先在ubuntu上调用malloc()看看结果

在ubuntu上调用malloc()过程如下图所示：



可以看到系统调用malloc()在程序刚开始的堆内存末尾地址为0x9dd000，在为i分配1个字节的地址空间后可以看到，堆内存末尾地址**从0x9dd000变为0x9fe000，**即内存扩充了0x9fe000-0x9dd000 = 1040384 – 905216 = 135168bits = 16896bytes = 16.5KB。

另外，可以看到指针i指向的地址为0x9dd420，即在malloc开辟的16.5KB这块大内存**中间的某个位置**，进行了内存地址分配，1个int类型数据为4bytes=32bits，换成16进制就是0x20，因此j指向的地址为i指向的地址+32 = i指向的地址+0x20 = 0x9dd440。

brk / sbrk

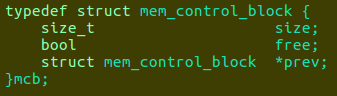
两个系统调用，用于在堆空间上分配新的内存空间，堆空间是一个线性地址，在分配内存时地址向上增长。sbrk()用法是传入int数值来分配内存，如sbrk(3);是分配3字节的堆内存，sbrk(21);是分配21字节的堆内存；brk()的用法是传入地址来分配内存，但是获取地址又要绕一层，因此通常用sbrk()函数。

注：sbrk(0)不是分配0个字节，而是得到堆内存当前的末尾位置。

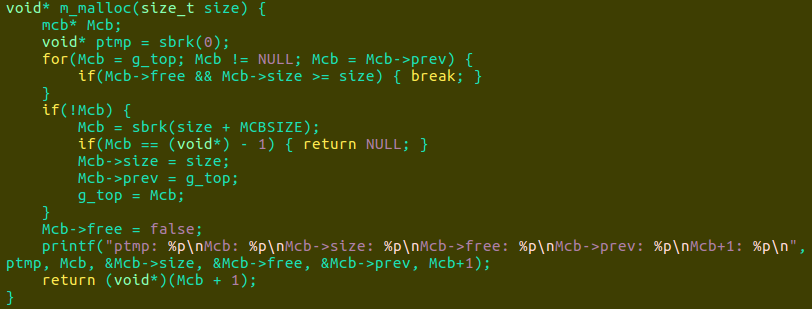
一个简单的m\_malloc / m\_free

函数原型：void\* m\_malloc(size\_t size);

控制堆内存的基本单位struct mem\_control\_block{…}如下图所示，可知堆内存是由一个个结构体连起来的链表组成的，每个控制块结构体里有该控制块的基本信息，它们都占据堆内存的空间。



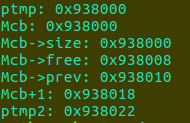
m\_malloc(…)实现如下图所示：



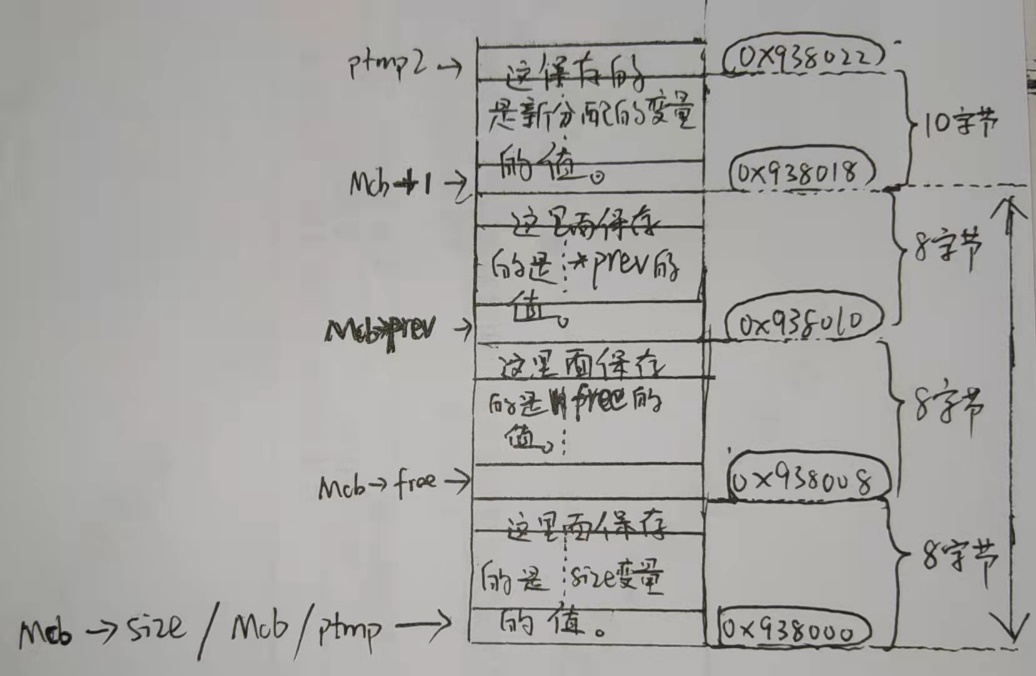
步骤0：先在现有的块里面搜索是否存在满足要求的块（块是free的并且块容量大于当前需要分配的内存的容量size，此时size==10），若找到了则break退出；

