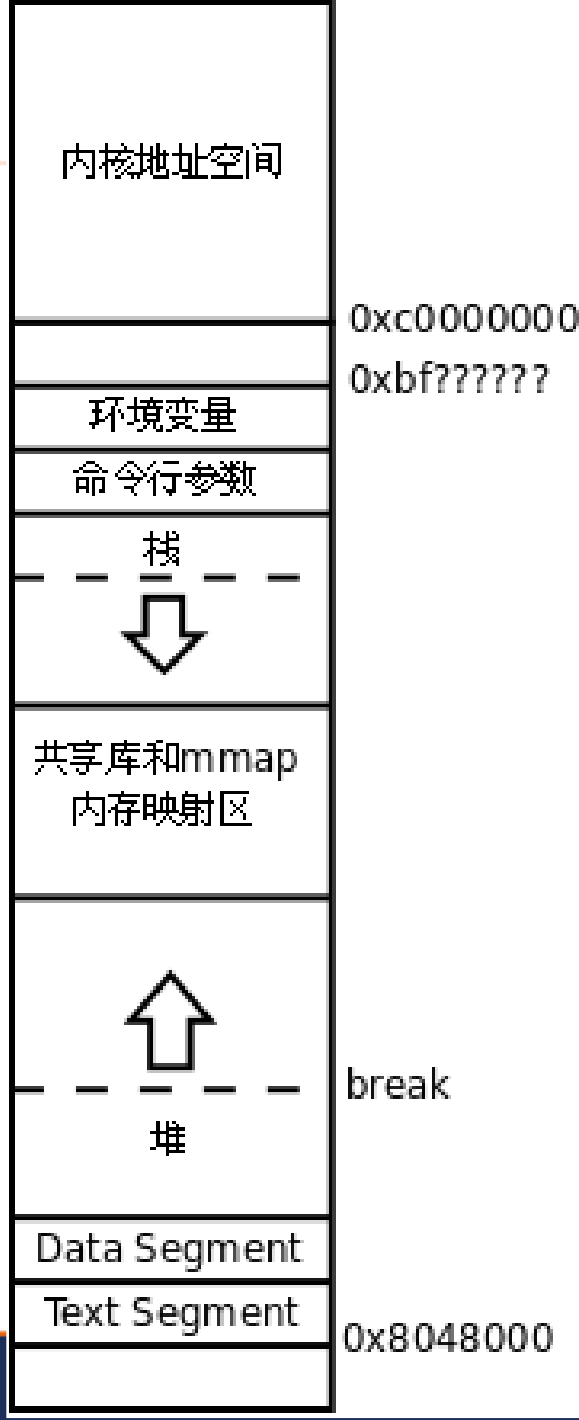
皓博
HAUBO
Haubo Training Center

# C语言入门与提高-5

张勇涛

# 进程地址分布

x86平台的虚拟地址空间是0x0000 0000~0xffff ffff，大致上前3GB（0x0000 0000~0xbfff ffff）是用户空间，后1GB（0xc000 0000~0xffff ffff）是内核空间



# 为什么使用动态内存分配

# malloc和free

```
#include <stdlib.h>
void * malloc(size_t size);
void free(void *pointer);
```

# calloc和realloc

```
#include <stdlib.h>
void *calloc( size_t num_elements,
                    size_t elements_size);
```

calloc的特殊之处: 将数组的元素都初始化为零。

```
void realloc(void *ptr,size_t new_size);
```

# 使用动态分配的内存

```
int * p;
p=malloc(100);
if(p==NULL)
{
    .....
}
```

# 奇怪的地方

- malloc(0)返回一个空指针还是指向0字节的指针?

