下：

printf("\

A favorite children's book,\

is 'muffy Gets It: the hilarious tale of a cat,\

a boy, and his machine gun'");

printf("A second favorite children's book is "

"'Thoms the tank engine and Naughty Enginedriver who"

" tied down Thoms's boiler safety value'");

## 运算顺序

.的优先级高于\*；\*p.f表示\*(p.f)，可以用p->f来表达(\*p).f

[]的优先级高于\*；int\*ap[]表示int\*(a[])，即数组元素是int指针。

函数()的优先级高于\*；int\*fp()表示int\*(fp())，表示函数返回值是int指针。

==和！=高于位操作和赋值；(val&mask!=0)表示val&(mask!=0)。c=getchar()!=EOF表示c=(getchar()!=EOF)

算数运算高于位移；msb<<4+lsb表示msb<<(4+lsb))

逗号运算符优先级最低；i=1,2表示(i=1),2，i的值是1，2被丢弃；

x = f() + g() \* h(); f,g,h三个函数的调用顺序并无确定顺序，确定的只有\*优先于+。函数参数调用中的计算顺序是不确定的。

所有的赋值符具有右结合性。位操作符具有左结合性。a = b = c，表示a = (b = c)。逻辑运算符是右结合性，以便更快得出结果，省去后续的计算。

谨记：算数运算符加入()来明确表明优先级，在运算表达式中不依赖于函数调用顺序；

## 关于结构

1、变量的定义和声明应该分开：

2、应该始终在结构定义中加入标签，在定义变量时减少重复编码。

//类型定义和变量声明分开

struct veg

{

int weight, price\_per\_1b;

};

struct veg onion, radish, turnip;

3、结构体在作为函数参数时是拷贝传值。

4、在结构体中定义指向自身的指针来实现链表、树等动态数据结构。

## 关于联合

1. 定义一组不可能同时存在的数据类型的组合；
2. 将同一数据解释成不同形式；

## 关于枚举

1. 缺省值从0开始，逐个增加1；
2. 枚举值类型是整型，且只能作为右值；
3. 枚举类型的变量可以作为左值，且可以被赋值为枚举值之外的值；

## 声明规则

1. 从它的最左边标识符开始读取，然后按照优先级顺序依次读取；
2. 优先级由高到低依次是：
   1. 声明中被括号括起来的那部分
   2. 后缀操作符：()表示这是一个函数，[]表示这是一个数组
   3. 前缀操作符：\*表示指向…的指针，函数的返回值类型，数组的元素类型；
3. 如果const、volatile关键字后面紧跟着类型说明符（如int，long等），那么它作用于类型说明符。在其他情况下，作用于它左边的紧邻的指针\*；

如 char\*const\*(\*next)();表示next是一个指针，指向一个函数，函数的返回值是一个指针，指针的类型是指向char的const指针。

再如char\*(\*c[10])(int \*\*p);表示c是一个数组，数组的类型是指针，指向一个函数，函数的参数是int的指针的指针，返回值是指向char的指针。

## 关于typedef

表示为原来的类型声明一个新的名字，并没有产生一个新的类型。

void ( \* signal **( int sig, void ( \* func ) ( int ) )** ) ( int ).

signal是一个函数，参数之一是一个整数，之二是一个函数指针，参数是证明，无返回值。Signal的返回值是一个指针，指向一个函数，函数的参数是int，无返回值。

typedef void(\*ptr\_to\_func)(int);表示ptr\_to\_func是一个函数指针，函数的参数是int，无返回值。

ptr\_to\_func signal(int sig, ptr\_to\_func);

1. 不要在一个typedef 中放入几个声明器：typedef int \*ptr, (\*fun)(), arr[5];
2. 不要把typedef嵌到声明中的中间部分。unsigned const long typedef int volatile \*kumquat;
3. typedef使用在数组，结构，指针及函数等组合类型、可移植类型；
4. typedef可以用于类型转换

