# Malloc的使用过程：CRT(C-Run-Time)

Company \*mircosoft = (Company \*)malloc(sizeof(Company));

1. 用户来确定分配内存空间的大小
2. Malloc返回空指针(void\*)，需要强制类型转换
3. Malloc可能内存分配失败，需要用户检查指针，内存分配是否成功
4. Malloc不会调用构造函数，需要用户自己来初始对象（最容易出错的地方）

如果你要delete的指针是0，没有什么事情会发生，因此可以在delete一个指针后，将其值设置为0，避免对一个指针delete两次。

# 使一个指针更像一个数组

Company\* const google = new Company[3];

定一个指向数组的指针，在使用过程中，有可能将它又指向了单个对象，这时在释放时就容易出错了。可以通过加上const来强制对象指针不能改变。

# 使用new来抛出异常

当new无法分配一段连续的空间时，会调用new\_handler.(检查一个指向函数的指针，如果该指针为非0，就会调用这个函数)。此时会抛出一个异常，因此如果在heap上分配内存，最好在自己的程序中用一个提示小时来替换new\_handler，并abort()程序，从而方便找到问题的来源。通过调用new.h中的set\_new\_handler()来替换new\_handler，向这个函数中传入你想安装的函数的地址。

#include <iostream>

#include <cstdlib>

#include <new>

void run\_out\_memory()

{

cerr<<"memory exhausted!"<<endl;

exit(1);

}

int main()

{

set\_new\_handler(run\_out\_memory);

while (1)

{

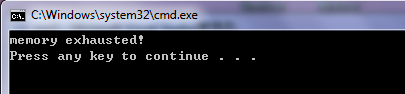
new int[1000]; //Exhausts the memory

}

return 0;

}

当程序内存耗尽会有如下提示：



# \_CrtMemBlockHeader结构

在VC里面，用release模式编译运行程序的时候，堆分配（Heap allocation）的时候调用的是malloc，如果你要分配10byte的空间，那么就会只分配10byte空间，而用debug模式的时候，堆分配调用的是\_malloc\_dbg，如果你只要分配10byte的空间，那么它会分配出除了你要的10byte之外，还要多出约36byte空间，用于存储一些薄记信息，debug堆分配出来之后就会按顺序连成一个链。

blockSize = sizeof(\_CrtMemBlockHeader) + nSize + nNoMansLandSize;

分配出的10byte空间的前面会有一个32byte的附加信息，存储的是一个\_CrtMemBlockHeader结构，可以在DBGINT.H中找到该结构的定义：

/\*

\* For diagnostic purpose, blocks are allocated with extra information and

\* stored in a doubly-linked list. This makes all blocks registered with

\* how big they are, when they were allocated, and what they are used for.

\*/

#define nNoMansLandSize 4

typedef struct \_CrtMemBlockHeader

{

struct \_CrtMemBlockHeader \* pBlockHeaderNext;

struct \_CrtMemBlockHeader \* pBlockHeaderPrev;

char \* szFileName;

int nLine;

#ifdef \_WIN64

/\* These items are reversed on Win64 to eliminate gaps in the struct

\* and ensure that sizeof(struct)%16 == 0, so 16-byte alignment is

\* maintained in the debug heap.

\*/

int nBlockUse;

size\_t nDataSize;

#else /\* \_WIN64 \*/

size\_t nDataSize;

int nBlockUse;

#endif /\* \_WIN64 \*/

long lRequest;

unsigned char gap[nNoMansLandSize];

/\* followed by:

\* unsigned char data[nDataSize];

\* unsigned char anotherGap[nNoMansLandSize];

\*/

} \_CrtMemBlockHeader;

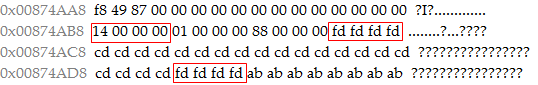
No Man's Land是指（一战中）两军交战的无人地带，是双方要争夺的地方。又称三不管地带。其范围通常为两个壕沟之间的区域。<http://en.wikipedia.org/wiki/No_man%27s_land>