# 常见的动态内存错误

1.忘记检查内存是否分配成功
2.访问越界
3. 释放并非动态分配的内存
4. 释放动态内存的一部分
5. 释放动态内存后继续使用

# 内存泄漏

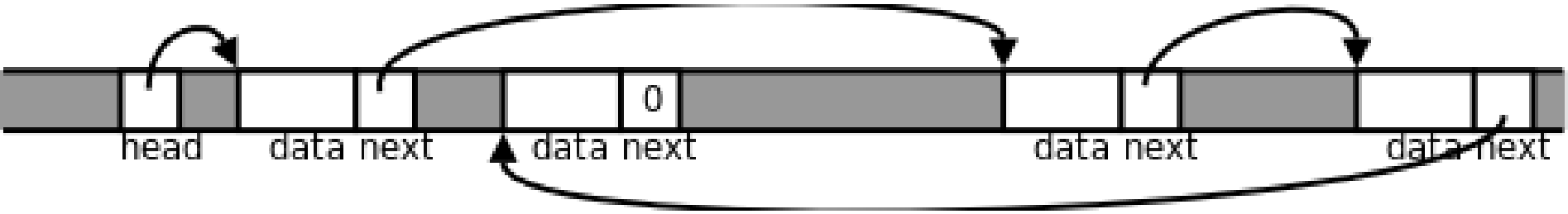
- 内存动态分配后,当它未不再被使用时未被释放.

