Snoop

Snoop is a freeware memory-leak tracking utility for Borland Delphi 2 and 3.

Snoop interfaces smoothly with your programs and takes note of all memory allocations and deallocations and where in your code they occur. When your program ends ***Snoop*** creates a report of all the memory blocks that didn't get deallocated.

Snoop checks memory handled by:

- The GetMem, ReallocMem, and FreeMem standard procedures
- The New and Dispose standard procedures
- Allocation and deallocation of objects through constructors and destructors
- Allocation and deallocation of dynamic strings

Snoop can also optionally check for corruption of memory when you write beyond the bounds of a memory block. ***Snoop*** can now be used to debug dlls. Version 2 is completely re-written for greater accuracy, stability, and speed.

Snoop Monitor

When ***Snoop*** terminates it sends a report to ***Snoop Monitor***.

If a monitor window is currently visible it will receive the report. Otherwise the stand-alone monitor is executed and used to view the report.

That's all there is to it.

Author and License

This version 2.08 of the ***Snoop*** package.

Snoop and ***Snoop Monitor*** are written by Robert R. Marsh, S.J. and are made available free of charge. You may use them and distribute it freely as long as you make no charge for fro and you include all the accompanying files.

Snoop is just another program. I am sure you will find bugs or cases in which it fails to work properly. I cannot see how it could damage your work but be warned—in programming anything is possible. Use ***Snoop*** at your own risk. I make no legally-binding promises whatsoever about ***Snoop***'s behavior or usefulness.

If you like ***Snoop*** and find it useful you might like to make a small donation to your favorite charity. Also, I always like to hear from users with comments, suggestions, or even bug-reports. I can be contacted at:

rrm@sprynet.com

If you are reporting a bug please specify which version of ***Snoop*** you are using and which version of Delphi.

The web page below is the home ***Snoop*** and my **Q**uick **D**ata**B**ase (***QDB***) components for Delphi 1, 2, & 3. Stop by and take a look:

http://home.sprynet.com/sprynet/rrm/

How to use Snoop

1) Make sure you have the correct version of **snoop.dcu** for your version of Delphi (2 or 3) and that **snoop.dcu** is in a directory accessible by your project (*e.g.*, the "...\Delphi 3\Lib" directory).

2) Include ***Snoop*** in the **uses** clause of your program's **project** (*.dpr) file. *E.g.*,

```
program snooper;

uses
  Snoop,
  Forms,
  other in 'other.pas' {Form1};
...
```

3) Change your project's compiler and linker options:

- debug information must be ***on***.
- stack frames must be ***on***.
- map file must be ***detailed***.

If you omit any of these settings your project will halt with an error message. Remember to rebuild the project after changing the settings.

4) Save your project. The ***Snoop Monitors*** rely on information stored in the project options file which must be up to date if they are to locate your source code.

5) Build your program and run it. When your application terminates it will produce a report.

NOTES:

Optimization
Unlike some other memory-tracking products ***Snoop*** does not demand that you switch off **optimization**. If you are having problems, however, a first move should be to switch off optimization and see if it makes a difference in your case.

Symbol Information
Also, though not necessary for Snoop itself to work, sometimes to help the Snoop Monitor Expert correctly locate all source code files you should enable generation of debug **symbol information**. Since this generates some extra large files and can considerably slow the compile/link cycle this option is not recommended unless unavoidable.

Directory Information
If you are specifying a search path or output directory in the compiler options page you should make sure that these optins have been saved to disk (Save All) so that ***Snoop*** can access the information.

Reports

Snoop reports are available in two forms: as a file dump and as a display in ***Snoop Monitor***.

The file dump has the form:

```
SNOOP Report on SNOOPER.EXE generated at 5/27/1998 7:30:15 PM

At peak 3,257 pointers were allocated
amounting to 567,233 bytes of memory

Memory Not Freed:
unitname     line    bytes     contents
other         40       26     m: 03 67 34 86 12 ...

other         49       44     o: TStringList

other         50      150     s: a bit of text

aunit         16        4     m: 00 00 00 00
 aunit         34
  other         43

Unit Map:
aunit             c:\otherplace\units\aunit.pas
other             c:\otherplace\other.pas
```

Snoop reports the **peak memory** demand of your application and shows both the number of pointers allocated and the total memory consumed at that point.

