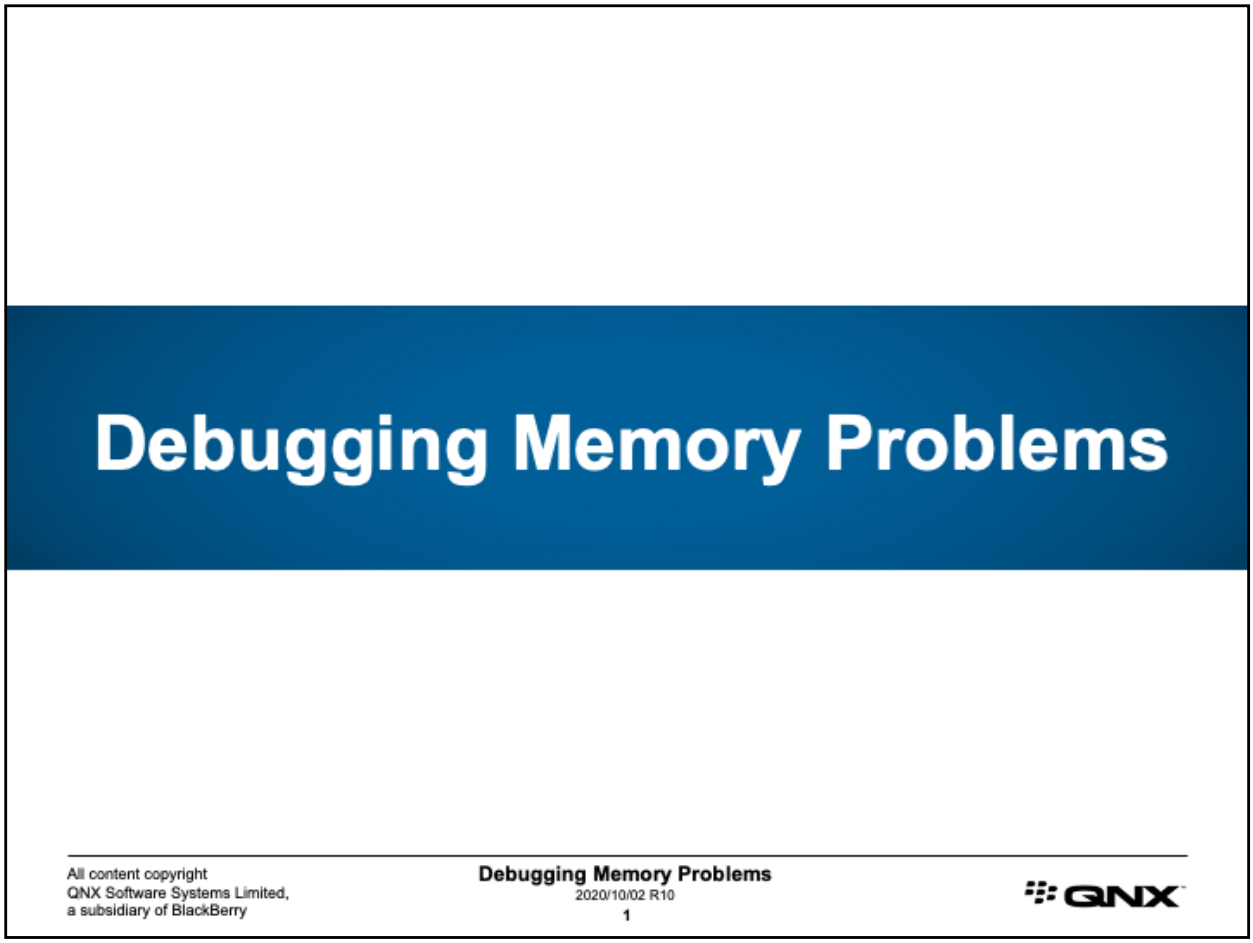


NOTES:

Introduction

You will learn:

- some techniques for finding:
  - memory corruption
  - excessive memory consumption
  - memory leaks

