CRT Debug Heap Details

Visual Studio 2015

The new home for Visual Studio documentation is Visual Studio 2017 Documentation on docs.microsoft.com.

The latest version of this topic can be found at CRT Debug Heap Details.

This topic provides a detailed look at the CRT debug heap.

Contents

Find buffer overruns with debug heap

Two of the most common and intractable problems that programmers encounter are overwriting the end of an allocated buffer and memory leaks (failing to free allocations after they are no longer needed). The debug heap provides powerful tools to solve memory allocation problems of this kind.

The Debug versions of the heap functions call the standard or base versions used in Release builds. When you request a memory block, the debug heap manager allocates from the base heap a slightly larger block of memory than requested and returns a pointer to your portion of that block. For example, suppose your application contains the call: `malloc( 10 )`. In a Release build, malloc would call the base heap allocation routine requesting an allocation of 10 bytes. In a Debug build, however, `malloc` would call _malloc_dbg, which would then call the base heap allocation routine requesting an allocation of 10 bytes plus approximately 36 bytes of additional memory. All the resulting memory blocks in the debug heap are connected in a single linked list, ordered according to when they were allocated.

The additional memory allocated by the debug heap routines is used for bookkeeping information, for pointers that link debug memory blocks together, and for small buffers on either side of your data to catch overwrites of the allocated region.

Currently, the block header structure used to store the debug heap's bookkeeping information is declared as follows in the DBGINT.H header file:

```
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;

/* In an actual memory block in the debug heap,
 * this structure is followed by:
 *   unsigned char data[nDataSize];
 *   unsigned char anotherGap[nNoMansLandSize];
 */
```

The `NoMansLand` buffers on either side of the user data area of the block are currently 4 bytes in size, and are filled with a known byte value used by the debug heap routines to verify that the limits of the user's memory block have not been overwritten. The debug heap also fills new memory blocks with a known value. If you elect to keep freed blocks in the heap's linked list as explained below, these freed blocks are also filled with a known value. Currently, the actual byte values used are as follows:

NoMansLand (0xFD)
The "NoMansLand" buffers on either side of the memory used by an application are currently filled with 0xFD.

Freed blocks (0xDD)
The freed blocks kept unused in the debug heap's linked list when the `_CRTDBG_DELAY_FREE_MEM_DF` flag is set are currently filled with 0xDD.

New objects (0xCD)
New objects are filled with 0xCD when they are allocated.

Contents

Types of blocks on the debug heap

Every memory block in the debug heap is assigned to one of five allocation types. These types are tracked and reported differently for purposes of leak detection and state reporting. You can specify a block's type by allocating it using a direct call to one of the debug heap allocation functions such as _malloc_dbg. The five types of memory blocks in the debug heap (set in the **nBlockUse** member of the **_CrtMemBlockHeader** structure) are as follows:

_NORMAL_BLOCK
A call to malloc or calloc creates a Normal block. If you intend to use Normal blocks only, and have no need for Client blocks, you may want to define _CRTDBG_MAP_ALLOC, which causes all heap allocation calls to be mapped to their debug equivalents in Debug builds. This will allow file name and line number information about each allocation call to be stored in the corresponding block header.

`_CRT_BLOCK`
The memory blocks allocated internally by many run-time library functions are marked as CRT blocks so they can be handled separately. As a result, leak detection and other operations need not be affected by them. An allocation must never allocate, reallocate, or free any block of CRT type.

`_CLIENT_BLOCK`
An application can keep special track of a given group of allocations for debugging purposes by allocating them as this type of memory block, using explicit calls to the debug heap functions. MFC, for example, allocates all **CObjects** as Client blocks; other applications might keep different memory objects in Client blocks. Subtypes of Client blocks can also be specified for greater tracking granularity. To specify subtypes of Client blocks, shift the number left by 16 bits and `OR` it with `_CLIENT_BLOCK`. For example:

```
#define MYSUBTYPE 4
freedbg(pbData, _CLIENT_BLOCK|(MYSUBTYPE<<16));
```

A client-supplied hook function for dumping the objects stored in Client blocks can be installed using _CrtSetDumpClient, and will then be called whenever a Client block is dumped by a debug function. Also, _CrtDoForAllClientObjects can be used to call a given function supplied by the application for every Client block in the debug heap.

_FREE_BLOCK
Normally, blocks that are freed are removed from the list. To check that freed memory is not still being written to or to simulate low memory conditions, you can choose to keep freed blocks on the linked list, marked as Free and filled with a known byte value (currently 0xDD).

_IGNORE_BLOCK
It is possible to turn off the debug heap operations for a period of time. During this time, memory blocks are kept on the list, but are marked as Ignore blocks.

