define定义详解

## ****1. 简单的define定义****

#define MAXTIME 1000

一个简单的MAXTIME就定义好了，它代表1000，如果在程序里面写

if(i<MAXTIME){.........}

编译器在处理这个代码之前会对MAXTIME进行处理替换为1000。

这样的定义看起来类似于普通的常量定义CONST，但也有着不同，因为define的定义更像是简单的文本替换，而不是作为一个量来使用，这个问题在下面反映的尤为突出。

## ****2.define的“函数定义”****

define可以像函数那样接受一些参数，如下

#define max(x,y) (x)>(y)?(x):(y);

这个定义就将返回两个数中较大的那个，看到了吗？因为这个“函数”没有类型检查，就好像一个函数模板似的，当然，它绝对没有模板那么安全就是了。可以作为一个简单的模板来使用而已。

但是这样做的话存在隐患，例子如下：  
#define Add(a,b) a+b;  
在一般使用的时候是没有问题的，但是如果遇到如：c \* Add(a,b) \* d的时候就会出现问题，代数式的本意是a+b然后去和c，d相乘，但是因为使用了define（它只是一个简单的替换），所以式子实际上变成了  
c\*a + b\*d

另外举一个例子：  
#define pin (int\*);  
pin a,b;  
本意是a和b都是int型指针，但是实际上变成int\* a,b;  
a是int型指针，而b是int型变量。  
这是应该使用typedef来代替define，这样a和b就都是int型指针了。

所以我们在定义的时候，养成一个良好的习惯，建议所有的层次都要加括号。

## ****3.宏的单行定义（少见用法）****

#define A(x) T\_##x  
#define B（x) #@x  
#define C（x) #x

我们假设：x=1，则有：

A(1)------〉T\_1  
B(1)------〉'1'  
C(1)------〉"1"

（这里参考了 hustli的文章）

## ****3.define的多行定义****

define可以替代多行的代码，例如MFC中的宏定义（非常的经典，虽然让人看了恶心）

#define MACRO(arg1, arg2) do { \  
/\* declarations \*/ \  
stmt1; \  
stmt2; \  
/\* ... \*/ \  
} while(0) /\* (no trailing ; ) \*/  
关键是要在每一个换行的时候加上一个"\"

## ****4.在大规模的开发过程中，特别是跨平台和系统的软件里，define最重要的功能是条件编译。****

就是：  
#ifdef WINDOWS  
......  
......  
#endif  
#ifdef LINUX  
......  
......  
#endif

可以在编译的时候通过#define设置编译环境

## ****`****

//定义宏  
#define [MacroName] [MacroValue]  
//取消宏  
#undef [MacroName]  
//普通宏  
#define PI (3.1415926)

带参数的宏  
#define max(a,b) ((a)>(b)? (a),(b))  
关键是十分容易产生错误，包括机器和人理解上的差异等等。

**6.条件编译**  
#ifdef XXX…(#else) … #endif  
例如  
#ifdef DV22\_AUX\_INPUT  
#define AUX\_MODE 3   
#else  
#define AUY\_MODE 3  
#endif  
#ifndef XXX … (#else) … #endif

**7.头文件(.h)可以被头文件或C文件包含；**  
重复包含（重复定义）  
由于头文件包含可以嵌套，那么C文件就有可能包含多次同一个头文件，就可能出现重复定义的问题的。  
通过条件编译开关来避免重复包含（重复定义）  
例如  
#ifndef \_\_headerfileXXX\_\_  
#define \_\_headerfileXXX\_\_  
…  
//文件内容  
…  
#endif

[C++宏定义详解](http://blog.chinaunix.net/uid-21372424-id-119797.html)

# 一、#define的基本用法

    #define是C语言中提供的宏定义命令，其主要目的是为程序员在编程时提供一定的方便，并能在一定程度上提高程序的运行效率，但学生在学习时往往不能 理解该命令的本质，总是在此处产生一些困惑，在编程时误用该命令，使得程序的运行与预期的目的不一致，或者在读别人写的程序时，把运行结果理解错误，这对 C语言的学习很不利。