# Windows中填充字符的含义

|  |  |  |
| --- | --- | --- |
| **Value** | **Name** | **Description** |
| 0xCD | **C**lean Memory | Allocated memory via malloc or new but never written by the application. |
| 0xDD | **D**ead Memory | Memory that has been released with delete or free. It is used to detect writing through dangling pointers. |
| 0xFD | **F**ence Memory | Also known as "no mans land." This is used to wrap the allocated memory (like surrounding it with fences) and is used to detect indexing arrays out of bounds. |
| 0xAB | (**A**llocated**B**lock?) | Memory allocated by LocalAlloc(). |
| 0xBAADF00D | Bad Food | Memory allocated by LocalAlloc() with LMEM\_FIXED, but not yet written to. |
| 0xCC |  | When the code is compiled with the /GZ option, uninitialized variables are automatically assigned to this value (at byte level). |

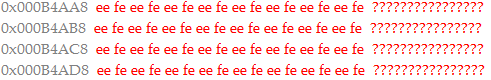
# 内置类型的new

执行int\* values = new int[5];



其中cd为分配的变量，fd为前后的gap，14h为分配的数据大小，当释放内存时，CRT会检查前后得到gap是否还是fd，如果不是就说明内存访问有问题。

在执行了 delete []values; 后



被重置为0xFEEEFEEE

# New的执行过程

void \*\_\_CRTDECL operator new[](size\_t count) \_THROW1(std::bad\_alloc) //数组会多调用这个函数，再转到下一个operator new

void \*\_\_CRTDECL operator new(size\_t size) \_THROW1(\_STD bad\_alloc)

Get a block of memory from the debug heap

extern "C" \_CRTIMP void \* \_\_cdecl malloc ( size\_t nSize )

Allocate of block of memory of at least size bytes from the debug heap and return a pointer to it. Assumes heap already locked.

extern "C" void \* \_\_cdecl \_nh\_malloc\_dbg ( size\_t nSize, int nhFlag, int nBlockUse, const char \* szFileName, int nLine )

extern "C" static void \* \_\_cdecl \_nh\_malloc\_dbg\_impl ( size\_t nSize, int nhFlag, int nBlockUse, const char \* szFileName, int nLine, int \* errno\_tmp )

Does heap allocation.这个函数做的工作比较多：

extern "C" static void \* \_\_cdecl \_heap\_alloc\_dbg\_impl( size\_t nSize, int nBlockUse, const char \* szFileName, int nLine, int \* errno\_tmp)

这个函数里有个锁堆的操作\_mlock(\_HEAP\_LOCK);

并在其中计算块的大小：

blockSize = sizeof(\_CrtMemBlockHeader) + nSize + nNoMansLandSize;

pHead = (\_CrtMemBlockHeader \*)\_heap\_alloc\_base(blockSize);这一句调用了函数

\_\_forceinline void \* \_\_cdecl \_heap\_alloc (size\_t size)，其中有一句

return HeapAlloc(\_crtheap, 0, size ? size : 1);

接下来填写\_CrtMemBlockHeader头信息

填充gap和分配的数据空间

/\* fill in gap before and after real block \*/

memset((void \*)pHead->gap, \_bNoMansLandFill, nNoMansLandSize);

memset((void \*)(pbData(pHead) + nSize), \_bNoMansLandFill, nNoMansLandSize);

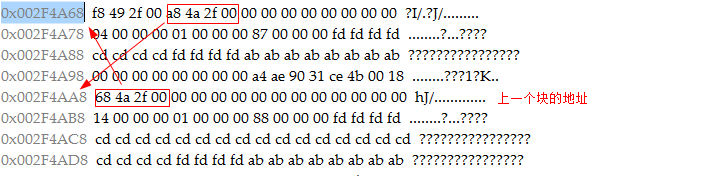
/\* fill data with silly value (but non-zero) \*/

memset((void \*)pbData(pHead), \_bCleanLandFill, nSize);