步骤1：若没有搜索到合适的块，则调用sbrk(…)系统调用开辟一块新的堆内存出来，大小为size + 结构体的大小MCBSIZE，如果Mcb到达堆内存最后一个位置，则返回NULL，否则向该Mcb输入相关信息（块的大小size，块的指针），然后将该块的free设置为false；

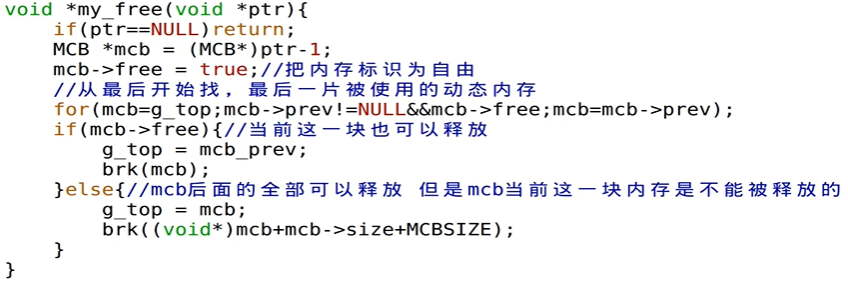
这里打印以下地址信息，就可知道这块区域的内存布局是怎样的：



1. 结合m\_malloc(…)代码，ptmp是堆内存末尾，由于该Mcb块是该程序堆内存空间第一个分配的块，因此Mcb地址 == ptmp地址；
2. 由于结构体的数据是size\_t size、bool free、mem\_control\_block \*prev，因此size变量被放在堆内存的第一个位置，因此size的起始地址为0x938000；
3. free变量放在堆内存的第二个位置，先通过指令uname -m得到返回值x86\_64，可知系统为64位，因此size变量大小为8Bytes，那么free的起始地址就是size的起始地址+size的大小，即free的起始地址为0x938000 + 8 = 0x938008；
4. prev同理，prev起始地址为0x938008 + 8 = 0x938010（注意都是16进制）
5. Mcb+1的地址等于prev+8（因为Mcb是mcb\*即指针类型的，因此Mcb+1就是内存地址往后跳8个字节），即Mcb+1就是跳过了结构体大小后的size的起始地址，可以看到Mcb大小等于0x938018 – 0x938000 = 24字节，其中size和\*prev都是8字节，由于内存对齐，bool类型的free也占用了8个字节。
6. 最后再调用一次sbrk(0)返回给ptmp2，得到ptmp2地址为0x938022，即0x938018 + size = 0x938022（注意16进制）。



函数原型：void\* m\_free(void\* ptr) {…}



由于malloc返回的指针地址为mcb+1，因此再函数里需要得到正确的堆内存地址，即mcb – 1，由于参数是指针类型，需要先转换成mcb类型再去-1，然后把该块内存的free设置为true，表示该块可释放。

for循环是从最顶端的块开始遍历，去搜索有多少free == true的块，直到条件不满足而退出，if、else是为了判断当前mcb指向的块是否是free的，如果是free就带上一块释放，不是free就不释放这个，**释放的方式就是通过brk调整mcb指向的位置，以及更新g\_top指向的位置**。

# malloclab实验简介

malloclab是CSAPP的第六个实验，网上说是难度最大的实验，我觉得只要能把书上那些#define宏定义搞明白基本就差不多理解了，当然下面的笔记也已经做了这些宏定义的解释，另外这个实验其实只是模拟了一个内存分配场景来让我学习malloc的原理，看一下mem\_sbrk(…)就知道它已经通过malloc()系统调用在堆内存上开辟了一大块内存，然后我们在这个基础上来实现数据结构。

书中介绍了4种数据结构：隐式空闲链表/显式空闲链表/分离链表/分离适配，我在实验中主要实现第1，2，4个，因为分离链表属于4的一部分就略了。 本质上3种结构的原理都是操作链表，隐式链表只操作1条链表（物理的），显式链表操作2条链表（1条物理的1条逻辑的），分离链表/分离适配操作n条链表（1条物理的n-1条逻辑的）。

**当前结果**

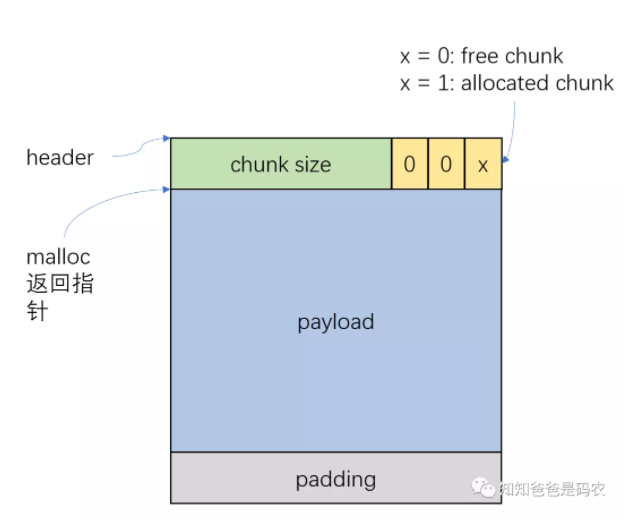
当前隐式链表代码已通过测试，显式链表代码已通过测试，分离适配代码已经bug free但仍然存在错误，导致测试结果有coredump，测试不通过。