## 关于可变参数宏

C99中规定宏可以像函数一样带有可变参数，比如

#define LOG(format, ...) fprintf(stdout, format, \_\_VA\_ARGS\_\_)

其中，...表示参数可变，\_\_VA\_ARGS\_\_在预处理中为实际的参数集所替换

GCC中同时支持如下的形式

#define LOG(format, args...) fprintf(stdout, format, args)

其用法和上面的基本一致，只是参数符号有变化

有一点需要注意，上述的宏定义不能省略可变参数，尽管你可以传递一个空参数，这里有必要提到"##"连接符号的用法。

"##"的作用是对token进行连接，在上例中，format、\_\_VA\_ARGS\_\_、args即是token，

如果token为空，那么不进行连接，所以允许省略可变参数(\_\_VA\_ARGS\_\_和args)，对上述变参宏做如下修改

#define LOG(format, ...) fprintf(stdout, format, ##\_\_VA\_ARGS\_\_)

#define LOG(format, args...) fprintf(stdout, format, ##args)

gcc用##解决该问题： 如果可变参数列表为空，就会将紧挨着")"的“,”去掉。

## 关于宏

当然宏定义非常重要的，它可以帮助我们防止出错，提高代码的可移植性和可读性等。

下面列举一些成熟软件中常用得宏定义

1，防止一个头文件被重复包含  
#ifndef COMDEF\_H  
#define COMDEF\_H

//头文件内容 …  
#endif

2，重新定义一些类型，防止由于各种平台和编译器的不同，而产生的类型字节数差异，方便移植。  
typedef  unsigned long int  uint32;      /\* Unsigned 32 bit value \*/

3，得到指定地址上的一个字节或字  
#define  MEM\_B( x )  ( \*( (byte \*) (x) ) )  
#define  MEM\_W( x )  ( \*( (word \*) (x) ) )

4，求最大值和最小值  
#define  MAX( x, y )  ( ((x) > (y)) ? (x) : (y) )  
#define  MIN( x, y )  ( ((x) < (y)) ? (x) : (y) )

5，得到一个field在结构体(struct)中的偏移量  
#define FPOS( type, field )   ( (dword) &(( type \*) 0)-> field )

6,得到一个结构体中field所占用的字节数  
#define FSIZ( type, field ) sizeof( ((type \*) 0)->field )

7，按照LSB格式把两个字节转化为一个word  
#define  FLIPW( ray ) ( (((word) (ray)[0]) \* 256) + (ray)[1] )

8，按照LSB格式把一个word转化为两个字节  
#define  FLOPW( ray, val )   
(ray)[0] = ((val) / 256);   
(ray)[1] = ((val) & 0xFF)

9，得到一个变量的地址（word宽度）  
#define  B\_PTR( var )  ( (byte \*) (void \*) &(var) )  
#define  W\_PTR( var )  ( (word \*) (void \*) &(var) )

10，得到一个字的高位和低位字节  
#define  WORD\_LO(xxx)  ((byte) ((word)(var) & 255))  
#define  WORD\_HI(xxx)  ((byte) ((word)(var) >> 8))

11，返回一个比X大的最接近的8的倍数  
#define RND8( x )       ((((x) + 7) / 8 ) \* 8 )

12，将一个字母转换为大写  
#define  UPCASE( c ) ( ((c) >= ’a' && (c) <= ’z') ? ((c) - 0×20) : (c) )

13，判断字符是不是10进值的数字  
#define  DECCHK( c ) ((c) >= ’0′ && (c) <= ’9′)

14，判断字符是不是16进值的数字  
#define  HEXCHK( c ) ( ((c) >= ’0′ && (c) <= ’9′) ||  
((c) >= ’A' && (c) <= ’F') ||  
((c) >= ’a' && (c) <= ’f') )

15，防止溢出的一个方法  
#define  INC\_SAT( val )  (val = ((val)+1 > (val)) ? (val)+1 : (val))

16，返回数组元素的个数  
#define  ARR\_SIZE( a )  ( sizeof( (a) ) / sizeof( (a[0]) ) )