## 1　#define命令剖析

### 1.1   #define的概念

    #define命令是C语言中的一个宏定义命令，它用来将一个标识符定义为一个字符串，该标识符被称为宏名，被定义的字符串称为替换文本。  
该命令有两种格式：一种是简单的宏定义，另一种是带参数的宏定义。

**(1)简单的宏定义：**

1. #define <宏名>　　<字符串>
2. 例： #define PI 3.1415926

**(2) 带参数的宏定义**

1. #define <宏名> (<参数表>) <宏体>
2. 例： #define A(x) x

    一个标识符被宏定义后，该标识符便是一个宏名。这时，在程序中出现的是宏名，在该程序被编译前，先将宏名用被定义的字符串替换，这称为宏替换，替换后才进行编译，宏替换是简单的替换。

### 1.2 宏替换发生的时机

    为了能够真正理解#define的作用，让我们来了解一下对C语言源程序的处理过程。当我们在一个集成的开发环境如Turbo C中将编写好的源程序进行编译时，实际经过了预处理、编译、汇编和连接几个过程。其中预处理器产生编译器的输出，它实现以下的功能：  
（1）文件包含  
    可以把源程序中的#include 扩展为文件正文，即把包含的.h文件找到并展开到#include 所在处。  
（2）条件编译  
    预处理器根据#if和#ifdef等编译命令及其后的条件，将源程序中的某部分包含进来或排除在外，通常把排除在外的语句转换成空行。  
（3）宏展开  
    预处理器将源程序文件中出现的对宏的引用展开成相应的宏 定义，即本文所说的#define的功能，由预处理器来完成。  
    经过预处理器处理的源程序与之前的源程序有所有不同，在这个阶段所进行的工作只是纯粹的替换与展开，没有任何计算功能，所以在学习#define命令时只要能真正理解这一点，这样才不会对此命令引起误解并误用。

## 2　#define使用中的常见问题解析

### 2.1 简单宏定义使用中出现的问题

    在简单宏定义的使用中，当替换文本所表示的字符串为一个表达式时，容易引起误解和误用。如下例：

1. 例1 #define N 2+2
2. void main()
3. {
4. int a=N\*N;
5. printf(“%d”,a);
6. }

**(1) 出现问题：**

    在此程序中存在着宏定义命令，宏N代表的字符串是2+2，在程序中有对宏N的使用，一般同学在读该程序时，容易产生的问题是先求解N为2＋2＝4，然后在程序中计算a时使用乘法，即N\*N=4\*4=16,其实该题的结果为8，为什么结果有这么大的偏差?

**(2) 问题解析：**

    如1节所述，宏展开是在预处理阶段完成的，这个阶段把替换文本只是看作一个字符串，并不会有任何的计算发生，在展开时是在宏N出现的地方 只是简单地使用串2＋2来代替N，并不会增添任何的符号，所以对该程序展开后的结果是a=2+2\*2+2，计算后=8，这就是宏替换的实质，如何写程序才能完成结果为16的运算呢？

**(3)解决办法：**

1. /\*将宏定义写成如下形式\*/
2. #define N (2+2)
3. /\*这样就可替换成(2+2)\*(2+2)=16\*/

### 2.2 带参数的宏定义出现的问题

    在带参数的宏定义的使用中，极易引起误解。例如我们需要做个宏替换能求任何数的平方，这就需要使用参数，以便在程序中用实际参数来替换宏定义中的参数。一般学生容易写成如下形式：

1. #define area(x) x\*x
2. /\*这在使用中是很容易出现问题的，看如下的程序\*/
3. void main()
4. {
5. int y = area(2+2);
6. printf(“%d”,y);
7. }