NOTES:

Debugging Memory Problems

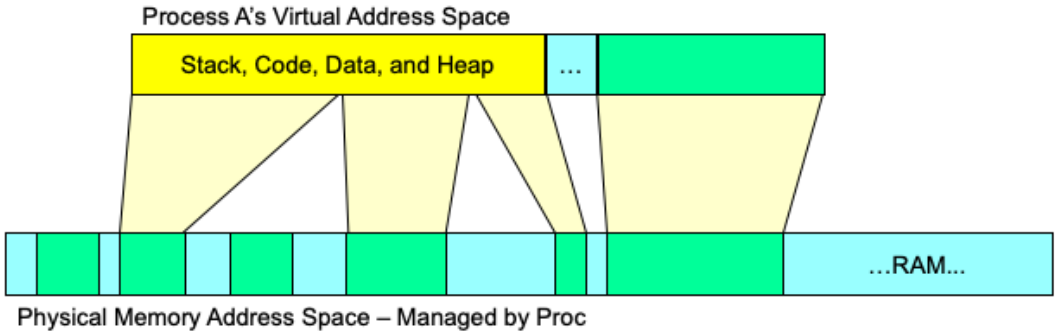
Topics:

- **Overview**
- **Finding Memory Corruption**
- **Excessive Memory Usage**
- **Finding Memory Leaks**
- **Importing and Exporting**
- **Conclusion**

NOTES:

Overview

Virtual to Physical Memory Mapping:

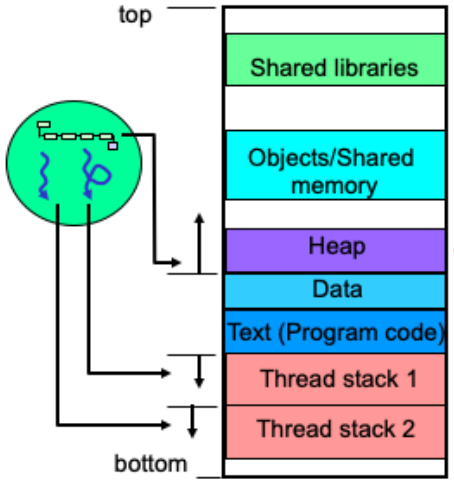


- all addresses you deal with in your process are virtual
- all of the memory related tools provide addresses and information about virtual addresses
- all allocations from system memory to processes is in multiples of page size (4k)

NOTES:

Overview

Virtual Address space of a process without ASLR:



for dynamic memory, what this section will be addressing

for local variables, function parameters, etc

- since each process has its own virtual address space, each process has its own set of these sections
- sizes and addresses for the sections are shown in:
  System Information perspective→Memory Information view

QNX

NOTES:

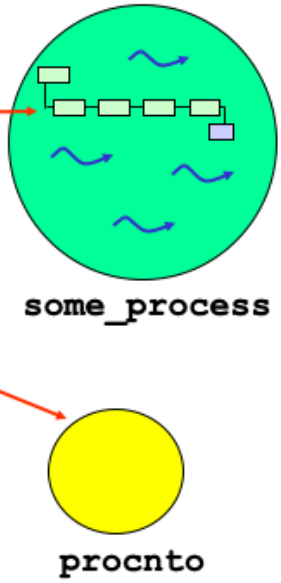
Address Space Layout Randomization (ASLR) is enabled by default. The diagram shows the layout without ASLR. With ASLR, the same sections will exist – but they will be randomly arranged within the address space of the process.

malloc() overview

What malloc() does (simplified):

e.g.: **ptr = malloc(114); //request 114 bytes**

- search the heap for acceptable memory to satisfy this request
- if satisfactory memory found
  - return address to caller
- else
  - ask procnto to grow heap (request one or more 4K chunks of memory from OS)
  - and return address of chunk from newly grown heap



NOTES:

The complexities come in how does malloc() carve up and recombine carved chunks of the heap. There are many different choices for how to do this, generally giving different trade-offs of speed for different situations (handling large single allocations, handling small allocation, quickly handling reallocations), overhead (how much data is burned in the library to represent the state of the heap), and fragmentation. No one algorithm will be best in every situation, and just about any algorithm can look really awful if you find the right test. Substituting a different malloc() library can help performance, as can tuning the existing one, which can be done through a set of environment variables.