下图为这两句指令执行后的内存情况

int\* value = new int;

int\* values = new int[5];



# delete []values;的过程如下：

void operator delete[]( void \* p )

void operator delete( void \*pUserData ) 先给堆加锁

/\* get a pointer to memory block header \*/

pHead = pHdr(pUserData);

\_free\_dbg( pUserData, pHead->nBlockUse ); //调用free函数

//Frees any type of supported block.

extern "C" \_CRTIMP void \_\_cdecl \_free\_dbg(void \* pUserData, int nBlockUse)

\_mlock(\_HEAP\_LOCK); //再次加锁

extern "C" void \_\_cdecl \_free\_dbg\_nolock( void \* pUserData, int nBlockUse )

extern "C" static int \_\_cdecl CheckBytes( unsigned char \* pb,unsigned char bCheck, size\_t nSize)

extern "C" \_CRTIMP int \_\_cdecl \_CrtIsValidHeapPointer( const void \* pUserData)

return HeapValidate( \_crtheap, 0, pHdr(pUserData) );

/\* check no-mans-land gaps \*/

if (!CheckBytes(pHead->gap, \_bNoMansLandFill, nNoMansLandSize))

更新链表并设置内存字节的值

/\* fill the entire block including header with dead-land-fill \*/

memset(pHead, \_bDeadLandFill, sizeof(\_CrtMemBlockHeader) + pHead->nDataSize + nNoMansLandSize);

然后在\_free\_base(pHead);中调用

retval = HeapFree(\_crtheap, 0, pBlock);

# 对于类对象的new

在执行为operator new之后，会跳转到类的构造函数处

Company\* twitter = new Company;

00B4487C push 4

00B4487E call operator new (0B41203h)

00B44883 add esp,4

00B44886 mov dword ptr [ebp-158h],eax

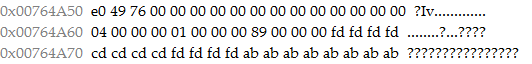
00B4488C mov dword ptr [ebp-4],0

00B44893 cmp dword ptr [ebp-158h],0 //验证new的返回值是否为0

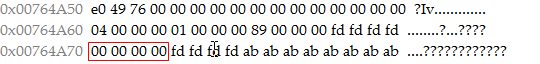
00B4489A je main+0FFh (0B448AFh)

00B4489C mov ecx,dword ptr [ebp-158h]

此时对象twitter在内存中的存储如下



00B448A2 call Company::Company (0B411C2h) 通过构造函数将对象的中赋值为0



00B448A7 mov dword ptr [ebp-19Ch],eax

00B448AD jmp main+109h (0B448B9h)

00B448AF mov dword ptr [ebp-19Ch],0

00B448B9 mov eax,dword ptr [ebp-19Ch]

00B448BF mov dword ptr [ebp-164h],eax

00B448C5 mov dword ptr [ebp-4],0FFFFFFFFh

00B448CC mov ecx,dword ptr [ebp-164h]

00B448D2 mov dword ptr [ebp-38h],ecx

# New一个对象数组

对象数组的情况，返回的地址不包括前面4字节数组的大小，而是第一个对象的地址

Company\* const google = new Company[3];

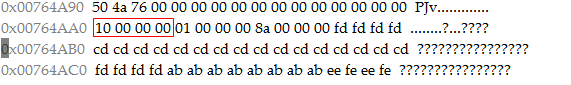
00B448D5 push 10h

00B448D7 call operator new[] (0B410E1h)

00B448DC add esp,4

00B448DF mov dword ptr [ebp-140h],eax

这次分配的内存大小是16个字节，比实际使用的多了4字节用来存储数组大小，而内置类型没有这4个字节



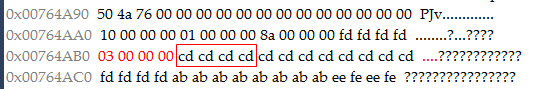
00B448E5 mov dword ptr [ebp-4],1

00B448EC cmp dword ptr [ebp-140h],0

00B448F3 je main+17Fh (0B4492Fh)