To determine the type and subtype of a given block, use the function _CrtReportBlockType and the macros **_BLOCK_TYPE** and **_BLOCK_SUBTYPE**. The macros are defined (in crtdbg.h), as follows:

```
#define _BLOCK_TYPE(block)          (block & 0xFFFF)
#define _BLOCK_SUBTYPE(block)       (block >> 16 & 0xFFFF)
```

Contents

Check for heap integrity and memory leaks

Many of the debug heap's features must be accessed from within your code. The following section describes some of the features and how to use them.

`_CrtCheckMemory`
You can use a call to _CrtCheckMemory, for example, to check the heap's integrity at any point. This function inspects every memory block in the heap, verifies that the memory block header information is valid, and confirms that the buffers have not been modified.

`_CrtSetDbgFlag`
You can control how the debug heap keeps track of allocations using an internal flag, _crtDbgFlag, which can be read and set using the _CrtSetDbgFlag function. By changing this flag, you can instruct the debug heap to check for memory leaks when the program exits and report any leaks that are detected. Similarly, you can specify that freed memory blocks not be removed from the linked list, to simulate low-memory situations. When the heap is checked, these freed blocks are inspected in their entirety to ensure that they have not been disturbed.

The **_crtDbgFlag** flag contains the following bit fields:

Bit field	Default value	Description
_CRTDBG_ALLOC_MEM_DF	On	Turns on debug allocation. When this bit is off, allocations remain chained together but their block type is **_IGNORE_BLOCK**.
_CRTDBG_DELAY_FREE_MEM_DF	Off	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** and filled with a special byte value.
_CRTDBG_CHECK_ALWAYS_DF	Off	Causes **_CrtCheckMemory** to be called at every allocation and deallocation. This slows execution, but catches errors quickly.
_CRTDBG_CHECK_CRT_DF	Off	Causes blocks marked as type **_CRT_BLOCK** to be included in leak-detection and state-difference operations. When this bit is off, the memory used internally by the run-time library is ignored during such operations.
_CRTDBG_LEAK_CHECK_DF	Off	Causes leak checking to be performed at program exit via a call to **_CrtDumpMemoryLeaks**. An error report is generated if the application has failed to free all the memory that it allocated.

Contents

Configure the debug heap

All calls to heap functions such as `malloc`, `free`, `calloc`, `realloc`, `new`, and `delete` resolve to debug versions of those functions that operate in the debug heap. When you free a memory block, the debug heap automatically checks the integrity of the buffers on either side of your allocated area and issues an error report if overwriting has occurred.

To use the debug heap

- Link the debug build of your application with a debug version of the C run-time library.

To change one or more _crtDbgFlag bit fields and create a new state for the flag

1. Call `_CrtSetDbgFlag` with the `newFlag` parameter set to `_CRTDBG_REPORT_FLAG` (to obtain the current `_crtDbgFlag` state) and store the returned value in a temporary variable.
2. Turn on any bits by `OR`-ing (bitwise | symbol) the temporary variable with the corresponding bitmasks (represented in the application code by manifest constants).
3. Turn off the other bits by `AND`-ing (bitwise & symbol) the variable with a `NOT` (bitwise ~ symbol) of the appropriate bitmasks.
4. Call `_CrtSetDbgFlag` with the `newFlag` parameter set to the value stored in the temporary variable to create the new state for `_crtDbgFlag`.

For example, the following lines of code turn on automatic leak detection and turn off checking for blocks of type `_CRT_BLOCK`:

```
// Get current flag
int tmpFlag = _CrtSetDbgFlag( _CRTDBG_REPORT_FLAG );

// Turn on leak-checking bit.
tmpFlag |= _CRTDBG_LEAK_CHECK_DF;

// Turn off CRT block checking bit.
tmpFlag &= ~_CRTDBG_CHECK_CRT_DF;

// Set flag to the new value.
_CrtSetDbgFlag( tmpFlag );
```

Contents

new, delete, and _CLIENT_BLOCKs in the C++ debug heap

The debug versions of the C run-time library contain debug versions of the C++ `new` and `delete` operators. If you use the `_CLIENT_BLOCK` allocation type, you must call the debug version of the `new` operator directly or create macros that replace the `new` operator in debug mode, as shown in the following example:

```
/* MyDbgNew.h
 Defines global operator new to allocate from
 client blocks
*/

#ifdef _DEBUG
   #define DEBUG_CLIENTBLOCK   new( _CLIENT_BLOCK, __FILE__, __LINE__)
#else
   #define DEBUG_CLIENTBLOCK
#endif // _DEBUG

/* MyApp.cpp
        Use a default workspace for a Console Application to
 *      build a Debug version of this code
*/

#include "crtdbg.h"
#include "mydbgnew.h"

#ifdef _DEBUG
#define new DEBUG_CLIENTBLOCK
#endif

int main( )   {
    char *p1;
    p1 =  new char[40];
    _CrtMemDumpAllObjectsSince( NULL );
}
```

The Debug version of the `delete` operator works with all block types and requires no changes in your program when you compile a Release version.

