MemMonD

A memory monitor for Delphi[1] applications.

Copyright © 1995 by Per B. Larsen

Overview

MemMonD is a utility for monitoring and analyzing a Delphi program's memory consumption and in particular for tracing instances of pointers that are never de-allocated or de-allocated with the wrong size.

[1] Delphi is a trademark of Borland International

What MemMonD does

MemMonD is a stand-alone application - itself a Delphi program - which will transparently "hook" into the application you've specified, and report the current status of memory and stack allocations as the program runs. When the program terminates, MemMonD will produce a report stating the global memory and stack consumption - along with a list of pointers which were never freed (if any), each with a reference to the line in source where the pointer was allocated.

MemMonD will also catch one of the nastier bugs you can have in a Delphi program - one which is occasionally introduced in programs by sloppy programmers like *your's truly* - namely the case where you do a FreeMem with a different size parameter from what you used for the GetMem or AllocMem. In fact, a case like this was the original inspiration for writing the code which is at the heart of this utility. This type of bug is particularly nasty because your program will typically run happily for a while after the bogus freemem and then, eventually, it will blow up.

For completeness, MemMonD will also track Windows' current resource state, reporting the amount of free USER and GDI resources to its screen as well as the application report.

MemMonD will let you record the application's current memory consumption in its report at any point either by pushing a button on the MemMonD panel or from within your programs code. This last bit is an option. In general, you shouldn't have to make any modifications to your code in order to use MemMonD.

MemMonD will monitor just about any Delphi application whether it's activated from within the Delphi environment, the Turbo Debugger, MemMonD itself or any other application.

What MemMonD doesn't do

MemMonD is not a complete resource monitor in the true sense of the word. Whereas Nu-Mega's BoundsChecker for Windows (which, unfortunately, doesn't work well with Delphi applications) is capable of tracing all Windows resources as they're allocated and returned to the system (or not), MemMonD only tracks global memory. What's more, MemMonD is only concerned with memory allocated directly or indirectly through the System unit. Allocations and de-allocations made directly through the Windows API via GlobalAlloc etc. are not seen by MemMonD.

Fortunately, global memory allocated through the normal heap memory system is by far the most important resource to keep track of in a Delphi application. Delphi's Visual Component Library encapsulates just about all Windows resources except memory and a few esoteric GDI resources, like regions. Normally, you don't deal directly with resources like pens, fonts, brushes, menus etc. Rather, you access them through a VCL object like the Canvas. Since the VCL is well-designed - it de-allocates all resources allocated in a constructor in the corresponding destructor - all you need to do in order to make your programs well behaved in terms of freeing all resources on termination is to make sure you've freed all object instances that are using Windows resources directly or indirectly. MemMonD should help you accomplish this. Indeed, the few resource leaks I've found within the VCL itself using MemMonD are without exception caused by classes forgetting to free a contained object within its destructor.

MemMonD is not currently capable of monitoring Delphi DLLs. There's a number of issues involved, and I simply didn't feel it worthwhile to address those. You can usually debug the code as a normal application before turning it into a DLL. I may add this capability in a later version, but don't hold your breath.

MemMonD only monitors the first running instance of an application. If you start more, MemMonD will simply ignore them.

How to use MemMonD

The user interface consists of a button bar at the top and a three pane notebook below. In addition, there's a pop-up menu available anywhere with a right button mouse-click or by pressing Alt-F10 on the keyboard.

Basically, you select an application on the "Run parameters page". You start it and watch the "Running status" page as the application is running. Then, you terminate the application and switch to the "Report" page to see how the application performed.

The button bar

The buttons on the bar have the following meaning:

Pressing this will terminate MemMonD immediately. If an application is being monitored, MemMonD will unhook itself from it, letting it proceed at normal speed.

Pressing this causes the MemMonD window to shrink to a size where just the caption and the buttons are visible. If a program is being monitored, MemMonD will display its name and the current memory consumption in the caption. When pressed, the Shrink button changes into a Grow button which will restore the previous window size when pressed.

