codediff之内存泄露

秦新良

2015年9月12日

目　录

1 引子

某天的codediff准时进行，但由于diff的代码改动非常小，不到十分钟代码便分享完成，于是小伙伴们开始走读本次修改代码对应特性的老代码。

走读代码的过程中，发现了一段如1所示的代码[1]，引起了我的注意。

```
1  void msg_proc(TMSG *msg)
2  {
3      B *pb = NULL;
4      process(msg, pb);
5      delete [] pb;
6
7      // something else to do ...
8
9      return;
10 }
```

代码 1: 内存释放

首先，该函数的内存使用没有遵守谁申请谁释放的原则。对于该场景，指针不需要在进程内传递，完全可以做到内存在同一函数内申请和释放。

其次，释放内存使用的是delete []，根据配对原则，在函数process中，必须使用new []申请内存。但跳转到process函数内，查看其申请内存的代码如2所示。

```
1  struct A
2  {
3      int  a;
4      int  b;
5      char desc[128];
6  };
7
8  struct B
9  {
10     int result;
11     int info_num;
12     A   info[];
13 };
14
15 void process(TMSG *msg, B* &pb)
16 {
17     // caculate the num of elements of struct A in struct B
18     int nelems = 12;
19
20     pb = new char [sizeof(B) + nelems*sizeof(A)];
```

[1] 本文中所引用的代码均是将业务逻辑剥离后的简化代码。

```
21
22      // logic processing....
23
24      return 0;
25  }
```

代码 2: 内存申请

我了个去，这是个什么鬼？这是我当时第一眼看到这个代码的感受。申请和释放内存的接口是配对使用了，但类型不匹配。

经过小伙伴们的激烈讨论，大家一致认为这个代码是有问题的，且问题是内存泄露，建议本次特性负责人在实际环境上给出验证结果。

但问题是否到此为至，除此之外是否还有其它问题？本文将探讨这种使用方法背后的原理及其可能引入的问题。

2 C 的 malloc 和 free

C 语言中申请和释放内存的接口分别是 malloc 和 free。程序调用 malloc 函数后，malloc 返回一个 void 类型的指针，指向新申请内存的起始地址；内存使用结束后，调用 free 函数，传入 malloc 返回的指针，即可完成内存的释放。

问题：调用 free 函数只传入一个指针，并没有传入该指针指向的内存大小，free 函数如何知道本次该释放多少内存？

要搞清楚这个问题，首先要搞清楚 malloc 是如何分配内存的。在 glibc[2] 的实现中，函数 __libc_malloc 是 malloc 函数的一个 strong_alias，可简单理解为该函数就是 malloc，其最终会调用 _int_malloc 函数来完成内存的分配，如代码3所示。

```
1   /*
2    *          call graph
3    * void *__libc_malloc (size_t bytes)
4    *             ||
5    *             \/
6    * void *_int_malloc (size_t bytes)
7    */
8
9   static void *_int_malloc (mstate av, size_t bytes)
10  {
11      INTERNAL_SIZE_T nb;         /* normalized request size */
```

[2] 本文引用的 glibc 代码的版本为 glibc-2.19。

```
12
13      /*
14        Convert request size to internal form by adding SIZE_SZ
15        bytes overhead plus possibly more to obtain necessary
16        alignment and/or to obtain a size of at least MINSIZE,
17        the smallest allocatable size. Also, checked_request2size
18        traps (returning 0) request sizes that are so large that
19        they wrap around zero when padded and aligned.
20      */
21
22      checked_request2size (bytes, nb);
23
24      /* ... */
25  }
```

代码 3: malloc 的实现

在代码3中，函数_int_malloc 的第二个参数bytes 是直接从__libc_malloc 函数透传下来的，即该参数的值就是程序调用malloc 函数传入的参数。在该函数的开始处，有一段注释，说明了紧接着注释下面的

checked_request2size (bytes, nb);

这行代码的作用：将请求的内存大小转换成内部实际分配的大小。具体的转换规则是外部请求大小 + SIZE_SZ + 字节对齐[3]，如果外部请求申请的内存太小，例如malloc(0)，这行代码会返回单次申请内存的最小值。这个转换过程是通过两个宏定义来实现的，如代码4所示。

```
1   /*  Same, except also perform argument check */
2   #define checked_request2size(req, sz)                           \
3       if (REQUEST_OUT_OF_RANGE (req)) {                           \
4           __set_errno (ENOMEM);                                   \
5           return 0;                                               \
6       }                                                           \
7       (sz) = request2size (req);
8
9
10  /* pad request bytes into a usable size -- internal version */
11  #define request2size(req)                                         \
12      (((req) + SIZE_SZ + MALLOC_ALIGN_MASK < MINSIZE)  ? \
13       MINSIZE :                                                    \
14       ((req) + SIZE_SZ + MALLOC_ALIGN_MASK) & ~MALLOC_ALIGN_MASK)
```

