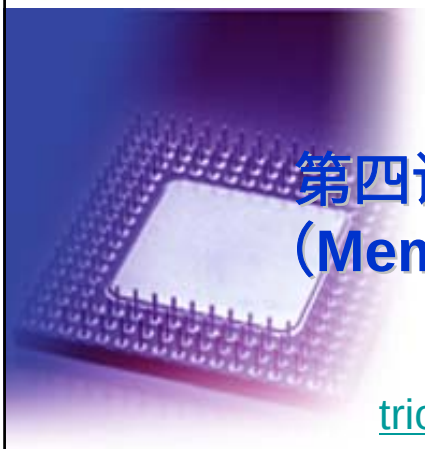
CNASIC



第四讲 存储器与指针（Memory & Pointer）

凌 明

trio@seu.edu.cn

东南大学国家专用集成电路系统工程技术研究中心

CNASIC

目 录

CNASIC

2.1 指针的基本概念

2.1.1 指针是什么？

指针是一个变量，它的值是另外一个变量的地址。

例1

上面例中的两个0x00C7有什么区别？

2.1.2 指针的类型

指针所存储的那个变量类型，就称为指针的类型。

例 2　有三个不同类型的指针：

```
int   I[2],  *pI  = &I[0];      右边的三个运算有何不同？    pI++;
char  C[2], *pC = &C[0];                                    pC++;
float F[2], *pF = &F[0];                                    pF++;
```


CNASIC

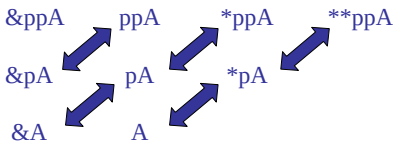
2.1.3 指针的三个要素

1. 指针指向的地址（指针的内容）；
2. 指针指向的地址上的内容；
3. 指针本身的地址。

例 3： int A, *pA, **ppA;

```
pA  = &A;
ppA = &pA;
```

在复杂的指针都可以通过下表来分析：



www.cnasic.com

2.1.4 指针的大小（指针变量占用的内存空间）

与所用的CPU寻址空间大小和类型有关，而与指针类型无关。

8位CPU的指针长度为1~2个字节（51单片机的情况较为复杂，是1~3个字节）；

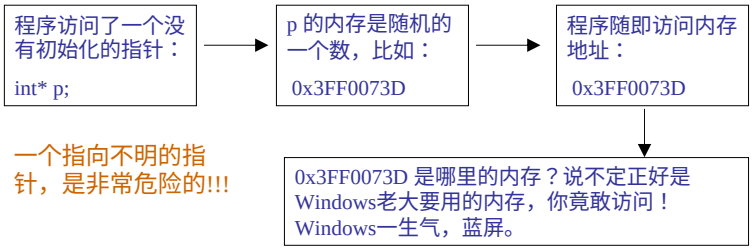
16位CPU的指针长度为2个字节（如MSP430）；

32位CPU的指针长度为4个字节（如Intel 80386）。

上面所述是通常情况，并不是全部符合。

2.1.5 指针的初始化

变量在没有赋值之前，其值不定的。对于指针变量，可以表述为：指向不明。



一个指向不明的指针，是非常危险的!!!

因此，指针在使用前一定要初始化；在使用前一定要确定指针是非空的！！！

CNASIC

2.2 数组与指针

对于数组的两个概念：

1. C语言中只有一维数组，数组的大小必须在编译时作为一个常数确定下来。数组的元素可以是任何类型，甚至是数组，由此可以方便地得到多维数组；

2. 数组的任何操作，即使采用数组下标进行的运算都等于对应的指针运算。可以用指针行为替代数组下标的运算。

例4 :

```
int a[4], *p;
p = a;          //等价于  p =&a[0];
*(a+2) = 0;     //等价于  a[2] = 0;
p[2] = 0;       //等价于  a[2] = 0;
```

但数组不同于指针：

数组名 a是指向数组起始位置的“常量”。

因此，不能对数组名进行赋值操作。

例 5：

```
int  a[4], *p;
p = a;                  //正确
a = p;                  //错误
p++;                    //正确
a++;                    //错误
```


CNASIC
第二章 指 针

2.3 空指针与通用指针

(1). 空指针

是个特殊指针值，也是唯一对任何指针类型都合法的指针值。一个指针变量具有空指针值，表示它当时没指向有意义的东西，处于闲置状态。空指针值用0 表示，这个值绝不会是任何程序对象的地址。给一个指针赋值0 就表示要它不指向任何有意义的东西。为了提高程序的可读性，标准库定义了一个与0 等价的符号常量NULL，程序里可以写：

```
p = NULL;           //注意不要与空字符NUL混淆，NUL等价于‘\0’
```