# 内存分配的方式

- 从静态存储区分配
- 从栈上分配
- 从堆上分配

# 链表

# 链表在物理内存上的存储



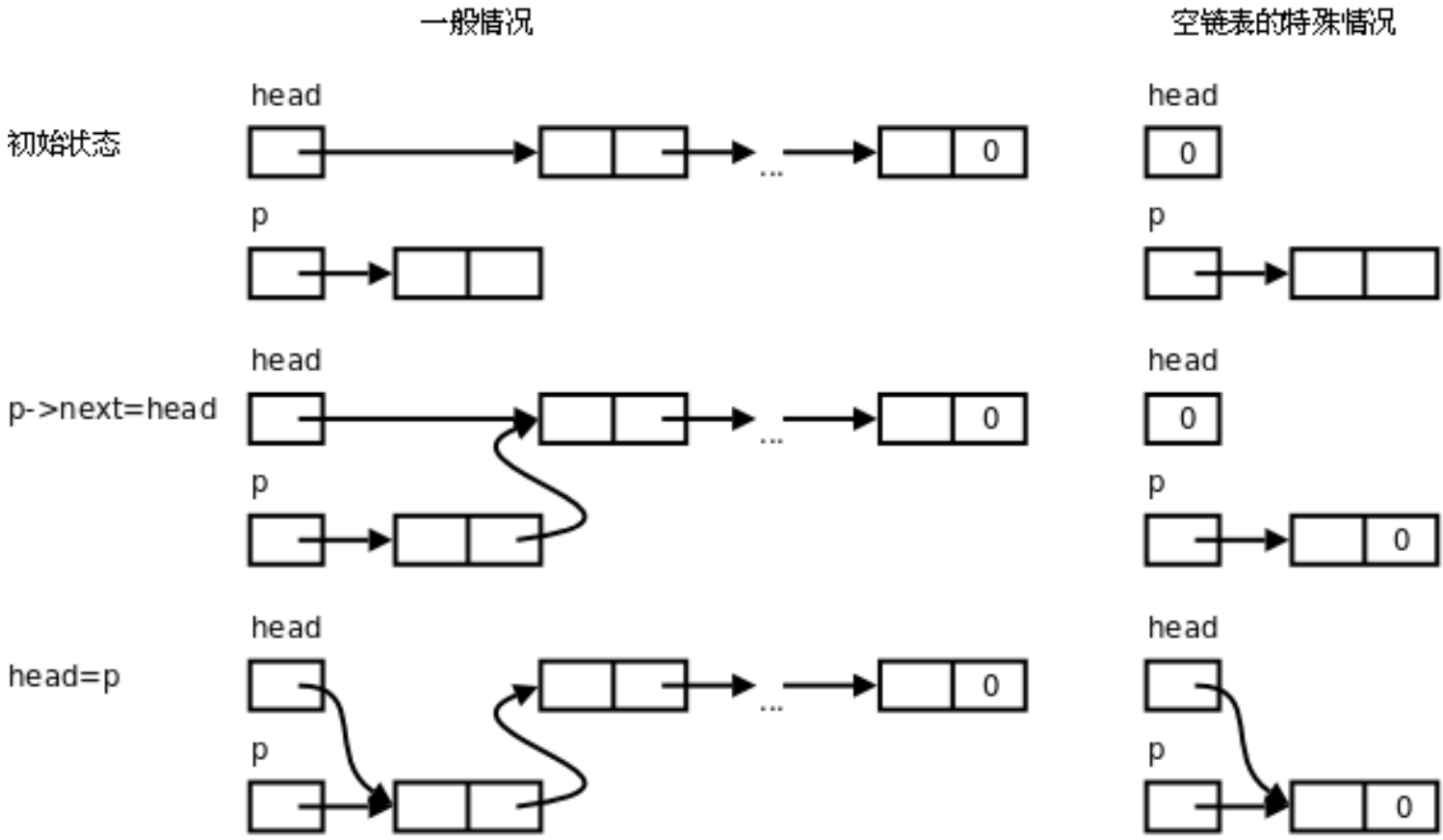

- head指针是链表的头指针，指向第一个节点，每个节点的next指针域指向下一个节点，最后一个节点的next指针域为NULL，在图中用0表示。

# 链表的特点

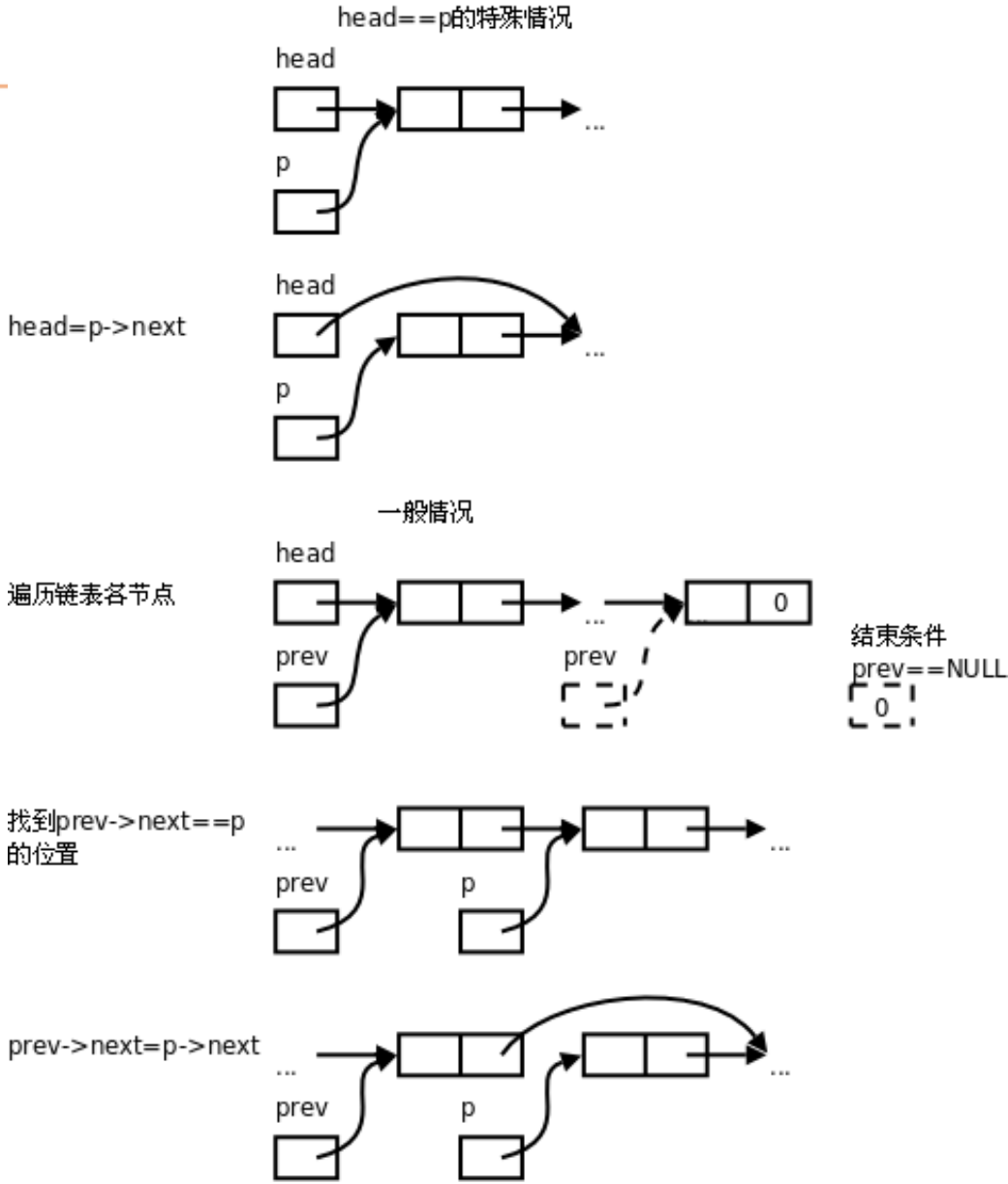
- 每个链表有一个头指针，通过头指针可以找到第一个节点，每个节点都可以通过指针域找到它的后继，最后一个节点的指针域为NULL，表示没有后继。
- 数组在内存中是连续存放的，而链表在内存中的布局是不规则的，我们知道访问某个数组元素b[n]时可以通过基地址+n×每个元素的字节数得到它地址，或者说数组支持随机访问，而链表是不支持随机访问的，只能通过前一个元素的指针域得知后一个元素的地址，因此只能从头指针开始顺序访问各节点。

