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。

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;

mainCRTStartup 的头几步工作是初始化一些预定义的全局变量, \_osplatform, \_winmajor, \_winminor, \_osver, \_winver等与Windows Version 有关的. 然后很重要的一步就是初始化 Heap. 大多的内存都是 Heap 上分配的, 如 new, malloc. 所以必须要先初始化才能干别的. 这个函数是\_heap\_init. 以前的Compiler 有些用自己的 Heap 管理, 但是现在我手上的 Visual Studio .NET 2003, 完全是用的 Windows 系统提供的 Heap management. 所以 \_heap\_init 很简单:  
  
int \_\_cdecl \_heap\_init (int mtflag)  
{  
    // Initialize the "big-block" heap first.  
    if ( (\_crtheap = HeapCreate( mtflag ? 0 : HEAP\_NO\_SERIALIZE,  
    BYTES\_PER\_PAGE, 0 )) == NULL )  
    return 0;  
    ...  
}  
  
系统默认的 new, malloc 等等的分配都是在这个\_crtheap 上进行的. 试试写个简单的程序:   
  
int main()  
{  
    int\* p = new int;  
    return 0;  
}  
  
在int\* p = new int; 这一行设个断点, 调试进去. 可以看见new 是这样的:   
  
void \* operator new( size\_t cb )  
{  
    void \*res = \_nh\_malloc( cb, 1 );  
    ...  
    return res;  
}  
  
下面以Debug version为例, 因为Debug version比较有意思. \_nh\_malloc 只是简单调用\_nh\_malloc\_dbg, 而malloc 也是调用\_nh\_malloc\_dbg来完成内存分配.   
  
\_nh\_malloc\_dbg最终调用\_heap\_alloc\_dbg, 在这里进行真正的分配工作. Debug Version中, 实际分配的是这样一个结构:   
  
typedef struct \_CrtMemBlockHeader  
{  
    struct \_CrtMemBlockHeader \* pBlockHeaderNext;  
    struct \_CrtMemBlockHeader \* pBlockHeaderPrev;  
    char \* szFileName;  
    int nLine;  
    size\_t nDataSize;  
    int nBlockUse;  
    long lRequest;  
    unsigned char gap[nNoMansLandSize];  
    /\* followed by:  
    \* unsigned char data[nDataSize];  
    \* unsigned char anotherGap[nNoMansLandSize];  
    \*/  
} \_CrtMemBlockHeader;  
  
假如你要分配一个大小为100的块, 则实际分配的块结构如下:   
  
\_CrtMemBlockHeader + <You Data> (100 bytes) + gap[nNoMansLandSize]  
  
\_CrtMemBlockHeader 最后有个gap[nNoMansLandSize], 这个nNoMansLandSize目前的值是 4, 所以在你的数据前后各有4个字节的 gap. \_heap\_alloc\_dbg会把 <Your Data> 的所有字节置为 0xCD, 前后的gap置成 0xFD. 如果你在自己的Data里写, 不小心越了界(前面或者后面), 系统在delete的时候通过检查 gap 的数据是否已被破坏，就知道你有没有越界. 当然了, 如果你恰好写的都是0xFD, 那就没法知道了. 试试如下程序：　  
  
int\* p = new int;  
p[1] = 0;  
delete p;  
  
在Debug 下运行时，delete 时系统会报错：　DAMAGE: after normal block (#59) at 0x00375C80.   
  
\_heap\_alloc\_dbg 调用 Windows System Call HeapAlloc 完成分配. HeapAlloc返回的指针, 要先初始化 \_CrtMemBlockHeader. 这个 Header 中有前后两个指针, 事实上所有的内存块连接在一起形成一个双向链表. 在 delete 或者 free 的时候, 链表指针需要调整. 如果没有内存泄漏, 程序结束的时候链表应该为空. 否则说明有内存泄漏. 如下面的程序:   
  
#include <stdio.h>  
#include <stdlib.h>  
#include <crtdbg.h>  
  
int main()  
{  
    int\* p = new int;  
    //delete p;  
  
    void \*px = malloc(100);  
    free(px);  
  
    \_CrtDumpMemoryLeaks();  
    return 0;  
}  
  
\_CrtDumpMemoryLeaks 通过检查分配链表, 来查找是否有泄漏. 在 Debug 下编译并且在 VC 中跟踪运行, 最后在 VC 的 Output 中会有如下输出:   
  
Detected memory leaks!  
Dumping objects ->  
{58} normal block at 0x00375FC0, 4 bytes long.  
Data: < > CD CD CD CD   
Object dump complete.  
  
这里只有分配的序号, 还不能知道到底是哪一行程序产生的泄漏. 但是注意看\_CrtMemBlockHeader, 事实上它还能记录源程序文件名和行号. 在 MFC 里就利用了这个技术. 在 afx.h 里, 有如下声明:  
  
// Memory tracking allocation  
void\* AFX\_CDECL operator new(size\_t nSize, LPCSTR lpszFileName, int nLine);  
#define DEBUG\_NEW new(THIS\_FILE, \_\_LINE\_\_)  
  
同时，使用　MFC　时产生的 cpp 文件开始都有如下定义：  
  
#ifdef \_DEBUG  
#define new DEBUG\_NEW  
  
这个 afx 的 new operator 把 new 时发生的源文件和行号传给 \_malloc\_dbg ．这样在 Dump memory leak 的时候就可以同时知道泄漏的数据最初是在什么地方分配的.   
  
delete 和 free 最终都是用的 \_free\_dbg. \_free\_dbg 首先检查前后的 gap 有没有被破坏, 然后把该块从链表中去掉, 最后把数据块全部置成 0xDD．这样如果你不小心使用了已经被删除的数据时，通常数据已经被破坏而出错．  
  
以上说的都是 Debug Version. 如果是 Release version, 内存分配更简单, 没有任何 overhead, 系统直接调用 HeapAlloc 分配所需的内存块. 同时分配的内存块也不会被初始化为 0xCD.

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>

# Win32 Debug CRT Heap Internals <http://www.nobugs.org/developer/win32/debug_crt_heap.html>

... by Andrew Birkett [(andy@nobugs.org)](mailto:andy@nobugs.org)

When you compile programs with DevStudio in debug mode, all of your calls to malloc() and free() use a special "debug" implementation. Rather than being blazingly fast, the debug heap concerns itself with spotting heap errors. It achieves this by surrounding your memory blocks with guard bytes (aka "no mans land", 0xFD) so that it can detect buffer overruns and underruns. It also initialises newly allocated memory to a fixed value (0xCD) to aid reproducability. Finally, it sets free()d blocks to a known value (0xDD) so that it can detect that people are writing through dangling pointers.

Mnemonics for remembering what each fill-pattern means:

* The newly allocated memory (0x**C**D) is **C**lean memory.
* The free()d memory (0x**D**D) is **D**ead memory.
* The guard bytes (0x**F**D) are like **F**ences around your memory.

The debug CRT heap defers most of the heavy work to the Win32 heap functions HeapAlloc() and HeapFree(). Therefore, you won't see any first-fit or "buddy system" code in the debug CRT. The 4Gb virtual memory space which you process has is sliced up and managed by the Win32 heap inside kernel32.dll.

