先看下简单的初始代码：注意其编译运行后的结果。  
  
root@xuanfei-desktop:~/cpropram/2# cat global.h //头文件  
#ifndef CLOBAL\_H  
        #define GLOBAL\_H  
        #include <stdio.h>  
        int funca(void);  
        int funcb(void);  
#endif  
root@xuanfei-desktop:~/cpropram/2# cat funca.c //函数a  
#include "global.h"  
int funca(void)  
{  
printf ("this is function\n");  
return 0;  
}  
root@xuanfei-desktop:~/cpropram/2# cat funcb.c //函数b  
#include "global.h"  
int funcb(void)  
{  
printf ("this is function\n");  
return 0;  
}  
root@xuanfei-desktop:~/cpropram/2# gcc -Wall funca.c funcb.c main.c //联合编译  
root@xuanfei-desktop:~/cpropram/2# ./a.out //运行  
this is main  
this is [**function**](javascript:;)  
this is main  
this is function  
this is main  
  
相同结果很难让人看出那里出错，下面我们用用 \_\_FILE\_\_,\_\_LINE\_\_,\_\_FUNCTION\_\_加入代码，看看有什么区别吗.  
把 \_\_FILE\_\_,\_\_LINE\_\_,\_\_FUNCTION\_\_加入到mail.c中  
root@xuanfei-desktop:~/cpropram/2# cat main.c  
#include "global.h"  
int main(int argc, char \*\*argv)  
{  
    printf("%s(%d)-%s: this is main\n",\_\_FILE\_\_,\_\_LINE\_\_,\_\_FUNCTION\_\_);  
    funca();  
    printf("%s(%d)-%s: this is main\n",\_\_FILE\_\_,\_\_LINE\_\_,\_\_FUNCTION\_\_);  
    funcb();  
    printf("%s(%d)-%s: this is main\n",\_\_FILE\_\_,\_\_LINE\_\_,\_\_FUNCTION\_\_);  
    return 0;  
}  
root@xuanfei-desktop:~/cpropram/2# gcc -Wall funca.c funcb.c main.c  
root@xuanfei-desktop:~/cpropram/2# ./a.out  
main.c(4)-main: this is main  
this is function  
main.c(6)-main: this is main  
this is function  
main.c(8)-main: this is main  
  
上面的结果main.c(4)-main:this is main 表示在mian.c源代码的第四行main函数里边打印出来的 this is main  
那样的话就很方便的让程序员对自己的程序进行排错！  
为了更方便的使用它我们可以通过在global.h代码中进行宏定义  
root@xuanfei-desktop:~/cpropram/2# cat global.h  
#ifndef CLOBAL\_H  
        #define GLOBAL\_H  
        #include <stdio.h>  
        int funca(void);  
        int funcb(void);  
        #define DEBUGFMT  "%s(%d)-%s"  
        #define DEBUGARGS \_\_FILE\_\_,\_\_LINE\_\_,\_\_FUNCTION\_\_  
#endif  
root@xuanfei-desktop:~/cpropram/2# cat funca.c  
#include "global.h"  
int funca(void)  
{  
printf (DEBUGFMT " this is function\n",DEBUGARGS);  
return 0;  
}  
root@xuanfei-desktop:~/cpropram/2# cat funcb.c  
#include "global.h"  
int funcb(void)  
{  
printf (DEBUGFMT " this is function\n",DEBUGARGS);  
return 0;  
}  
root@xuanfei-desktop:~/cpropram/2# cat main.c  
#include "global.h"  
int main(int argc, char \*\*argv)  
{  
    printf(DEBUGFMT "this is main\n", DEBUGARGS);  
    funca();  
    printf(DEBUGFMT "this is main\n", DEBUGARGS);  
    funcb();  
    printf(DEBUGFMT "this is main\n", DEBUGARGS);  
    return 0;  
}  
root@xuanfei-desktop:~/cpropram/2# gcc -Wall funca.c funcb.c main.c  
root@xuanfei-desktop:~/cpropram/2# ./a.out  
main.c(4)-mainthis is main  
funca.c(4)-funca this is function  
main.c(6)-mainthis is main  
funcb.c(4)-funcb this is function  
main.c(8)-mainthis is main  
root@xuanfei-desktop:~/cpropram/2#  
  
这就是通过定义\_\_FILE\_\_,\_\_LINE\_\_,FUNCTION\_\_的宏来简单实现代码的跟踪调试：）  
  
下面是一个可供调试用的头文件  
#ifndef \_GOLD\_DEBUG\_H  
#define \_GOLD\_DEBUG\_H

#ifdef \_\_cplusplus  
#if \_\_cplusplus  
extern "C"{  
#endif  
#endif /\* \_\_cplusplus \*/

//#define GI\_DEBUG

#ifdef GI\_DEBUG

#define GI\_DEBUG\_POINT()   printf("\n\n[File:%s Line:%d] Fun:%s\n\n", \_\_FILE\_\_, \_\_LINE\_\_, \_\_FUNCTION\_\_)  
#define dbg\_printf(arg...)   printf(arg);

#define GI\_ASSERT(expr)                                     \  
    do{                                                     \  
        if (!(expr)) { \  
            printf("\nASSERT failed at:\n  >File name: %s\n  >Function : %s\n  >Line No. : %d\n  >Condition: %s\n", \  
                    \_\_FILE\_\_,\_\_FUNCTION\_\_, \_\_LINE\_\_, #expr);\  
        } \  
    }while(0);