# 隐式空闲链表

参考：<https://zhuanlan.zhihu.com/p/126341872>

<https://zhuanlan.zhihu.com/p/386793474>

## chunk



隐式空闲链表的基本单元是一个个的chunk，1个chunk包含以下几个元素：

1. header(1) chunk size；
2. header(2) A/F flag；
3. payload；
4. padding area；

## header（chunk size和a/f flag）

header大小为4字节共32bit，其中chunk size里保存的是块的大小，由于要8字节对齐，因此低三位是没用的，（因为低三位就是1/2/4/0这4个数字各种加减，除了0以外，怎么加加减减得到的结果都没法被8整除，因此低三位统统为0），用低三位编码其他信息，这里只用了第0位来表示“已分配/空闲”，其他两个bit位没有用，**空闲**的话就是0b000 == 0x0，**已分配**的话就是0b001 == 0x1。

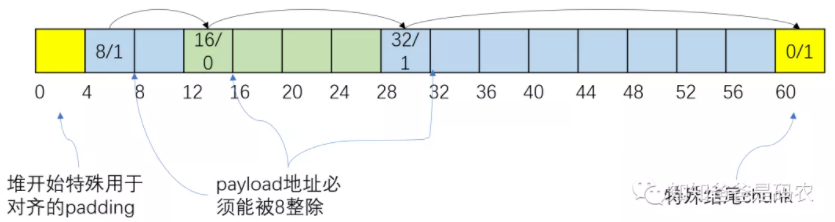
header的计算方式举例如下：

128字节的已分配块，它的header为0x00000080 | 0x1 == 0x00000081；

96字节的空闲块，它的header为0x00000060 | 0x0 == 0x00000060；

## 隐式空闲链表

把chunk数据结构组织成一个连续链表，遍历过程中可以通过a/f flag标志位判断得到块是否是空闲的，因此叫它隐式空闲链表，假定堆开始地址为0，方格大小是4bytes，灰色表示已分配，蓝色表示空闲，如下图所示：

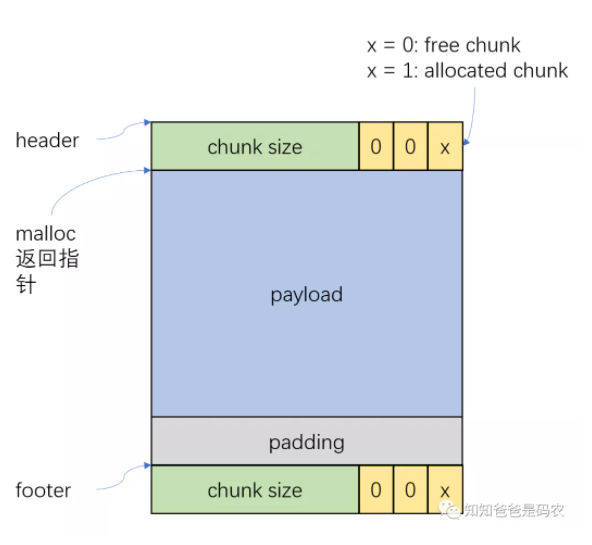


注意点1：由于8字节对齐，因此chunk最小的大小size为8bytes；

注意点2：为让payload地址能被8整除，在堆内存开始位置放了4bytes的padding，即黄色块。

注意点3：为判断链表结束，最后1个chunk size强制为0，af flag为1。

## footer



在遍历链表时，只能按序访问下一个块地址（块地址是连续的，知道当前块地址和当前块大小，可计算下一个块的地址），但不能访问上一个块地址（**知道当前块的地址和大小，但不知道上个块的块大小，因此无法计算上一个块的地址**），因此是链表是单向的，在查找时效率较低，例如在调用free场景中，需要判断当前块的前一个块的a/f flag标志位是不是也是free的（如果是就一块合并掉），现在只能从头再次遍历直到当前块的前一个块，那么在这里就加一个footer，相当于加了一个向前访问的指针，效率这不就更高了。

**隐式空闲链表宏定义解释**

**注：**这里参考着“footer”的图看就很清晰明白了。



解释：打包size值和alloc值，这俩的结构为（aaaaaaaaaaaaaaaaaaaaaaaaaaaaabbb），a是size的bit位，b是alloc的bit位。



解释：获得chunk size数据。由于chunk size数据放在了header的前29位上，又“0x7”意思是最后3位bit位 == 7 == 4 + 2 + 1 = 0b111，“~”是求补码，因此~0x7 == 11111111111111111111111111111000，1总共有29个，最后3个为0。