If you call malloc(8), the debug CRT will request a 48 byte block from HeapAlloc(). It needs the extra 40 bytes to store information about the memory blocks - such as the file/line where malloc() was called, and space for links to the next/prev heap block. In the [table below](http://www.nobugs.org/developer/win32/debug_crt_heap.html#table), all of the debug CRT information is colored in shades of red.

HeapAlloc() itself needs bookkeeping information. The HeapAlloc(40) call will, in turn, reserve a total of 80 bytes from your process's address space. Eight bytes of bookkeeping appear below the requested 40 bytes, and the other 32 bytes appear above it. In the [table below](http://www.nobugs.org/developer/win32/debug_crt_heap.html#table), the Win32 heap bookkeeping is colored grey. The memory which you get back from HeapAlloc() is always initialised to the 4 byte pattern 0xBAADF00D.

*(As an aside, when you request pages from the VM manager via VirtualAlloc, they are initialized to zero, so HeapAlloc is actually doing additional work to re-initialize them to this pattern).*

Once the debug CRT has got it's 40 byte block, it will fill in it's book-keeping information. The first 2 words are links to the previous and next blocks on the CRT heap. The choice of names is confusing, because the "next" pointer actually takes you the block which was allocated *before* this one chronologically, while the "previous" pointer takes you to the one allocated *subsequently*. The reason for the naming is that the linked list starts at the last-allocated block, and progresses back in time as you follow "next" links. The debug CRT heap also internally maintains pointers to the first and last blocks (\_pFirstBlock and \_pLastBlock) to allow heap-checking code to traverse all the blocks.

If the filename and line number of the malloc() call are known, they are stored in the next 2 words. Following that, the next word tells you how many bytes were requested. The next word gives a type field. Usually this is "1" which means a normal block, allocated by malloc/new. It will be "2" for blocks allocate by the CRT for its own internal purposes, and "0" for blocks which have been freed but not released back to the win32 heap (see below for more info). The final word is a simple counter which increases everytime an allocation is made.

Surrounding the 8-byte malloc()'d memory there are areas of "no mans land". These are filled with a known value (0xFD), and when the block is free()d, the CRT will check that they still have the right value. If they've changed, then your program contains an error. Unfortunately, the corruption may have happened a long time ago. You can use Purify or Boundschecker to stop at the corruption point, or if you don't fancy spending any money, you can wait a few days until I write an article telling you how to do it using only a bit of ingenuity!

The eight bytes which were originally requested are initialised with 0xCD. If you see this pattern appearing in your variables, you have forgotten to initialise something.

When you call free() on the 8-byte block, the CRT sets the whole 48-byte block (including its bookkeeping) to 0xDDDDDDDD. This means that it can tell if the block gets subsequently altered (ie. via a dangling pointer) by checking that the pattern is still there.

At this point, two things can happen. Normally, HeapFree() will be called to return the block to the win32 heap. This causes the block to be overwritten with the win32 heap's "freed memory" pattern, which is 0xFEEEFEEE. Note that the debug CRT does not maintain any "free lists" - all of that is done within the black box of HeapFree().

However, you can also tell the debug heap to keep hold of free()d blocks. You do this by passing the \_CRTDBG\_DELAY\_FREE\_MEM\_DF flag to \_CrtSetDbgFlag(). In this case, the debug CRT will keep hold of the block. This is useful if you are trying to track down a dangling pointer error, since memory blocks will not be reused and you should expect them to remain filled with 0xDDDDDDDD unless someone is writing to the free()d block. You can call \_CrtCheckMemory() and it will tell you if any of these values have been tampered with.

## Here's an allocation I prepared earlier ...

I called malloc(8) followed by free() and stepped through the CRT calls to see how the memory was changed. Read the columns from left to right, and you will see what values appear in memory at various stages during malloc() and free(). The call to malloc(8) returned a block at address 0x00321000, and I've included offsets from that address so that you can find out the information for one of your allocations.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Address | Offset | After HeapAlloc() | After malloc() | During free() | After HeapFree() | Comments |
| 0x00320FD8 | -40 | 0x01090009 | 0x01090009 | 0x01090009 | 0x0109005A | Win32 heap info |
| 0x00320FDC | -36 | 0x01090009 | 0x00180700 | 0x01090009 | 0x00180400 | Win32 heap info |
| 0x00320FE0 | -32 | 0xBAADF00D | 0x00320798 | 0xDDDDDDDD | 0x00320448 | Ptr to next CRT heap block (allocated earlier in time) |
| 0x00320FE4 | -28 | 0xBAADF00D | 0x00000000 | 0xDDDDDDDD | 0x00320448 | Ptr to prev CRT heap block (allocated later in time) |
| 0x00320FE8 | -24 | 0xBAADF00D | 0x00000000 | 0xDDDDDDDD | 0xFEEEFEEE | Filename of malloc() call |
| 0x00320FEC | -20 | 0xBAADF00D | 0x00000000 | 0xDDDDDDDD | 0xFEEEFEEE | Line number of malloc() call |
| 0x00320FF0 | -16 | 0xBAADF00D | 0x00000008 | 0xDDDDDDDD | 0xFEEEFEEE | Number of bytes to malloc() |
| 0x00320FF4 | -12 | 0xBAADF00D | 0x00000001 | 0xDDDDDDDD | 0xFEEEFEEE | Type (0=Freed, 1=Normal, 2=CRT use, etc) |
| 0x00320FF8 | -8 | 0xBAADF00D | 0x00000031 | 0xDDDDDDDD | 0xFEEEFEEE | Request #, increases from 0 |
| 0x00320FFC | -4 | 0xBAADF00D | 0xFDFDFDFD | 0xDDDDDDDD | 0xFEEEFEEE | No mans land |
| **0x00321000** | **+0** | **0xBAADF00D** | **0xCDCDCDCD** | **0xDDDDDDDD** | **0xFEEEFEEE** | **The 8 bytes you wanted** |
| **0x00321004** | **+4** | **0xBAADF00D** | **0xCDCDCDCD** | **0xDDDDDDDD** | **0xFEEEFEEE** | **The 8 bytes you wanted** |
| 0x00321008 | +8 | 0xBAADF00D | 0xFDFDFDFD | 0xDDDDDDDD | 0xFEEEFEEE | No mans land |
| 0x0032100C | +12 | 0xBAADF00D | 0xBAADF00D | 0xDDDDDDDD | 0xFEEEFEEE | Win32 heap allocations are rounded up to 16 bytes |
| 0x00321010 | +16 | 0xABABABAB | 0xABABABAB | 0xABABABAB | 0xFEEEFEEE | Win32 heap bookkeeping |
| 0x00321014 | +20 | 0xABABABAB | 0xABABABAB | 0xABABABAB | 0xFEEEFEEE | Win32 heap bookkeeping |
| 0x00321018 | +24 | 0x00000010 | 0x00000010 | 0x00000010 | 0xFEEEFEEE | Win32 heap bookkeeping |
| 0x0032101C | +28 | 0x00000000 | 0x00000000 | 0x00000000 | 0xFEEEFEEE | Win32 heap bookkeeping |
| 0x00321020 | +32 | 0x00090051 | 0x00090051 | 0x00090051 | 0xFEEEFEEE | Win32 heap bookkeeping |
| 0x00321024 | +36 | 0xFEEE0400 | 0xFEEE0400 | 0xFEEE0400 | 0xFEEEFEEE | Win32 heap bookkeeping |
| 0x00321028 | +40 | 0x00320400 | 0x00320400 | 0x00320400 | 0xFEEEFEEE | Win32 heap bookkeeping |
| 0x0032102C | +44 | 0x00320400 | 0x00320400 | 0x00320400 | 0xFEEEFEEE | Win32 heap bookkeeping |

(I've tried to helpfully color-code things. Blue and grey is for Win32 heap stuff, and reds are for the debug crt heap stuff. If you hate the color scheme, the colors are set up using CSS at the top of this .html file - go edit them yourself!)