代码 4: request2size 的实现

[3]32位程序是8字节对齐。

从上面的分析及源代码可以看出，glibc分配的内存一定大于程序申请的内存。经过glibc分配的内存，其布局如图1所示。

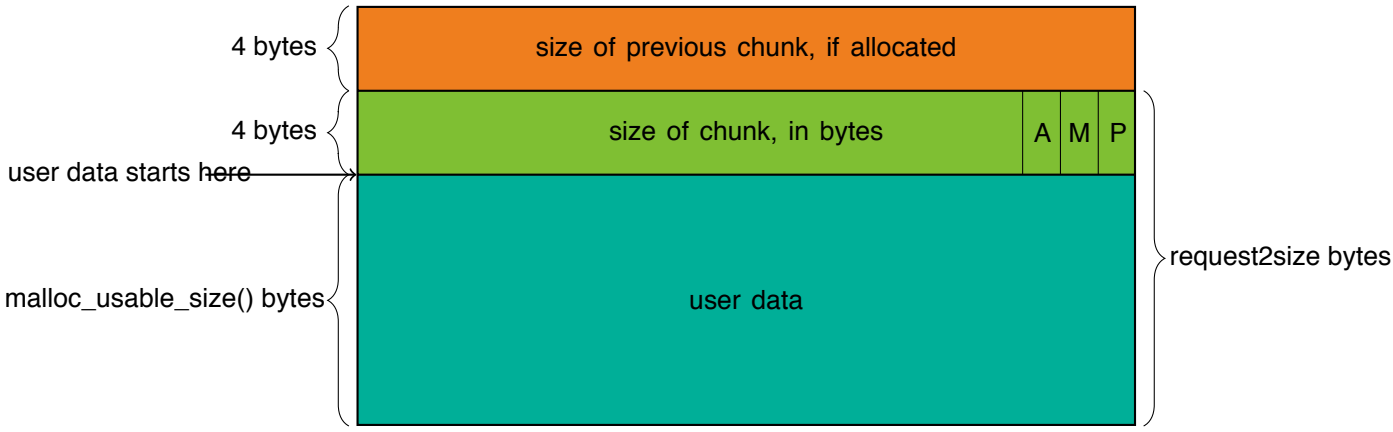

图 1: 内存布局

图1中的user data即glibc返回给应用程序的可使用内存，该块内存大小大于等于程序申请的内存。紧接着这片内存的前4字节内存，保存了本次malloc实际分配的内存大小，即request2size的结果。

32位程序glibc分配内存是8字节对齐的，所以4字节的size其低3位肯定为0，glibc使用这三个bit保存额外的管理信息，即图1中的的A、M和P三个bit。

glibc提供了malloc_usable_size()接口供应用程序获取实际分配（可使用）的内存大小[4]，使用该接口即可验证上面的分析，验证代码如5所示。

```
1  //
2  //  size.c
3  //
4  //  Created by q00148943 on 18/08/15.
5  //  Copyright (c) 2015 q00148943. All rights reserved.
6  //
7
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <malloc.h>
11
12 int main(int argc, char *argv[])
13 {
14     void   *p = malloc(atoi(argv[1]));
15     size_t *size = (char*)p - 4;
16
```

[4] 虽然glibc提供了该接口，但不建议应用程序调用该接口获取glic多分配的内存并使用该内存，应用程序还是按照自己申请的大小来使用。

```
17      printf("address = %p size = %d malloc_usable_size = %d\n",
18             size,
19             (*size)&(~7),        /* 8 bytes aligned */
20             malloc_usable_size(p));
21
22      free(p);
23
24      return 0;
25  }
```

代码 5: 验证代码

编译[5]代码5，针对申请不同的内存大小做验证，结果如6所示。从验证结果可以看出，malloc_usable_size() 大于等于应用程序申请的内存大小，而size等于malloc_usable_size() + 4，且是8的整数倍。

```
1  [q00148943@Inspiron-3421 Ubuntu]$ ./size 0
2  address = 0x850f004 size = 16 malloc_usable_size = 12
3  [q00148943@Inspiron-3421 Ubuntu]$ ./size 8
4  address = 0x839d004 size = 16 malloc_usable_size = 12
5  [q00148943@Inspiron-3421 Ubuntu]$ ./size 16
6  address = 0x937a004 size = 24 malloc_usable_size = 20
7  [q00148943@Inspiron-3421 Ubuntu]$ ./size 20
8  address = 0x848c004 size = 24 malloc_usable_size = 20
9  [q00148943@Inspiron-3421 Ubuntu]$ ./size 32
10 address = 0x97de004 size = 40 malloc_usable_size = 36
```