解释：类似GET\_SIZE宏，这里”0x1” == 0b001，看这个比特位是空闲（1）还是已分配（0）。



解释：HDRP意思是header pointer，bp是block pointer（块指针），它指向第一个有效载荷字节（参考“加入footer”小节那张图里的“malloc返回指针”就是bp），bp指向的地址-WSIZE（4字节）就是header的地址。



解释：FTRP意思是footer pointer，GET\_SIZE(HDRP(bp))是该块的大小，bp加上该值就超了（因为bp指向payload首地址的原因，因此还要要减去1个DSIZE（2个WSIZE）才是本块的footer地址。



解释：指向下个块的指针，bp + 本块的大小size就是下一个块的bp应该指向的地址（即下一个块payload的首地址）。



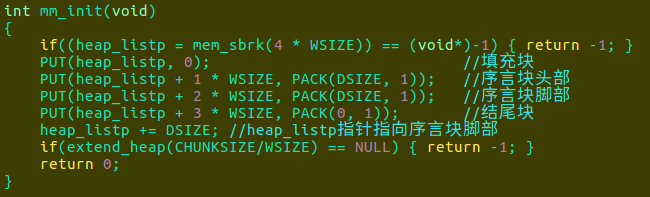
解释：GET\_SIZE里面的bp – DSIZE是上一个块的footer的地址，因此GET\_SIZE得到的是上一块的大小size，bp – 上一块的大小就是上一个块的bp应该指向的地址（即上一块的payload的首地址）。

注：bp指针指向的是**payload的首地址**。

## 代码示例

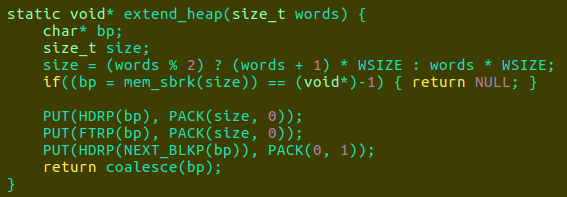
位置：mm.c

### 初始化



参考上文的隐式链表结构可知该段代码意思。

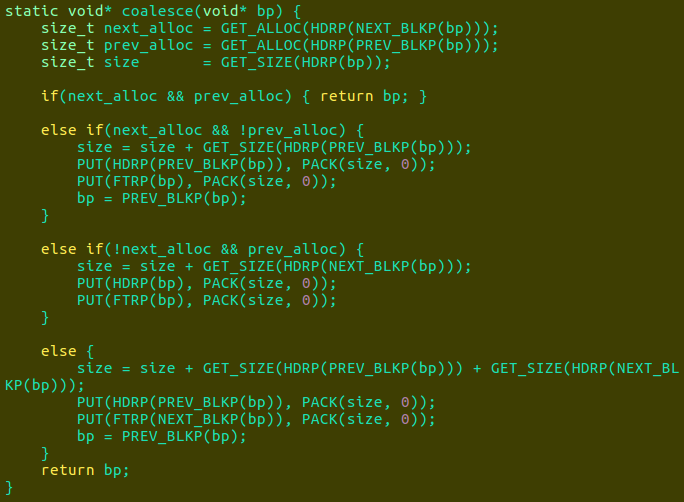
### 扩充堆空间



形参为words，扩充的时候需要 \* 4才是字节，WSIZE就是4，考虑到8字节对齐，words大小不能为奇数，必须被2整除，得到size后调用系统调用mem\_sbrk(…)（其实mem\_sbrk不是系统调用，在memlib.c里可以看到，该实验只是模拟了这个事），给该块的header和footer输出size和flag，并在该块下一个4字节位置定为结尾快，因此需要输入PACK(0, 1)，即大小为0，且已分配。