# Inside CRT: Debug Heap Management

[**http://www.codeguru.com/cpp/w-p/win32/tutorials/article.php/c9535**](http://www.codeguru.com/cpp/w-p/win32/tutorials/article.php/c9535)

When you compile a debug build of your program with Visual Studio and run it in debugger, you can see that the memory allocated or deallocated has funny values, such as 0xCDCDCDCD or 0xDDDDDDDD. This is the result of the work Microsoft has put in to detect memory corruption and leaks in the Win32 platform. In this article, I will explain how memory allocation/deallocation is done via new/delete or malloc/free.

First, I will explain what all these values that you see, like CD, DD, and so forth, mean.

|  |  |  |
| --- | --- | --- |
| **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). |

# If you take a look at DBGHEAP.C, you can see how some of these values are defined:

static unsigned char \_bNoMansLandFill = 0xFD; /\* fill no-man's land with this \*/

static unsigned char \_bDeadLandFill = 0xDD; /\* fill free objects with this \*/

static unsigned char \_bCleanLandFill = 0xCD; /\* fill new objects with this \*/

Before going any further, take a look at the memory management function that I will refer in this article.

|  |  |
| --- | --- |
| **Function** | **Description** |
| malloc | C/C++ function that allocates a block of memory from the heap. The implementation of the C++ operator new is based on malloc. |
| \_malloc\_dbg | Debug version of malloc; only available in the debug versions of the run-time libraries. \_malloc\_dbg is a debug version of the malloc function. When \_DEBUG is not defined, each call to \_malloc\_dbg is reduced to a call to malloc. Both malloc and \_malloc\_dbg allocate a block of memory in the base heap, but \_malloc\_dbg offers several debugging features: buffers on either side of the user portion of the block to test for leaks, a block type parameter to track specific allocation types, and *filename*/*linenumber* information to determine the origin of allocation requests. |
| free | C/C++ function that frees an allocated block. The implementation of C++ operator delete is based on free. |
| \_free\_dbg | Debug version of free; only available in the debug versions of the run-time libraries. The \_free\_dbg function is a debug version of the free function. When \_DEBUG is not defined, each call to \_free\_dbg is reduced to a call to free. Both free and \_free\_dbg free a memory block in the base heap, but \_free\_dbg accommodates two debugging features: the ability to keep freed blocks in the heap's linked list to simulate low memory conditions and a block type parameter to free specific allocation types. |
| LocalAlloc GlobalAlloc | Win32 API to allocate the specified number of bytes from the heap. Windows memory management does not provide a separate local heap and global heap. |
| LocalFree GlobalFree | Win32 API free the specified local memory object and invalidates its handle. |
| HeapAlloc | Win32 API allocates a block of memory from a heap. The allocated memory is not movable. |
| HeapFree | Win32 API frees a memory block allocated from a heap by the HeapAlloc or HeapReAlloc function. |

There are many other functions that deal with memory management. For a complete view please refer to MSDN.

Note: Because this article is about memory management in a debug build, all the references to malloc and free in the following are actually references to their debug versions, \_malloc\_dbg and \_free\_dbg.

Compile the following code and run it in the debugger, walking step by step into it to see how memory is allocated and deallocated.

int main(int argc, char\* argv[])

{

char \*buffer = new char[12];

delete [] buffer;

return 0;

}

Here, 12 bytes are dynamically allocated, but the CRT allocates more than that by wrapping the allocated block with bookkeeping information. For each allocated block, the CRT keeps information in a structure called \_CrtMemBlockHeader, which is declared in DBGINT.H:

#define nNoMansLandSize 4

typedef struct \_CrtMemBlockHeader

{

struct \_CrtMemBlockHeader \* pBlockHeaderNext;

struct \_CrtMemBlockHeader \* pBlockHeaderPrev;

char \* szFileName;

int nLine;

size\_t nDataSize;

int nBlockUse;

long lRequest;

unsigned char gap[nNoMansLandSize];

/\* followed by:

\* unsigned char data[nDataSize];

\* unsigned char anotherGap[nNoMansLandSize];

\*/

} \_CrtMemBlockHeader;

It stores the following information:

|  |  |
| --- | --- |
| **Field** | **Description** |
| pBlockHeaderNext | A pointer to the next block allocated, but next means the previous allocated block because the list is seen as a stack, with the latest allocated block at the top. |
| pBlockHeaderPrev | A pointer to the previous block allocated; this means the block that was allocated after the current block. |
| szFileName | A pointer to the name of the file in which the call to malloc was made, if known. |
| nLine | The line in the source file indicated by szFileName at which the call to malloc was made, if known. |
| nDataSize | Number of bytes requested |
| nBlockUse | 0 - Freed block, but not released back to the Win32 heap 1 - Normal block (allocated with new/malloc) 2 - CRT blocks, allocated by CRT for its own use |
| lRequest | Counter incremented with each allocation |
| gap | A zone of 4 bytes (in the current implementation) filled with 0xFD, fencing the data block, of nDataSize bytes. Another block filled with 0xFD of the same size follows the data. |

Most of the work of heap block allocation and deallocation are made by HeapAlloc() and HeapFree(). When you request 12 bytes to be allocated on the heap, malloc() will call HeapAlloc(), requesting 36 more bytes.

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

malloc requests space for the 12 bytes we need (nSize), plus 32 bytes for the \_CrtMemBlockHeaderstructure and another nNoMansLandSize bytes (4 bytes) to fence the data zone and close the gap.

But, HeapAlloc() will allocate even more bytes: 8 bytes below the requested block (that is, at a lower address) and 32 above it (that is, at a bigger address). It also initializes the requested block to 0xBAADF00D (bad food).

Then, malloc() fills the \_CrtMemBlockHeader block with information and initializes the data block with 0xCD and no mans land with 0xFD.