17，对于IO空间映射在存储空间的结构，输入输出处理  
#define inp(port)         (\*((volatile byte \*) (port)))  
#define inpw(port)        (\*((volatile word \*) (port)))  
#define inpdw(port)       (\*((volatile dword \*)(port)))

#define outp(port, val)   (\*((volatile byte \*) (port)) = ((byte) (val)))  
#define outpw(port, val)  (\*((volatile word \*) (port)) = ((word) (val)))  
#define outpdw(port, val) (\*((volatile dword \*) (port)) = ((dword) (val)))

18，使用一些宏跟踪调试  
ANSI标准说明了五个预定义的宏名。它们是：  
\_\_LINE\_\_  
\_\_FILE\_\_  
\_\_DATE\_\_  
\_\_TIME\_\_  
\_\_STDC\_\_

如果编译不是标准的，则可能仅支持以上宏名中的几个，或根本不支持。记住编译程序 也许还提供其它预定义的宏名。  
是行连接符，会将下一行和前一行连接成为一行，即将物理上的两行连接成逻辑上的一行  
\_\_FILE\_\_ 是内置宏 代表源文件的文件名  
\_\_LINE\_\_ 是内置宏，代表该行代码的所在行号  
\_\_DATE\_\_宏指令含有形式为月/日/年的串，表示源文件被翻译到代码时的日期。  
源代码翻译到目标代码的时间作为串包含在\_\_TIME\_\_ 中。串形式为时：分：秒。  
如果实现是标准的，则宏\_\_STDC\_\_含有十进制常量1。如果它含有任何其它数，则实现是非标准的。

可以定义宏，例如:  
当定义了\_DEBUG，输出数据信息和所在文件所在行

#ifdef \_DEBUG  
#define DEBUGMSG(msg,date) printf(msg);printf(“%d%d%d”,date,\_LINE\_,\_FILE\_)  
#else  
#define DEBUGMSG(msg,date)  
#endif

19，宏定义防止使用是错误  
用小括号包含。  
例如：#define ADD(a,b) （a+b）

用do{}while(0)语句包含多语句防止错误

例如：#difne DO(a,b) a+b;  
a++;  
应用时：if(….)  
DO(a,b); //产生错误  
else  
解决方法: #difne DO(a,b) do{a+b;  
a++;}while(0)

为什么需要do{…}while(0)形式?

总结了以下几个原因：

1)，空的宏定义避免warning:

#define foo() do{}while(0)

2)，存在一个独立的block，可以用来进行变量定义，进行比较复杂的实现。

3)，如果出现在判断语句过后的宏，这样可以保证作为一个整体来是实现：

#define foo(x)

action1();

action2();

在以下情况下：

if(NULL == pPointer)

    foo();

就会出现action2必然被执行的情况，而这显然不是程序设计的目的。

4)，以上的第3种情况用单独的{}也可以实现，但是为什么一定要一个do{}while(0)呢，看以下代码：

#define switch(x,y) {int tmp; tmp=x;x=y;y=tmp;}

if(x>y)

   switch(x,y);

else        //error, parse error before else

   otheraction();

在把宏引入代码中，会多出一个分号，从而会报错。

使用do{….}while(0) 把它包裹起来，成为一个独立的语法单元，从而不会与上下文发生混淆。同时因为绝大多数的编译器都能够识别do{…}while(0)这种无用的循环并进行优化，所以使用这种方法也不会导致程序的性能降低。

为什么很多linux内核中宏#defines用do { … } while(0)？

有很多原因：

（Dave Miller的说法）：

编译器对于空语句会给出告警，这是为什么#define FOO do{ }while(0);