Mark increments the internal tag counter, and puts the current memory statistics in the report. You can trigger the Mark functionality from within your own code. See The Report page for details on this.

The tag counter is an internal counter value which is reset to zero when the application under inspection begins execution. Each pointer is stamped with the tag value as it is put in the pointer cache. Tags are convenient for filtering pointers in the View snap-shot dialog.

You seem to have figured out what this one is for.

Between the Shrink/Grow and Help buttons, MemMonD shows a bitmap indicating what state it's in:

This indicates that MemMonD is idle, waiting for you to specify an application to monitor.

This indicates that a .EXE file has been selected for monitoring and that MemMonD is on the lookout for its activation. This might also mean that the application under inspection has terminated.

This, and the other wide-eyed variations, indicates that MemMonD is actively monitoring an application. MemMonD will animate the eyeballs whenever it updates its statistics on screen.

Run parameters

This is where you specify information about the application you wish to monitor.

Pressing the Select button will give you the familiar open file dialog which will let you select the application you're interested in. Note the history list.

When you've selected a .EXE file, MemMonD will immediately look for a .MAP file for the application in the same directory. If none is found, MemMonD will complain, but let you continue. For reasons which should become clear in a moment, you'll want to re-compile your application, generating a detailed .MAP file, before attempting to monitor it with MemMonD. That is, of course, unless you're looking at some application for which you don't have the code ;-)

The Select button does not execute the program. You can do that from the pop-up menu, if you wish.

MemMonD's use of .MAP files

MemMonD looks for a .MAP file for two purposes:

The first is to translate addresses in the report into file names and line numbers. It's nicer to look at "MEMMOND.DPR line 38" than "0001:0C33", though you can also use the Search/Find Error... dialog in Delphi to find out what location in your code a particular address corresponds to.

If a location does show up as XXXX : XXXX in the report, though you've generated a detailed .MAP file, it's either because a) the address is in a unit for which you don't have source (TABS, perhaps), b) the unit was not on your project's specified search path (Options/ Project/ Directories) or c) you - or someone else - have specified $D- and/or $L- for the unit or the entire project.
Then, of course, there's also the remote chance that you've found a bug in MemMonD ;-)

The second - perhaps, less obvious - use for the .MAP file is to let you specify what units to report errors for. Initially, all heap errors are attributes to the System unit. Notably the internal routines NewMemory and DisMemory, who are responsible for all allocations and de-allocations as far as MemMonD is concerned. MemMonD will always trace back to the first public entry point, but still, a report stating that "a bunch of pointers weren't freed property and they were all allocated with New" doesn't help very much.

In theory, the best way to report errors would be to produce a stack-dump for each one, specifying the entire call hierarchy - typically from Application.Run - down (or up, if you prefer) to the point where the error was detected in System. That will have to remain a theory, though: MemMonD doesn't know if an allocation will have a corresponding de-allocation until it sees it, and storing a complete stack trace for each allocation would require way too much memory and it would also make MemMonD extremely slow (as opposed to relatively slow :-|).

The unit pane

Instead, MemMonD offers you the opportunity to specify what units you want errors attributes to.

All units parsed from the .MAP file are presented in the list box along with the code address interval (you needn't worry about that) and an indication of whether you want MemMonD to trace out of each unit.

You toggle between "Yes" and "No" by double-clicking or by pressing the space bar. By choosing "Yes" for a particular unit, you specify that if MemMonD wants to report an address within this unit, it should instead walk further up the call stack. If the next frame is also in a unit where you specified "yes", the walk will

continue - possibly all the way up to the main project file, which is always the top-most "stack frame". From there, there would be no place to go, which is why the main project doesn't appear in the list.