最后，调用合并函数coalesce(…)，看该新分配块前面是否也是空闲的，若是空闲的就合并掉。

### 合并函数



用来合并空闲chunk的，原理很简单，4次判断：

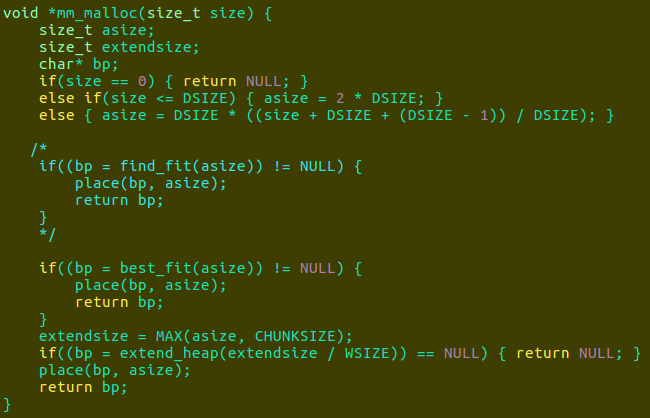
（1）该块前后两个块都不空闲，则什么都不干；

（2）后块不空闲前块空闲，则合并前块；

（3）后块空闲前块不空闲，则合并后块；

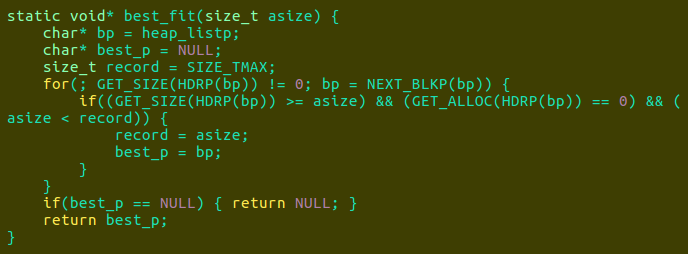
（4）后块前块全都空闲，则前后都合并；

### mm\_malloc



asize用来确定最终得分配多少字节，考虑到内存对齐，最小不小于16字节，且必须被8整除，接下来调用搜索函数find\_fit或best\_fit（这里可能写错了应该是first\_fit），若返回NULL说明没有合适的空闲块了，因此调用extend\_heap扩充内存空间，然后通过place函数把申请块放进新扩充的块里去。

### best\_fit

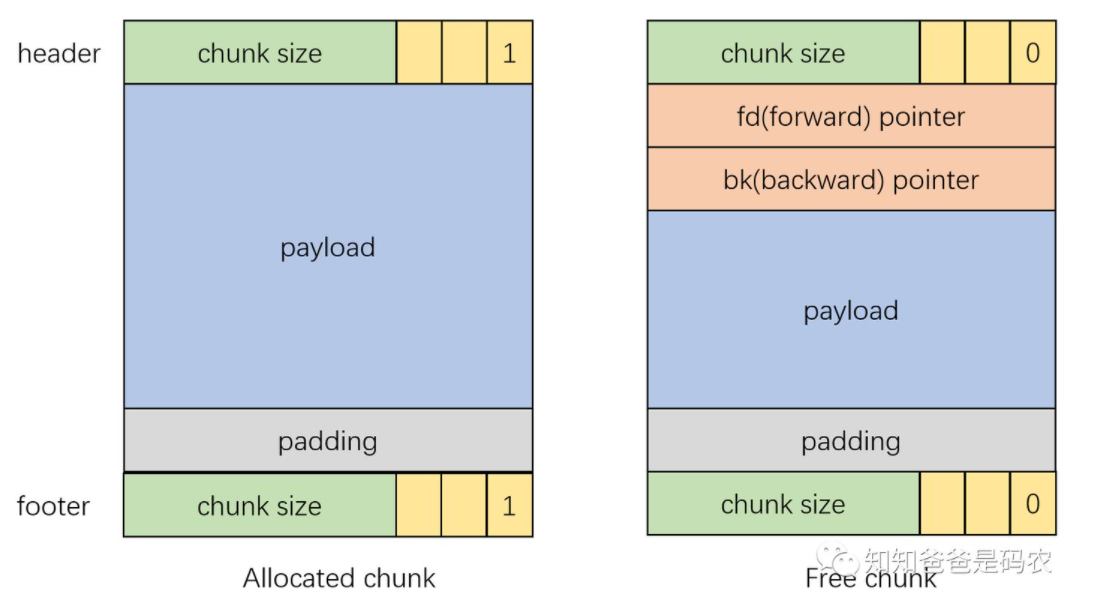


遍历所有块得到块大小和asize最接近的那个块，return它。

# 显式空闲链表

参考：https://zhuanlan.zhihu.com/p/378352199

## chunk



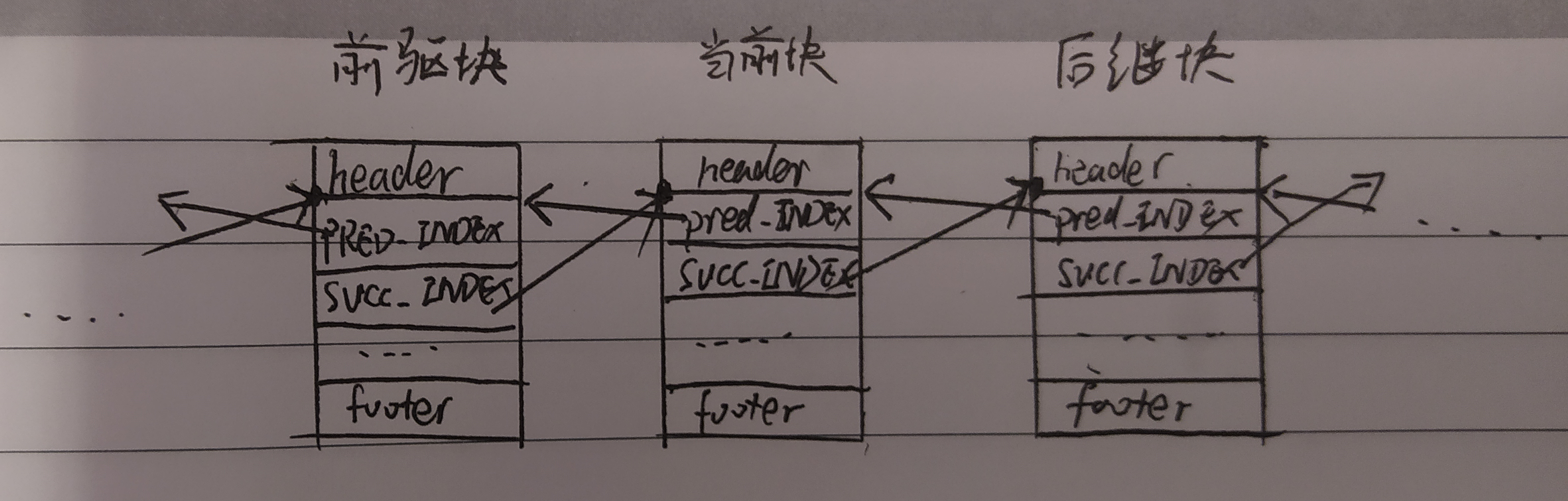
空闲块里多了俩指针，每个指针大小4Bytes，里面保存的是前驱块和后继块的**地址**（其实就是前驱块/后继块的PRED\_INDEX/SUCC\_INDEX的地址），其他跟隐式空闲链表一样。

