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 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 */
```

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)(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

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。

_LINE_及_FILE_宏指令在有关#line 的部分中已讨论，这里讨论其余的宏名。

_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。

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
```

20. 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)
```

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>
```

```
＃i nclude<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("%d\n", 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=%d\n", TOW, TOW, MUL(TOW,TOW));
```

这行的宏会被展开为：

```
printf("%d*%d=%d\n", (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: %s\n",    STR(INT_MAX));       // INT_MAX  ＃i nclude<climits>
```

这行会被展开为：

```
printf("int max: %s\n", "INT_MAX");
printf("%s\n", CONS(A, A));                    // compile error
```

这一行则是：

```
printf("%s\n", 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:  %s\n",  STR(INT_MAX));              // INT_MAX,int 型的最大值，为一个变量 ＃i nclude<climits>
```

输出为: int max: 0x7fffffff

STR(INT_MAX) -->   _STR(0x7fffffff) 然后再转换成字符串；

printf("%d\n", 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];

==========
==========
====

C（和 C++）中的宏（Macro）属于编译器预处理的范畴，属于编译期概念（而非运行期概念）。下面对常遇到的宏的使用问题做了简单总结。

宏使用中的常见的基础问题

#符号和##符号的使用

...符号的使用

宏的解释方法

我们能碰到的宏的使用

宏使用中的陷阱

常见的基础性问题

关于#和##

在 C 语言的宏中，#的功能是将其后面的宏参数进行字符串化操作（Stringfication），简单说就是在对它所引用的宏变量通过替换后在其左右各加上一个双引号。比如下面代码中的宏：

```
#define WARN_IF(EXP)       \
    do{ if (EXP)       \
            fprintf(stderr, "Warning: " #EXP "\n"); }      \
    while(0)
```

那么实际使用中会出现下面所示的替换过程：

```
WARN_IF (divider == 0);
```

被替换为

```
do {
    if (divider == 0)
            fprintf(stderr, "Warning" "divider == 0" "\n");
} while(0);
```

这样每次 divider（除数）为 0 的时候便会在标准错误流上输出一个提示信息。

而##被称为连接符（concatenator），用来将两个 Token 连接为一个 Token。注意这里连接的对象是 Token 就行，而不一定是宏的变量。比如你要做一个菜单项命令名和函数指针组成的结构体的数组，并且希望在函数名和菜单项命令名之间有直观的、名字上的关系。那么下面的代码就非常实用：

```
struct command
{
    char * name;
    void (*function) (void);
};

#define COMMAND(NAME) { NAME, NAME ## _command }

// 然后你就用一些预先定义好的命令来方便的初始化一个 command 结构的数组了：

struct command commands[] = {
    COMMAND(quit),
    COMMAND(help),
    ...
}
```

COMMAND 宏在这里充当一个代码生成器的作用，这样可以在一定程度上减少代码密度，间接地

也可以减少不留心所造成的错误。我们还可以 n 个##符号连接 n+1 个 Token，这个特性也是#符号所不具备的。比如：

```
#define LINK_MULTIPLE(a,b,c,d) a##_##b##_##c##_##d

typedef struct _record_type LINK_MULTIPLE(name,company,position,salary);
// 这里这个语句将展开为：
//    typedef struct _record_type name_company_position_salary;
```

关于...的使用

...在 C 宏中称为 Variadic Macro，也就是变参宏。比如：

```
#define myprintf(templt,...) fprintf(stderr,templt,__VA_ARGS__)
// 或者
#define myprintf(templt,args...) fprintf(stderr,templt,args)
```

第一个宏中由于没有对变参起名，我们用默认的宏__VA_ARGS__来替代它。第二个宏中，我们显式地命名变参为 args，那么我们在宏定义中就可以用 args 来代指变参了。同 C 语言的 stdcall 一样，变参必须作为参数表的最有一项出现。当上面的宏中我们只能提供第一个参数 templt 时，C 标准要求我们必须写成：

```
myprintf(templt,);
```

的形式。这时的替换过程为：

```
myprintf("Error!\n",);
```

替换为：

```
fprintf(stderr,"Error!\n",);
```

这是一个语法错误，不能正常编译。这个问题一般有两个解决方法。首先，GNU CPP 提供的解决方法允许上面的宏调用写成：

```
myprintf(templt);
```

而它将会被通过替换变成：

```
fprintf(stderr,"Error!\n",);
```

很明显，这里仍然会产生编译错误（非本例的某些情况下不会产生编译错误）。除了这种方式外，c99 和 GNU CPP 都支持下面的宏定义方式：

```
#define myprintf(templt, ...) fprintf(stderr,templt, ##__VAR_ARGS__)
```

这时，##这个连接符号充当的作用就是当__VAR_ARGS__为空的时候，消除前面的那个逗号。那么此时的翻译过程如下：

```
myprintf(templt);
```

