以下内容转自：候捷 的 Effective C++

标题的另外一层意思：尽量用编译器，少用预处理器。

因为#define 经常被认为好象不是语言本身的一部分。这是问题之一。再看下面的语句：  
      #define ASPECT\_RATIO 1.653

编译器会永远也看不到ASPECT\_RATIO 这个符号名，因为在源码进入编译器之前，它会被预处理程序去掉，于是ASPECT\_RATIO 不会加入到符号列表中。如果涉及到这个常量的代码在编译时报错，就会很令人费解，因为报错信息指的是1.653，而不是ASPECT\_RATIO。如果ASPECT\_RATIO 不是在你自己写的头文件中定义的，你就会奇怪1.653 是从哪里来的，甚至会花时间跟踪下去。这个问题也会出现在符号调试器中，因为同样地，你所写的符号名不会出现在符号列表中。

解决这个问题的方案很简单：不用预处理宏，定义一个常量：  
const double ASPECT\_RATIO = 1.653;

这种方法很有效。但有两个特殊情况要注意。

首先，定义指针常量时会有点不同。因为常量定义一般是放在头文件中（许多源文件会包含它），除了指针所指的类型要定义成const 外，重要的是指针也经常要定义成const。例如，要在头文件中定义一个基于char\*的字符串常量，  
你要写两次const：

const char \* const authorName = "Scott Meyers";

另外，定义某个类(class)的常量一般也很方便，只有一点点不同。要把常量限制在类中，首先要使它成为类的成员；为了保证常量最多只有一份拷贝，还要把它定义为静态成员：  
class GamePlayer {  
private:  
static const int NUM\_TURNS = 5;   // constant declaration  
int scores[NUM\_TURNS];               // use of constant  
...  
};  
还有一点，正如你看到的，上面的语句是NUM\_TURNS 的声明，而不是定义，  
所以你还必须在类的实现代码文件中定义类的静态成员：  
const int GamePlayer::NUM\_TURNS; // mandatory definition;  
// goes in class impl. file

你不必过于担心这种小事。如果你忘了定义，链接器会提醒你。

旧一点的编译器会不接受这种语法，因为它认为类的静态成员在声明时定义初始值是非法的；而且，类内只允许初始化整数类型(如：int, bool, char 等)，还只能是常量。  
在上面的语法不能使用的情况下，可以在定义时赋初值：  
class EngineeringConstants { // this goes in the class  
private: // header file  
static const double FUDGE\_FACTOR;  
...  
};  
// this goes in the class implementation file  
const double EngineeringConstants::FUDGE\_FACTOR = 1.35;  
大多数情况下你只要做这么多。唯一例外的是当你的类在编译时需要用到这个类的常量的情况，例如上面GamePlayer::scores 数组的声明(编译过程中编译器一定要知道数组的大小)。所以，为了弥补那些(不正确地)禁止类内进行整型类常量初始化的编译器的不足，可以采用称之为“借用enum”的方法来解决。这种技术很好地利用了当需要int 类型时可以使用枚举类型的原则，所以  
GamePlayer 也可以象这样来定义：  
class GamePlayer {  
private:  
enum { NUM\_TURNS = 5 }; // "the enum hack" — makes  
// NUM\_TURNS a symbolic name  
// for 5

int scores[NUM\_TURNS]; // fine  
...  
};  
除非你正在用老的编译器(即写于1995 年之前)，你不必借用enum。当然，知道有这种方法还是值得的，因为这种可以追溯到很久以前的时代的代码可是不常见的哟。  
回到预处理的话题上来。另一个普遍的#define 指令的用法是用它来实现那些看起来象函数而又不会导致函数调用的宏。典型的例子是计算两个对象的最大值：  
#define max(a,b) ((a) > (b) ? (a) : (b))  
这个语句有很多缺陷，光想想都让人头疼，甚至比在高峰时间到[高速公路](https://www.baidu.com/s?wd=%E9%AB%98%E9%80%9F%E5%85%AC%E8%B7%AF&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)去开车还让人痛苦。  
无论什么时候你写了象这样的宏，你必须记住在写宏体时对每个参数都要加上括号；否则，别人调用你的宏时如果用了表达式就会造成很大的麻烦。但是即使你象这样做了，还会有象下面这样奇怪的事发生：  
int a = 5, b = 0;  
max(++a, b); // a 的值增加了2 次  
max(++a, b+10); // a 的值只增加了1 次

这种情况下，max 内部发生些什么取决于它比较的是什么值！

幸运的是你不必再忍受这样愚笨的语句了。你可以用普通函数实现宏的效率，再加上可预计的行为和类型安全，这就是内联函数(见条款33)：  
inline int max(int a, int b) { return a > b ? a : b; }  
不过这和上面的宏不大一样，因为这个版本的max 只能处理int 类型。但模板可以很轻巧地解决这个问题：  
template<class T>  
inline const T& max(const T& a, const T& b)  
{ return a > b ? a : b; }  
这个模板产生了一整套函数，每个函数拿两个可以转换成同种类型的对象进行比较然后返回较大的(常量)对象的引用。因为不知道T 的类型，返回时传递引用可以提高效率(见条款22)。  
顺便说一句，在你打算用模板写象 max 这样有用的通用函数时，先检查一下标准库(见条款49)，看看他们是不是已经存在。比如说上面说的max，你会惊喜地发现你可以后人乘凉：max 是C++标准库的一部分。  
有了const 和inline，你对预处理的需要减少了，但也不能完全没有它。抛弃#include 的日子还很远，#ifdef/#ifndef 在控制编译的过程中还扮演重要角色。  
预处理还不能退休，但你一定要计划给它经常放长假。