Here is a table that shows how memory looks after the call to HeapAlloc() and after malloc() returns. For a complete situation, see the last table. (Note: All values are in hex.)

|  |  |  |
| --- | --- | --- |
| **Address** | **after HeapAlloc()** | **after malloc()** |
| 00320FD8 00320FDC 00320FE0 00320FE4 00320FE8 00320FEC 00320FF0 00320FF4 00320FF8 00320FFC 00321000 00321004 00321008 0032100C 00321010 00321014 00321018 0032101C 00321020 00321024 00321028 0032102C | 09 00 09 01 E8 07 18 00 0D F0 AD BA 0D F0 AD BA 0D F0 AD BA 0D F0 AD BA 0D F0 AD BA 0D F0 AD BA 0D F0 AD BA 0D F0 AD BA 0D F0 AD BA 0D F0 AD BA 0D F0 AD BA 0D F0 AD BA AB AB AB AB AB AB AB AB 00 00 00 00 00 00 00 00 79 00 09 00 EE 04 EE 00 40 05 32 00 40 05 32 00 | 09 00 09 01 E8 07 18 00 98 07 32 00 00 00 00 00 00 00 00 00 00 00 00 00 0C 00 00 00 01 00 00 00 2E 00 00 00 FD FD FD FD CD CD CD CD CD CD CD CD CD CD CD CD FD FD FD FD AB AB AB AB AB AB AB AB 00 00 00 00 00 00 00 00 79 00 09 00 EE 04 EE 00 40 05 32 00 40 05 32 00 |

Colors:

* Green: win32 bookkeeping info
* Blue: block size requested by malloc and filled with bad food
* Magenta: \_CrtMemBlockHeader block
* Red: no mans land
* Black: requested data block

In this example, after the call to malloc() returns, buffer will point to memory address 0x00321000.

When you call delete/free, the CRT will set the block it requested from HeapAlloc() to 0xDD, indicating this is a free zone. Normally after this, free() will call HeapFree() to give back the block to the Win32 heap, in which case the block will be overwritten with 0xFEEEEEEE, to indicate Win32 heap free memory.

You can avoid this by using the CRTDBG\_DELAY\_FREE\_MEM\_DF flag to \_CrtSetDbgFlag(). It prevents memory from actually being freed, as for simulating low-memory conditions. When this bit is on, freed blocks are kept in the debug heap's linked list but are marked as \_FREE\_BLOCK. This is useful if you want to detect dangling pointers errors, which can be done by verifying if the freed block is written with 0xDD pattern or something else. Use \_CrtCheckMemory() to verify the heap.s integrity.

The next table shows how the memory looks during the free(), before HeapFree() is called and afterwards.

|  |  |  |
| --- | --- | --- |
| **Address** | **Before HeapFree()** | **After HeapFree()** |
| 00320FD8 00320FDC 00320FE0 00320FE4 00320FE8 00320FEC 00320FF0 00320FF4 00320FF8 00320FFC 00321000 00321004 00321008 0032100C 00321010 00321014 00321018 0032101C 00321020 00321024 00321028 0032102C | 09 00 09 01 5E 07 18 00 DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD DD AB AB AB AB AB AB AB AB 00 00 00 00 00 00 00 00 79 00 09 00 EE 04 EE 00 40 05 32 00 40 05 32 00 | 82 00 09 01 5E 04 18 00 E0 2B 32 00 78 01 32 00 EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE |

Colors:

* Green: win32 bookkeeping info
* Blue: CRT block filled with dead memory
* Gray: memory given back to win32 heap

The two tables above are put in a single, more detailed, table below:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Address (hex)** | **Offset** | **HeapAlloc** | **malloc** | **Free before HeapFree** | **Free after HeapFree** | **Description** |
| 00320FD8 | -40 | 01090009 | 01090009 | 01090009 | 01090082 | Win32 Heap info |
| 00320FDC | -36 | 001807E8 | 001807E8 | 0018075E | 0018045E | Win32 Heap info |
| 00320FE0 | -32 | BAADF00D | 00320798 | DDDDDDDD | 00322BE0 | pBlockHeaderNext |
| 00320FE4 | -28 | BAADF00D | 00000000 | DDDDDDDD | 00320178 | pBlockHeaderPrev |
| 00320FE8 | -24 | BAADF00D | 00000000 | DDDDDDDD | FEEEEEEE | szFileName |
| 00320FEC | -20 | BAADF00D | 00000000 | DDDDDDDD | FEEEEEEE | nLine |
| 00320FF0 | -16 | BAADF00D | 0000000C | DDDDDDDD | FEEEEEEE | nDataSize |
| 00320FF4 | -12 | BAADF00D | 00000001 | DDDDDDDD | FEEEEEEE | nBlockUse |
| 00320FF8 | -8 | BAADF00D | 0000002E | DDDDDDDD | FEEEEEEE | lRequest |
| 00320FFC | -4 | BAADF00D | FDFDFDFD | DDDDDDDD | FEEEEEEE | gap (no mans land) |
| 00321000 | 0 | BAADF00D | CDCDCDCD | DDDDDDDD | FEEEEEEE | Data requested |
| 00321004 | +4 | BAADF00D | CDCDCDCD | DDDDDDDD | FEEEEEEE | Data requested |
| 00321008 | +8 | BAADF00D | CDCDCDCD | DDDDDDDD | FEEEEEEE | Data requested |
| 0032100C | +12 | BAADF00D | FDFDFDFD | DDDDDDDD | FEEEEEEE | No mans land |
| 00321010 | +16 | ABABABAB | ABABABAB | ABABABAB | FEEEEEEE | Win32 Heap info |
| 00321014 | +20 | ABABABAB | ABABABAB | ABABABAB | FEEEEEEE | Win32 Heap info |
| 00321018 | +24 | 00000000 | 00000000 | 00000000 | FEEEEEEE | Win32 Heap info |
| 0032101C | +28 | 00000000 | 00000000 | 00000000 | FEEEEEEE | Win32 Heap info |
| 00321020 | +32 | 00090079 | 00090079 | 00090079 | FEEEEEEE | Win32 Heap info |
| 00321024 | +36 | 00EE04EE | 00EE04EE | 00EE04EE | FEEEEEEE | Win32 Heap info |
| 00321028 | +40 | 00320540 | 00320540 | 00320540 | FEEEEEEE | Win32 Heap info |
| 0032102C | +44 | 00320540 | 00320540 | 00320540 | FEEEEEEE | Win32 Heap info |

## About the Author

Marius Bancila is a Microsoft MVP for VC++. He works as a software developer for a Norwegian-based company. He is mainly focused on building desktop applications with MFC and VC#. He keeps a blog at www.mariusbancila.ro/blog, focused on Windows programming. He is the co-founder of codexpert.ro, a community for Romanian C++/VC++ programmers.