被转化为：

```
fprintf(stderr,templt);
```

这样如果 templt 合法，将不会产生编译错误。

宏是如何解释的

宏在日常编程中的常见使用

宏使用中的陷阱

这里列出了一些宏使用中容易出错的地方，以及合适的使用方式。

错误的嵌套－Misnesting

宏的定义不一定要有完整的、配对的括号，但是为了避免出错并且提高可读性，最好避免这样使用。

由操作符优先级引起的问题－Operator Precedence Problem

由于宏只是简单的替换，宏的参数如果是复合结构，那么通过替换之后可能由于各个参数之间的操作符优先级高于单个参数内部各部分之间相互作用的操作符优先级，如果我们不用括号保护各个宏

参数，可能会产生预想不到的情形。比如：

```
#define ceil_div(x, y) (x + y - 1) / y
```

那么

```
a = ceil_div( b & c, sizeof(int) );
```

将被转化为：

```
a = ( b & c + sizeof(int) - 1) / sizeof(int);
    // 由于+/-的优先级高于&的优先级，那么上面式子等同于：
a = ( b & (c + sizeof(int) - 1)) / sizeof(int);
```

这显然不是调用者的初衷。为了避免这种情况发生，应当多写几个括号：

```
#define ceil_div(x, y) (((x) + (y) - 1) / (y))
```

消除多余的分号－Semicolon Swallowing

通常情况下，为了使函数模样的宏在表面上看起来像一个通常的 C 语言调用一样，通常情况下我们在宏的后面加上一个分号，比如下面的带参宏：

```
MY_MACRO(x);
```

但是如果是下面的情况：

```
#define MY_MACRO(x) {           \
    /* line 1 */    \
    /* line 2 */    \
    /* line 3 */ }

//...

if (condition())
    MY_MACRO(a);
else
    {...}
```

这样会由于多出的那个分号产生编译错误。为了避免这种情况出现同时保持 MY_MACRO(x);的这种写法，我们需要把宏定义为这种形式：

```
#define MY_MACRO(x) do {
    /* line 1 */    \
    /* line 2 */    \
    /* line 3 */ } while(0)
```

这样只要保证总是使用分号，就不会有任何问题。

Duplication of Side Effects

这里的 Side Effect 是指宏在展开的时候对其参数可能进行多次 Evaluation（也就是取值），但是如果这个宏参数是一个函数，那么就有可能被调用多次从而达到不一致的结果，甚至会发生更严重的错误。比如：

```
#define min(X,Y) ((X) > (Y) ? (Y) : (X))

    //...

c = min(a,foo(b));
```

这时 foo()函数就被调用了两次。为了解决这个潜在的问题，我们应当这样写 min(X,Y)这个宏：

```
#define min(X,Y) ({ \
```

```
    typeof (X) x_ = (X);    \
    typeof (Y) y_ = (Y);    \
    (x_ < y_) ? x_ : y_; })
```

({...})的作用是将内部的几条语句中最后一条的值返回，它也允许在内部声明变量（因为它通过大括号组成了一个局部 Scope）。

自引用宏的使用－Self-Referential Macros

Argument Prescan

==========

==========

几个常用道的 macro

1、macro

(1)#error msg 指令使編譯器停止執行並打印一條語句，

(2)printf("%d,%s",_LINE_,_FILE_)打印當前行號和文件名

(3)#pragma arg 設置了編譯器所使用的條件.

(4)#str 將字符串放入被引號括起來的字符串中，如：

```
#define pr(s) puts(#s)
pr(hello world);->puts("hello world");
```

(5)將兩段文字拼接在一起，如：

```
#define call(verb,adj,do) verb##_##adj(do)
call(shrink, all, trees);->shrink_all(trees);
#define SIGNAL(x) "SIGNAL:" #x
SIGNAL(finishedParsing())
```

2、給結構賦初值

```
struct AAA
{
  AAA():a(10){}
  int a;
  int b;
};
```

3、使用 macro 批定義變量

```
#define STR_EACH_DECLARE(macro)\
 macro(age)\
 macro(name)
#define STR_DECLARE(name) char name ## Str;
STR_EACH_DECLARE(STR_DECLARE)
void main()
{
 int a=1;
 nameStr='a';
 ageStr='b';
}
```

4、定位參數
5、指針和常量
const type *
type const *
type * const
6、友元函數和友元類
7、拷貝構造函數和賦值語句
8、#pragma once
==========
==========
====
用字符串调用同名的函数
str.def:
my_Macro(func1),
my_Macro(func2)
str.h:
#pragma once
void func1();
void func2();
#define my_Macro(a) a
typedef void (*func)();
func fArray[]=
{
#include "str.def"
};
#undef my_Macro
#define my_Macro(a) #a
char* str[]=
{
#include "str.def"
};
str.cpp:
#include "str.h"
#include "string.h"
void func1(){}
void func2(){}
void main()
{
char call[]="func1";
int len=sizeof(fArray)/sizeof(fArray[0]);
for(int i=0; i<len; i++)
{
if(strcmp(str[i],call)==0)
{

```
    fArray[i]();
    break;
   }
  }
}
```

==========

浅析 C++里面的宏

说到宏，恐怕大家都能说出点东西来：一种预处理，没有分号（真的吗？）。然后呢？

嗯.....茫然中.....

好吧，我们就从这开始说起。

最常见的宏恐怕是#include 了，其次就是#define 还有.....