## 前驱/后继指针

相比隐式空闲链表，显式空闲链表的空闲块多了两个指针，在上图中写的是fd(forward) pointer和bk(backward) pointer，在实验中写的是PRED\_INDEX和SUCC\_INDEX。

PRED\_INDEX指向逻辑上的前驱chunk，SUCC\_INDEX指向逻辑上的后继chunk（他们都是free的，因为只有free chunk才有这俩指针，才能挂到空闲链表上），具体来说，指向了bp位置（这和隐式链表一样），只是在隐式链表中bp指向的是payload的首地址，然而在显式链表中这个位置是PRED\_INDEX。

因此，对于当前块的PRED\_INDEX，它里面保存的数据是（逻辑上）**前一个块的PRED\_INDEX位置的地址**，对于当前块的SUCC\_INDEX，它里面保存的数据是（逻辑上）**下一个块的PRED\_INDEX位置的地址**，如下图所示：



（注意图中黑点，它意思就是PRED\_INDEX的首地址）

## 代码示例

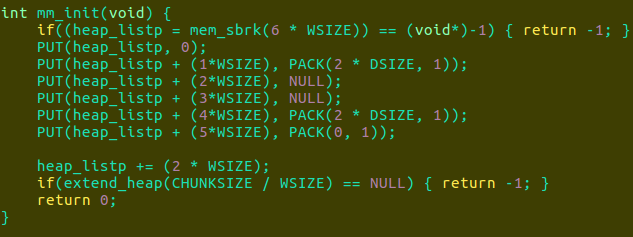
位置：explicit\_mm.c

### 宏定义



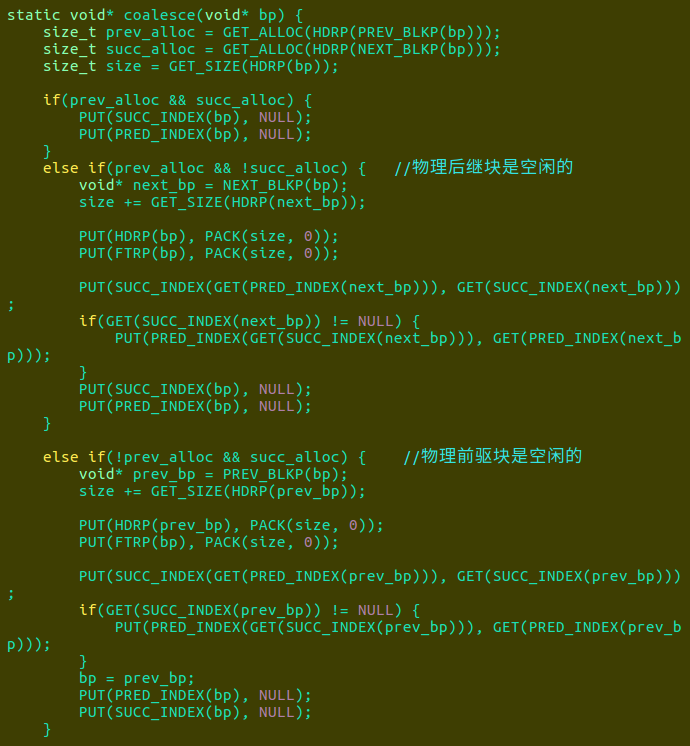
由于显式链表新加入了前驱和后继指针，因此需要在宏上定义这俩指针的位置。

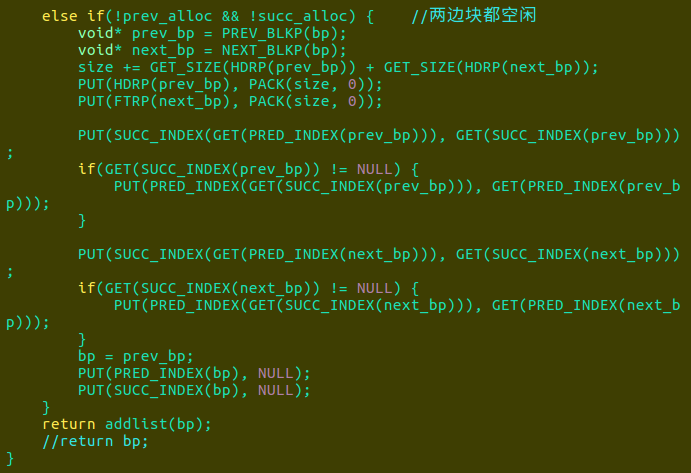
### 初始化



初始化步骤和隐式的类似，只是序言块多了8字节存放前驱指针和后继指针，可以看到初始分配的时候指针都指向NULL。

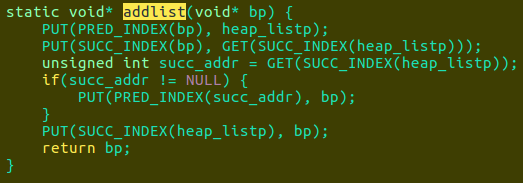
### 合并





合并部分代码原理跟隐式链表式一样的，但是操作上更复杂一些，因为除了物理块的链表，显式链表还要维护1条逻辑上的链表。因此在物理前驱或物理后继块空闲可以合并的时候，需要判断该物理前驱块或物理后继块的逻辑前驱和逻辑后继是否存在（或者说它的前驱后继指针是否指着其他块），若存在，则需要断掉逻辑上与前驱和后继的指针，把该块的指针连接到当前合并块上来，连接不了的就置为NULL。最后，调用addlist把该新块加入逻辑上的空闲链表上去。