Overview

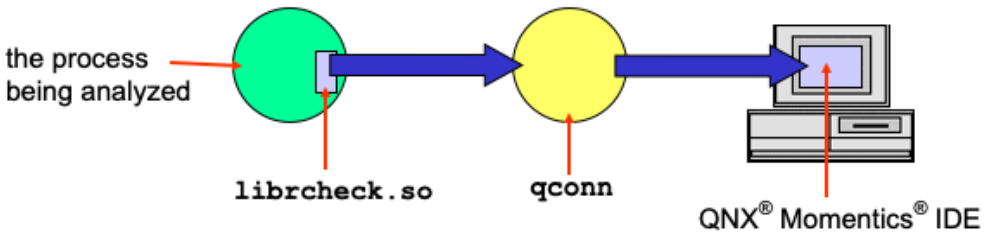
In this section, we'll look at debugging memory problems with:

- System Information perspective
  - relevant views:
    - System Resources view
    - Malloc Information view
- QNX Analysis perspective
  - QNX Memory Analysis

NOTES:

librcheck.so

QNX memory analysis relies on `librcheck`:



– when a *malloc()* related event happens, the *malloc()* code in **`librcheck.so`** will pass the data on to **`qconn`**, which will pass it on to the IDE

☞ make sure you have **`librcheck.so`** on your target (located somewhere in LD_LIBRARY_PATH)

NOTES:

librcheck.so

Memory related function are also replaced with instrumented versions:

- e.g.: *memcpy(), strcpy(), strcmp(), strlen(), memset()*
- parameters will be:
  - range checked
  - checked for NULL pointers

NOTES:

librcheck.so

The views in System Information don't require `librcheck.so`:

- Malloc Information and System Resources use instrumentation from the standard memory allocator and `procnto`

NOTES:

Debugging Memory Problems

Topics:

- Overview
- → Finding Memory Corruption
- Excessive Memory Usage
- Finding Memory Leaks
- Importing and Exporting
- Conclusion

NOTES:

Debugging Memory Corruption

For a memory area allocated with *malloc()* or *new*, you can catch:

- access past the end
- access before the beginning
- doing a *free()* with an invalid address
- bad values passed into memory related library functions, e.g. *memcpy()* and *strcpy()*

NOTES:

Debugging Memory Corruption

When a memory error occurs:

- IDE can:
  - report the error and continue
    - log memory error without stopping the process
    - ☞ if the error corrupts memory block headers, it may affect:
      - the execution of the program
      - validity of further memory events
  - terminate the process

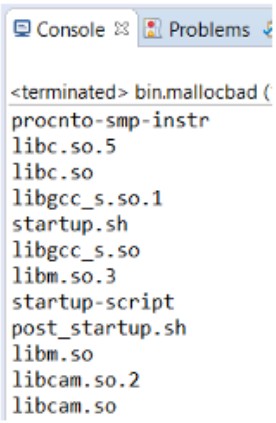
NOTES:

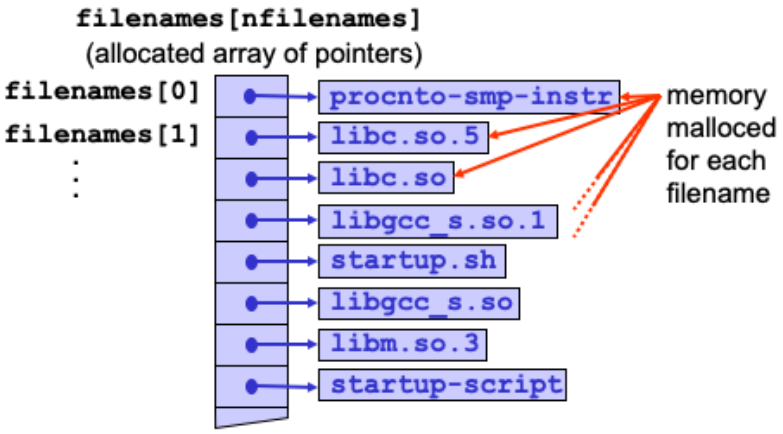
There is also a ‘launch the debugger’ option, but it is not functioning correctly in SDP 7.1 (or SDP 7.0).

