Memory Leak Detection
Or, How to not Shoot Yourself in the Foot

Alan Wu

Version History

2006.12.19 - Version 1.0 - Initial document.

Contents

1 Introduction

C is widely used for its speed and power, primarily through direct control of memory via dynamic allocation and pointers. However, it is very easy to literally shoot oneself in the foot while working with memory. With the advent of large game solvers for Gamesman, it is becoming readily apparent that constraining memory use is of utmost concern. While dynamic allocation is a powerful tool, failure to release dynamically allocated memory that is no longer needed can quickly lead to mushrooming memory usage, and, eventually, segmentation faults. The purpose of this document is to demonstrate how to use the library Memwatch, which may be used to catch memory leaks. Unfortunately, C++ support is not recommended (due to classes being allowed their own memory management, and the possibility of Memwatch's preprocessor approach wreaking havoc with class-specific memory management).

2 About Memwatch

Memwatch is a C library, written by Johan Lindh, used for memory debugging. It can be downloaded from http://www.linkdata.se/sourcecode.html. From the webpage, Memwatch is:

> A memory leak detection tool. Basically, you add a header file to your souce code files, and compile with MEMWATCH defined or not. The header file MEMWATCH.H contains detailed instructions. This is a list of some of the features present in version 2.71:
> - ANSI C
> - Logging to file or user function using TRACE() macro
> - Fault tolerant, can repair it's own data structures
> - Detects double-frees and erroneous free's
> - Detects unfreed memory
> - Detects overflow and underflow to memory buffers
> - Can set maximum allowed memory to allocate, to stress-test app
> - ASSERT(expr) and VERIFY(expr) macros
> - Can detect wild pointer writes
> - Support for OS specific address validation to avoid GP's (segmentation faults)
> - Collects allocation statistics on application, module or line level
> - Rudimentary support for threads (see FAQ for details)
> - Rudimentary support for C++ (disabled by default, use with care!)
> - ...and more

Memwatch is a very powerful tool, and it is highly recommended that every Gamescrafters developer use it frequently in order to find deleterious memory bugs.

3 Using Memwatch

Memwatch has been included in Gamesman for convenience; the Makefiles are a bit of black magic, and it is best if they are left alone, lest something nasty happened. Ensure that you have the latest version from the CVS repository; then run the following commands from your Gamesman root directory:

```
./configure
```

make clean
make memdebug

After completing the above steps, run the executable of your choice (preferably the one being debugged!). Memwatch will create a log file in the executable's directory (usually 'bin') called Memwatch.log. This file, described in the next section, will provide various memory allocation statistics, as well as list any "anomalies". It should be noted that Memwatch will inevitably slow down Gamesman; this is due to the overhead of logging all dynamic memory requests.

3.1 Understanding the log file

Below is a sample memwatch.log file. It was created by running msim with an existing, solved database.

```
============= MEMWATCH 2.71 Copyright (C) 1992-1999 Johan Lindh =============

Started at Tue Dec 19 17:28:26 2006

Modes: __STDC__ 32-bit mwDWORD==(unsigned long)
mwROUNDALLOC==4 sizeof(mwData)==32 mwDataSize==32



Stopped at Tue Dec 19 17:28:32 2006

unfreed: <15> openPositions.c(38), 14348907 bytes at 0xb2a9c02c
         {00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................}
unfreed: <14> openPositions.c(37), 14348907 bytes at 0xb384c02c
         {00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................}
unfreed: <13> openPositions.c(36), 57395628 bytes at 0xb45fc02c
         {F8 0F 00 00 F8 0F 00 00 F8 0F 00 00 F8 0F 00 00 ................}
Memory usage statistics (global):
 N)umber of allocations made: 37
 L)argest memory usage      : 98649355
 T)otal of all alloc() calls: 98649533
 U)nfreed bytes totals      : 86093442
MEMWATCH detected 3 anomalies
```

The first few lines list miscellaneous information that is somewhat self-explanatory. Next come the "anomalies". These are, in short, various events that should not have occurred. After the anomalies comes a listing of various statistics.

The anomalies present in this particular log file are listed below:

```
unfreed: <15> openPositions.c(38), 14348907 bytes at 0xb2a9c02c
         {00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................}
unfreed: <14> openPositions.c(37), 14348907 bytes at 0xb384c02c
         {00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................}
unfreed: <13> openPositions.c(36), 57395628 bytes at 0xb45fc02c
         {F8 0F 00 00 F8 0F 00 00 F8 0F 00 00 F8 0F 00 00 ................}
```

All three anomalies are of the same type, unfreed. This refers to the fact that some amount of dynamically allocated memory was never freed before the program terminated. There are other types of anomalies as well, including wild reads/writes, double frees, etc. The file **src/core/memwatch.h** contains a full listing of possible anomalies.

unfreed is probably the most common memory bug encountered. While it is acceptable to simply never free memory and just depend on program termination to free dynamically allocated memory, this is not recommended. One nasty instance of this "bug" was discovered in the loopy solver about half a year ago: a list of nodes was never freed. This particular bug went unnoticed until the analysis team began performing automated mass analyses of games. Since the list was never freed, and the automated mass analysis solved each variant before terminating, this led to skyrocketing memory usage and, eventually, the dreaded segmentation fault. It was in fact this incident that prompted the first use of Memwatch with Gamesman.

4 Changes made to support Memwatch

First, Memwatch itself needed to be modified. Memwatch works by using a macro to replace all calls to malloc(), free(), etc., with calls to its own internal functions, thus allowing it to record memory allocation as well the location of each call. However, Gamesman uses the functions SafeMalloc() and SafeFree() to provide (safe) dynamic memory allocation; essentially, they perform error checking for the programmer. However, this leads to a problem: since all malloc() and free() calls are routed through these two functions, it is impossible to know where exactly an offending line is in the event of a segmentation fault or other memory bug. Thus, the following macros were added to **src/core/memwatch.h**:

```
#define SafeMalloc(n) mwMalloc(n,__FILE__,__LINE__)
#define SafeRealloc(p,n) mwRealloc(p,n,__FILE__,__LINE__)
#define SafeFree(p) mwFree(p,__FILE__,__LINE__)
```

These three macros replace all calls to the Safe*() functions with calls to the Memwatch equivalents.

Next, **src/core/gamesman.h** was edited to include **src/core/memwatch.h**.

When compiling a program that uses Memwatch, MEMWATCH must be defined, or else Memwatch will not activate. A new target, **memdebug** was added to each Makefile. This target would add **-DMEMWATCH** to the **CFLAGS** variable, in order to activate Memwatch. The files **src/core/memwatch.c** and **src/core/memwatch.h** were also added in order to tell the Makefile to actually compile Memwatch.

The prototypes of the Safe*() functions should only be defined when MEMWATCH is defined, otherwise the macro replaces them and GCC complains. So the following lines were added around the Safe*() prototypes in **src/core/misc.h**:

```
#ifndef MEMWATCH
#endif
```

The above two lines were also added around the Safe*() definitions in **src/core/misc.h**.

One final change made, this time to the majority of the game files. Most of the game sources declare the Safe*() functions as externs. These too had

```
#ifndef MEMWATCH
#endif
```

placed around them to prevent GCC complaints.

5 The future

It is highly unlikely that Memwatch will be modified or touched until the Gamesman refactor. However, since Gamesman++ is slated to use C++, and Memwatch support for C++ is tricky, it is highly likely that a different tool will need to be found for future use. Until then, Memwatch will work nicely for memory debugging.