<https://blog.csdn.net/vevenlcf/article/details/47036127>

realloc   
       原型：extern void \*realloc(void \*mem\_address, unsigned int newsize);   
       用法：#include <stdlib.h> 有些编译器需要#include <alloc.h>   
       功能：改变mem\_address所指内存区域的大小为newsize长度。   
       说明：如果重新分配成功则返回指向被分配内存的指针，否则返回空指针NULL。   
                 当内存不再使用时，应使用free()函数将内存块释放。 

       注意：这里原始内存中的数据还是保持不变的。

下面 通过 代码来认识一下：

**[cpp]** [view plain](https://blog.csdn.net/vevenlcf/article/details/47036127) [copy](https://blog.csdn.net/vevenlcf/article/details/47036127)

1. #include <stdlib.h>
2. #include<stdio.h>
4. main()
5. {
6. **char** \*p;
7. p=(**char** \*)malloc(100);
8. **if**(p)
9. printf("Memory Allocated at: %p\n",p);
10. **else**
11. printf("Not Enough Memory!/n");
12. **char** \*lcf=(**char** \*)realloc(p,256);
13. **if**(lcf)
14. printf("Memory Reallocated at: %p\n",lcf);
15. **else**
16. printf("Not Enough Memory!/n");
17. free(lcf);
18. **return** 0;
19. }

详细说明及注意要点：   
1、如果有足够空间用于扩大mem\_address指向的内存块，则分配额外内存，并返回mem\_address   
这里说的是“扩大”，我们知道，realloc是从堆上分配内存的，当扩大一块内存空间时， realloc()试图直接从堆上现存的数据后面的那些字节中获得附加的字节，如果能够满足，自然天下太平。也就是说，如果原先的内存大小后面还有足够的空闲空间用来分配，加上原来的空间大小＝ newsize。那么就ok。得到的是一块连续的内存。   
2、如果原先的内存大小后面没有足够的空闲空间用来分配，那么从堆中另外找一块newsize大小的内存。   
并把原来大小内存空间中的内容复制到newsize中。返回新的mem\_address指针。（数据被移动了）。   
老块被放回堆上。   
例如：   
#include <malloc.h>   
char \*p，\*q;   
p = (char \* ) malloc (10);   
q=p;   
p = (char \* ) realloc (p,20);   
…………………………   
这段程序也许在编译器中没有办法通过，因为编译器可能会为我们消除一些隐患！在这里我们只是增加了一个记录原来内存地址的指针q，然后记录了原来的内存地址p，如果不幸的话，数据发生了移动，那么所记录的原来的内存地址q所指向的内存空间实际上已经放回到堆上了!这样一来，我们应该终于意识到问题的所在和可怕了吧！   
3、返回情况   
返回的是一个void类型的指针，调用成功。（这就再你需要的时候进行强制类型转换）   
返回NULL，当需要扩展的大小（第二个参数）为0并且第一个参数不为NULL，此时原内存变成了“freed（游离）”的了。   
返回NULL，当没有足够的空间可供扩展的时候，此时，原内存空间的大小维持不变。   
4、特殊情况   
如果mem\_address为null，则realloc()和malloc()类似。分配一个newsize的内存块，返回一个指向该内存块的指针。  
如果newsize大小为0，那么释放mem\_address指向的内存，并返回null。   
如果没有足够可用的内存用来完成重新分配（扩大原来的内存块或者分配新的内存块），则返回null.而原来的内存块保持不变。

引用：

==============================================================

void\* malloc(unsigned size); void\* calloc(size\_t nelem, size\_t elsize); 和void\* realloc(void\* ptr, unsigned newsize);都在stdlib.h函数库内，是C语言的标准内存分配函数。  
1. 函数malloc()和calloc()都可以用来动态分配内存空间。malloc()函数有一个参数，即要分配的内存空间的大小，malloc 在分配内存时会保留一定的空间用来记录分配情况，分配的次数越多，这些记录占用的空间就越多。另外，根据 malloc 实现策略的不同，malloc 每次在分配的时候，可能分配的空间比实际要求的多些，多次分配会导致更多的这种浪费。当然，这些都和 malloc 的实现有关；calloc()函数有两个参数，分别为元素的数目和每个元素的大小，这两个参数的乘积就是要分配的内存空间的大小。如果调用成功，它们都将返回所分配内存空间的首地址。  
2. 函数malloc()和函数calloc()的主要区别是前者不能初始化所分配的内存空间，而后者能。 char\* a = (char\*)calloc(1, sizeof(int));  
3. realloc可以对给定的指针所指的空间进行扩大或者缩小，无论是扩张或是缩小，原有内存的中内容将保持不变。当然，对于缩小，则被缩小的那一部分的内容会丢失。  
4. realloc 并不保证调整后的内存空间和原来的内存空间保持同一内存地址。相反，realloc 返回的指针很可能指向一个新的地址。所以在代码中，我们必须将realloc返回的值，重新赋值给 p :  
p = (int \*) realloc (p, sizeof(int) \*15);

==================================================================

关于realloc函数说明的补充：  
函数定义：  
void \*realloc(void \*ptr, size\_t size);  
上面的分析基本没有问题，但有两点要注意：  
1、返回值可能与ptr的值不同，如果是不同的话，那么realloc函数完成后，ptr指向的旧内存已被free掉了。  
2、如果返回NULL值，则分配不成功，而原来的ptr指向的内存还没有被free掉，要求程序显式free.  
  