还是从宏的用途分类吧：

1、#include 主要用于包含引用文件，至今其地位无人能替代；

2、注释掉代码。例如：

```
#if 0
.....
#endif;
```

这种机制是目前注释掉代码的最佳选择，为摩托罗拉公司员工所普遍采用；

3、代码版本管理。例如：

```
#ifdef DEBUG
file://调试版本
#else
file://非调试版本
#endif;
```

4、声明宏。例如：

```
#define DECLARE_MESSAGE(x) x();~x()     file://有没有分号？哈哈
//.....
class A
{
public:
        DECLARE_MESSAGE(A);
.............
}
```

想起什么了，呵呵：）对,VC 里面有好多这样的东东，有空我会写《我的 VC 历程》，到时候会把 VC 里的各种宏详细的解释一下，那可是一个庞大的工程：）

5、符号常量。例如：

```
#define PI 3.14159
```

6、内联函数。例如：

```
#define CLEAR(x) ((x)=0)
```

7、泛型函数。例如:

```
#define ABS(x) ((x)>0? (x):-(x))
```

x=3 没问题！ x=1.3 也没问题！

如果是这样呢:

```
#include <iostream.h>
#define A(x) ((x)>0? (x):-(x))
void main()
{
int i=-1;
cout<<A(1)<<endl;
cout<<A(++i)<<endl;
}
```

有问题了，不过以后再说，大概讲 const or inline 时会说的:）

8、泛型类型。例如：

```
#define Stack(T)  Stack__ ##T
#define Stackdeclare(T)  class Stack(T) {.....}
Stackdeclare(int);
Stackdeclare(char);
.......
Stack(int) s1;
Stack(char) s2;
```

9、语法扩展。例如：

```
Set<int> s;//假设 Set 为一个描述集合的类
int i;
FORALL(i,s);
.......
```

宏最大的问题便是易引起冲突，例如：

libA.h:

```
#define MACRO stuff
```

同时：

libB.h:

```
#define MACRO stuff
```

下面我们对他们进行引用：

user.cpp:

```
#include "libA.h"
#include "libB.h"
.............
```

糟糕，出现了重定义！

还有一种冲突的可能：

libB.h:（没有定义宏 MACRO)

```
class x    { void MACRO();   .....};
```

那么程序运行期间，libA.h 中的宏讲会改变 libB.h 中的成员函数的名字，导致不可预料的结果。

宏的另一个问题，便是如 7 中出现的问题，如果你把 7 中的 x 设为'a'，程序也不会给出任何警告，所以他是不安全的。

针对以上的问题，我们说：

1、尽可能的少用公用宏,能替换掉就替换掉；

2、对那些不能替换的宏，使用命名约定；

1、符号常量预处理程序我们可以用 const or enum 来代替：

```
const int TABLESIZE=1024;
enum {    TABLESIZE=1024    };
```

2、非泛型内联函数的预处理程序可以使用真正的内联函数来代替：

```
inline void clear(int& x) {x=0;}
```

奥，对了，还有这样一种情况：

```
#define    CONTROL(c)    ((c)-64)
.........
switch(c)
{
    case CONTROL('a') :    ......
    case CONTROL('b') :    ......
    case CONTROL('c') :    ......
    case CONTROL('d') :    ......
```

详细出处参考：http://www.itqun.net/content-detail/71013_2.html

```
.........
    }
```

这时候就不能单独使用内联函数来取代了，因为 case 标签禁止函数调用，我们只好做如下转换：

```
inline char control(char c) { return c+64; }
.........
switch(control(c))
{
   case 'a':.....
   case 'b':.....
   case 'c':.....
   case 'd':.....
   ........
}
```

当然这样做是以牺牲时间作为代价的（你想想为什么:)）

3、对于泛型预处理程序，我们可以用函数模板或类默板来代替：

```
template<class T>
T ABS(const T& t) { return t>0 ? t : -t;    }
template<class T>
Class Stack     {    .............      };
```

4、最后对于语法扩展程序几乎都可以用一个或多个 C++类代替：

```
Set<int> s;
int i;
Set_iter<int> iter(s);
while(iter.next(i))
...........
```

与使用宏相比，我们只是牺牲了一点程序的简洁性而已。

当然并不是所有的宏都能替换（我们也并不主张替换掉所有的宏!），对于不能替换的宏，我们应该对他们实行命名约定，例如：

#define COMPANY_XYZ_LIBABC_MACRO stuff

同时我们也要采取一定的方法，进行预防：

#ifndef COMPANY_XYZ_LIBABC_MACRO
#define COMPANY_XYZ_LIBABC_MACRO stuff
#endif

当然，在程序库实现内部定义的宏没有这个约束：

my.cpp:
#define MACRO stuff
.....

我们给出几个常见的宏：

#define A(x) T_##x
#define Bx) #@x
#define Cx) #x

我们假设：x=1，则有：

A(1)=======T_1
B(1)======'1'
C(1)======"1"

还有一个比较常见的宏：_T

TCHAR tStr[] = _T("t code");

_T 宏的作用就是转换成 TCHAR

详细出处参考：http://www.itqun.net/content-detail/71013_3.html