# 链表的插入操作

```
void insert(link p)
{
        p->next = head;
        head = p;
}
```

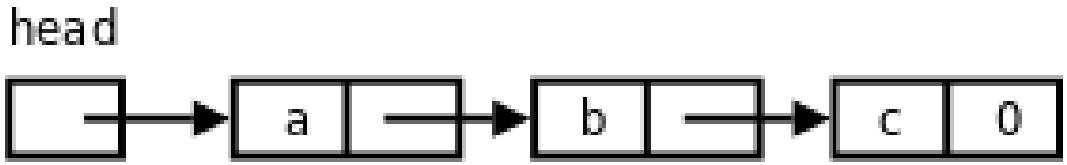


# 链表的删除操作

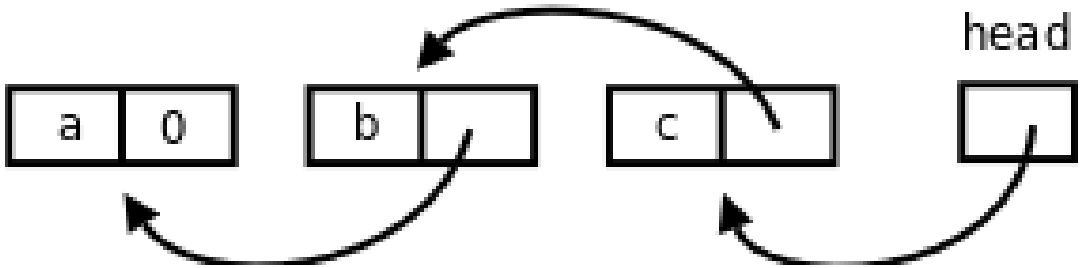

head==p的特殊情况
head
p
head=p->next
head
p
一般情况
遍历链表各节点
head
prev
prev
0
结束条件
prev==NULL
0
找到prev->next==p
的位置
prev
p
prev->next=p->next
prev
p