00B448F5 mov eax,dword ptr [ebp-140h]

00B448FB mov dword ptr [eax],3 //将数组元素的个数写入内存区域中，第一个对象的地址为0x00764ab4，现在的值为cd



00B44901 push offset Company::~Company (0B41019h) //压入析构

00B44906 push offset Company::Company (0B411C2h) //压入构造

00B4490B push 3 //数组元素的个数

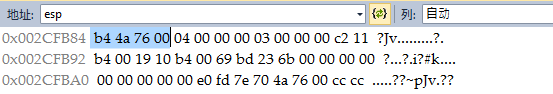
00B4490D push 4 //数组元素的长度

00B4490F mov ecx,dword ptr [ebp-140h]

00B44915 add ecx,4

//加4取得第一个对象的地址，因为内存空间的前四个字节是数组元素的个数

00B44918 push ecx //此时堆栈中压入了第一个对象的地址



00B44919 call `eh vector constructor iterator' (0B41181h) //调用构造函数

00B4491E mov edx,dword ptr [ebp-140h]

00B44924 add edx,4

00B44927 mov dword ptr [ebp-19Ch],edx

00B4492D jmp main+189h (0B44939h)

00B4492F mov dword ptr [ebp-19Ch],0

00B44939 mov eax,dword ptr [ebp-19Ch]

00B4493F mov dword ptr [ebp-14Ch],eax

00B44945 mov dword ptr [ebp-4],0FFFFFFFFh

00B4494C mov ecx,dword ptr [ebp-14Ch]

00B44952 mov dword ptr [ebp-44h],ecx

根据上一步的入栈情况，下面函数中ebp+8为对象的地址，ebp+0Ch为每个元素的大小，ebp+10h为元素个数，ebp+14h为构造函数地址，ebp+18h为析构函数地址

`eh vector constructor iterator':

00B42440 mov edi,edi

00B42442 push ebp

00B42443 mov ebp,esp

00B42445 push 0FFFFFFFEh

00B42447 push offset \_\_\_rtc\_tzz+210h (0B48D60h)

00B4244C push offset @ILT+140(\_\_except\_handler4) (0B41091h)

00B42451 mov eax,dword ptr fs:[00000000h]

00B42457 push eax

00B42458 add esp,0FFFFFFF0h

00B4245B push ebx

00B4245C push esi

00B4245D push edi

00B4245E mov eax,dword ptr [\_\_\_security\_cookie (0B49000h)]

00B42463 xor dword ptr [ebp-8],eax

00B42466 xor eax,ebp

00B42468 push eax

00B42469 lea eax,[ebp-10h]

00B4246C mov dword ptr fs:[00000000h],eax

00B42472 mov dword ptr [ebp-20h],0

00B42479 mov dword ptr [ebp-4],0

00B42480 mov dword ptr [ebp-1Ch],0

00B42487 jmp `eh vector constructor iterator'+52h (0B42492h)

00B42489 mov eax,dword ptr [ebp-1Ch]

00B4248C add eax,1 //将已经构造的对象个数增加1，从0开始计数

00B4248F mov dword ptr [ebp-1Ch],eax //保存已经构造的元素个数

00B42492 mov ecx,dword ptr [ebp-1Ch]

00B42495 cmp ecx,dword ptr [ebp+10h] //与元素个数进行比较

00B42498 jge `eh vector constructor iterator'+6Bh (0B424ABh)

00B4249A mov ecx,dword ptr [ebp+8] //将对象地址传给构造函数

00B4249D call dword ptr [ebp+14h] //调用构造函数

00B424A0 mov edx,dword ptr [ebp+8]

//将对象的地址增加每个对象的大小，从而指向下一个对象地址

00B424A3 add edx,dword ptr [ebp+0Ch]

00B424A6 mov dword ptr [ebp+8],edx // ebp+8中总是存储需对象的地址

00B424A9 jmp `eh vector constructor iterator'+49h (0B42489h)

00B424AB mov dword ptr [ebp-20h],1

00B424B2 mov dword ptr [ebp-4],0FFFFFFFEh

# 删除一个对象数组的情况：

delete []google;