或者：

```
p = 0;
```

注意：

在编程时，应该将处于闲置的指针赋为空指针；

在调用指针前一定要判断是否为空指针，只有在非空情况下才能调用。

(2).通用指针

通用指针可以指向任何类型的变量。通用指针的类型用(void *)表示，因此也称为void 指针。

下面的第三行定义了两个通用指针：

```
int       n, *p;
double  *q;
void     *gp1, *gp2;
```

可以直接把任何变量的地址赋给通用指针。

例如，有了上面定义，下面赋值是合法的：

```
gp1 = (void *) &n;
```

可以把通用指针的值赋给普通的指针。如果被赋值指针与通用指针所指变量的类型不符，需要写强制转换：

```
p = (int *)gp1;
```


2.4 函数指针

2.4.1 函数指针的定义

函数指针即指向函数地址的指针。利用该指针可以知道函数在内存中的位置。因此也可以利用函数指针调用函数。

函数指针的定义方法：

<类型> (* <函数指针名>)(.....)

例如：

```
int (*func)(void )
```

这里，func就是一个函数指针。

注意：int *func(void)和int (*func)(void)的区别

```
int *func(void);          //这是返回一个整型指针的函数
int (*func)(void)；       //这是一个函数指针
```

CNASIC

2.4.2 函数指针的使用

例6 ：假定有下面的函数声明

```
int ptr；
int fn(int);
int (*fp)(int);
```

指出下面的语句是否合法？，为什么？。

```
fp = fn;            //正确，将函数fn的地址赋给fp
fp = fn(5);         //错误，返回给fp的结果不是一个函数地址。
fp = &ptr；         // 错误，ptr的地址不在程序代码区，两种数据类型不能转换。
```

从上面的例子可以看出：

（1） 不能将普通变量的地址赋给函数指针；
（2） 不能将函数的调用赋给函数指针
（3） 可以将函数名赋给一个函数指针

CNASIC

2.4.2 函数指针的用途

一旦函数可以通过指针被传递、被记录，这开启了许多应用，特别是下列三者：

1. 多态 (polymorphism)：指用一个名字定义不同的函数，这函数执行不同但又类似的操作，从而实现“一个接口，多种方法”。
2. 多线程 (multithreading)：将函数指针传进负责建立多线程的 API 中：例如 Win32 的 CreateThread(...pF...)。
3. 回调 (call-back)：所谓的回调机制就是：「当发生某事件时，自动呼叫某段程序代码」。事件驱动 (event-driven) 的系统经常透过函数指针来实现回调机制，例如 Win32 的 WinProc 其实就是一种回调，用来处理窗口的讯息。

CNASIC

2.4.3 函数指针数组

例7 ：在一个计算器的例子中，有如下一些语句：

```
switch(oper){
case ADD:
    result=add(op1,op2);  break;
case SUB:
    result=sub(op1,op2);  break;
... }
```

对于一个复杂的计算器，switch语句将非常长。我们可以用函数指针数组来完成。

```
double add(double,double);
double sub(double,double);
...
double (*oper_func[])(double,double)={add,sub,...};
```

第2个步骤是用下面语句替换前面整条switch语句：

```
result=oper_func[oper](op1,op2);
```

oper从数组中选择正确的函数指针，而函数调用操作符将执行这个函数。

CNASIC

ASIX Window中的函数指针

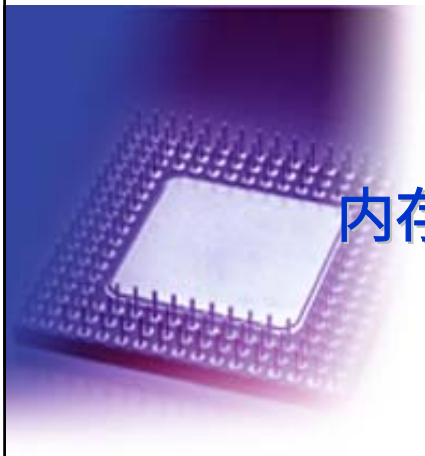
```
typedef struct window_class
{
    U8              wndclass_id;

    STATUS   (*create)(char *caption, U32 style, U16 x, U16 y, U16 width,
    U16 hight, U32 wndid, U32 menu, void **ctrl_str, void *exdata);
    STATUS   (*destroy)(void *ctrl_str);
    STATUS   (*msg_proc)( U32 win_id, U16 asix_msg, U32 lparam, void
    *data, U16 wparam, void *reserved);
    STATUS   (*msg_trans)(void *ctrl_str, U16 msg_type, U32 areald,
    P_U16 data, U32 size, PMSG trans_msg);
    STATUS   (*repaint)(void *ctrl_str, U32 lparam);
    STATUS   (*move)(void *ctrl_str, U16 x, U16 y, U16 width, U16 hight,
    void *reserved);
    STATUS   (*enable)(void *ctrl_str, U8 enable);
    STATUS   (*caption)(void *ctrl_str, char *caption, void *exdata);
    STATUS   (*information)(void *ctrl_str, struct asix_window *wndinfo);

} WNDCLASS;
```