Visual Studio 提供了一系列的 CRT 调试 API ， CRT 提供了一个调试内存分配堆，可以跟踪和管理内存在什么地方分配，当你在这个堆上分配内存的时候，每一次内存分配调用例如malloc 或者 new ， CRT 都会额外分配大约 36 个字节用来保存例如这个内存块分配的文件名、行号、内存块的大小等信息，最后 CRT 将这些内存块使用一个双链表链接起来。每一次内存释放的时候， free 或者 delete 函数就从这个内存块链表里面将要释放的内存块删除，因此在需要检查内存泄漏的时候，只要遍历这个双链表依次打印出这些内存块就可以发现所有未释放的内存了。下面是 CRT 内存块的原始声明：

|  |
| --- |
| typedef struct \_CrtMemBlockHeader  {      // Pointer to the block allocated just before this one:      struct \_CrtMemBlockHeader \*pBlockHeaderNext;      // Pointer to the block allocated just after this one:      struct \_CrtMemBlockHeader \*pBlockHeaderPrev;      char \*szFileName;    // File name      int nLine;           // Line number      size\_t nDataSize;    // Size of user block      int nBlockUse;       // Type of block      long lRequest;       // Allocation number      // Buffer just before (lower than) the user's memory:      unsigned char gap[nNoMansLandSize];  } \_CrtMemBlockHeader; |

下面的代码演示了如何使用 CRT 提供的调试 API 来修改刚才的源文件检测未释放的内存空间（注意红色添加的部分）：

|  |
| --- |
| // 未释放内存空间 .cpp : Defines the entry point for the console application.  //    #include "stdafx.h"  #include <windows.h>  #include <string>  #include <iostream>  // 使用 CRT 调试 API  **#include <crtdbg.h>**    using namespace std;    // 将所有的内存分配函数 new 替换成 CRT 提供的调试 new  **#ifdef \_DEBUG**  **#   define DEBUG\_CLIENTBLOCK new (\_CLIENT\_BLOCK, \_\_FILE\_\_, \_\_LINE\_\_)**  **#else**  **#   define DEBUG\_CLIENTBLOCK**  **#endif**    **#ifdef \_DEBUG**  **#    define new DEBUG\_CLIENTBLOCK**  **#endif**    class CTestClass  {  public :      CTestClass(LPWSTR szName)      {          m\_lpName = new wstring(szName);      }        ~CTestClass()      {      }        void PrintName()      {          wcout << \*m\_lpName << endl;      }    private :      wstring \*m\_lpName;  };    HRESULT CreateTestClass(LPWSTR szName, CTestClass \*\*ppObject)  {      \*ppObject = new CTestClass(szName);      if ( (\*ppObject) == NULL )          return E\_FAIL;      else          return S\_OK;  }    int \_tmain(int argc, \_TCHAR\* argv[])  {  // 设置 CRT 调试 API 的报表输出模式，将所有的错误、警告还有断言都输出到控制台  **\_CrtSetReportMode( \_CRT\_WARN, \_CRTDBG\_MODE\_FILE );**  **\_CrtSetReportFile( \_CRT\_WARN, \_CRTDBG\_FILE\_STDOUT );**  **\_CrtSetReportMode( \_CRT\_ERROR, \_CRTDBG\_MODE\_FILE );**  **\_CrtSetReportFile( \_CRT\_ERROR, \_CRTDBG\_FILE\_STDOUT );**  **\_CrtSetReportMode( \_CRT\_ASSERT, \_CRTDBG\_MODE\_FILE );**  **\_CrtSetReportFile( \_CRT\_ASSERT, \_CRTDBG\_FILE\_STDOUT );**        CTestClass \*pObject = NULL;      HRESULT hr = CreateTestClass(L"This is a Test" , &pObject);      if ( hr != S\_OK )      {          return -1;      }      else      {          pObject->PrintName();    // 检查未释放的内存  **\_CrtDumpMemoryLeaks();**          return 0;      }  } |

这里面是我们的输出结果：



从上面的输出我们可以看出，在“未释放内存空间 .cpp ”的第 27 行和第 45 行所分配的内存没有被适当释放，查看源代码可以发现就是 CTestClass 和 CTestClass 的成员变量m\_Name 没有被释放。

2.\_CrtDumpMemLeaks()

msdn说明

The **\_CrtDumpMemoryLeaks** function determines whether a memory leak has occurred since the start of program execution. When a leak is found, the debug header information for all of the objects in the heap is dumped in a user-readable form. When [\_DEBUG](http://blog.csdn.net/ly402609921/article/details/_crt__debug.htm) is not defined, calls to **\_CrtDumpMemoryLeaks** are removed during preprocessing.

**\_CrtDumpMemoryLeaks** is frequently called at the end of program execution to verify that all memory allocated by the application has been freed. The function can be called automatically at program termination by turning on the **\_CRTDBG\_LEAK\_CHECK\_DF** bit field of the [\_crtDbgFlag](http://blog.csdn.net/ly402609921/article/details/_crt__crtdbgflag.htm) flag using the [\_CrtSetDbgFlag](http://blog.csdn.net/ly402609921/article/details/_crt__crtsetdbgflag.htm)function.

**\_CrtDumpMemoryLeaks** calls [\_CrtMemCheckpoint](http://blog.csdn.net/ly402609921/article/details/_crt__crtmemcheckpoint.htm) to obtain the current state of the heap and then scans the state for blocks that have not been freed. When an unfreed block is encountered,**\_CrtDumpMemoryLeaks** calls [\_CrtMemDumpAllObjectsSince](http://blog.csdn.net/ly402609921/article/details/_crt__crtmemdumpallobjectssince.htm) to dump information for all of the objects allocated in the heap from the start of program execution.

By default, internal C run-time blocks (**\_CRT\_BLOCK** ) are not included in memory dump operations. The[\_CrtSetDbgFlag](http://blog.csdn.net/ly402609921/article/details/_crt__crtsetdbgflag.htm) function can be used to turn on the **\_CRTDBG\_CHECK\_CRT\_DF** bit of **\_crtDbgFlag** to include these blocks in the leak detection process.

For more information about heap state functions and the **\_CrtMemState** structure, see the [Heap State Reporting Functions](http://blog.csdn.net/ly402609921/article/details/_core_heap_state_reporting_functions.htm) . For information about how memory blocks are allocated, initialized, and managed in the debug version of the base heap, see [Memory Management and the Debug Heap](http://blog.csdn.net/ly402609921/article/details/_core_memory_management_and_the_debug_heap.htm) .

应用：

#include "stdafx.h"  
#include <assert.h>  
#ifdef \_DEBUG  
#define DEBUG\_CLIENTBLOCK   new( \_CLIENT\_BLOCK, \_\_FILE\_\_, \_\_LINE\_\_)  
#else  
#define DEBUG\_CLIENTBLOCK  
#endif  
#define \_CRTDBG\_MAP\_ALLOC  
#include <crtdbg.h>  
#ifdef \_DEBUG  
#define new DEBUG\_CLIENTBLOCK  
#endif //此部分用于使\_CrtDumpMemoryLeaks输出内存泄漏文件名和行号信息默认不会输出相关信息  
void Exit  
{  
    int i = \_CrtDumpMemoryLeaks;  
    assert( i == 0);  
}  
int \_tmain(int argc, \_TCHAR\* argv[])  
{  
    atexit(Exit);  
    int\* p = new int;  
    return 0;  
}

不含红色部分输出：

Detected memory leaks!  
Dumping objects ->  
{112} normal block at 0x003AA770, 4 bytes long.  
 Data: <    > 00 00 00 00   
Object dump complete.

含红色部分输出：

Detected memory leaks!  
Dumping objects ->  
d:/code/consoletest/consoletest.cpp(21) : {112} client block at 0x003A38B0, subtype 0, 4 bytes long.  
 Data: <    > 00 00 00 00   
Object dump complete.