代码 6: 验证结果

一句话总结malloc：malloc在返回给应用程序内存地址的前4个字节地址处保存了本次分配的内存大小。

知道了malloc分配内存的原理，那如何在不传入长度的情况下正确的释放内存就很清楚了：根据应用程序传入的地址，向前偏移4个字节获取本次释放内存的大小并释放该大小的内存即可。

在glibc中，free函数的实现是__libc_free，该函数首先将应用程序传的的指针转换成chunk结构类型的指针，然后再调用_int_free完成内存的释放。在函数_int_free的开始便计算出了本次释放内存的大小，见代码7。

```
1  /*
2   *          call graph
3   * void *__libc_free (void *mem)
4   *             ||
5   *             \/
```

[5]编译环境信息：OS(Ubuntu 14.04 x86_64)、编译器(gcc version 4.8.2 (Ubuntu 4.8.2-19ubuntu1))、编译参数(gcc -m32)。

```
 * void *_int_free (mstate av, mchunkptr p, int have_lock)
 */

struct malloc_chunk {

    INTERNAL_SIZE_T prev_size;  /* Size of previous chunk (if
       free).  */
    INTERNAL_SIZE_T size;       /* Size in bytes, including
       overhead. */

    struct malloc_chunk* fd;   /* double links -- used only if
       free. */
    struct malloc_chunk* bk;

    /* Only used for large blocks: pointer to next larger size.
        */
    struct malloc_chunk* fd_nextsize; /* double links -- used
       only if free. */
    struct malloc_chunk* bk_nextsize;
};

typedef struct malloc_chunk* mchunkptr

#define mem2chunk(mem) ((mchunkptr)((char*)(mem) - 2*SIZE_SZ))

#define PREV_INUSE 0x1

#define IS_MMAPPED 0x2

#define NON_MAIN_ARENA 0x4

#define SIZE_BITS (PREV_INUSE | IS_MMAPPED | NON_MAIN_ARENA)

/* Get size, ignoring use bits */
#define chunksize(p)         ((p)->size & ~(SIZE_BITS))


void __libc_free (void *mem)
{
    mstate ar_ptr;
    mchunkptr p;      /* chunk corresponding to mem */

    if (mem == 0)     /* free(0) has no effect */
        return;

    p = mem2chunk (mem);

    ar_ptr = arena_for_chunk (p);
    _int_free (ar_ptr, p, 0);
}
```

```
51
52 static void _int_free (mstate av, mchunkptr p, int have_lock)
53 {
54     INTERNAL_SIZE_T size;        /* its size */
55
56     size = chunksize (p);
57 }
```

代码 7: free 的实现

mem2chunk 将 mem 指针减 2*SIZE_SZ 后直接强转成 malloc_chunk 类型的指针。32 位程序 SIZE_SZ 等于 4，mem2chunk 相当于 mem 指针减 8 再强转。chunksize 则直接通过 malloc_chunk 指针取其 size 成员。size 是 malloc_chunk 的第二个成员，相当于是计算 malloc_chunk 指针加 4 地址的值。结合 mem2chunk 的定义，该取值相当于是取（mem - 8 + 4）地址的值，即 free 函数是通过应用程序传入地址的前 4 个字节来计算本次释放内存的大小。

3 C++ 的 new 和 delete

C++ 中申请和释放内存的接口分别是 new 和 delete。程序调用 new 函数后，new 返回指定类型的指针，指向新申请对象的起始地址；内存使用结束后，调用 delete 函数，传入 new 返回的指针，即可完成内存的释放。

代码8和代码9分别是 gcc[6] 的 libstdc++ 中 new 和 delete 函数的实现。

```
1  // A freestanding C runtime may not provide "malloc" -- but
   there is no
2  // other reasonable way to implement "operator new".
3  extern "C" void *malloc (std::size_t);
4  #endif
5
6  _GLIBCXX_WEAK_DEFINITION void *
7  operator new (std::size_t sz) _GLIBCXX_THROW (std::bad_alloc)
8  {
9      void *p;
10
11     /* malloc (0) is unpredictable; avoid it.  */
12     if (sz == 0)
13         sz = 1;
14
15     while (__builtin_expect ((p = malloc (sz)) == 0, false))
16     {
```