CNASIC

ASIX Window中的函数指针

```
WNDCLASS                                WindowClass[] = {
      {WNDCLASS_WIN,    wn_create,      wn_destroy,   wn_msgproc,wn_msgtrans, wn_repaint,  NULL,NULL,wn_caption, NULL},

      {WNDCLASS_BUTTON,Btn_create,Btn_destroy, Btn_msg_proc,Btn_msg_trans,Btn_repaint,NULL,Btn_enable,Btn_caption,
      NULL},

      {WNDCLASS_SELECT,sl_create, sl_destroy, sl_msg_proc, sl_msg_trans, sl_repaint,NULL, sl_enable,  sl_caption, NULL},

      {WNDCLASS_SELECTCARD,NULL, NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL},

      {WNDCLASS_MENU,menu_create, menu_destroy, menu_msgproc, menu_msgtrans, mn_repaint, NULL, NULL,NULL,NULL},

      {WNDCLASS_LIST,   Lbox_create, Lbox_destroy, Lbox_msgproc,  Lbox_msgtrans, lb_repaint, NULL, NULL,NULL,NULL},

      {WNDCLASS_KEYBD, kbd_create,kbd_destroy, kbd_msgproc, kbd_msgtrans, kbd_repaint, NULL, NULL,NULL,NULL},

      {WNDCLASS_SCROLL,sb_create, sb_destroy,   sb_msgproc, sb_msgtrans, sb_repaint,   NULL,sb_enable, NULL,NULL},

      {WNDCLASS_KEYBAR,kb_create, kb_destroy,   kb_msgproc, kb_msgtrans,NULL,NULL,  NULL,NULL,  NULL},

      #ifdef ASIX_DEBUG
      {WNDCLASS_TEST,tst_create, tst_destroy, tst_msgproc, tst_msgtrans, NULL,NULL,NULL,NULL,NULL}
#endif

};
```


CNASIC



内存陷阱！

www.cnasic.com

CNASIC

看看这段代码有什么问题？

```
char *DoSomething(…)
{
    char i[32*1024];
    memset(i,0,32*1024);
    …
    return i;
}
```

两个重大问题：
1，临时变量是通过堆栈实现的，太大的临时变量数组会冲掉堆栈
2，返回堆栈中的地址是非常危险的，因为堆栈中的值永远是不确定的

CNASIC

看看这段代码有什么问题？

```
void DoSomething(…)
{
    int i;
    int j;
    int k;
    memset(&k,0,3*sizeof(int)
    );
```

这段代码的作用是将3个临时变量清零
但是这段代码有两个假设：
1，编译器将I,j,k三个变量通过堆栈表示
2，压栈顺序是 I, j , k (假设堆栈是满递减堆栈)
3，如果K在寄存器怎么办？对K取地址操作将产生Data Aboart

CNASIC

关于临时变量

- 不要对临时变量作取地址操作，因为你不知道编译器是否将这个变量映射到了寄存器
- 不要返回临时变量的地址，或临时指针变量,因为堆栈中的内容是不确定的（出了这个函数，存放在堆栈中的局部变量就没有意义了！）
- 不要在申请大的临时变量数组，你的临时变量是在堆栈中实现的，你有多大的堆栈呢？

CNASIC

问题？

- 现在要为一个矩形区域申请一块内存保存这块的数据，如果每个Pixle占用2个bit，如何分配内存？

```
Char *buffer;
buffer = malloc (x*y/4);
```

Buffer = malloc(x*y/4 + 1);

CNASIC

看看这段代码有什么问题？

```
char *DoSomething(…)
{
    char *p, *q;
    if ( (p = malloc(1024)) == NULL ) return NULL;
    if ( (q = malloc(2048)) == NULL ) return NULL;
    …
    return p;
{
```

