How To Find Memory Leaks

BigWorld Technology 2.0. Released 2010.

Software designed and built in Australia by BigWorld.

Level 2, Wentworth Park Grandstand, Wattle St
Glebe NSW 2037, Australia
www.bigworldtech.com

Table of Contents

Chapter 1. Introduction

Memory leaks occur when allocated memory is never deallocated as a result of program error. Memory leaks can result in degraded performance over time for all applications on the system, and are especially problematic for real-time or embedded applications.

Non-trivial C++ programs can contain thousands of explicit memory management calls, resulting in millions of allocations and deallocations throughout the life of a program. Worse still, commonly used libraries such as the STL can add thousands of implicit allocations that are far less obvious. The burden of managing these and correctly freeing allocated memory is generally placed on the programmer.

When memory leaks occur they are often very hard to find manually. Numerous third party tools like BoundsChecker and Purify are available for Windows. However these are *active* methods of finding memory issues, whereas we require a *passive* method. That is, we need to know exactly when leaks occur without having to add instrumentation, build a particular configuration, or run an external tool. External tools are more useful for investigating known leaks when more data is needed.

Chapter 2. Memory Tracker

BigWorld provides a singleton class called `MemTracker`, which provides the following functionality:

- Automatic leak detection and reporting.
- Allocation id.
- A stack trace for each allocation that leaked.

The first two features are enabled by default without adding any instrumentation or running any external programs. Once leaks are identified, users can re-run the application and break on a given allocation id.

`MemTracker` works on both Windows and Linux platforms. The stack trace functionality is only available in debug configurations.

Chapter 3. How To

3.1. Find out if your program leaks

Link your program against the `cstdmf` library, and ensure every source file includes the `cstdmf.hpp` header.

Then, run the program in the `DEBUG` configuration, and if leaks occur they will be output in the debugger, when the program exits, in the following format:

```
MemTracker detected the following leaks:
Slot: Default, Id: 5007 - 88 bytes
Slot: Default, Id: 8695 - 44 bytes
Slot: Default, Id: 8883 - 60 bytes
Slot: Default, Id: 8884 - 60 bytes
Slot: Default, Id: 9435 - 28 bytes
Slot: Default, Id: 9436 - 20 bytes
Slot: Default, Id: 20975 - 116 bytes
Slot: Default, Id: 20976 - 16 bytes
Slot: Default, Id: 392363 - 12 bytes
```

Note `MemTracker` can be enabled for any build configuration by defining preprocessor symbol `ENABLE_MEMTRACKER` for that configuration.

3.2. Break on a given memory allocation.

To enter the debugger on the first allocation in the example above, add the following line to your program (outside of a function body).

```
MEMTRACKER_BREAK_ON_ALLOC( Default, 5007 );
```

3.3. Defining slots

Often it is convenient to divide up allocations into slots based on function scope. To do this, `MemTracker` allows you to define slots.

```
MEMTRACKER_DECLARE( Foo, "Foo's allocations", 0 );

void Foo()
{
    MEMTRACKER_SCOPED( Foo );

    DoFooStuff();
    DoOtherStuff();
}
```

This will mark any allocations in `Foo()`, `DoFooStuff()` and `DoOtherStuff()` as belonging to the `"Foo"` slot, and each allocation will be numbered uniquely within the `"Foo"` slot. Slots are organised into a stack, so `DoFooStuff()` may contain its own definition for a slot, and its own leaks could be reported there.

If you do not supply any slots, then the top level slot called `Default` is used.

3.4. Ignoring leaks

Sometimes it isn't convenient to fix leaks as soon as they are found. If this is the case, you can prevent leaks from being reported by declaring a slot with the `FLAG_DONT_REPORT` flag. In the example above, if we wished to ignore all leaks in the Foo slot, we would declare the slot like so:

```
MEMTRACKER_DECLARE( Foo, "Foo's allocations", FLAG_DONT_REPORT );
```

3.5. See the call stack for leaks

If you require a stack trace for a leak, you should define the `ENABLE_CALLSTACK` symbol as `1` in `memory_tracker.h` and add the `FLAG_CALLSTACK` flag to the definition. Use of this flag requires a large amount of overhead per allocation, so it is recommended for use within a small scope (for example, do not enable it in a slot that contains more than a few hundred allocations).

Here is an example stack trace from a leak, reading from bottom to top, the offending function is the one that calls `bw_malloc()`, in this case `foo()`:

```
e:\mf_19_0\src\lib\cstdmf\memory_tracker.cpp (514): bw_malloc
e:\mf_19_0\src\lib\cstdmf\unit_test\test_memory_tracker.cpp (73): foo
e:\mf_19_0\src\lib\cstdmf\unit_test\test_memory_tracker.cpp (79): bar
e:\mf_19_0\src\lib\cstdmf\unit_test\test_memory_tracker.cpp (84): wuu
e:\mf_19_0\src\lib\cstdmf\unit_test\test_memory_tracker.cpp (90):
 MemoryTracker_testCallstackTest::RunTest
e:\mf\src\lib\third_party\cppunitlite2\src\test.cpp (40): Test::Run
e:\mf\src\lib\third_party\cppunitlite2\src\testregistry.cpp (37):
 TestRegistry::Run
e:\mf\src\lib\unit_test_lib\unit_test.cpp (59): BWUnitTest::runTest
e:\mf\src\lib\cstdmf\unit_test\main.cpp (12): main
f:\sp\vctools\crt_bld\self_x86\crt\src\crtexe.c (597): __tmainCRTStartup
f:\sp\vctools\crt_bld\self_x86\crt\src\crtexe.c (414): mainCRTStartup
ERROR: SymGetLineFromAddr64, GetLastError: 487 (Address: 76CE4911)
76CE4911 (kernel32): (filename not available): BaseThreadInitThunk
ERROR: SymGetLineFromAddr64, GetLastError: 487 (Address: 7743E4B6)
7743E4B6 (ntdll): (filename not available): RtlInitializeExceptionChain
ERROR: SymGetLineFromAddr64, GetLastError: 487 (Address: 7743E489)
7743E489 (ntdll): (filename not available): RtlInitializeExceptionChain
```