002B4992 mov eax,dword ptr [ebp-44h]

002B4995 mov dword ptr [ebp-110h],eax //存入对象google的地址

002B499B mov ecx,dword ptr [ebp-110h]

002B49A1 mov dword ptr [ebp-11Ch],ecx

002B49A7 cmp dword ptr [ebp-11Ch],0

002B49AE je main+215h (2B49C5h)

002B49B0 push 3 //压入数组的个数

002B49B2 mov ecx,dword ptr [ebp-11Ch]

002B49B8 call Company::`vector deleting destructor' (2B118Bh)

002B49BD mov dword ptr [ebp-19Ch],eax

真正执行析构的地方

Company::`vector deleting destructor':

002B18D0 push ebp

002B18D1 mov ebp,esp

002B18D3 sub esp,0CCh

002B18D9 push ebx

002B18DA push esi

002B18DB push edi

002B18DC push ecx

002B18DD lea edi,[ebp-0CCh]

002B18E3 mov ecx,33h

002B18E8 mov eax,0CCCCCCCCh

002B18ED rep stos dword ptr es:[edi]

002B18EF pop ecx

002B18F0 mov dword ptr [ebp-8],ecx //对象的地址

002B18F3 mov eax,dword ptr [ebp+8] //数组元素个数

002B18F6 and eax,2 //0标志ZF为1就跳转

002B18F9 je Company::`vector deleting destructor'+61h (2B1931h)

002B18FB push offset Company::~Company (2B1019h)

002B1900 mov eax,dword ptr [this]

002B1903 mov ecx,dword ptr [eax-4] //将元素个数存入ecx中

002B1906 push ecx //元素个数入栈

002B1907 push 4

002B1909 mov edx,dword ptr [this]

002B190C push edx //对象地址入栈

002B190D call `eh vector destructor iterator' (2B1267h) //见下面代码片段

002B1912 mov eax,dword ptr [ebp+8]

002B1915 and eax,1

002B1918 je Company::`vector deleting destructor'+59h (2B1929h)

002B191A mov eax,dword ptr [this]

002B191D sub eax,4

002B1920 push eax

002B1921 call operator delete[] (2B1028h)

002B1926 add esp,4

002B1929 mov eax,dword ptr [this]

002B192C sub eax,4

002B192F jmp Company::`vector deleting destructor'+80h (2B1950h)

002B1931 mov ecx,dword ptr [this]

002B1934 call Company::~Company (2B1019h) //调用析构函数

002B1939 mov eax,dword ptr [ebp+8]

002B193C and eax,1

002B193F je Company::`vector deleting destructor'+7Dh (2B194Dh)

002B1941 mov eax,dword ptr [this]

002B1944 push eax

002B1945 call operator delete (2B10AAh)

002B194A add esp,4

002B194D mov eax,dword ptr [this]

002B1950 pop edi

002B1951 pop esi

002B1952 pop ebx

002B1953 add esp,0CCh

002B1959 cmp ebp,esp

002B195B call @ILT+425(\_\_RTC\_CheckEsp) (2B11AEh)

002B1960 mov esp,ebp

002B1962 pop ebp

下面函数中ebp+8为对象的地址，ebp+0Ch为每个元素的大小，ebp+10h为元素个数，ebp+14h为析构函数地址

`eh vector destructor iterator':

00FE27C0 mov edi,edi

00FE27C2 push ebp

00FE27C3 mov ebp,esp

00FE27C5 push 0FFFFFFFEh

00FE27C7 push offset \_\_\_rtc\_tzz+230h (0FE8D80h)

00FE27CC push offset @ILT+140(\_\_except\_handler4) (0FE1091h)

00FE27D1 mov eax,dword ptr fs:[00000000h]

00FE27D7 push eax

00FE27D8 add esp,0FFFFFFF4h

00FE27DB push ebx

00FE27DC push esi

00FE27DD push edi

00FE27DE mov eax,dword ptr [\_\_\_security\_cookie (0FE9000h)]