-------------------------------------------------------------------------------------------

1.\_CrtDumpMemoryLeaks

确定自程序开始执行以来是否发生过内存泄漏，如果发生过，则转储所有已分配对象。如果已使用 \_CrtSetDumpClient 安装了挂钩函数，那么，\_CrtDumpMemoryLeaks每次转储 \_CLIENT\_BLOCK 块时，都会调用应用程序所提供的挂钩函数。

CrtDumpMemoryLeaks()就是显示当前的内存泄漏。 注意是“当前”，也就是说当它执行时，所有未销毁的对象均会报内存泄漏。因此尽量让这条语句在程序的最后执行。它所反映的是检测到泄漏的地方。  
一般用在MFC中比较准确，在InitInstance里面调用\_CrtDumpMemoryLeaks

2.信息输出  
Detected memory leaks!  
Dumping objects ->  
{52} normal block at 0x006D2498, 512 bytes long.  
?Data: <??????????????? > 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   
{51} normal block at 0x006D2440, 24 bytes long.  
?Data: < 4????????????? > 10 34 14 00 FF FF FF FF 00 00 00 00 00 00 00 00   
Object dump complete.  
3.\_CrtSetBreakAlloc  
知道某个错误分配块的分配请求编号后，可以将该编号传递给 \_CrtSetBreakAlloc 以创建一个断点  
\_CrtSetBreakAlloc(51);这样可以快速在{51}次内存泄漏处设上断点。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

# 最快速度找到内存泄漏

许式伟  
2006年11月某日

内存管理是C++程序员的痛。我的《[内存管理变革](http://blog.csdn.net/xushiweizh/archive/2006/11/16/1388982.aspx)》系列就是试图讨论更为有效的内存管理方式，以杜绝（或减少）内存泄漏，减轻C++程序员的负担。由于工作忙的缘故，这个系列目前未完，暂停。

这篇短文我想换个方式，讨论一下如何以最快的速度找到内存泄漏。

# 确认是否存在内存泄漏

我们知道，MFC程序如果检测到存在内存泄漏，退出程序的时候会在调试窗口提醒内存泄漏。例如：

class  CMyApp :  public  CWinApp  
{  
public :  
   BOOL InitApplication()  
   {  
        int \*  leak  =   new   int [ 10 ];  
        return  TRUE;  
   }  
};

产生的内存泄漏报告大体如下：

Detected memory leaks !   
Dumping objects  ->   
c:/work/test.cpp( 186 ) : { 52 } normal block at  0x003C4410 ,  40  bytes  long .  
 Data:  <                  >  CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD   
Object dump complete.

这挺好。问题是，如果我们不喜欢MFC，那么难道就没有办法？或者自己做？

呵呵，这不需要。其实，MFC也没有自己做。内存泄漏检测的工作是VC++的C运行库做的。也就是说，只要你是VC++程序员，都可以很方便地检测内存泄漏。我们还是给个样例：

#include  < crtdbg.h >   
  
inline  void  EnableMemLeakCheck()  
{  
   \_CrtSetDbgFlag(\_CrtSetDbgFlag(\_CRTDBG\_REPORT\_FLAG)  |  \_CRTDBG\_LEAK\_CHECK\_DF);  
}  
  
void  main()  
{  
   EnableMemLeakCheck();  
    int \*  leak  =   new   int [ 10 ];  
}

 运行（提醒：不要按Ctrl+F5，按F5），你将发现，产生的内存泄漏报告与MFC类似，但有细节不同，如下：

Detected memory leaks !   
Dumping objects  ->   
{ 52 } normal block at  0x003C4410 ,  40  bytes  long .  
 Data:  <                  >  CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD   
Object dump complete.

为什么呢？看下面。

# 定位内存泄漏由于哪一句话引起的

你已经发现程序存在内存泄漏。现在的问题是，我们要找泄漏的根源。

一般我们首先确定内存泄漏是由于哪一句引起。在MFC中，这一点很容易。你双击内存泄漏报告的文字，或者在Debug窗口中按F4，IDE就帮你定位到申请该内存块的地方。对于上例，也就是这一句：

   int\* leak = new int[10];

这多多少少对你分析内存泄漏有点帮助。特别地，如果这个new仅对应一条delete（或者你把delete漏写），这将很快可以确认问题的症结。

我们前面已经看到，不使用MFC的时候，生成的内存泄漏报告与MFC不同，而且你立刻发现按F4不灵。那么难道MFC做了什么手脚？

其实不是，我们来模拟下MFC做的事情。看下例：

inline  void  EnableMemLeakCheck()  
{  
   \_CrtSetDbgFlag(\_CrtSetDbgFlag(\_CRTDBG\_REPORT\_FLAG)  |  \_CRTDBG\_LEAK\_CHECK\_DF);  
}  
  
#ifdef \_DEBUG  
#define  new   new(\_NORMAL\_BLOCK, \_\_FILE\_\_, \_\_LINE\_\_)   
#endif   
  
void  main()  
{  
   EnableMemLeakCheck();  
    int \*  leak  =   new   int [ 10 ];  
}

再运行这个样例，你惊喜地发现，现在内存泄漏报告和MFC没有任何分别了。

# 快速找到内存泄漏

单确定了内存泄漏发生在哪一行，有时候并不足够。特别是同一个new对应有多处释放的情形。在实际的工程中，以下两种情况很典型：

* 创建对象的地方是一个类工厂（ClassFactory）模式。很多甚至全部类实例由同一个new创建。对于此，定位到了new出对象的所在行基本没有多大帮助。
* COM对象。我们知道COM对象采用Reference Count维护生命周期。也就是说，对象new的地方只有一个，但是Release的地方很多，你要一个个排除。

那么，有什么好办法，可以迅速定位内存泄漏？

答：有。

在内存泄漏情况复杂的时候，你可以用以下方法定位内存泄漏。这是我个人认为通用的内存泄漏追踪方法中最有效的手段。

我们再回头看看crtdbg生成的内存泄漏报告：

Detected memory leaks !   
Dumping objects  ->   
c:/work/test.cpp( 186 ) : { 52 } normal block at  0x003C4410 ,  40  bytes  long .  
 Data:  <                  >  CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD   
Object dump complete.

除了产生该内存泄漏的内存分配语句所在的文件名、行号为，我们注意到有一个比较陌生的信息：{52}。这个整数值代表了什么意思呢？

其实，它代表了第几次内存分配操作。象这个例子，{52}代表了第52次内存分配操作发生了泄漏。你可能要说，我只new过一次，怎么会是第52次？这很容易理解，其他的内存申请操作在C的初始化过程调用的呗。:)

有没有可能，我们让程序运行到第52次内存分配操作的时候，自动停下来，进入调试状态？所幸，crtdbg确实提供了这样的函数：即 long **\_CrtSetBreakAlloc**(long nAllocID)。我们加上它：

inline  void  EnableMemLeakCheck()  
{  
   \_CrtSetDbgFlag(\_CrtSetDbgFlag(\_CRTDBG\_REPORT\_FLAG)  |  \_CRTDBG\_LEAK\_CHECK\_DF);  
}  
  
#ifdef \_DEBUG  
#define  new   new(\_NORMAL\_BLOCK, \_\_FILE\_\_, \_\_LINE\_\_)   
#endif   
  
void  main()  
{  
   EnableMemLeakCheck();  
   **\_CrtSetBreakAlloc** ( 52 );  
    int \*  leak  =   new   int [ 10 ];  
}