给定一个基本块（局部可视域），定义很多局部变量；

（Ben Collins的说法）：

在条件代码中，允许定义复杂的宏。可以想像有很多行宏，如下代码

#define FOO(x)   
printf(“arg is %sn”, x);   
do\_something\_useful(x);  
现在，想像下面的应用:  
if (blah == 2)  
FOO(blah);  
展开后代码为:  
if (blah == 2)  
printf(“arg is %sn”, blah);  
do\_something\_useful(blah);;  
就像你看到的，if仅仅包含了printf（），而do\_something\_useful()调用是无条件调用。因此，如果用do { … } while(0)，结果是：  
if (blah == 2)  
do {  
printf(“arg is %sn”, blah);  
do\_something\_useful(blah);  
} while (0);  
这才是所期望的结果。  
（Per Persson的说法）：  
像 Miller and Collins指出的那样，需要一个块语句包含多个代码行和声明局部变量。但是，本质如下面例子代码：  
#define exch(x,y) { int tmp; tmp=x; x=y; y=tmp; }  
上面代码在有些时候却不能有效工作，下面代码是一个有两个分支的if语句：  
if (x > y)  
exch(x,y);               // Branch 1  
else  
do\_something();      // Branch 2  
展开后代码如下：  
if (x > y)

{                // Single-branch if-statement!!!  
int tmp;            // The one and only branch consists  
tmp = x;           // of the block.  
x = y;  
y = tmp;  
}  
;                // empty statement  
else                             // ERROR!!! “parse error before else”  
do\_something();  
问题是分号（；）出现在块后面。解决这个问题可以用do{}while（0）：  
if (x > y)  
do {  
int tmp;  
tmp = x;  
x = y;  
y = tmp;  
} while(0);  
else  
do\_something();  
（ Bart Trojanowski的说法）：  
Gcc加入了语句解释，它提供了一个替代do-while-0块的方法。对于上面的解决方法如下，并且更加符合常理  
#define FOO(arg) ({   
typeof(arg) lcl;    
lcl = bar(arg);     
lcl;                    
})  
这是一个奇怪的循环，它根本就只会运行一次，为什么不去掉外面的do{..}while结构呢？我曾一度在心里把它叫做“怪圈”。原来这也是非常巧妙的技巧。在工程中可能经常会引起麻烦，而上面的定义能够保证这些麻烦不会出现。下面是解释：  
假设有这样一个宏定义  
#define macro(condition)   
if(condition) dosomething()  
现在在程序中这样使用这个宏：  
if(temp)  
macro(i);  
else  
doanotherthing();  
一切看起来很正常，但是仔细想想。这个宏会展开成：  
if(temp)  
if(condition) dosomething();  
else  
doanotherthing();  
这时的else不是与第一个if语句匹配，而是错误的与第二个if语句进行了匹配，编译通过了，但是运行的结果一定是错误的。  
为了避免这个错误，我们使用do{….}while(0) 把它包裹起来，成为一个独立的语法单元，从而不会与上下文发生混淆。同时因为绝大多数的编译器都能够识别do{…}while(0)这种无用的循环并进行优化，所以使用这种方法也不会导致程序的性能降低。

另一个讲解  
这是为了含多条语句的宏的通用性  
因为默认规则是宏定义最后是不能加分号的，分号是在引用的时候加上的  
比如定义了一个宏fw(a,b)，那么在c文件里一定是这样引用  
fw(a,b);  
如果不用do…while，那么fw就得定义成:  
#define fw(a,b) {read((a));write((b));}  
那这样fw(a,b);展开后就成了:  
{read(a);write(b);};  
最后就多了个分号，这是语法错误  
而定义成do…while的话，展开后就是:  
do{read(a);write(b);}while(0);    完全正确  
所以要写一个包含多条语句的宏的话，不用do…while是不可能的

宏中#和##的用法

一、一般用法  
我们使用#把宏参数变为一个字符串,用##把两个宏参数贴合在一起.  
用法:  
＃include<cstdio>  
＃include<climits>  
using namespace std;

#define STR(s)     #s  
#define CONS(a,b)  int(a##e##b)

int main()

{

printf(STR(vck));               // 输出字符串vck  
printf(%dn, CONS(2,3));  // 2e3 输出:2000  
return 0;  
}

二、当宏参数是另一个宏的时候  
需要注意的是凡宏定义里有用’#'或’##’的地方宏参数是不会再展开.