Contents

Heap State Reporting Functions

_CrtMemState

To capture a summary snapshot of the state of the heap at a given time, use the _CrtMemState structure defined in CRTDBG.H:

```
typedef struct _CrtMemState
{
    // Pointer to the most recently allocated block:
    struct _CrtMemBlockHeader * pBlockHeader;
    // A counter for each of the 5 types of block:
    size_t lCounts[_MAX_BLOCKS];
    // Total bytes allocated in each block type:
    size_t lSizes[_MAX_BLOCKS];
    // The most bytes allocated at a time up to now:
    size_t lHighWaterCount;
    // The total bytes allocated at present:
    size_t lTotalCount;
} _CrtMemState;
```

This structure saves a pointer to the first (most recently allocated) block in the debug heap's linked list. Then, in two arrays, it records how many of each type of memory block (_NORMAL_BLOCK, `_CLIENT_BLOCK`, _FREE_BLOCK, and so on) are in the list and the number of bytes allocated in each type of block. Finally, it records the highest number of bytes allocated in the heap as a whole up to that point, and the number of bytes currently allocated.

Other CRT Reporting Functions

The following functions report the state and contents of the heap, and use the information to help detect memory leaks and other problems.

Function	Description
_CrtMemCheckpoint	Saves a snapshot of the heap in a **_CrtMemState** structure supplied by the application.
_CrtMemDifference	Compares two memory state structures, saves the difference between them in a third state structure, and returns TRUE if the two states are different.
_CrtMemDumpStatistics	Dumps a given **_CrtMemState** structure. The structure may contain a snapshot of the state of the debug heap at a given moment or the difference between two snapshots.
_CrtMemDumpAllObjectsSince	Dumps information about all objects allocated since a given snapshot was taken of the heap or from the start of execution. Every time it dumps a **_CLIENT_BLOCK** block, it calls a hook function supplied by the application, if one has been installed using **_CrtSetDumpClient**.

Function	Description
_CrtDumpMemoryLeaks	Determines whether any memory leaks occurred since the start of program execution and, if so, dumps all allocated objects. Every time **_CrtDumpMemoryLeaks** dumps a **_CLIENT_BLOCK** block, it calls a hook function supplied by the application, if one has been installed using **_CrtSetDumpClient**.

Contents

Track Heap Allocation Requests

Although pinpointing the source file name and line number at which an assert or reporting macro executes is often very useful in locating the cause of a problem, the same is not as likely to be true of heap allocation functions. While macros can be inserted at many appropriate points in an application's logic tree, an allocation is often buried in a special routine that is called from many different places at many different times. The question is usually not what line of code made a bad allocation, but rather which one of the thousands of allocations made by that line of code was bad and why.

Unique Allocation Request Numbers and _crtBreakAlloc

The simplest way to identify the specific heap allocation call that went bad is to take advantage of the unique allocation request number associated with each block in the debug heap. When information about a block is reported by one of the dump functions, this allocation request number is enclosed in braces (for example, "{36}").

Once you know the allocation request number of an improperly allocated block, you can pass this number to _CrtSetBreakAlloc to create a breakpoint. Execution will break just before allocating the block, and you can backtrack to determine what routine was responsible for the bad call. To avoid recompiling, you can accomplish the same thing in the debugger by setting **_crtBreakAlloc** to the allocation request number you are interested in.

Creating Debug Versions of Your Allocation Routines

A somewhat more complicated approach is to create Debug versions of your own allocation routines, comparable to the **_dbg** versions of the heap allocation functions. You can then pass source file and line number arguments through to the underlying heap allocation routines, and you will immediately be able to see where a bad allocation originated.

For example, suppose your application contains a commonly used routine similar to the following:

```
int addNewRecord(struct RecStruct * prevRecord,
                 int recType, int recAccess)
{
    // ...code omitted through actual allocation...
    if ((newRec = malloc(recSize)) == NULL)
    // ... rest of routine omitted too ...
}
```

In a header file, you could add code such as the following:

```
#ifdef _DEBUG
#define  addNewRecord(p, t, a) \
           addNewRecord(p, t, a, __FILE__, __LINE__)
```

```
#endif
```

Next, you could change the allocation in your record-creation routine as follows:

```
int addNewRecord(struct RecStruct *prevRecord,
                 int recType, int recAccess
#ifdef _DEBUG
                , const char *srcFile, int srcLine
#endif
    )
{
    /* ... code omitted through actual allocation ... */
    if ((newRec = _malloc_dbg(recSize, _NORMAL_BLOCK,
            srcFile, scrLine)) == NULL)
    /* ... rest of routine omitted too ... */
}
```

Now the source file name and line number where `addNewRecord` was called will be stored in each resulting block allocated in the debug heap and will be reported when that block is examined.

Contents

See Also

Debugging Native Code