[6] 本文引用的 gcc 代码的版本为 gcc-5.2.0。

```
17          new_handler handler = std::get_new_handler ();
18          if (! handler)
19              _GLIBCXX_THROW_OR_ABORT(bad_alloc());
20          handler ();
21      }
22
23      return p;
24  }
```

代码 8: new

```
1   // A freestanding C runtime may not provide "free" -- but there
        is no
2   // other reasonable way to implement "operator delete".
3   namespace std
4   {
5       _GLIBCXX_BEGIN_NAMESPACE_VERSION
6       extern "C" void free(void*);
7       _GLIBCXX_END_NAMESPACE_VERSION
8   } // namespace
9
10  _GLIBCXX_WEAK_DEFINITION void
11  operator delete(void* ptr) _GLIBCXX_USE_NOEXCEPT
12  {
13      std::free(ptr);
14  }
```

代码 9: delete

new 函数的核心功能是调用 malloc 申请指定大小的内存，除此之外都是异常处理。而 delete 函数则更为简单，直接调用 C 的 free 函数释放内存，无任何其它处理。从 new 和 delete 函数的实现来看，其本质和 C 的 malloc、free 没有任何差异。

在 C++ 中，使用 new 动态创建一个类的对象后，会自动调用该类的构造函数；使用 delete 销毁一个对象前，会自动调用其析构函数。但从代码8和代码9中 new 和 delete 的实现上来看，这两个函数本身并不会调用类的构造和析构函数。

构造和析构函数的调用，其实并不是 new 和 delete 函数的职责，而是编译器的责任[7]。代码在编译阶段，编译器检测到对象的创建，便生成调用对象构造函数的代码；同理，检测到对象的销毁，便生成调用对象析构函数的代码[8]。

代码11是代码10编译后反汇编的结果。第 9 行调用 new 申请内存，第 8 行是压栈 new 函数的第一个参数。new 函数只有一个参数，即申请内存的大小，而代码10中类 A

[7] 从局部变量在作用域开始生成并调用构造函数、作用域结束调用析构函数并销毁也可以说明构造和析构与 new 和 delete 无关，因为这种场景下根本不会有 new 和 delete 的参与。

[8] 包括局部自动生成的对象和使用 new 动态创建的对象。

只有一个string类型的成员变量，类的大小是4个字节，*movl $0x4,(%esp)*即表示调用new时，传入的入参是4。第11—12行是将new的返回值[9]压栈后调用类A的构造函数。第21—22行同样是先压栈new的返回值，然后调用类A的析构函数。第23—24行，压栈new的返回值后调用delete函数销毁对象。

```
1  #include <stdio.h>
2  #include <string>
3
4  using namespace std;
5
6  class A
7  {
8  public:
9      A(){};
10     ~A(){};
11
12 private:
13     string m_strData;
14 };
15
16 int main(int argc, char *argv[])
17 {
18     A *pa = new A;
19
20     printf("%p\n", pa);
21     delete pa;
22
23     return 0;
24 }
```

代码 10: C++代码

```
1  0804868d <main>:
2   804868d:        push   %ebp
3   804868e:        mov    %esp,%ebp
4   8048690:        push   %esi
5   8048691:        push   %ebx
6   8048692:        and    $0xfffffff0,%esp
7   8048695:        sub    $0x20,%esp
8   8048698:        movl   $0x4,(%esp)
9   804869f:        call   8048560 <_Znwj@plt>
10  80486a4:        mov    %eax,%ebx
11  80486a6:        mov    %ebx,(%esp)
12  80486a9:        call   8048700 <_ZN1AC1Ev>
13  80486ae:        mov    %ebx,0x1c(%esp)
14  80486b2:        mov    0x1c(%esp),%eax
15  80486b6:        mov    %eax,0x4(%esp)
16  80486ba:        movl   $0x80487c0,(%esp)
```

[9]即调用对象构造函数时传入的this指针。

```
80486c1:        call   8048550 <printf@plt>
80486c6:        mov    0x1c(%esp),%ebx
80486ca:        test   %ebx,%ebx
80486cc:        je     80486de <main+0x51>
80486ce:        mov    %ebx,(%esp)
80486d1:        call   8048714 <_ZN1AD1Ev>
80486d6:        mov    %ebx,(%esp)
80486d9:        call   8048520 <_ZdlPv@plt>
80486de:        mov    $0x0,%eax
80486e3:        jmp    80486f9 <main+0x6c>
80486e5:        mov    %eax,%esi
80486e7:        mov    %ebx,(%esp)
80486ea:        call   8048520 <_ZdlPv@plt>
80486ef:        mov    %esi,%eax
80486f1:        mov    %eax,(%esp)
80486f4:        call   8048580 <_Unwind_Resume@plt>
80486f9:        lea    -0x8(%ebp),%esp
80486fc:        pop    %ebx
80486fd:        pop    %esi
80486fe:        pop    %ebp
80486ff:        ret
```