# 单链表的反转

初始状态

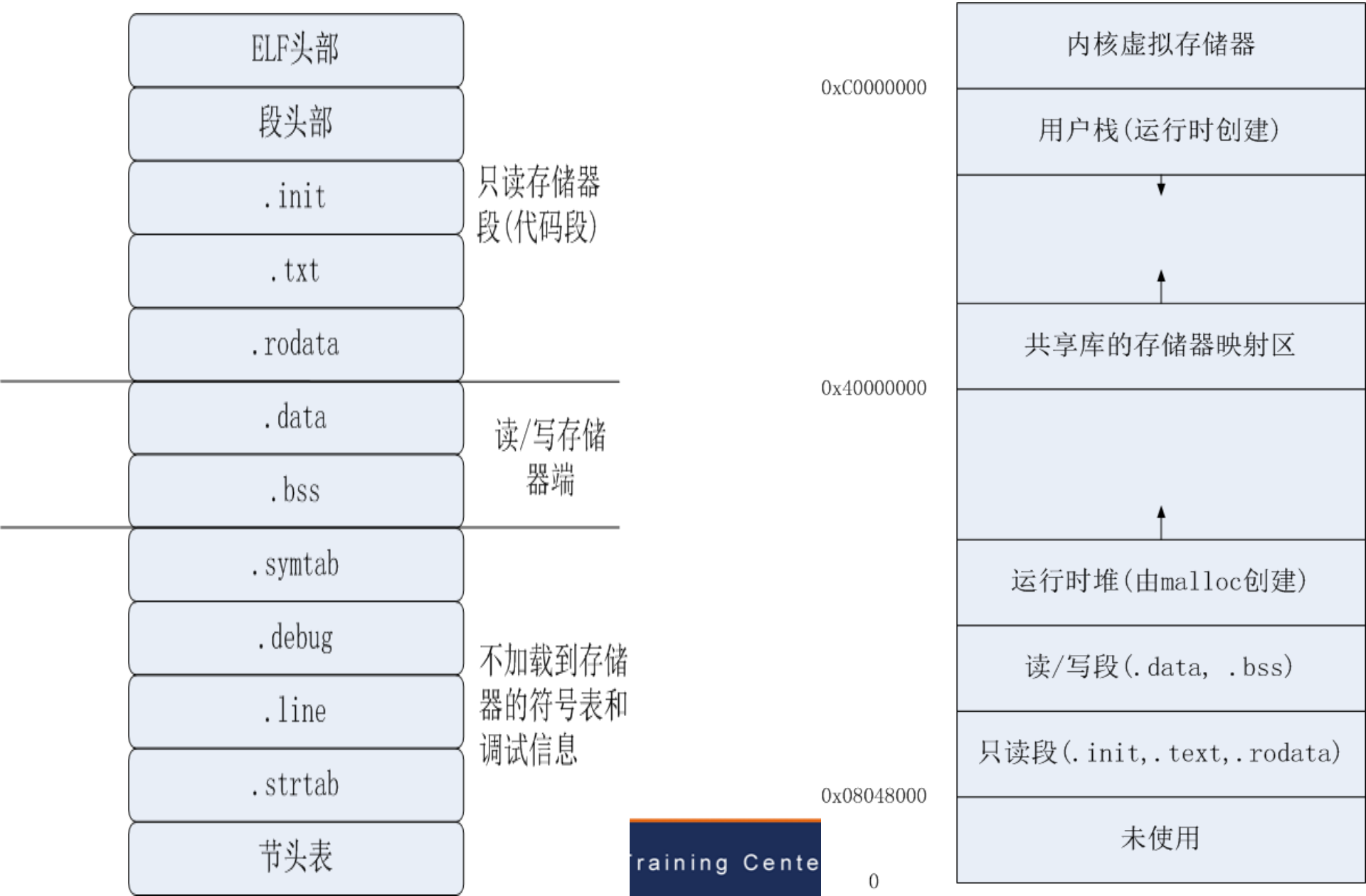



反转之后



# ELF可执行目标文件内容

# Linux运行时存储器映像



# 预处理

# 预处理的步骤

1、把三连符替换成相应的单字符。例如用??=表示#字符

2、把用\字符续行的多行代码接成一行

3、把注释（不管是单行注释还是多行注释）都替换成一个空格。

4、经过以上两步之后去掉了一些换行，有的换行在续行过程中去掉了，有的换行在多行注释之中，也随着注释一起去掉了，剩下的代码行称为逻辑代码行

5、在Token中识别出预处理指示，做相应的预处理动作

6、找出字符常量或字符串中的转义序列，用相应的字节来替换它，比如把\n替换成字节0x0a

7、把相邻的字符串连接起来

8、经过以上处理之后，把空白字符丢掉，把Token交给C编译器做语法解析，这时就不再是预处理Token，而称为C Token了

# 宏定义

函数式宏定义（Function-like Macro）。

```
#define MAX(a, b) ((a)>(b)?(a):(b))
 k = MAX(i&0x0f, j&0x0f)
k = ((i&0x0f)>(j&0x0f)?(i&0x0f):(j&0x0f))
```

# 函数式宏定义和函数调用有什么不同

1、函数式宏定义的参数没有类型，预处理器只负责做形式上的替换，而不做参数类型检查，所以传参时要格外小心.

2、调用真正函数的代码和调用函数式宏定义的代码编译生成的指令不同。

3、定义这种宏要格外小心，如果上面的定义写成#define MAX(a, b) (a>b?a:b)，省去内层括号，则宏展开就成了k = (i&0x0f>j&0x0f?i&0x0f:j&0x0f)，运算的优先级就错了.

4、调用函数时先求实参表达式的值再传给形参，如果实参表达式有Side Effect，那么这些Side Effect只发生一次。例如MAX(++a, ++b)，如果MAX是个真正的函数，a和b只增加一次。但如果MAX是上面那样的宏定义，则要展开成k = ((++a)>(++b)?(++a):(++b))，a和b就不一定是增加一次还是两次了。

5、即使实参没有Side Effect，使用函数式宏定义也往往会导致较低的代码执行效率

# 函数式宏定义的优点

- 尽管函数式宏定义和真正的函数相比有很多缺点，但只要小心使用还是会显著提高代码的执行效率，毕竟省去了分配和释放栈帧、传参、传返回值等一系列工作，因此那些简短并且被频繁调用的函数经常用函数式宏定义来代替实现。

# 内联函数(inline)

- inline关键字告诉编译器，这个函数的调用要尽可能快，可以当普通的函数调用实现，也可以用宏展开的办法实现.

# #运算符

```
#define PSQR(X) printf( “the square of X is %d.\n” ,((X)*(X))
#define PSQR(X) printf( “the square of” #X ” is %d.\n” ,((X)*(X))
```

# #include

# 条件预处理指示

```
#ifndef HEADER_FILENAME
#define HEADER_FILENAME
/* body of header */
#endif


#if MACHINE == 68000
    int x;
#elif MACHINE == 8086
    long x;
#else    /* all others */
    #error UNKNOWN TARGET MACHINE
#endif
```

#undef
#if
#elif
#endif

# 预定义宏

__FILE__
__LINE__
__DATE__
__TIME__

C99新特性
__func__

# 断言

- assert宏的原型定义在<assert.h>中，其作用是如果它的条件返回错误，则终止程序执行，原型定义：
  #include <assert.h>
  void assert( int expression );
- assert的作用是现计算表达式 expression ，如果其值为假（即为0），那么它先向stderr打印一条出错信息，
  然后通过调用 abort 来终止程序运行。

再见！