    按理说给的参数是2+2，所得的结果应该为4\*4=16，但是错了，因为该程序的实际结果为8，仍然是没能遵循纯粹的简单替换的规则，又是先计算再替换 了，在这道程序里，2+2即为area宏中的参数，应该由它来替换宏定义中的x，即替换成2+2\*2+2=8了。那如果遵循(1)中的解决办法，把2+2 括起来，即把宏体中的x括起来，是否可以呢？#define area(x) (x)\*(x)，对于area(2+2)，替换为(2+2)\*(2+2)=16，可以解决，但是对于area(2+2)/area(2+2)又会怎么样呢，有的学生一看到这道题马上给出结果，因为分子分母一样，又错了，还是忘了遵循先替换再计算的规则了，这道题替换后会变为 (2+2)\*(2+2)/(2+2)\*(2+2)即4\*4/4\*4按照乘除运算规则，结果为16/4\*4=4\*4=16，那应该怎么呢？解决方法是在整个宏体上再加一个括号，即#define   area(x) ((x)\*(x))，不要觉得这没必要，没有它，是不行的。  
    要想能够真正使用好宏定义，那么在读别人的程序时，一定要记住先将程序中对宏的使用全部替换成它所代表的字符串，不要自作主张地添加任何其他符号，完全展开后再进行相应的计算，就不会写错运行结果。

    如果是自己编程使用宏替换，则在使用简单宏定义时，当字符串中不只一个符号时，加上括号表现出优先级，如果是带参数的宏定义，则要给宏体中的每个参数加上括号，并在整个宏体上再加一个括号。看到这里，不禁要问，用宏定义这么麻烦，这么容易出错，可不可以摒弃它， 那让我们来看一下在C语言中用宏定义的好处吧。

如：

1. #include <iostream.h>
2. #define product(x)    x\*x
3. int main()
4. {
5. int i=3;
6. int j,k;
7. j = product(i++);
8. cout<<"j="<<j<<endl;
9. cout<<"i="<<i<<endl;
10. k = product(++i);
11. cout<<"k="<<k<<endl;
12. cout<<"i="<<i<<endl;
13. return 0;
14. }

依次输出结果：

j=9;i=5;k=49;i=7

## 3 宏定义的优点

**(1)   方便程序的修改**

    使用简单宏定义可用宏代替一个在程序中经常使用的常量，这样在将该常量改变时，不用对整个程序进行修改，只修改宏定义的字符串即可，而且当常量比较长时， 我们可以用较短的有意义的标识符来写程序，这样更方便一些。我们所说的常量改变不是在程序运行期间改变，而是在编程期间的修改，举一个大家比较熟悉的例子，圆周率π是在数学上常用的一个值，有时我们会用3.14来表示，有时也会用3.1415926等，这要看计算所需要的精度，如果我们编制的一个程序中 要多次使用它，那么需要确定一个数值，在本次运行中不改变，但也许后来发现程序所表现的精度有变化，需要改变它的值， 这就需要修改程序中所有的相关数值，这会给我们带来一定的不便，但如果使用宏定义，使用一个标识符来代替，则在修改时只修改宏定义即可，还可以减少输入 3.1415926这样长的数值多次的情况，我们可以如此定义 #define   pi   3.1415926，既减少了输入又便于修改，何乐而不为呢？

**(2) 提高程序的运行效率**

    使用带参数的宏定义可完成函数调用的功能，又能减少系统开销，提高运行效率。正如C语言中所讲，函数的使用可以使程序更加模块化，便于组织，而且可重复利用，但在发生函数调用时，需要保留调用函数的现场，以便子 函数执行结束后能返回继续执行，同样在子函数执行完后要恢复调用函数的现场，这都需要一定的时间，如果子函数执行的操作比较多，这种转换时间开销可以忽 略，但如果子函数完成的功能比较少，甚至于只完成一点操作，如一个乘法语句的操作，则这部分转换开销就相对较大了，但使用带参数的宏定义就不会出现这个问 题，因为它是在预处理阶段即进行了宏展开，在执行时不需要转换，即在当地执行。宏定义可完成简单的操作，但复杂的操作还是要由函数调用来完成，而且宏定义所占用的目标代码空间相对较大。所以在使用时要依据具体情况来决定是否使用宏定义。