代码 11: 汇编代码

搞清楚了new和delete的工作原理后，接着分析new[]和delete[]的原理。代码12和代码13分别是gcc中libstdc++实现的new[]和delete[]函数。

```
_GLIBCXX_WEAK_DEFINITION void*
operator new[] (std::size_t sz) _GLIBCXX_THROW (std::bad_alloc)
{
    return ::operator new(sz);
}
```

代码 12: new[]函数

```
_GLIBCXX_WEAK_DEFINITION void
operator delete[] (void *ptr) _GLIBCXX_USE_NOEXCEPT
{
    ::operator delete (ptr);
}
```

代码 13: delete[]

从实现来看，new[]、delete[]就是new、delete的简单封装，没有任何区别，但在功能上，两者有很大差异。几乎相同的代码却有着不同的功能，这个差异也是由编译器在编译阶段完成的。

将代码10稍做修改，修改为调用new[]申请0x10个类A的对象，在函数退出前调用delete[]将这些对象销毁，修改后的代码如14所示。

```
#include <stdio.h>
```

```
#include <string>

using namespace std;

class A
{
public:
    A(){};
    ~A(){};

private:
    string m_strData;
};

int main(int argc, char *argv[])
{
    A *pa = new A [0x10];

    printf("%p\n", pa);
    delete [] pa;

    return 0;
}
```

代码 14: 测试代码

代码14编译后反汇编的代码如下所示:

```
0804868d <main>:
 804868d:       push   %ebp
 804868e:       mov    %esp,%ebp
 8048690:       push   %edi
 8048691:       push   %esi
 8048692:       push   %ebx
 8048693:       and    $0xfffffff0,%esp
 8048696:       sub    $0x20,%esp
 8048699:       movl   $0x44,(%esp)
 80486a0:       call   8048550 <_Znaj@plt>
 80486a5:       mov    %eax,%ebx
 80486a7:       movl   $0x10,(%ebx)
 80486ad:       lea    0x4(%ebx),%esi
 80486b0:       mov    $0xf,%edi
 80486b5:       mov    %esi,0xc(%esp)
 80486b9:       jmp    80486cf <main+0x42>
 80486bb:       mov    0xc(%esp),%eax
 80486bf:       mov    %eax,(%esp)
 80486c2:       call   804877a <_ZN1AC1Ev>
 80486c7:       addl   $0x4,0xc(%esp)
 80486cc:       sub    $0x1,%edi
 80486cf:       cmp    $0xffffffff,%edi
 80486d2:       jne    80486bb <main+0x2e>
 80486d4:       lea    0x4(%ebx),%eax
```

```
80486d7:        mov    %eax,0x1c(%esp)
80486db:        mov    0x1c(%esp),%eax
80486df:        mov    %eax,0x4(%esp)
80486e3:        movl   $0x8048840,(%esp)
80486ea:        call   8048540 <printf@plt>
80486ef:        cmpl   $0x0,0x1c(%esp)
80486f4:        je     804872f <main+0xa2>
80486f6:        mov    0x1c(%esp),%eax
80486fa:        sub    $0x4,%eax
80486fd:        mov    (%eax),%eax
80486ff:        lea    0x0(,%eax,4),%edx
8048706:        mov    0x1c(%esp),%eax
804870a:        lea    (%edx,%eax,1),%ebx
804870d:        cmp    0x1c(%esp),%ebx
8048711:        je     8048720 <main+0x93>
8048713:        sub    $0x4,%ebx
8048716:        mov    %ebx,(%esp)
8048719:        call   804878e <_ZN1AD1Ev>
804871e:        jmp    804870d <main+0x80>
8048720:        mov    0x1c(%esp),%eax
8048724:        sub    $0x4,%eax
8048727:        mov    %eax,(%esp)
804872a:        call   8048560 <_ZdaPv@plt>
804872f:        mov    $0x0,%eax
8048734:        jmp    8048772 <main+0xe5>
8048736:        mov    %eax,0xc(%esp)
804873a:        test   %esi,%esi
804873c:        je     804875c <main+0xcf>
804873e:        mov    $0xf,%eax
8048743:        sub    %edi,%eax
8048745:        shl    $0x2,%eax
8048748:        lea    (%esi,%eax,1),%edi
804874b:        cmp    %esi,%edi
804874d:        je     804875c <main+0xcf>
804874f:        sub    $0x4,%edi
8048752:        mov    %edi,(%esp)
8048755:        call   804878e <_ZN1AD1Ev>
804875a:        jmp    804874b <main+0xbe>
804875c:        mov    0xc(%esp),%esi
8048760:        mov    %ebx,(%esp)
8048763:        call   8048560 <_ZdaPv@plt>
8048768:        mov    %esi,%eax
804876a:        mov    %eax,(%esp)
804876d:        call   8048580 <_Unwind_Resume@plt>
8048772:        lea    -0xc(%ebp),%esp
8048775:        pop    %ebx
8048776:        pop    %esi
8048777:        pop    %edi
8048778:        pop    %ebp
```

```
74 8048779:       ret
```