You can think of the units that have "Trace out of" set to "Yes" as being *below the surface.* For instance, if you're allocated a zstring with the SysUtils function StrNew from within your SomeUnit.pas and you're never calling StrDispose for it, you'll initially get an error report with an address within the System unit. If you change the System unit to "Yes" - effectively pushing it below the surface of visibility - you'll get the same error reported, this time in SysUtils. Now, if you change that setting to "Yes" - drowning SysUtils - you'll get the error reported in SomeUnit.pas which is probably what you want - unless you're interested in what exactly goes on in SysUtils.

In general, you'll want to always specify "Yes" for at least the System unit. MemMonD will let you specify a set of default units which should always be traced out of from the pop-up menu.

You can toggle the order in which units are shown between address and unit name by clicking the list header or by pressing F5 on the keyboard.

Running status

The Running status page of MemMonD displays information about the current memory consumption of the application under analysis.

The display shows how much memory is presently allocated and the program's maximum memory consumption since it was started. Below these figures, MemMonD shows the number of live pointers currently in the cache.

Similarly, the current amount of stack use and the maximum is shown.

Below, two gauges show how many free GDI and USER resources Windows reports - and their minimum value since the application was started.

The Running status displayed is updated about once every second, provided Windows is able to schedule MemMonD for execution that often:

MemMonD will not be able to update its display while your program sits in a loop without calling Application.ProcessMessages. This is a fact of 16 bit Windows. That doesn't affect the accuracy of MemMonD's global memory and stack usage sampling for your application, though. When MemMonD monitors a program, it's running within that program's context. The application is calling MemMonD as if it were a part of the application, so memory reporting is always accurate.

It does, however, affect the resource display: MemMonD only calls Windows to inquire about free resources in its own context; immediately before updating the running status - and then, immediately before and after the application is run as well. The resource display can't be totally accurate anyway, so I decided not to do more than absolutely necessary from within the applications context.

A word about Windows resources: Often, you'll see a drop in the free resource percentage the first time an application is run, but no drop on subsequent runs. This is normal Windows behaviour, and it doesn't mean you have a resource leak in your code. The reason is that Windows caches certain resources. Windows doesn't create those resource until someone (your application, in this case) asks for them, but once they're created, Windows hangs on to them.

Report

The Report page of MemMonD's display is where collected information about the program being monitored is shown.

Most of the information is generated after the program has ended, but some information is generated on the fly.

When the application starts, MemMonD checks the amount of free resources and writes the program name and the free resource percentages to the report.

While the application is executing, MemMonD will not write to the report unless you ask it to, except on two occasions:

If MemMonD runs out of buffer space for live pointers, it will display a message stating that the rest of the report cannot be trusted completely.

If you de-allocate an invalid pointer or de-allocate it with the wrong size (as in Getmem(P,24); Freemem(P,8);), MemMonD will inform you immediately in the report. You'll get a message beep, too, as you might want to shut down the application before it crashes. MemMonD doesn't actually prevent you from doing this type of thing - it simply reports that it happened.

If you pass a totally bogus pointer (or a pointer to a truly global memory object, which has already been freed) to Dispose, you're likely to get a GPF. MemMonD won't ever get to see those, so they will not appear in the report.

MemMonD displays the address of the bogus de-allocation it does see, but doesn't convert the address into a unit name/line number - even if you have a .MAP file. When MemMonD reports this error in real-time, it's still running on the application's context, so it can't do stack and memory intensive things like parsing a .MAP file. You can find the error spot in Delphi with Search/Find error - or by inspecting the .MAP file yourself.

You can force MemMonD to write the current memory state to the report at any time by pressing the Mark button. You can also do this with code from within the application under inspection with the following procedure:

```
type
   TMark = procedure(const Msg : String);

procedure Mark(const S : String);
var
   M : TMark;
begin
   @M := GetProcAddress(GetModuleHandle('MEMMOND'),'MARK');
   if Assigned(M) then
      M(S);
end;
```

This will let you supply a message as well. This is convenient if you want to analyze memory consumption between two specific points in code where you're not in a state where a switch to MemMonD can be made manually.