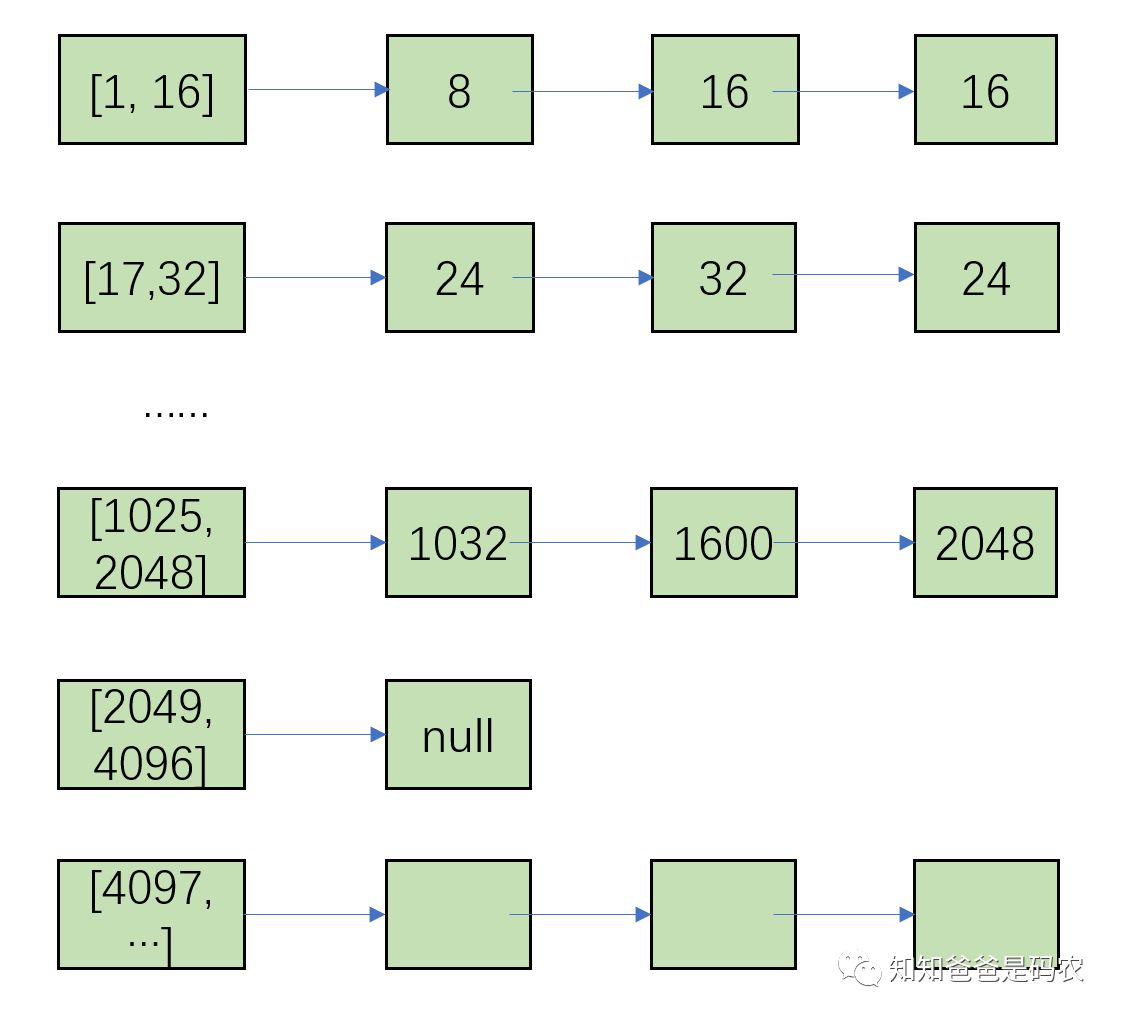
### addlist



把该块连接到heap\_listp的下一个节点上去，heap\_listp作为逻辑链表的头节点，后继指针连接的都是空闲块，每次要添加新空闲块到空闲链表上时候都调用该函数，添加的位置也都是heap\_listp指向的下一个位置。

# 分离存储

该算法把所有的free chunk用多个单向链表来维护，每个链表维护的free chunk大小都是相同范围内的，如图所示：



分配一个chunk，要先确定请求大小属于哪个范围，在哪一个空闲链表上，定位链表后在该链表上使用查找算法（first\_fit、best\_fit等）来确定分配到哪个空闲chunk上，存在2种情况：

1. 对应list == NULL，说明这个范围的链表之前没有申请chunk的请求没有挂载任何空闲块，因此调用extend\_heap(…)在该链表上申请堆内存，考虑到8字节对齐因此可能产生多余的空闲块。例如，申请198字节空间，且该属于128-256链表上，但按照内存对齐需要分配200字节（因为可被8整除），多余的2字节就挂到1-16链表上；
2. 对应list != NULL，存在两种情况：