如果q没有申请到，首先应该释放p，然后再返回NULL！

CNASIC

看看这段代码有什么问题？

```
void FreeWindowsTree(windows *Root)
{
    if(Root != NULL)
    {
        window *pwnd;
        /* 释放pwndRoot 的子窗口.. */
        for(pwnd = Root->Child;pwnd != NULL;pwnd = pwnd->Sibling)
            FreeWindowTree(pwnd);
        if(Root->strWndTitle != NULL)
            FreeMemory(Root->strWndTitle);
        FreeMemory(Root);
    }
}
```

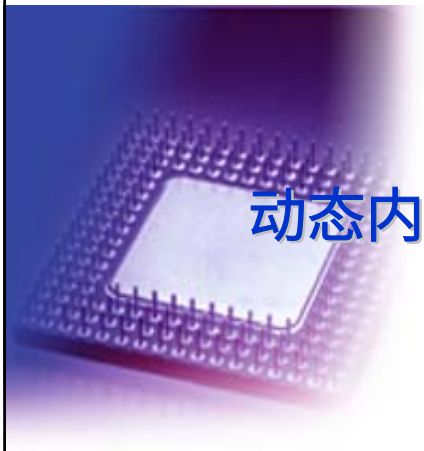
Pwnd已经被释放了，但是在for循环中被再次引用

CNASIC

关于动态内存

- 总是检查动态内存分配是否成功后再引用该指针！
- 在分配struct空间是总是使用sizeof
- 分配内存时宁滥勿缺（别忘了加一）
- 总是Free由malloc()函数返回的指针
  - 按照ANSI C 标准Free函数是没有返回值的
- 错误处理时不要忘了其他已分配空间的释放