Under the heading "Memory Not Freed:" **unitname** and **line** show the location of any dangling pointers. The size of the memory leak is given by **bytes**. The final column contains the **contents** of the allocated memory in a format appropriate to its type.

- Ordinary memory allocations are prefixed with **m:** and show a hex dump of up to 20 bytes of the memory's contents
- Object allocations are prefixed with **o:** and show the object's class name
- String allocations (long or AnsiStrings) are prefixed with **s:** and show the string value

The location given is the place of allocation of the memory in question **except** in the case of object allocations which are hard to track down accurately: instead ***Snoop*** reports the location of the **previous** memory allocation. In conjunction with the class name this is usually enough information to trace forward in your code and locate the dangling object.

The indented lines of the above report which only contain unit and line information give information about the call stack of the leak follow. In the case above 4 bytes of memory were allocated at line 16 of *aunit.pas* and never freed. Line 16 is within a routine which was called from line 34 of the same unit which in its turn was called from line 43 of *other.pas*. Note that call stack locations are only reported when they lie within code with available debug information.

The final section, headed "Unit Map:" lists the units referenced in the map file of your project alongside the physical source files to which they correspond. If you find any files listed incorrectly setting the compiler debug option to include "local symbols" might correct the problem

Snoop Monitor displays reports in a more accessible manner.

Debugging DLLs

Dynamic link libraries are not directly executable but run as part of a host application.

To use ***Snoop*** to track memory leaks in a DLL you need to include **snoop.dcu** in the **uses** clause of the **DLL** and ensure that the DLL projects compiler and linker options are set to generate debug information, stack frames, and a detailed map file.

When you run the DLL's host application you must supply the command line parameter /**SnoopDLL=**<*dllname*>, where <*dllname*> is the name of the DLL. The DLL must be in the same directory as the host application.

N.B. The memory-leak report generated will only cover the address space of the DLL and not the host application.

Checking for memory corruption

Snoop offers limited checking for memory corruption arising from overwritten memory.

For example, it is quite common to allocate an array with N elements (*i.e.*, from 0..N-1) and then assign to the Nth element, thus overwriting memory belonging to another object. If range-checking were enabled such a bug would be flagged immediately but if for some reason you have switched range-checking off the effect can be delayed and lead to mysterious bugs that are hard to track down.

If you start ***Snoop*** with the command-line parameter /**SnoopCorruption,** ***Snoop*** checks every block about to be freed or reallocated to see if it has been corrupted and, if so, triggers an exception. If your project is running in the IDE the exception will occur at the place where the memory was allocated. ***Snoop*** also checks the list of dangling pointers to see if any have been corrupted.

N.B. If a memory-corruption exception occurs any memory-leak reports will be spurious. You should fix the corruption problem before checking for memory leaks.

N.B.B. Memory corruption errors are nasty. Since ***Snoop*** checks for corruption when it frees blocks there is no guarantee that it will catch them **before** they cause a problem. Even when Snoop does catch one there may be a crash in the process.

Options

Location of the File Report
Report Modes
Activation at Start Up
Selective Snooping
Disabling Snoop
Internal Memory Management
Corruption Checking
Snooping on the dpr file
Command-Line Parameters

Installation

Snoop Monitor comes in two versions:

- A stand-alone application which must be placed in your **windows** directory
- A Delphi Expert which may be placed anywhere but must be registered. You register ***Snoop Monitor*** using the supplied ***SnoopReg*** program which registers the expert in its current location and also adds a registry entry locating the Snoop help file.

How to use Snoop Monitor

Snoop Monitor displays the same information as the file report but adds some capabilities:

- different kinds of memory allocation are color-coded
- the display can be sorted by clicking the head of a grid column
- a hint shows the hex dump as a string

- leak reports can be saved (File|Save As...) for future viewing (File|Open...)
- the monitor is configurable and its state can be saved
- clicking the button in the title bar rolls up the monitor

The expert version of Snoop Monitor has all the functionality of the stand-alone application and more.