/\*调试宏, 用于暂停\*/  
#define GI\_DEBUG\_PAUSE()           \  
 do               \  
 {               \  
  GI\_DEBUG\_POINT();          \  
  printf("pause for debug, press 'q' to exit!\n");  \  
  char c;             \  
  while( ( c = getchar() ) )        \  
   {             \  
    if('q' == c)         \  
     {           \  
      getchar();        \  
      break;         \  
     }           \  
   }             \  
 }while(0);  
#define GI\_DEBUG\_PAUSE\_ARG(arg...)          \  
  do               \  
  {               \  
   printf(arg);           \  
   GI\_DEBUG\_PAUSE()          \  
  }while(0);

#define GI\_DEBUG\_ASSERT(expression)      \  
if(!(expression))                        \  
{                                  \  
    printf("[ASSERT],%s,%s:%d\n", \_\_FILE\_\_,  \_\_FUNCTION\_\_, \_\_LINE\_\_);\  
    exit(-1);             \  
}  
#else  
#define GI\_ASSERT(expr)  
#define GI\_DEBUG\_PAUSE()  
#define GI\_DEBUG\_PAUSE\_ARG(arg...)   
#define GI\_DEBUG\_POINT()  
#define dbg\_printf(arg...)  
#define GI\_DEBUG\_ASSERT(expression)

#endif

#ifdef \_\_cplusplus  
#if \_\_cplusplus  
}  
#endif  
#endif /\* \_\_cplusplus \*/

#endif

**C语言常用宏定义**

**01:** 防止一个头文件被重复包含  
#ifndef COMDEF\_H  
#define COMDEF\_H  
//头文件内容  
#endif  
**02:**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 char      boolean;     /\* Boolean value type. \*/  
typedef  unsigned long int  uint32;      /\* Unsigned 32 bit value \*/  
typedef  unsigned short     uint16;      /\* Unsigned 16 bit value \*/  
typedef  unsigned char      uint8;       /\* Unsigned 8  bit value \*/  
typedef  signed long int    int32;       /\* Signed 32 bit value \*/  
typedef  signed short       int16;       /\* Signed 16 bit value \*/  
typedef  signed char        int8;        /\* Signed 8  bit value \*/  
  
//下面的不建议使用  
typedef  unsigned char     byte;         /\* Unsigned 8  bit value type. \*/  
typedef  unsigned short    word;         /\* Unsinged 16 bit value type. \*/  
typedef  unsigned long     dword;        /\* Unsigned 32 bit value type. \*/  
typedef  unsigned char     uint1;        /\* Unsigned 8  bit value type. \*/  
typedef  unsigned short    uint2;        /\* Unsigned 16 bit value type. \*/  
typedef  unsigned long     uint4;        /\* Unsigned 32 bit value type. \*/  
typedef  signed char       int1;         /\* Signed 8  bit value type. \*/  
typedef  signed short      int2;         /\* Signed 16 bit value type. \*/  
typedef  long int          int4;         /\* Signed 32 bit value type. \*/  
typedef  signed long       sint31;       /\* Signed 32 bit value \*/  
typedef  signed short      sint15;       /\* Signed 16 bit value \*/  
typedef  signed char       sint7;        /\* Signed 8  bit value \*/  
  
**03:**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)))  
  
**04:**求最大值和最小值  
#define  MAX(x,y) (((x)>(y)) ? (x) : (y))  
#define  MIN(x,y) (((x) < (y)) ? (x) : (y))  
  
**05:**得到一个field在结构体(struct)中的偏移量  
#define FPOS(type,field) ((dword)&((type \*)0)->field)  
  
**06:**得到一个结构体中field所占用的字节数  
#define FSIZ(type,field) sizeof(((type \*)0)->field)  
  
**07:**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])  
  
**08:**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)  
  
**09:** 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)(xxx) & 255))  
#define WORD\_HI(xxx)  ((byte) ((word)(xxx) >> 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) - 0x20) : (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:** 返回一个无符号数n尾的值MOD\_BY\_POWER\_OF\_TWO(X,n)=X%(2^n)  
#define MOD\_BY\_POWER\_OF\_TWO( val, mod\_by ) ((dword)(val) & (dword)((mod\_by)-1))  
  
**18:** 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)))  
  
**19:** 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\_\_  
C++中还定义了 \_\_cplusplus  
  
如果编译器不是标准的,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。  
  
\_\_LINE\_\_ 及 \_\_FILE\_\_ 宏指示，#line指令可以改变它的值，简单的讲，编译时，它们包含程序的当前行数和文件名。  
  
\_\_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。  
\_\_TIME\_\_ 宏指令包含程序编译的时间。时间用字符串表示，其形式为： 分：秒  
\_\_STDC\_\_ 宏指令的意义是编译时定义的。一般来讲，如果\_\_STDC\_\_已经定义，编译器将仅接受不包含任何非标准扩展的标准C/C++代码。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。  
\_\_cplusplus 与标准c++一致的编译器把它定义为一个包含至少6为的数值。与标准c++不一致的编译器将使用具有5位或更少的数值。  
  
  
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  
   
  
**20：** 宏定义防止错误使用小括号包含。  
例如：  
有问题的定义：#define DUMP\_WRITE(addr,nr) {memcpy(bufp,addr,nr); bufp += nr;}  
应该使用的定义： #difne DO(a,b) do{a+b;a++;}while(0)  
例如：  
if(addr)  
    DUMP\_WRITE(addr,nr);  
else   
    do\_somethong\_else();  
宏展开以后变成这样:  
if(addr)  
    {memcpy(bufp,addr,nr); bufp += nr;};  
else  
    do\_something\_else();  
  
gcc 在碰到else前面的“；”时就认为if语句已经结束，因而后面的else不在if语句中。而采用do{} while(0)的定义，在任何情况下都没有问题。而改为 #difne DO(a,b) do{a+b;a++;}while(0) 的定义则在任何情况下都不会出错。