CNASIC



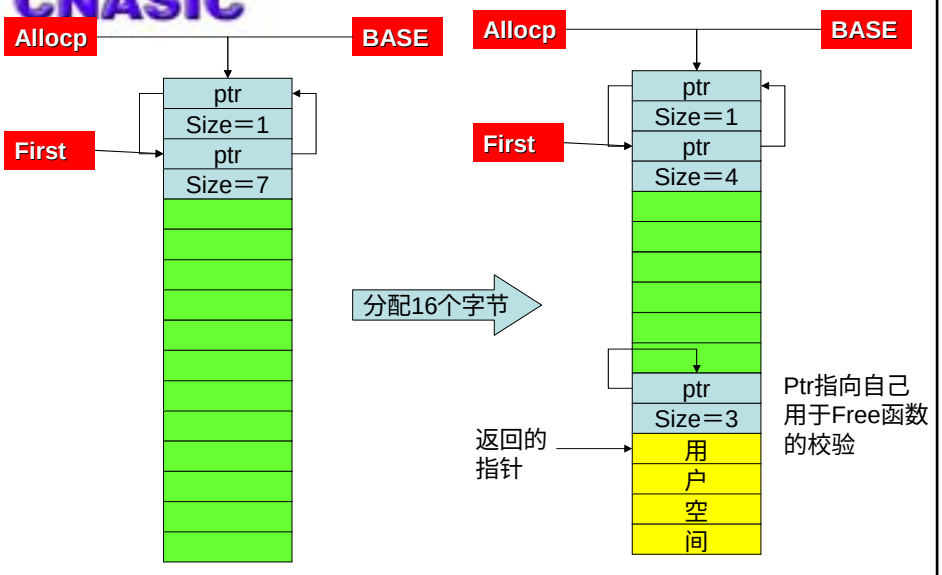
动态内存分配算法

CNASIC

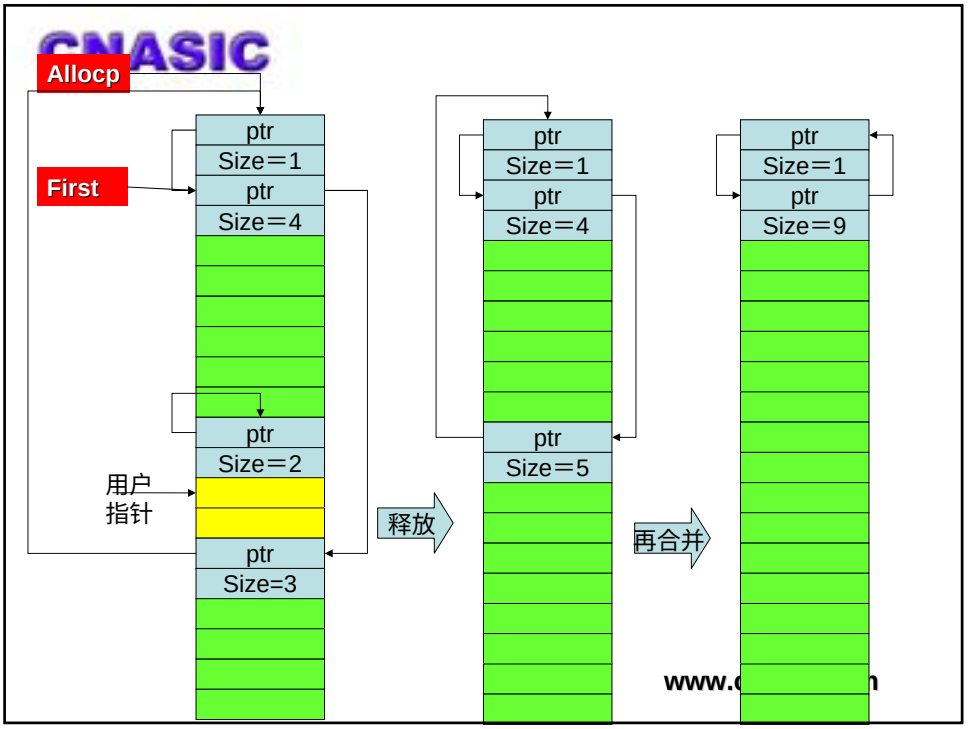
问题的提出

- 按照调用者的要求分配合适大小的Mem，返回该内存块的首指针。
- 如果没有足够的内存返回空指针。
- 用户不再使用该内存时可以调用Free函数释放该内存块
- 快速分配算法并尽量减少内存碎片的情况

CNASIC



为了方便管理，我们将内存按照8个字节进行组织—分配块
每个分配块包含两个部分：Free链表指针和本块的大小

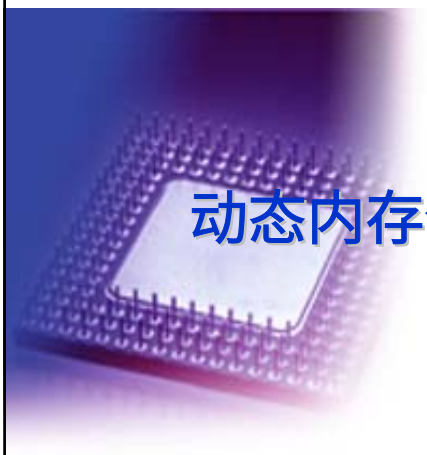


CNASIC

分配和释放策略

- 通过分配块为单位简化管理，并使得内存块更规整，便于以后的合并操作
- 总是在空闲块的高端地址分配内存，减少内存碎片
- 通过ptr指针实现空闲块链表，对于已分配内存块使用该指针作为校验
- 释放内存块的时候进行空闲块合并操作

CNASIC



动态内存分配代码讲解

CNASIC

头部的定义（分配块）

```
union header {
   struct {
        union header  *ptr;
        unsigned long size;
   } s;
   char c[8];// For debugging; also ensure size is 8 bytes
};
typedef union header HEADER;
```

CNASIC

为了方便大家理解的头部

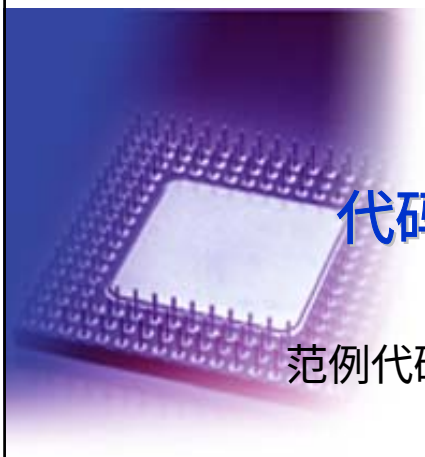
```
struct sheader {
   struct sheader      *ptr;
   unsigned long       size;
};
```


CNASIC

两个宏定义

```
#define    ABLKSIZE       (sizeof (HEADER))
/*将用户申请字节数转换成为分配块＋1*/
#define    BTOU(nb)       ((((nb) +
   ABLKSIZE - 1) / ABLKSIZE) + 1)
```

CNASIC



代码的分析

范例代码已在VC＋＋编译通过，
Just Try It！