00FE27E3 xor dword ptr [ebp-8],eax

00FE27E6 xor eax,ebp

00FE27E8 push eax

00FE27E9 lea eax,[ebp-10h]

00FE27EC mov dword ptr fs:[00000000h],eax

00FE27F2 xor eax,eax

00FE27F4 jne `eh vector destructor iterator'+32h (0FE27F2h)

00FE27F6 mov dword ptr [ebp-1Ch],0

00FE27FD mov ecx,dword ptr [ebp+0Ch] //单个对象的大小给ecx

00FE2800 imul ecx,dword ptr [ebp+10h] //对象大小\*对象个数

00FE2804 add ecx,dword ptr [ebp+8] //再加上对象的地址，指向后4字节fd

00FE2807 mov dword ptr [ebp+8],ecx //

00FE280A mov dword ptr [ebp-4],0

00FE2811 mov edx,dword ptr [ebp+10h]

00FE2814 sub edx,1 //元素个数减一

00FE2817 mov dword ptr [ebp+10h],edx //如果符号标志位被置位就跳转

00FE281A js `eh vector destructor iterator'+6Dh (0FE282Dh)

00FE281C mov eax,dword ptr [ebp+8]

00FE281F sub eax,dword ptr [ebp+0Ch] //指向倒数第一个对象地址

00FE2822 mov dword ptr [ebp+8],eax

00FE2825 mov ecx,dword ptr [ebp+8] //ecx现在是倒数第一个对象地址

00FE2828 call dword ptr [ebp+14h] //调用析构函数

00FE282B jmp `eh vector destructor iterator'+51h (0FE2811h) //跳转到元素个数减一操作指令出

00FE282D mov dword ptr [ebp-1Ch],1

00FE2834 mov dword ptr [ebp-4],0FFFFFFFEh

00FE283B call $LN11 (0FE2842h)

00FE2840 jmp $LN14 (0FE285Eh)

# Delete删除一个对象数组，只能调用第一个对象的析构

delete google;

00354992 mov eax,dword ptr [ebp-44h]

00354995 mov dword ptr [ebp-110h],eax

0035499B mov ecx,dword ptr [ebp-110h]

003549A1 mov dword ptr [ebp-11Ch],ecx

003549A7 cmp dword ptr [ebp-11Ch],0

003549AE je main+215h (3549C5h)

003549B0 push 1 //压入元素个数为1

003549B2 mov ecx,dword ptr [ebp-11Ch]

003549B8 call Company::`scalar deleting destructor' (351285h)

//调用单个对象的析构函数，数组为call Company::`vector deleting destructor'

003549BD mov dword ptr [ebp-19Ch],eax

Company::`scalar deleting destructor':

00351970 push ebp

00351971 mov ebp,esp

00351973 sub esp,0CCh

00351979 push ebx

0035197A push esi

0035197B push edi

0035197C push ecx

0035197D lea edi,[ebp-0CCh]

00351983 mov ecx,33h

00351988 mov eax,0CCCCCCCCh

0035198D rep stos dword ptr es:[edi]

0035198F pop ecx

00351990 mov dword ptr [ebp-8],ecx

00351993 mov ecx,dword ptr [this]

00351996 call Company::~Company (351019h)

0035199B mov eax,dword ptr [ebp+8]

0035199E and eax,1

003519A1 je Company::`scalar deleting destructor'+3Fh (3519AFh)

003519A3 mov eax,dword ptr [this]

003519A6 push eax

003519A7 call operator delete (3510AAh)

003519AC add esp,4

003519AF mov eax,dword ptr [this]

003519B2 pop edi

Company::`scalar deleting destructor':

00351970 push ebp

00351971 mov ebp,esp

00351973 sub esp,0CCh

00351979 push ebx

0035197A push esi

0035197B push edi

0035197C push ecx

0035197D lea edi,[ebp-0CCh]

00351983 mov ecx,33h

00351988 mov eax,0CCCCCCCCh

0035198D rep stos dword ptr es:[edi]