代码 15: 汇编代码

代码15的第 10 行调用 new[] 申请内存，第 9 行将new[] 的参数压栈，但压栈的值是0x44，而不是0x40。类 A 的大小是 4 字节，0x10 个 A 的对象应该是0x40 字节，但这两行指令的功能是申请0x44 字节的内存，这就是new[] 和new 最本质的不同。多申请的 4 字节作用是什么，接着看后面的代码。

11—12 行的功能是获取new[] 的返回值，即本次申请内存的起始地址，然后将该地址的值赋为 0x10，即在 new[] 返回地址的起始 4 个字节保存本次申请对象的个数。

13—23 行的功能是从new[] 返回的地址 +4 开始，调用 0x10 次类 A 的构造函数，每次调用构造函数传入的 this 指针是：new[] 返回的地址 + 4 + 4n，其中 n 为从 0 开始循环 0x10 次的循环计数。

24—29 行的作用是调用 printf 函数打印地址（new 返回的地址 + 4，即代码14中的 pb 指针）。

30—43 行代码的作用是调用申请的 0x10 个对象的析构函数，注意析构函数的调用顺序正好和构造函数相反，构造函数的调用是从低地址对象一直到高地址对象，而析构是先析构高地址对象再析构低地址对象。

44—47 行代码的作用是调用 delete[] 函数释放内存，注意调用 delete[] 函数时，传入的指针是new[] 返回的地址，而不是代码14中的 pb 指针指向的地址。

结合前面分析的 malloc 的内存布局，画出代码14中申请 0x10 个类 A 对象的内存布局如图2所示[10]。

[10] 图2中所示的内存布局有一个前提假设是 malloc 分配的可使用内存正好等于上层应用申请的内存。

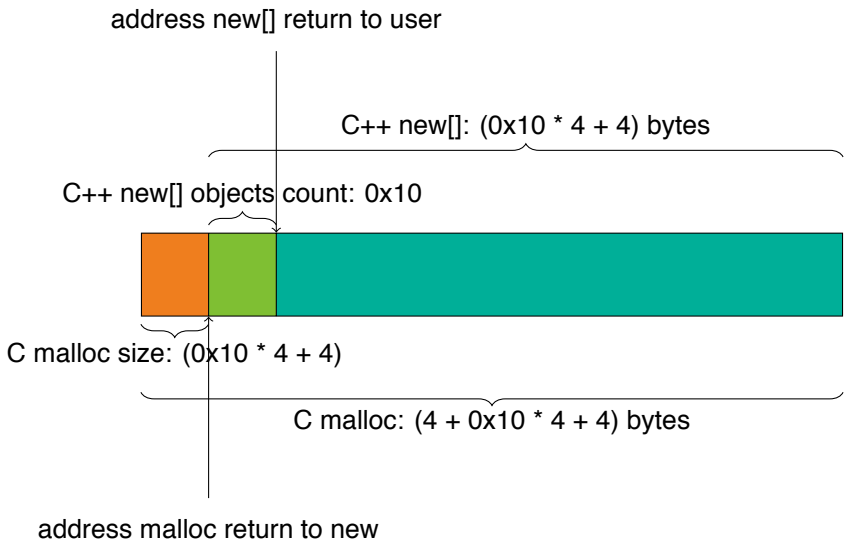

图 2

综上所述，new[] 的实现原理是：调用 new 申请比实际申请内存多 4 字节的内存，并在多出的 4 字节内存地址处保存动态申请的对象个数，最后再从 new 返回的地址加 4 开始，循环调用类的构造函数；delete[] 则是 new[] 的逆向过程，先从 new[] 返回的地址读取要释放对象的个数，然后调用每个对象的析构函数，最后调用 delete 释放内存。

4 内存泄露?