You might use it like this:

```
..
```

```
    Mark('Before MyForm was created');
    fMyForm := TfMyForm.Create(Application);
    fMyForm.ShowModal;
    fMyForm.Free;
    Mark('After MyForm was freed');
  ..
```

The Mark procedure shown does nothing if MemMonD isn't currently active and monitoring your application.

You can also add your own commentary to the MemMonD report directly in the edit box. All MemMonD ever does is append to it, so you can type anything you want.

When the program has ended, you can save the report to a text file from the pop-up menu.

After program termination, MemMonD adds the current resource state, the resource state at the lowest point during application execution and the maximum global memory and stack consumption to the report. If some pointers were never freed, their size and the location where they were allocated (with respect to visibility) are reported as well.

Note, that once the application is being executed a second time, the previous report is erased.

The Pop-up menu

The pop-up menu is available from anywhere within the MemMonD window by pressing Alt-F10 on the keyboard or by clicking the right mouse-button.

Save report...

This point becomes enabled when the application being monitored terminates. It lets you save the report in a text file.

The report window is a standard edit box, so you can also select parts of it and copy them to the clipboard with Ctrl-Ins, if you prefer.

Execute from here

MemMonD will start monitoring the application when it starts - wherever it's started from. This is convenient when you're working with the application from within the Delphi environment. At times, however, you'll want to execute the program right away after having specified it. This menu command is for that.

Default units...

The dialog's edit control lets you specify a list of units that should automatically be set to "Trace out of" in the unit pane on the Run parameters page when the unit list is loaded - regardless of what application you're working with. You might want to specify all the standard Delphi units in this list - there's a button for that in the dialog. You can always override this default "trace out of" setting manually. Specifying all the standard Delphi units will attribute all missing de-allocations to your code. You can then "drill down" by "un-hiding" selected units and re-running the application.

View snap-shot....

This command takes a snap-shot of the pointer cache as it looks right now and presents it in a dialog.

See snap-shot dialog.

Stay on top

This toggles whether the MemMonD window should stay on the top of the desktop when it's de-activated.

Hide reports from project

This toggle has connection to the Default units dialog described above. The VCL allocates a lot of objects that are never de-allocated. Nearly all of them are of a type that will never exist in more than one instance in a running application. Examples of this include the global Application object and some of the internal VCL cache structures like the font cache, the class list etc. For most of these objects, it's ok that the VCL doesn't de-allocate them (though, in my humble opinion, it's a bit untidy): These objects are used throughout the life of a VCL application, and the memory they consume will be reclaimed by Windows when the application terminates. They do show up as errors in the MemMonD report, though.

The objects mentioned in the previous paragraph are allocated during program initialization. If you specify that all VCL units are to be traced out of, these VCL "errors" will be attributed to the main project file. The "Hide reports from project" lets you suppress the corresponding error messages. Note, however, that memory leaks caused by custom components loaded on the auto-created forms will disappear as well,

since they are created from the project file, too.

Consolidate by address

This toggle determines how MemMonD reports pointer that were never de-allocated. When the toggle is off, as it is by default, MemMonD reports each individual instance of un-freed pointers sorted by the address of the allocating statement.

A typical report might look something like this:

```
Pointers not freed:

Bytes  Allocated at

12     UHEAPCHK.PAS line 46
28     UHEAPCHK.PAS line 46
100    SYSUTILS.PAS line 1501
12     UHEAPCHK.PAS line 46
28     UHEAPCHK.PAS line 46
12     UHEAPCHK.PAS line 46
28     UHEAPCHK.PAS line 46
100    SYSUTILS.PAS line 1501
```

If, for instance, you've left out a Free in a busy class, you're likely to get a lot of error reports that are actually the result of that same omission. Consolidate by address lets you compress the error report so that the information shown above would be presented like this:

```
Pointers not freed:

Bytes  Pointers     Allocated at

120    [ 6 ]        UHEAPCHK.PAS line 46
200    [ 2 ]        SYSUTILS.PAS line 1501
```

About...