你发现，程序运行到 int \*  leak  =   new   int [ 10 ]; 一句时，自动停下来进入调试状态。细细体会一下，你可以发现，这种方式你获得的信息远比在程序退出时获得文件名及行号有价值得多。因为报告泄漏文件名及行号，你获得的只是静态的信息，然而\_CrtSetBreakAlloc 则是把整个现场恢复，你可以通过对函数调用栈分析（我发现很多人不习惯看函数调用栈，如果你属于这种情况，我强烈推荐你去补上这一课，因为它太重要了）以及其他在线调试技巧，来分析产生内存泄漏的原因。通常情况下，这种分析方法可以在5分钟内找到肇事者。

当然，**\_CrtSetBreakAlloc** 要求你的程序执行过程是可还原的（多次执行过程的内存分配顺序不会发生变化）。这个假设在多数情况下成立。不过，在多线程的情况下，这一点有时难以保证。

# 附加说明：

对“内存管理”相关的技术感兴趣？这里可以看到我的所有[关于内存管理的文章](http://blog.csdn.net/xushiweizh/category/265099.aspx) 。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**VC使用CRT调试功能来检测内存泄漏**

作者：JerryZ

C/C++ 编程语言的最强大功能之一便是其动态分配和释放内存，但是中国有句古话：“最大的长处也可能成为最大的弱点”，那么 C/C++ 应用程序正好印证了这句话。在 C/C++ 应用程序开发过程中，动态分配的内存处理不当是最常见的问题。其中，最难捉摸也最难检测的错误之一就是内存泄漏，即未能正确释放以前分配的内存的错误。偶 尔发生的少量内存泄漏可能不会引起我们的注意，但泄漏大量内存的程序或泄漏日益增多的程序可能会表现出各种 各样的征兆：从性能不良（并且逐渐降低）到内存完全耗尽。更糟的是，泄漏的程序可能会用掉太多内存，导致另外一个程序垮掉，而使用户无从查找问题的真正根 源。此外，即使无害的内存泄漏也可能殃及池鱼。

　　幸运的是，Visual Studio 调试器和 C 运行时 (CRT) 库为我们提供了检测和识别内存泄漏的有效方法。下面请和我一起分享收获——如何使用 CRT 调试功能来检测内存泄漏？  
  
**一、如何启用内存泄漏检测机制**  
　　VC++ IDE 的默认状态是没有启用内存泄漏检测机制的，也就是说即使某段代码有内存泄漏，调试会话的 Output 窗口的 Debug 页不会输出有关内存泄漏信息。你必须设定两个最基本的机关来启用内存泄漏检测机制。  
  
　　一是使用调试堆函数：

#define \_CRTDBG\_MAP\_ALLOC   
#include<stdlib.h>   
  
#include<crtdbg.h>

　　注意：#include 语句的顺序。如果更改此顺序，所使用的函数可能无法正确工作。  
  
　　通过包含 crtdbg.h 头文件，可以将 malloc 和 free 函数映射到其“调试”版本 \_malloc\_dbg 和 \_free\_dbg，这些函数会跟踪内存分配和释放。此映射只在调试（Debug）版本（也就是要定义 \_DEBUG）中有效。发行版本（Release）使用普通的 malloc 和 free 函数。#define 语句将 CRT 堆函数的基础版本映射到对应的“调试”版本。该语句不是必须的，但如果没有该语句，那么有关内存泄漏的信息会不全。  
  
　　二是在需要检测内存泄漏的地方添加下面这条语句来输出内存泄漏信息：  
  
\_CrtDumpMemoryLeaks();

　　当在调试器下运行程序时，\_CrtDumpMemoryLeaks 将在 Output 窗口的 Debug 页中显示内存泄漏信息。比如： Detected memory leaks!

Dumping objects ->  
  
C:/Temp/memleak/memleak.cpp(15) : {45} normal block at 0x00441BA0, 2 bytes long.  
  
Data: <AB> 41 42  
  
  
  
c:/program files/microsoft visual studio/vc98/include/crtdbg.h(552) : {44} normal   
block at 0x00441BD0, 33 bytes long.  
  
Data: < C > 00 43 00 CD CD CD CD CD CD CD CD CD CD CD CD CD  
  
  
  
c:/program files/microsoft visual studio/vc98/include/crtdbg.h(552) : {43} normal   
block at 0x00441C20, 40 bytes long.  
  
Data: < C > 08 02 43 00 16 00 00 00 00 00 00 00 00 00 00 00  
  
  
  
Object dump complete.

　　如果不使用 #define \_CRTDBG\_MAP\_ALLOC 语句，内存泄漏的输出是这样的：

Detected memory leaks!  
  
Dumping objects ->  
  
{45} normal block at 0x00441BA0, 2 bytes long.  
Data: <AB> 41 42   
  
{44} normal block at 0x00441BD0, 33 bytes long.  
Data: < C > 00 43 00 CD CD CD CD CD CD CD CD CD CD CD CD CD   
  
{43} normal block at 0x00441C20, 40 bytes long.  
Data: < C > C0 01 43 00 16 00 00 00 00 00 00 00 00 00 00 00   
  
Object dump complete.

　　根据这段输出信息，你无法知道在哪个源程序文件里发生了内存泄漏。下面我们来研究一下输出信息的格式。第一行和第二行没有什么可说的，从第三行开始：

xx}：花括弧内的数字是内存分配序号，本文例子中是 {45}，{44}，{43}；  
block：内存块的类型，常用的有三种：normal（普通）、client（客户端）或 CRT（运行时）；本文例子中是：normal block；   
用十六进制格式表示的内存位置，如：at 0x00441BA0 等；  
以字节为单位表示的内存块的大小，如：32 bytes long；   
前 16 字节的内容（也是用十六进制格式表示），如：Data: 41 42 等；

　　仔细观察不难发现，如果定义了 \_CRTDBG\_MAP\_ALLOC ，那么在内存分配序号前面还会显示在其中分配泄漏内存的文件名，以及文件名后括号中的数字表示发生泄漏的代码行号，比如：

C:/Temp/memleak/memleak.cpp(15)

　　双击 Output 窗口中此文件名所在的输出行，便可跳到源程序文件分配该内存的代码行（也可以选中该行，然后按 F4，效果一样） ，这样一来我们就很容易定位内存泄漏是在哪里发生的了，因此，\_CRTDBG\_MAP\_ALLOC 的作用显而易见。  
  
使用 **\_CrtSetDbgFlag**   
　 　如果程序只有一个出口，那么调用 \_CrtDumpMemoryLeaks 的位置是很容易选择的。但是，如果程序可能会在多个地方退出该怎么办呢？在每一个可能的出口处调用 \_CrtDumpMemoryLeaks 肯定是不可取的，那么这时可以在程序开始处包含下面的调用：**\_CrtSetDbgFlag** ( \_CRTDBG\_ALLOC\_MEM\_DF | \_CRTDBG\_LEAK\_CHECK\_DF );这条语句无论程序在什么地方退出都会自动调用 \_CrtDumpMemoryLeaks。注意：这里必须同时设置两个位域标志：\_CRTDBG\_ALLOC\_MEM\_DF 和 \_CRTDBG\_LEAK\_CHECK\_DF。  
  