搞清楚 malloc/free、new/delete 和 new[]/delete[] 的原理后，再回到问题的开始，代码1和2这种使用方式的问题在哪儿?

代码2中，struct B 的成员 info 定义为 struct A 的数组，但该数组没有指定长度。可能原作者将这个数组理解为一个变长数组，这样的话，使用代码

```
pb = new char [sizeof(B) + nelems*sizeof(A)];
```

申请内存后，nelems*sizeof(A) 变成了 B 的成员变量，即 info 数组中的元素，所以在使用

```
delete [] pb;
```

释放内存时，这段内存也会一起释放。但实际上 info 在 B 中是一个 0 长数组，只是一个占位符，在 B 的内存模型中，info 不占任何内存。0 长数组的使用是 C 语言编码的一个

技巧，最常用的场景就是TLV编码。TLV编码是为了在进程间传递消息，这就要求编码的结构其内存是连续的，如代码16所示。

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct TLV
{
    int  tag;
    int  length;
    char value[]; /* equals to char value[0] */
}TLV;

int main(int argc, char *argv[])
{
    TLV *tlv1 = malloc(sizeof(struct TLV) + 0x10);

    tlv1->tag    = 0x0A;
    tlv1->length = 0x10;

    strcpy(tlv1->value, "0123456789ABCDEF");

    printf("%c %c %c ...\n", tlv1->value[13], tlv1->value[14],
       tlv1->value[15]);


    TLV *tlv2 = malloc(sizeof(struct TLV) + 10);

    tlv2->tag    = 0x0A;
    tlv2->length = 10;

    strcpy(tlv2->value, "0123456789");

    printf("%c %c %c ...\n", tlv2->value[0], tlv2->value[1],
       tlv2->value[2]);

    free(tlv1);
    free(tlv2);
    return 0;
}
```

代码 16: TLV

读者可以猜测一下代码16的输出结果。只看这个样例代码，似乎正好说明了TLV中的value就是一个变长数组：第一个例子中，其长度是0x10，第二个例子中，其长度是10。除此之外，value的值同样可以使用数组下标的方式来获取。但这仅仅是因为申请tlv1和tlv2的内存是连续的，而并不是因为value是动态的，因为对于任何一个类型，

编译器在编译时必须确定其大小，否则代码中使用到该类型的变量时，编译器将无法生成为其分配内存的代码。

前面说过value[0]或value[]其实是一个0长数组，只是一个占位符，在TLV结构中，其所占内存大小为0，所以struct TLV的sizeof结果是8。那这个占位符究竟有何作用？试想下，如果不定义这个占位符，进程间使用TLV结构通信，那接收端在解析TLV时，先解析到TLV的起始地址，然后从该地址加8开始取数据；但定义了这个占位符后，代码就可以这样写：

```
strcpy(target, TLV->value, TLV->length);
```

即由编译器在生成代码时自动从TLV + 8开始拷贝length长度的内容，而不需要由程序员考虑实际的value到底是从哪个地址开始的。

了解了0长数组的功能后，那么代码1中通过delete [] pb释放代码2中通过new char [sizeof(B) + nelems*sizeof(A)];申请的内存肯定是有问题的，因为delete [] pb只会释放B对应的8字节内存，剩余的nelems*sizeof(A)字节的内存会泄露。实际情况是不是这样，继续分析。

按照前面分析的new []和delete []的原理，new char [sizeof(B) + nelems*sizeof(A)];会申请(sizeof(B) + nelems*sizeof(A) + 4)个字节的内存，然后在申请的内存的开始4字节保存本次申请的char对象的个数，即(sizeof(B) + nelems*sizeof(A))；而delete [] pb会从pb指向内存地址的前4个字节读取对象B的个数，然后再循环调用B的析构函数，最后通过delete (pb + 4)释放内存。按照这个分析，这段代码在运行时有很大概率会出现进程异常退出，因为通过new char [sizeof(B) + nelems*sizeof(A)];申请的内存的范围是0—(sizeof(B) + nelems*sizeof(A))[11]，而通过delete [] pb释放内存前，调用B的构造函数访问的内存范围是0—(sizeof(B) + nelems*sizeof(A))*8[12]，这会因为内存越界而导致进程异常退出。

综上所述，这段代码会导致进程退出，而不是内存泄露。但对这段代码对应的功能反复测试，结果一切正常，即不会内存泄露，也不会进程退出。为了搞清楚这段代码到

[11] 实际通过new申请的地址不会从0开始，这里只是为了说明问题简化描述。

[12] B的大小是8字节，从0开始遍历(sizeof(B) + nelems*sizeof(A))个B的对象，其操作的内存范围是0 (sizeof(B) + nelems*sizeof(A))*8。

底有没有问题，把代码1和代码2合并成一份代码17。代码17编译反汇编后的结果如代码18所示。