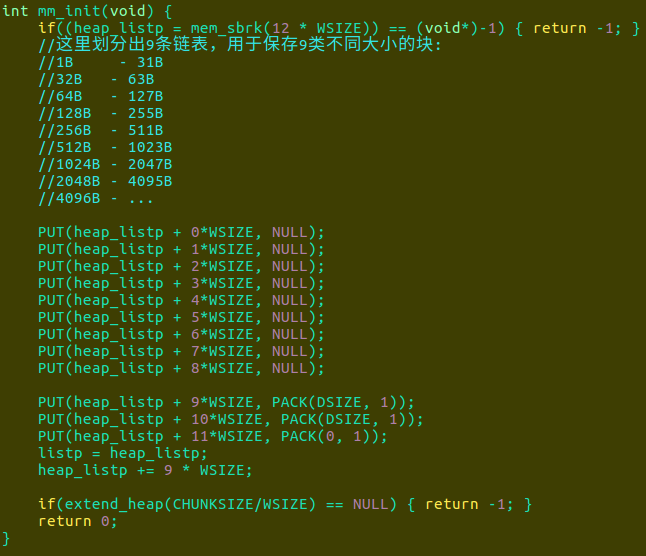
2.1该链表上没有足够大小的空闲chunk，那么参照情况1解决方法；

2.2 该链表上有空闲块且大小 > 需要的大小，例如，该链表上有1个空闲块大小为54字节，然而我只需要49字节，多出来5字节就分配到1-16链表上去。

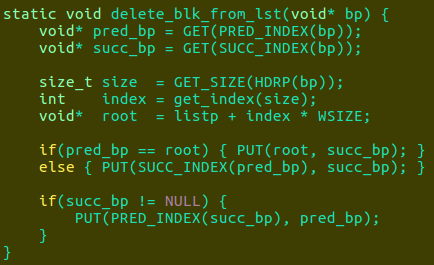
释放一个chunk，先判断该chunk的物理前驱后继，看前驱和后继是否也是空闲，若空闲，则把该前驱或后继chunk所在链表上的（逻辑的）的前驱后继链删除掉，然后把该前驱或后继chunk和该chunk合并，求出大小后，加入到对应大小的链表上去。

## 代码示例

### 初始化

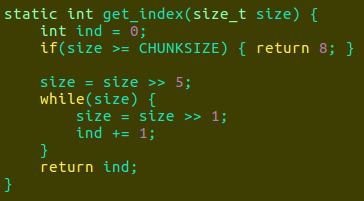


### 把块从空闲链表上删除



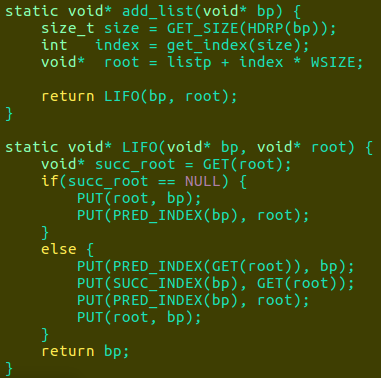
由于需要频繁把空闲链表上的块从块上摘下来删除，因此把这块操作单独写成函数。

### 索引函数



形参是该块的大小，计算该块大小对应在哪个链表块上，得到索引值。

### 加入空闲链表



原理跟显式链表一样的，不做赘述。