0035198F pop ecx

00351990 mov dword ptr [ebp-8],ecx

00351993 mov ecx,dword ptr [this]

00351996 call Company::~Company (351019h) //调用析构函数

0035199B mov eax,dword ptr [ebp+8]

0035199E and eax,1

003519A1 je Company::`scalar deleting destructor'+3Fh (3519AFh)

003519A3 mov eax,dword ptr [this]

003519A6 push eax

003519A7 call operator delete (3510AAh) //在析构函数后调用delete

003519AC add esp,4

003519AF mov eax,dword ptr [this]

003519B2 pop edi

003519B3 pop esi

003519B4 pop ebx

003519B5 add esp,0CCh

003519BB cmp ebp,esp

003519BD call @ILT+425(\_\_RTC\_CheckEsp) (3511AEh)

003519C2 mov esp,ebp

003519C4 pop ebp

# 对单个对象调用delete []:

delete [] twitter;

013C4955 mov eax,dword ptr [ebp-38h]

013C4958 mov dword ptr [ebp-128h],eax

013C495E mov ecx,dword ptr [ebp-128h]

013C4964 mov dword ptr [ebp-134h],ecx

013C496A cmp dword ptr [ebp-134h],0

013C4971 je main+1D8h (13C4988h)

013C4973 push 3

013C4975 mov ecx,dword ptr [ebp-134h]

013C497B call Company::`vector deleting destructor' (13C118Bh)

013C4980 mov dword ptr [ebp-19Ch],eax

013C4986 jmp main+1E2h (13C4992h)

013C4988 mov dword ptr [ebp-19Ch],0

Company::`vector deleting destructor':

013C18D0 push ebp

013C18D1 mov ebp,esp

013C18D3 sub esp,0CCh

013C18D9 push ebx

013C18DA push esi

013C18DB push edi

013C18DC push ecx

013C18DD lea edi,[ebp-0CCh]

013C18E3 mov ecx,33h

013C18E8 mov eax,0CCCCCCCCh

013C18ED rep stos dword ptr es:[edi]

013C18EF pop ecx

013C18F0 mov dword ptr [ebp-8],ecx

013C18F3 mov eax,dword ptr [ebp+8]

013C18F6 and eax,2

013C18F9 je Company::`vector deleting destructor'+61h (13C1931h)

013C18FB push offset Company::~Company (13C1019h)

013C1900 mov eax,dword ptr [this]

013C1903 mov ecx,dword ptr [eax-4]

//将数组个数传给ecx入栈就会出错，因为此时eax-4是fdfdfdfd，而不是元素的个数

013C1906 push ecx

013C1907 push 4

013C1909 mov edx,dword ptr [this]

013C190C push edx

总结：

New一个对象：

1.call operator new (0AD1203h) //new分配内存

2.call Company::Company (0AD11C2h) //调用构造函数

new一个对象数组

1.call operator new[]

2.将析构函数，构造函数，数组元素个数，元素大小，第一个对象地址压入栈中，调用call `eh vector constructor iterator' ，并在这个函数中依次对每个对象调用构造函数

3. call dword ptr [ebp+14h] 调用构造函数

Delete一个对象

1. push 1

2. call Company::`scalar deleting destructor'

3. call Company::~Company //调用析构函数

4. call operator delete //调用delete操作符

Delete []一个对象数组

1. push 3 数组个数入栈

2. call Company::`vector deleting destructor'

3. 将析构函数、数组元素个数、每个元素的大小，对象地址入栈

4. call `eh vector destructor iterator'

5.从倒数第一个对象开始逐个调用析构函数call dword ptr [ebp+14h]

6. call operator delete[]

对于内建的普通数据类型，没有构造和析构函数，所以也就没有将数组元素个数压栈的过程，用delete和delete[]效果是一样的，只是一个用operator delete，一个用operator delete[]。如果用delete删除一个对象数组，只能析构第一个对象，后面的元素都执行不到。如果用delete[]删除一个对象，则会把对象前的0xfdfdfdfd作为数组大小进行析构操作。