故p = (int \*) realloc (p, sizeof(int) \*15);语句有这么一个问题，  
调用前p指向一个已分配成功的内存,而调用realloc时却失败（即返回NULL），此时，p原来指向的内存还没有free掉，而现在又找不到地址，这样就出现memory leak了。  
  
关于这一点的确要注意，最好如下：

**[cpp]** [view plain](https://blog.csdn.net/vevenlcf/article/details/47036127) [copy](https://blog.csdn.net/vevenlcf/article/details/47036127)

1. **int** \*q
2. q = (**int** \*) realloc (p, **sizeof**(**int**) \*15);
3. **if**(!q) p =q;

最后 还是引用 大神博客文章： http://blog.chinaunix.net/uid-23629988-id-371240.html

[编写安全代码：小心使用realloc](http://blog.chinaunix.net/uid-23629988-id-371240.html)

在良好的代码风格中，其中有一项要求就是，一个函数只做一件事情。如果该函数实现了多个功能，那基本上可以说这不是一个设计良好的函数。

今天看C库中的函数realloc。其原型是void

\*realloc(void \*ptr, size\_t size);函数说明如下：

realloc() changes the size of the memory block pointed to by ptr to size bytes. The contents will be unchanged to the minimum of the old and new sizes; newly allocated memory will be uninitialized. If ptr is NULL, the call is equivalent to malloc(size); if sizeis equal to zero, the call is equivalent to free(ptr). Unless ptr is NULL, it must have been returned by an earlier call to malloc(), calloc() or realloc(). If the area pointed to was moved, a free(ptr) is done.

总结一下，有以下几种行为：

1. 与名字相符，真正的realloc，参数ptr和size均不为NULL，重新调整内存大小，并将新的内存指针返回，并保证最小的size的内容不变；

2. 参数ptr为NULL，但size不为0，那么行为就等于malloc(size)；

3. 参数size为0，则realloc的行为为free(ptr)；这时原有的指针已经被free掉，不能继续使用。而此时realloc的返回值为NULL。这意味着不检查realloc的返回值，直接使用，会导致crash。

看，一个简单C库函数，却赋予了三种行为，所以这个realloc并不是设计良好的库函数。估计也是为了兼容性，才容忍这个函数一直在C库中。虽然在编码中，realloc会提供一定的方便，但是也很容易引发bug。

下面就举两个例子，来说明一下。

1. realloc第一种行为引发的bug

1. void \*ptr = realloc(ptr, new\_size);
2. if (!ptr) {
3. 错误处理
4. }

这里就引出了一个内存泄露的问题，当realloc分配失败的时候，会返回NULL。但是参数中的ptr的内存是没有被释放的。如果直接将realloc的返回值赋给ptr。那么当申请内存失败时，就会造成ptr原来指向的内存丢失，造成泄露。

正确的处理应该是这样

1. void \*new\_ptr = realloc(ptr, new\_size);
2. if (!new\_ptr) {
3. 错误处理。
4. }
5. ptr = new\_ptr

2. 第三种行为引发的bug

这种bug由一种不好的编程习惯引发的。即认为申请内存始终可以成功，因此并不检查malloc的返回值。这在一般情况下，不会引发问题。但是对于realloc来说，当new\_size为0时，realloc返回NULL。而在后面的代码上，继续使用new\_ptr，比如会导致程序crash。

1. void \*new\_ptr = realloc(old\_ptr, new\_size);

 //其它代码

 ...... ......

从上面可以看出，在面对这个设计并非良好的API时，我们需要小心小心再小心。上面只是举了两个例子，其实还有一些其它的小问题

最后 再次 上代码已是说明：

**[cpp]** [view plain](https://blog.csdn.net/vevenlcf/article/details/47036127) [copy](https://blog.csdn.net/vevenlcf/article/details/47036127)

1. #include <stdlib.h>
2. #include<stdio.h>
4. main()
5. {
6. **char** \*p;
7. p=(**char** \*)malloc(100);
8. **if**(p)
9. printf("Memory Allocated at: %p\n",p);
10. **else**
11. printf("Not Enough Memory!/n");
12. **char** \*new\_p=(**char** \*)realloc(p,256);
13. **if**(new\_p)
14. {
15. printf("Memory Reallocated at: %p\n",new\_p);
16. }
17. **else**
18. {
19. printf("Not Enough Memory!/n");
20. free(p);
21. p = NULL;
22. **return** -1;
23. }
24. p = new\_p;
26. free(p);
27. **return** 0;
28. }