设置 CRT 报告模式  
　　默认情况下，\_CrtDumpMemoryLeaks 将内存泄漏信息 dump 到 Output 窗口的 Debug 页， 如果你想将这个输出定向到别的地方，可以使用 \_CrtSetReportMode 进行重置。如果你使用某个库，它可能将输出定向到另一位置。此时，只要使用以下语句将输出位置设回 Output 窗口即可：

\_CrtSetReportMode( \_CRT\_ERROR, \_CRTDBG\_MODE\_DEBUG );

　　有关使用 \_CrtSetReportMode 的详细信息，请参考 MSDN 库关于 \_CrtSetReportMode 的描述。  
  
**二、解释内存块类型**   
  
　　前面已经说过，内存泄漏报告中把每一块泄漏的内存分为 normal（普通块）、client（客户端块）和 CRT 块。事实上，需要留心和注意的也就是 normal 和 client，即普通块和客户端块。  
　　1.normal block（普通块）：这是由你的程序分配的内存。  
　　2.client block（客户块）：这是一种特殊类型的内存块，专门用于 MFC 程序中需要析构函数的对象。MFC new 操作符视具体情况既可以为所创建的对象建立普通块，也可以为之建立客户块。  
　 　3.CRT block（CRT 块）：是由 C RunTime Library 供自己使用而分配的内存块。由 CRT 库自己来管理这些内存的分配与释放，我们一般不会在内存泄漏报告中发现 CRT 内存泄漏，除非程序发生了严重的错误（例如 CRT 库崩溃）。

　　除了上述的类型外，还有下面这两种类型的内存块，它们不会出现在内存泄漏报告中：  
　　1.free block（空闲块）：已经被释放(free)的内存块。   
　　2.Ignore block（忽略块）：这是程序员显式声明过不要在内存泄漏报告中出现的内存块。

**三、如何在内存分配序号处设置断点**　　在内存泄漏报告中，的文件名和行号可告诉分配泄漏的内存 的代码位置，但仅仅依赖这些信息来了解完整的泄漏原因是不够的。因为一个程序在运行时，一段分配内存的代码可能会被调用很多次，只要有一次调用后没有释放 内存就会导致内存泄漏。为了确定是哪些内存没有被释放，不仅要知道泄漏的内存是在哪里分配的，还要知道泄漏产生的条件。这时内存分配序号就显得特别有用 ——这个序号就是文件名和行号之后的花括弧里的那个数字。

　　例如，在本文例子代码的输出信息中，“45”是内存分配序号，意思是泄漏的内存是你程序中分配的第四十五个内存块：

Detected memory leaks!  
  
Dumping objects ->  
  
C:/Temp/memleak/memleak.cpp(15) : {45} normal block at 0x00441BA0, 2 bytes long.  
  
Data: <AB> 41 42   
  
......  
  
Object dump complete.

　　CRT 库对程序运行期间分配的所有内存块进行计数，包括由 CRT 库自己分配的内存和其它库（如 MFC）分配的内存。因此，分配序号为 N 的对象即为程序中分配的第 N 个对象，但不一定是代码分配的第 N 个对象。（大多数情况下并非如此。）这样的话，你便可以利用分配序号在分配内存的位置设置一个断点。方法是在程序起始附近设置一个位置断点。当程序在该点 中断时，可以从 QuickWatch（快速监视）对话框或 Watch（监视）窗口设置一个内存分配断点：  
  
　　例如，在 Watch 窗口中，在 Name 栏键入下面的表达式：

\_crtBreakAlloc

　　如果要使用 CRT 库的多线程 DLL 版本（/MD 选项），那么必须包含上下文操作符，像这样：

{,,msvcrtd.dll}\_crtBreakAlloc

　　现在按下回车键，调试器将计算该值并把结果放入 Value 栏。如果没有在内存分配点设置任何断点，该值将为 –1。

　　用你想要在其位置中断的内存分配的分配序号替换 Value 栏中的值。例如输入 45。这样就会在分配序号为 45 的地方中断。

　　在所感兴趣的内存分配处设置断点后，可以继续调试。这时，运行程序时一定要小心，要保证内存块分配的顺序不会改变。当程序在指定的内存分配处中 断时，可以查看 Call Stack（调用堆栈）窗口和其它调试器信息以确定分配内存时的情况。如果必要，可以从该点继续执行程序，以查看对象发生了什么情况，或许可以确定未正确 释放对象的原因。

　　尽管通常在调试器中设置内存分配断点更方便，但如果愿意，也可在代码中设置这些断点。为了在代码中设置一个内存分配断点，可以增加这样一行（对于第四十五个内存分配）：

\_crtBreakAlloc = 45;

　　你还可以使用有相同效果的 \_CrtSetBreakAlloc 函数：

\_CrtSetBreakAlloc(45);

**四、如何比较内存状态**   
  
　　定位内存泄漏的另一个方法就是在关键点获取应用程序内存状态的快照。CRT 库提供了一个结构类型 \_CrtMemState。你可以用它来存储内存状态的快照：

\_CrtMemState s1, s2, s3;

　　若要获取给定点的内存状态快照，可以向 \_CrtMemCheckpoint 函数传递一个 \_CrtMemState 结构。该函数用当前内存状态的快照填充此结构：

\_CrtMemCheckpoint( &s1 );

　　通过向 \_CrtMemDumpStatistics 函数传递 \_CrtMemState 结构，可以在任意地方 dump 该结构的内容：

\_CrtMemDumpStatistics( &s1 );

　　该函数输出如下格式的 dump 内存分配信息：

0 bytes in 0 Free Blocks.  
75 bytes in 3 Normal Blocks.  
5037 bytes in 41 CRT Blocks.  
0 bytes in 0 Ignore Blocks.  
0 bytes in 0 Client Blocks.  
Largest number used: 5308 bytes.  
Total allocations: 7559 bytes.

　　若要确定某段代码中是否发生了内存泄漏，可以通过获取该段代码之前和之后的内存状态快照，然后使用 \_CrtMemDifference 比较这两个状态：

\_CrtMemCheckpoint( &s1 );// 获取第一个内存状态快照  
  
// 在这里进行内存分配  
  
\_CrtMemCheckpoint( &s2 );// 获取第二个内存状态快照  
  
// 比较两个内存快照的差异  
if ( \_CrtMemDifference( &s3, &s1, &s2) )  
\_CrtMemDumpStatistics( &s3 );// dump 差异结果

　　顾名思义，\_CrtMemDifference 比较两个内存状态（前两个参数），生成这两个状态之间差异的结果（第三个参数）。在程序的开始和结尾放置 \_CrtMemCheckpoint 调用，并使用 \_CrtMemDifference 比较结果，是检查内存泄漏的另一种方法。如果检测到泄漏，则可以使用 \_CrtMemCheckpoint 调用通过二进制搜索技术来分割程序和定位泄漏。

**五、结论**  
　　尽管 VC ++ 具有一套专门调试 MFC 应用程序的机制，但本文上述讨论的内存分配很简单，没有涉及到 MFC 对象，所以这些内容同样也适用于 MFC 程序。在 MSDN 库中可以找到很多有关 VC++ 调试方面的资料，如果你能善用 MSDN 库，相信用不了多少时间你就有可能成为调试高手。