Expert version

In addition to all the functionality of the stand-alone ***Snoop Monitor*** the expert version:

- lets you click on a line of the report and be taken directly to the place in your source code were a leak occurred
- when a leak has a call stack to report the unit name is followed by "..." and can be right-clicked to popup a menu with the call locations (selecting one of these jumps you into source code)

Issues

- If clicking on a line in the ***Snoop Monitor*** report doesn't jump you to the right place in your source code try setting the compiler debug option to include symbol information.

- Delphi 2 map files don't handle included files too well so the line numbers reported by ***Snoop*** may be incorrect under those circumstances.

- Under Delphi 2 ***Snoop*** may report leaks apparently arising from **snoop.pas**. These are object allocations which, in fact, occur outside **snoop.pas** but are listed under the last memory allocation. You can ignore such object allocations but if you discover memory or string allocations attributed to snoop.pas I would be glad to hear of them.

Location of the File Report

By default the ***Snoop*** report is sent to a file with the same name as the project being debugged and the extension **snp**. If you wish to change the name or location of the report file you can simply assign the new filename to the **SnoopLogFile** variable somewhere in your program, *e.g.*:

```
SnoopLogFile:='AnotherFileName.txt';
```

Report Modes

By default ***Snoop*** generates both a file report and report for ***Snoop Monitor.*** Each of these modes of reporting can be suppressed *via* command line switches.

/**SnoopNoFileShow**	suppresses the file report
/**SnoopNoMonShow**	suppresses the monitor report

Activation at Start Up

In normal operation the ***Snoop*** memory manager is actively snooping on memory allocations from (nearly) the very first moments of your program's execution. If this is undesirable it can be overridden by supplying as a command-line parameter /**SnoopDormant,** *e.g.*,

```
snooper /snoopdormant;
```

If you initiate your program with snooping deactivated in this way you can switch snooping on selectively.

Selective Snooping

The ***Snoop*** memory manager can be switched off and on via the **Switch** procedure. The current status of the memory manager is returned by the **Snooping** function. To use these routines within a unit ***Snoop*** must be referenced in the unit's **uses** clause.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  MyObject: TMyObject;
begin
  Switch(Onn);
  MyObject:=TMyObject.Create;
  MyObject.DoStuff;
  MyObject.Free;
  Switch(Off);
end;
```

Notice the predefined constants **Onn** (not a misspelling!) and **Off** (with the values **true** and **false**, respectively).

You need to take care when switching ***Snoop*** on and off that you place the switches *symmetrically*, *i.e.*, in such a way that you expect the program's memory to be in the same state at each point. If you mismatch them the results will be misleading.

Disabling Snoop

Because ***Snoop*** degrades the performance of your program it is important that ***Snoop*** not be active in the release version of your application. You can, of course, remove any references to ***Snoop*** from your program code but, even without changing your source, ***Snoop*** will recognize when it is running outside the Delphi IDE and avoid loading, so cutting out any performance hit. This behavior can be overridden *via* the command line.

If you want ***Snoop*** to be loaded when running *outside* the IDE use the /**SnoopAlive** parameter. If you want ***Snoop*** *not* to be loaded when running *inside* the IDE use the /**SnoopDead** parameter.

N.B. The /**SnoopDead** parameter has a different effect from the /**SnoopDormant** parameter. **SnoopDormant** loads the ***Snoop*** memory manager but leaves it dormant (ready to be made active using **Switch**). **SnoopDead** doesn't load the memory manager at all.

Internal Memory Management

Snoop expects initially to track 10,000 pointers but whenever it runs out of space it doubles the size of its internal structures. Repeated doublings will impose a performance hit which is best avoided by persuading ***Snoop*** to prepare to track the correct number of pointers from the start *via* the command-line parameter /**SnoopNum**, *e.g.*, the command line below prepares ***Snoop*** to track 200,000 pointers.