Displays the familiar about dialog. Don't bother looking for hidden secrets: Ain't got no swimming pool ;-)

How MemMonD works

When the application starts, MemMonD stops it momentarily in its tracks and plugs into the semi-documented HeapCheck pointer in the System unit. MemMonD also takes this opportunity to initialize the free part of the programs stack space to some known value.

When the application allocates and de-allocates memory via New, GetMem, Create, Destroy, Free and FreeMem, etc, the System unit calls MemMonD via the HeapCheck pointer, informing it of the event.

MemMonD stores allocated pointers with their size in a hash table and removes them as they're returned to the pool. While the application under inspection is running, you can look at the hash table via the pop-up menu's View snap-shot command.

When the application terminates, MemMonD inspects its stack and records the maximum stack use and then reports all pointers still in the hash table.

If you terminate MemMonD while your application is being monitored, MemMonD simply restores the HeapCheck pointer to its default NIL value. This causes the System unit to stop calling MemMonD. If the application was started from within MemMonD, MemMonD will let you terminate it at this point, too.

If, for some reason, you want to prevent others from running your application through MemMonD, all you need to do is explicitly set the HeapCheck pointer to NIL in the initialization block of a unit and specify this unit as the first in your project's USES list.

Limitations

MemMonD must play a few tricks with Windows in order to work. This means that MemMonD might not work in your setup. MemMonD was developed using Windows 3.11/debug. It has been tested and found to be working correctly running on Win 3.1 as well as Windows 95. If you get an initialization error in the 600 range, or if you have other difficulties, I'd like to hear from you - but I'm not making any promises.

MemMonD can currently handle at most 32760 live pointer at any one time, assuming there's sufficient memory available. If your application uses more than that or there isn't enough memory available for MemMonD to allocate for its pointer cache, MemMonD will issue an error message to that effect in the report.

MemMonD chops a fair bit of performance out of your application, when it's being monitored. I might be able to improve on that with careful scrutiny and optimizations, but I didn't think it worthwhile. I find the performance hit acceptable when you consider that this is just a debugging utility.

Pricing and availability

MemMonD is copyrighted freeware. I grant you the right to use it and pass on copies of it as much as you like, as long as you include this file. You may also upload it to BBS'es or Internet sites or what have you - again, as long as you include this file with the upload. What you cannot do is include this utility in compilations - be it on disk, CD-ROM or any other media - and charge money for it - neither directly nor as part of bonus packages distributed with other software or magazines - without a prior written permission from me. If you, yourself, got this utility from a source where you paid for it - and this source isn't mentioned in the About box for this program, call them for a full refund!

Why MemMonD is free

> *Don't want money. Got money. Want admiration.*
>
> The Stone Soup Group (makers of Fract-Int)

No, actually, that's not it. In my opinion, there's no such thing as "enough money"<g>. One reason, MemMonD is free, is that I don't believe - whatever the price tag - that this utility would be able to make a significant impact on my financial status. However, the more important reason is that I'm far from certain that this program will work in all environments and generate valid information in all cases. I wouldn't feel comfortable charging money for something that might not work for you. Please consider the price in the - not entirely unlikely - event that you encounter irregularities while using this utility.

How to get the latest version

The home of MemMonD is the Delphi forum on CompuServe. This is where you can always get the latest version - that is, if I ever release another version. Feel free to write me on any of the addresses below and ask whether there's a newer version available, but don't expect me to mail it to you.

I can say right away that I'm definitely not going to do a 32 bit version of this program in a foreseeable future. Win32 isn't quite as forgiving as Win16.

How to reach me

I usually hang out in the Delphi forum on CompuServe, so if you're having trouble using this program or interpreting its output, you might post a question there.