## 4 结语

本文对C语言中宏定义#define在使用时容易出现的问题进行了解析，并从C源程序处理过程的角度对#define的处理进行了分析，也对它的优点进行 了阐述。只要能够理解宏展开的规则，掌握使用宏定义时，是在预处理阶段对源程序进行替换，只是用对应的字符串替换程序中出现的宏名，这样就可在正确使用的 基础上充分享受使用宏定义带来的方便和效率了

# 二、define中的三个特殊符号：#，##，#@

1. #define Conn(x,y) x##y
2. #define ToChar(x) #@x
3. #define ToString(x) #x

**(1)x##y表示什么？表示x连接y，举例说：**

1. int n = Conn(123,456); /\* 结果就是n=123456;\*/
2. char\* str = Conn("asdf", "adf"); /\*结果就是 str = "asdfadf";\*/

**（2）再来看**[**#@x**](mailto:#@x)**，其实就是给x加上单引号，结果返回是一个const char。举例说：**

char a = ToChar(1);结果就是a='1';  
做个越界试验char a = ToChar(123);结果就错了;  
但是如果你的参数超过四个字符，编译器就给给你报错了！

error C2015: too many characters in constant   ：P

**(3）最后看看#x,估计你也明白了，他是给x加双引号**

char\* str = ToString(123132);就成了str="123132";

# 三、常用的一些宏定义

**1 防止一个头文件被重复包含**

1. #ifndef BODYDEF\_H
2. #define BODYDEF\_H
3. //头文件内容
4. #endif

**2 得到指定地址上的一个字节或字**

1. #define MEM\_B( x ) ( \*( (byte \*) (x) ) )
2. #define MEM\_W( x ) ( \*( (word \*) (x) ) )

用法如下：

1. #include <iostream>
2. #include <windows.h>
3. #define MEM\_B(x) (\*((byte\*)(x)))
4. #define MEM\_W(x) (\*((WORD\*)(x)))
5. int main()
6. {
7. int bTest = 0x123456;
8. byte m = MEM\_B((&bTest));/\*m=0x56\*/
9. int n = MEM\_W((&bTest));/\*n=0x3456\*/
10. return 0;
11. }

**3 得到一个field在结构体(struct)中的偏移量**

1. #define OFFSETOF( type, field ) ( (size\_t) &(( type \*) 0)-> field )

**请参考文章：详解写宏定义：得到一个field在结构体（struct type）中的偏移量**。

**4 得到一个结构体中field所占用的字节数**

1. #define FSIZ( type, field ) sizeof( ((type \*) 0)->field )

**5 得到一个变量的地址（word宽度）**

1. #define B\_PTR( var ) ( (byte \*) (void \*) &(var) )
2. #define W\_PTR( var ) ( (word \*) (void \*) &(var) )

**6 将一个字母转换为大写**

1. #define UPCASE( c ) ( ((c) >= ''a'' && (c) <= ''z'') ? ((c) - 0x20) : (c) )

**7 判断字符是不是10进值的数字**

1. #define DECCHK( c ) ((c) >= ''0'' && (c) <= ''9'')

**8 判断字符是不是16进值的数字**

1. #define HEXCHK( c ) ( ((c) >= ''0'' && (c) <= ''9'') ||((c) >= ''A'' && (c) <= ''F'') ||((c) >= ''a'' && (c) <= ''f'') )

**9 防止溢出的一个方法**

1. #define INC\_SAT( val ) (val = ((val)+1 > (val)) ? (val)+1 : (val))

**10 返回数组元素的个数**

1. #define ARR\_SIZE( a ) ( sizeof( (a) ) / sizeof( (a[0]) ) )

**11 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：

1. \_LINE\_ /\*(两个下划线)，对应%d\*/
2. \_FILE\_ /\*对应%s\*/
3. \_DATE\_ /\*对应%s\*/
4. \_TIME\_ /\*对应%s\*/