```
#include <stdio.h>

struct A
{
    int  a;
    int  b;
    char desc[128];
};

struct B
{
    int result;
    int info_num;
    A   info[];
};

int main(int argc, char *argv[])
{
    B *pb = (B*)new char [sizeof(B) + 12*sizeof(A)];
    printf("%p\n", pb);
    delete [] pb;

    return 0;
}
```

代码 17: 测试代码

```
0804853d <main>:
 804853d:       push   %ebp
 804853e:       mov    %esp,%ebp
 8048540:       and    $0xfffffff0,%esp
 8048543:       sub    $0x20,%esp
 8048546:       mov    $0x660,%eax
 804854b:       add    $0x8,%eax
 804854e:       mov    %eax,(%esp)
 8048551:       call   8048420 <_Znaj@plt>
 8048556:       mov    %eax,0x1c(%esp)
 804855a:       mov    0x1c(%esp),%eax
 804855e:       mov    %eax,0x4(%esp)
 8048562:       movl   $0x8048620,(%esp)
 8048569:       call   8048410 <printf@plt>
 804856e:       cmpl   $0x0,0x1c(%esp)
 8048573:       je     8048581 <main+0x44>
 8048575:       mov    0x1c(%esp),%eax
 8048579:       mov    %eax,(%esp)
 804857c:       call   8048430 <_ZdaPv@plt>
 8048581:       mov    $0x0,%eax
 8048586:       leave
 8048587:       ret
```

```
8048588:        xchg   %ax,%ax
804858a:        xchg   %ax,%ax
804858c:        xchg   %ax,%ax
804858e:        xchg   %ax,%ax
```

代码 18: 汇编代码

代码18的第 9 行调用 new [] 申请内存，第 8 行将%eax 寄存器中的值压栈作为 new [] 的入参，即本次申请内存的大小；而%eax 寄存器中的值是 0x8 + 0x660，即本次申请内存的实际大小，并没有像之前分析的 new [] 操作会多申请 4 个字节的内存保存本次申请对象的个数。

第 14 行调用 printf 打印指针 pa；15 行判断 pb 指针是否为空，如果不为空则执行 17 行，而 17—18 行的作用是将 pb 指针压栈，作为 19 行调用 delete [] 的入参，最终由 19 行的 delete [] 将内存释放。

这样分析后发现，整个内存申请和释放过程中，没有任何对象的构造和析构函数被调用。再结合最前面的分析，new [] 和 delete [] 其实就是 new 和 delete 的简单封装，而 new 和 delete 最终是通过 malloc 和 free 申请和释放内存的，按照这个顺序一步一步往下推导，最终可以得出这样的结论：代码17中 main 函数的功能和代码19中 main 函数的功能完全等价。

```
int main(int argc, char *argv[])
{
    B *pb = (B*)malloc(sizeof(B) + 12*sizeof(A));
    printf("%p\n", pb);
    free(pb);

    return 0;
}
```

代码 19: 测试代码

对代码19中内存的申请和释放，相信不会有任何人会怀疑其有内存泄露或会由于内存越界导致进程退出。这就是代码1和代码2在数据类型不配套的情况下使用 new [] 和 delete [] 即不会内存泄露也不会引起进程重启的真正原因，因为在该场景下，new [] 和 delete [] 最终会“退化”成了 malloc 和 free。

问题搞清楚了，结论是该场景下这样使用没问题，前提条件是 new 的数据类型无构造函数、delete 的的数据类型无析构函数。也许有人会问，char 类型没有构造和析构函数很好理解，为什么 struct B 也即没有构造也没有析构函数？在不显式定义类的构造和析构的场景下，编译器难道不会为该类自动生成这两个函数？答案是确实不会，为什么

不会，感兴趣的同学可以参考下C++大师Stanley B. Lippman的《深度探索C++对象模型》这本书，在书里作者详细描述了构造、析构以及拷贝构造这些函数哪些场景下编译器会自动生成，哪些场景下不会。

最后，给代码17中的B显示定义一个空的析构函数B() return;，编译并运行该程序，结果如下：

```
[q00148943@Inspiron-3421 Ubuntu]$ ./test
0x9b0e008
*** Error in `./test': free(): invalid pointer: 0x09b0e004 ***
Aborted (core dumped)
```

代码 20: core dumped

进程会core dumped，感兴趣的同学可以思考下，这个core dumped是因为引用了图2中的哪个地址导致的。