1, 非’#'和’##’的情况  
#define TOW      (2)  
#define MUL(a,b) (a\*b)

printf(%d\*%d=%dn, TOW, TOW, MUL(TOW,TOW));  
这行的宏会被展开为：  
printf(%d\*%d=%dn, (2), (2), ((2)\*(2)));  
MUL里的参数TOW会被展开为(2).

2, 当有’#'或’##’的时候  
#define A          (2)  
#define STR(s)     #s  
#define CONS(a,b)  int(a##e##b)

printf(“int max: %sn”,  STR(INT\_MAX));    // INT\_MAX ＃include<climits>  
这行会被展开为：  
printf(“int max: %sn”, #INT\_MAX);

printf(%sn, CONS(A, A));               // compile error  
这一行则是：  
printf(%sn, int(AeA));

INT\_MAX和A都不会再被展开, 然而解决这个问题的方法很简单. 加多一层中间转换宏.  
加这层宏的用意是把所有宏的参数在这层里全部展开, 那么在转换宏里的那一个宏(\_STR)就能得到正确的宏参数.

#define A           (2)  
#define \_STR(s)     #s  
#define STR(s)      \_STR(s)                 // 转换宏  
#define \_CONS(a,b)  int(a##e##b)  
#define CONS(a,b)   \_CONS(a,b)       // 转换宏

printf(int max: %sn, STR(INT\_MAX));          // INT\_MAX，int型的最大值，为一个变量 ＃include<climits>  
输出为: int max: 0x7fffffff  
STR(INT\_MAX) –>  \_STR(0x7fffffff) 然后再转换成字符串；

printf(%dn, CONS(A, A));  
输出为：200  
CONS(A, A)  –>  \_CONS((2), (2))  –> int((2)e(2))

三、’#'和’##’的一些应用特例  
1、合并匿名变量名  
#define  \_\_ANONYMOUS1(type, var, line)  type  var##line  
#define  \_ANONYMOUS0(type, line)  \_\_ANONYMOUS1(type, \_anonymous, line)  
#define  ANONYMOUS(type)  \_ANONYMOUS0(type, \_\_LINE\_\_)  
例：ANONYMOUS(static int);  即: static int \_anonymous70;  70表示该行行号；  
第一层：ANONYMOUS(static int);  –>  \_\_ANONYMOUS0(static int, \_\_LINE\_\_);  
第二层：–>  \_\_\_ANONYMOUS1(static int, \_anonymous, 70);  
第三层：–>  static int  \_anonymous70;  
即每次只能解开当前层的宏，所以\_\_LINE\_\_在第二层才能被解开；

2、填充结构  
#define  FILL(a)   {a, #a}

enum IDD{OPEN, CLOSE};  
typedef struct MSG{  
IDD id;  
const char \* msg;  
}MSG;

MSG \_msg[] = {FILL(OPEN), FILL(CLOSE)};  
相当于：  
MSG \_msg[] = {{OPEN, “OPEN”},  
{CLOSE, ”CLOSE“}};

3、记录文件名  
#define  \_GET\_FILE\_NAME(f)   #f  
#define  GET\_FILE\_NAME(f)    \_GET\_FILE\_NAME(f)  
static char  FILE\_NAME[] = GET\_FILE\_NAME(\_\_FILE\_\_);

4、得到一个数值类型所对应的字符串缓冲大小  
#define  \_TYPE\_BUF\_SIZE(type)  sizeof #type  
#define  TYPE\_BUF\_SIZE(type)   \_TYPE\_BUF\_SIZE(type)  
char  buf[TYPE\_BUF\_SIZE(INT\_MAX)];  
–>  char  buf[\_TYPE\_BUF\_SIZE(0x7fffffff)];  
–>  char  buf[sizeof 0x7fffffff];  
这里相当于：  
char  buf[11];