```
snooper /snoopnum=200000
```

You can use ***Snoop***'s report of peak pointer allocation to judge a good value for /**SnoopNum**.

If memory is tight, however, you might want to start ***Snoop*** with smaller internal structures. You cannot, though, make /**SnoopNum** less than 1,000—which corresponds to roughly 40K of memory used internally.

Corruption Checking

Corruption checking can be turned on and off selectively *via* the **CheckForCorruption** procedure, *e.g.*,

```
CheckForCorruption(Onn);
```

Snooping on the dpr file

Generally any leaks which occur "beneath" the level of your own code will percolate upwards and be captured. Most of these leaks end up registering in the project (dpr) file where they are of very little diagnostic use. By default ***Snoop*** suppresses reporting of all leaks in the dpr file but there may be circumstances where you need to see what is going on in there. The /**SnoopProject** command line option turns on the reporting of leaks in the dpr file.

Command-Line Parameters

Parameter	Equivalent	Purpose
SnoopAlive	$A	Load ***Snoop*** even if outside IDE
SnoopDead	$D	Don't load ***Snoop*** even if in IDE
SnoopDormant	$N	***Snoop*** is dormant until switched on
NoSnooping	$N	same as SnoopDormant
SnoopCorruption	$C	When active check for corruption
SnoopNum=nnnn		Adjust memory requirements
SnoopDLL=<dllname>		Track the named DLL
SnoopNoFileShow	$F	Suppress the file report
SnoopNoMonShow	$M	Suppress the monitor report
SnoopProject	$P	Enable report of leaks in dpr file

Snoop's parameters can appear anywhere on the command line, even mixed among the parameters of your program. If conflicting switches are present the latest in the command line takes effect. Note that some of the parameters have a hierarchy.

SnoopAlive/SnoopDead > NoSnooping/SnoopDormant > SnoopCorruption

For example, SnoopCorruption achieves nothing if SnoopDead is present.

The standard parameters must be separated by spaces or slashes but the $-equivalents can be collapsed into a single block, *e.g.*, **$ANCF** is equivalent to /**SnoopAlive** /**SnoopDormant** /**SnoopCorruption** /**SnoopNoFileShow**

Version History

This is version 2.08

2.08 Released July 25, 1998
Fixed an obscure bug to do with D2 map files.

2.07 Fixed a problem with Snoop Monitor (not) appearing on screen in some circumstances
Continued to improve the location of source files
Note: D2's map files seem to be less reliable than D3 ... I'm working on it

2.06 Fixed a problem with D2 map files ("map corrupted")
Fixed problem jumping into source files in Search Path

2.05 Released June 1, 1998
Fixed problem with jumping into source code from call stack popup list
Fixed call stack information for reallocations

2.04 Added command-line option to show leaks in the dpr file

2.03 Added a roll-up button in the Snoop Monitor title bar
Added extra call stack information.

2.02 Corrected a problem with the memory dump
Extended corruption checking to dangling pointers

2.01 Released May 16, 1998
Fixed a bug in expert not locating source files in some circumstances

2.0 Released May 16, 1998
Complete re-write with new algorithm.
More accurate.
Distinguishes object creation from ordinary memory allocation.
Improved Snoop Monitor

1.4 Released April 30, 1998
Improved performance
Tracks peak memory allocation
Works with DLL files
Changed behavior of /SnoopNum
Added ***Snoop Monitor*** and monitor reporting
Added /SnoopNoFileShow and /SnoopNoMinShow

1.3 Released October 25, 1997
Made thread-safe
Added /SnoopNum parameter

1.2 Released May 24, 1997
Versions for both Delphi 2 and Delphi 3
Filters out leak warnings for '**end**.' statements

1.11 Fixed a rare list-index problem

1.10 Released May 18, 1997
Fixed a bug with the /NoSnooping parameter
No longer needs to have stack frames enabled

Added overwrite checking (/SnoopCorruption)
Now possible to leave Snoop in finished programs
SnoopLogFile now defaults to <YourExeName>.snp
Made some ***considerable*** speed enhancements

1.01 First public version—released April 20, 1997

1.00 First private version