If you absolutely must, you can write me directly on one of the following addresses:

CompuServe 100121,1514	-	(Preferred) This is my corporate account. I check for mail on this account on a daily basis.
CompuServe 75470,1320	-	This is my personal account. It's checked less often, but is slightly more private.

From the Internet, you reach CompuServe mail like so:

> 100121.1514@compuserve.com

You can also write me by snail mail as

Per Larsen
Sct. Anna Gade 53
DK-8000 Aarhus C
Denmark

So long, and thanks for all the fish

Nonsense license agreement.

1. User Obligations.

1.1 User assumes full responsibility that this program meets the specifications, capacity, capabilities, and other requirements of said user, and agrees not to bother the author if the program does not perform as expected, or performs other than expected, or does not perform at all.

1.2 User assumes full responsibility for any deaths or injuries that may result from the normal or abnormal operation of this program. In the event of casualties exceeding 1000 persons or property damage in excess of US$10 million, user agrees that he or she has stolen the program and I didn't even know he or she had it.

1.3 User agrees not to say bad things about the program or the author to anyone claiming to be from "60 minutes".

2. Limited Warranty.

2.1 For a period of 90 minutes, commencing from the time you first thought about getting this program, I warrant that this program may or may not be free of any manufacturing defects. It will be replaced during the warranty period upon payment of an amount equal to the original purchase price plus US$100 for handling. This warranty is void if the program has been examined or run by the user, or if the documentation has been read.

2.2 This program is distributed on an AS WAS basis. The author makes no warranty that this is, in fact, what I say it is in my propaganda, or that it will perform any useful function. I have no obligation whatsoever other than to provide you with this fine disclaimer.

2.3 Some countries do not allow limitations as to how long an implied warranty lasts, so I refuse to imply anything.

2.4 There is an extremely small but non-zero chance that, through a process known as "tunnelling", this program may spontaneously disappear from its present location and reappear at any random place in the universe, including your neighbours computer system. The author will not be responsible for any damages or inconvenience that may result.

3. Limitation of liability.

3.1 I have no liability or responsibility to the user, the user's agents, my creditors, your creditors, or anyone else.

How to un-install MemMonD

Should you decide that you want to get rid of this useless piece of software, it's quite easy:

After having been run the first time, MemMonD consists of four files:

MEMMOND.EXE
MEMMOND.DLL
MEMMOND.HLP
MEMMOND.INI

The first three files reside where you put them <s>. The .INI files resides in the Windows directory. To remove MemMonD from your system, just delete all of these files.

MemMonD makes no changes to any other settings, INI files or the registry.

The Snap-shot dialog

This function is only available while the application under inspection is running. It takes a snap-shot of the current pointer cache for the application and presents it in a list box dialog.

You can toggle the sort order by pressing the headers of the list box.

At any time, you can dump the contents of the current pane to the report.

At the upper right is a spin edit which lets you filter what pointer you wish to see. Pointers are filtered based on their tag value. Each pointer is stamped with the current tag value when it's put in the pointer cache. The current tag value is incremented each time the Mark functionality is used. A filter value of zero means no filtering. All other values will cause the list box to only show pointers with that value.

You can use this to analyze your program's memory consumption by pressing Mark and executing a particular function in your program. Then, if you take a snap-shot of MemMonD's pointer cache, you can filter out all pointers but the ones what were allocated since the tag was incremented (the current value is shown in the dialog). This will show you what memory blocks your particular function is using.

This dialog doesn't show program locations with file names and line numbers, but as usual, you can find them in Delphi with Search/Find error.

The Consolidate button compresses the report based on locations. That is, all entries with the same value for "Allocated at" are combined into one. After consolidation, the Pointer column will change into Pointers, showing the number of allocations at each program address. The Size column now shows the total allocation.

Note! The snap-shot dialog cannot presently display more than 5460 pointers at a time. If you have more live pointers than that, the list is truncated.