Finding Memory Leaks Using the CRT Library

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 Finding Memory Leaks Using the CRT Library.

Memory leaks, defined as the failure to correctly deallocate memory that was previously allocated, are among the most subtle and hard-to-detect bugs in C/C++ applications. A small memory leak might not be noticed at first, but over time, a progressive memory leak can cause symptoms that range from decreased performance to crashing when the application runs out of memory. Worse, a leaking application that uses up all available memory can cause another application to crash, creating confusion as to which application is responsible. Even seemingly harmless memory leaks might be symptomatic of other problems that should be corrected.

The Visual Studio debugger and C Run-Time (CRT) libraries provide you with the means for detecting and identifying memory leaks.

Enabling Memory Leak Detection

The primary tools for detecting memory leaks are the debugger and the C Run-Time Libraries (CRT) debug heap functions.

To enable the debug heap functions, include the following statements in your program:

```
#define _CRTDBG_MAP_ALLOC
#include <stdlib.h>
#include <crtdbg.h>
```

For the CRT functions to work correctly, the `#include` statements must follow the order shown here.

Including crtdbg.h maps the `malloc` and the free functions to their debug versions, _malloc_dbg and `free`, which track memory allocation and deallocation. This mapping occurs only in debug builds, which have `_DEBUG`. Release builds use the ordinary `malloc` and `free` functions.

The `#define` statement maps a base version of the CRT heap functions to the corresponding debug version. If you omit the `#define` statement, the memory leak dump will be less detailed.

After you have enabled the debug heap functions by using these statements, you can place a call to `_CrtDumpMemoryLeaks` before an application exit point to display a memory-leak report when your application exits:

```
_CrtDumpMemoryLeaks();
```

If your application has multiple exits, you do not need to manually place a call to _CrtDumpMemoryLeaks at every exit point. A call to `_CrtSetDbgFlag` at the beginning of your application will cause an automatic call to `_CrtDumpMemoryLeaks` at each exit point. You must set the two bit fields shown here:

```
_CrtSetDbgFlag ( _CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF );
```

By default, `_CrtDumpMemoryLeaks` outputs the memory-leak report to the **Debug** pane of the **Output** window. You can use `_CrtSetReportMode` to redirect the report to another location.

If you use a library, the library might reset the output to another location. In that case, you can set the output location back to the **Output** window, as shown here:

```
_CrtSetReportMode( _CRT_ERROR, _CRTDBG_MODE_DEBUG );
```

Interpreting the Memory-Leak Report

If your application does not define `_CRTDBG_MAP_ALLOC`, _CrtDumpMemoryLeaks displays a memory-leak report that looks like this:

```
Detected memory leaks!
Dumping objects ->
{18} normal block at 0x00780E80, 64 bytes long.
 Data: <                > CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD
Object dump complete.
```

If your application defines `_CRTDBG_MAP_ALLOC`, the memory-leak report looks like this:

```
Detected memory leaks!
Dumping objects ->
c:\users\username\documents\projects\leaktest\leaktest.cpp(20) : {18}
 normal block at 0x00780E80, 64 bytes long.
 Data: <                > CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD
Object dump complete.
```

The difference is that the second report shows name of the file and the line number where the leaked memory is first allocated.

Whether you define `_CRTDBG_MAP_ALLOC` or not, the memory-leak report will display the following information:

- The memory allocation number, which is `18` in this example
- The block type, which is `normal` in this example.

- The hexadecimal memory location, which is `0x00780E80` in this example.
- The size of the block, `64 bytes` in this example.
- The first 16 bytes of data in the block, in hexadecimal form.

The memory-leak report identifies a block of memory as normal, client, or CRT. A *normal block* is ordinary memory allocated by your program. A *client block* is a special type of memory block used by MFC programs for objects that require a destructor. The MFC `new` operator creates either a normal block or a client block, as appropriate for the object being created. A *CRT block* is allocated by the CRT library for its own use. The CRT library handles the deallocation for these blocks. Therefore, it is unlikely you will see these in the memory leak report unless something is significantly wrong, for example, the CRT library is corrupted.

There are two other types of memory blocks that never appear in memory-leak reports. A *free block* is memory that has been released. That means it is not leaked, by definition. An *ignore block* is memory that you have explicitly marked to exclude it from the memory-leak report.

These techniques work for memory allocated using the standard CRT `malloc` function. If your program allocates memory using the C++ `new` operator, however, you may only see the file and line number where the implementation of global `operator new` calls `_malloc_dbg` in the memory-leak report. Because that behavior is not very useful, you can change it to report the line that made the allocation by using a macro that looks like this:

C++

```
#ifdef _DEBUG
    #define DBG_NEW new ( _NORMAL_BLOCK , __FILE__ , __LINE__ )
    // Replace _NORMAL_BLOCK with _CLIENT_BLOCK if you want the
    // allocations to be of _CLIENT_BLOCK type
#else
    #define DBG_NEW new
#endif
```

Now you can replace the `new` operator by using the `DBG_NEW` macro in your code. In debug builds, this uses an overload of global `operator new` that takes additional parameters for the block type, file, and line number. This overload of `new` calls `_malloc_dbg` to record the extra information. When you use `DBG_NEW`, the memory leak reports show the filename and line number where the leaked objects were allocated. In retail builds, it uses the default `new`. (We do not recommend you create a preprocessor macro named `new`, or any other language keyword.) Here's an example of the technique:

C++

```
// debug_new.cpp
// compile by using: cl /EHsc /W4 /D_DEBUG /MDd debug_new.cpp
#define _CRTDBG_MAP_ALLOC
#include <cstdlib>
#include <crtdbg.h>

#ifdef _DEBUG
    #define DBG_NEW new ( _NORMAL_BLOCK , __FILE__ , __LINE__ )
    // Replace _NORMAL_BLOCK with _CLIENT_BLOCK if you want the
    // allocations to be of _CLIENT_BLOCK type
#else
    #define DBG_NEW new
#endif

struct Pod {
    int x;
};

void main() {
    Pod* pPod = DBG_NEW Pod;
    pPod = DBG_NEW Pod; // Oops, leaked the original pPod!
    delete pPod;

    _CrtDumpMemoryLeaks();
}
```

When you run this code in the debugger in Visual Studio, the call to `_CrtDumpMemoryLeaks` generates a report in the **Output** window that looks similar to this:

Output

```
Detected memory leaks!
Dumping objects ->
c:\users\username\documents\projects\debug_new\debug_new.cpp(20) : {75}
 normal block at 0x0098B8C8, 4 bytes long.
 Data: <    > CD CD CD CD
Object dump complete.
```

This tells you that the leaked allocation was on line 20 of debug_new.cpp.

Setting Breakpoints on a Memory Allocation Number

The memory allocation number tells you when a leaked memory block was allocated. A block with a memory allocation number of 18, for example, is the 18th block of memory allocated during the run of the application. The CRT report counts all memory-block allocations during the run. This includes allocations by the CRT library and other libraries such as MFC. Therefore, a block with a memory allocation number of 18 may not be the 18th memory block allocated by your code. Typically, it will not be.

You can use the allocation number to set a breakpoint on the memory allocation.

To set a memory-allocation breakpoint using the Watch window

1. Set a breakpoint near the start of your application, and then start your application.

2. When the application breaks at the breakpoint, the **Watch** window.

3. In the **Watch** window, type `_crtBreakAlloc` in in the **Name** column.

   If you are using the multithreaded DLL version of the CRT library (the /MD option), include the context operator: `{,,ucrtbased.dll}_crtBreakAlloc`

4. Press **RETURN**.

   The debugger evaluates the call and places the result in the **Value** column. This value will be –1 if you have not set any breakpoints on memory allocations.

5. In the **Value** column, replace the value shown with the allocation number of the memory allocation where you want to break.

After you set a breakpoint on a memory-allocation number, you can continue to debug. Be careful to run the program under the same conditions as the previous run so that the memory-allocation order does not change. When your program breaks at the specified memory allocation, you can use the **Call Stack** window and other debugger windows to determine the conditions under which the memory was allocated. Then, you can continue execution to observe what happens to the object and determine why it is not correctly deallocated.

Setting a data breakpoint on the object might also be helpful. For more information, see Using Breakpoints.

You can also set memory-allocation breakpoints in code. There are two ways to do this:

```
_crtBreakAlloc = 18;
```

or:

```
_CrtSetBreakAlloc(18);
```

Comparing Memory States

Another technique for locating memory leaks involves taking snapshots of the application's memory state at key points. To take a snapshot of the memory state at a given point in your application, create a **_CrtMemState** structure and pass it to the `_CrtMemCheckpoint` function. This function fills in the structure with a snapshot of the current memory state:

```
_CrtMemState s1;
_CrtMemCheckpoint( &s1 );
```

`_CrtMemCheckpoint` fills in the structure with a snapshot of the current memory state.

To output the contents of a **_CrtMemState** structure, pass the structure to the _ `CrtMemDumpStatistics` function:

```
_CrtMemDumpStatistics( &s1 );
```

_ `CrtMemDumpStatistics` outputs a dump of memory state that looks like this:

```
0 bytes in 0 Free Blocks.
0 bytes in 0 Normal Blocks.
3071 bytes in 16 CRT Blocks.
0 bytes in 0 Ignore Blocks.
0 bytes in 0 Client Blocks.
Largest number used: 3071 bytes.
```

```
Total allocations: 3764 bytes.
```

To determine whether a memory leak has occurred in a section of code, you can take snapshots of the memory state before and after the section, and then use _ `CrtMemDifference` to compare the two states:

```
_CrtMemCheckpoint( &s1 );
// memory allocations take place here
_CrtMemCheckpoint( &s2 );

if ( _CrtMemDifference( &s3, &s1, &s2) )
   _CrtMemDumpStatistics( &s3 );
```

`_CrtMemDifference` compares the memory states `s1` and `s2` and returns a result in (`s3`) that is the difference of `s1` and `s2`.

One technique for finding memory leaks begins by placing `_CrtMemCheckpoint` calls at the beginning and end of your application, then using `_CrtMemDifference` to compare the results. If `_CrtMemDifference` shows a memory leak, you can add more `_CrtMemCheckpoint` calls to divide your program using a binary search until you have isolated the source of the leak.

False positives

In some cases, `_CrtDumpMemoryLeaks` can give false indications of memory leaks. This might occur if you use a library that marks internal allocations as _NORMAL_BLOCKs instead of `_CRT_BLOCK`s or `_CLIENT_BLOCK`s. In that case, `_CrtDumpMemoryLeaks` is unable to tell the difference between user allocations and internal library allocations. If the global destructors for the library allocations run after the point where you call `_CrtDumpMemoryLeaks`, every internal library allocation is reported as a memory leak. Older versions of the Standard Template Library, earlier than Visual Studio .NET, caused `_CrtDumpMemoryLeaks` to report such false positives, but this has been fixed in recent releases.

See Also

CRT Debug Heap Details
Debugger Security
Debugging Native Code