Debugging Memory Corruption - Demo

Memory corruption demo:

- run **`mallocbad`** (in the **`memory_problems`** project)
- it displays a list of filenames in the Console view.
- it allocates an array of pointers to strings (character arrays) for storing the list





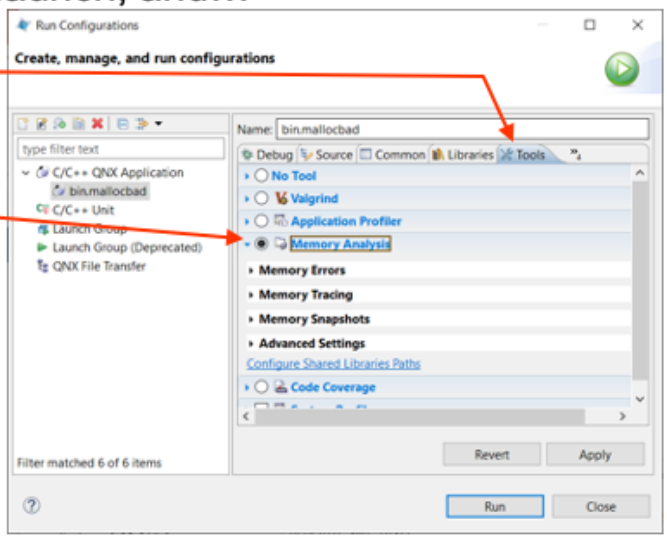
NOTES:

Launching Applications with Memory Analysis

To launch an application for doing Memory Analysis:

- create a normal launch, and…

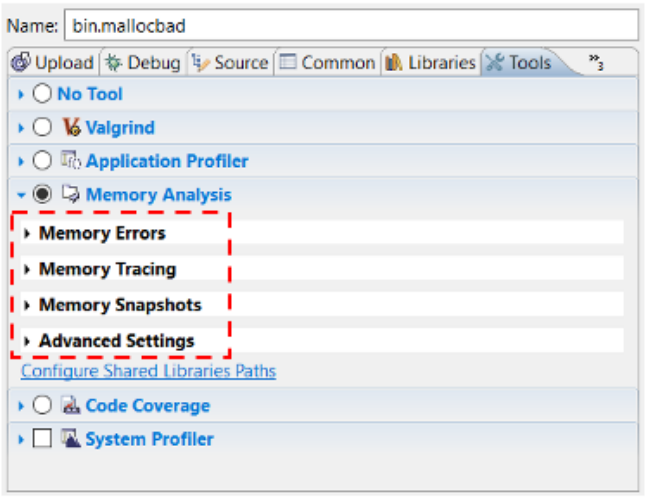
1. go to the Tools tab
2. check Memory Analysis



NOTES:

Launching Applications with Memory Analysis

There are various options to configure:

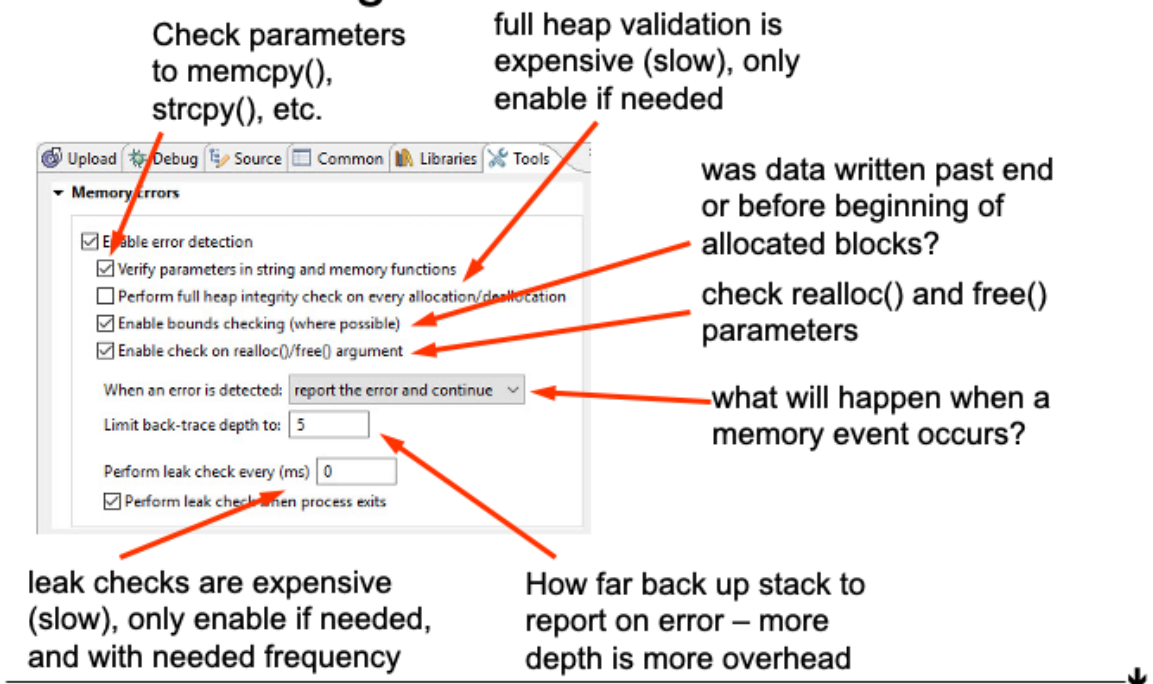


- in this section we'll deal with the Memory Errors configuration
- leak detection will use memory tracing

NOTES:

Debugging Memory Corruption - Demo

Choose settings:



NOTES:

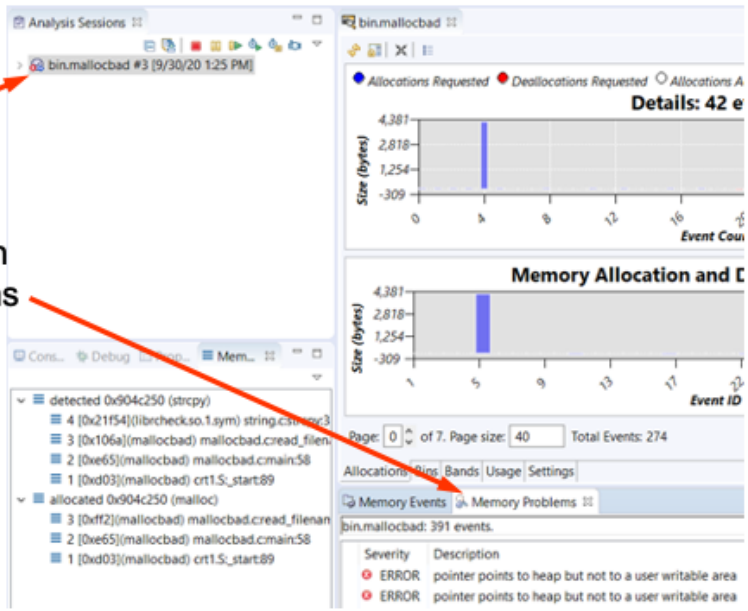
Several of the choices here are overhead vs data collection. The more data you collect, the closer you can pinpoint your problem – but the slower your program will run, and the longer it will take to get to the problem point. As memory leaks or errors may take a while to reproduce, this sort of tuning of the debug parameters can be quite important.

QNX Analysis Perspective

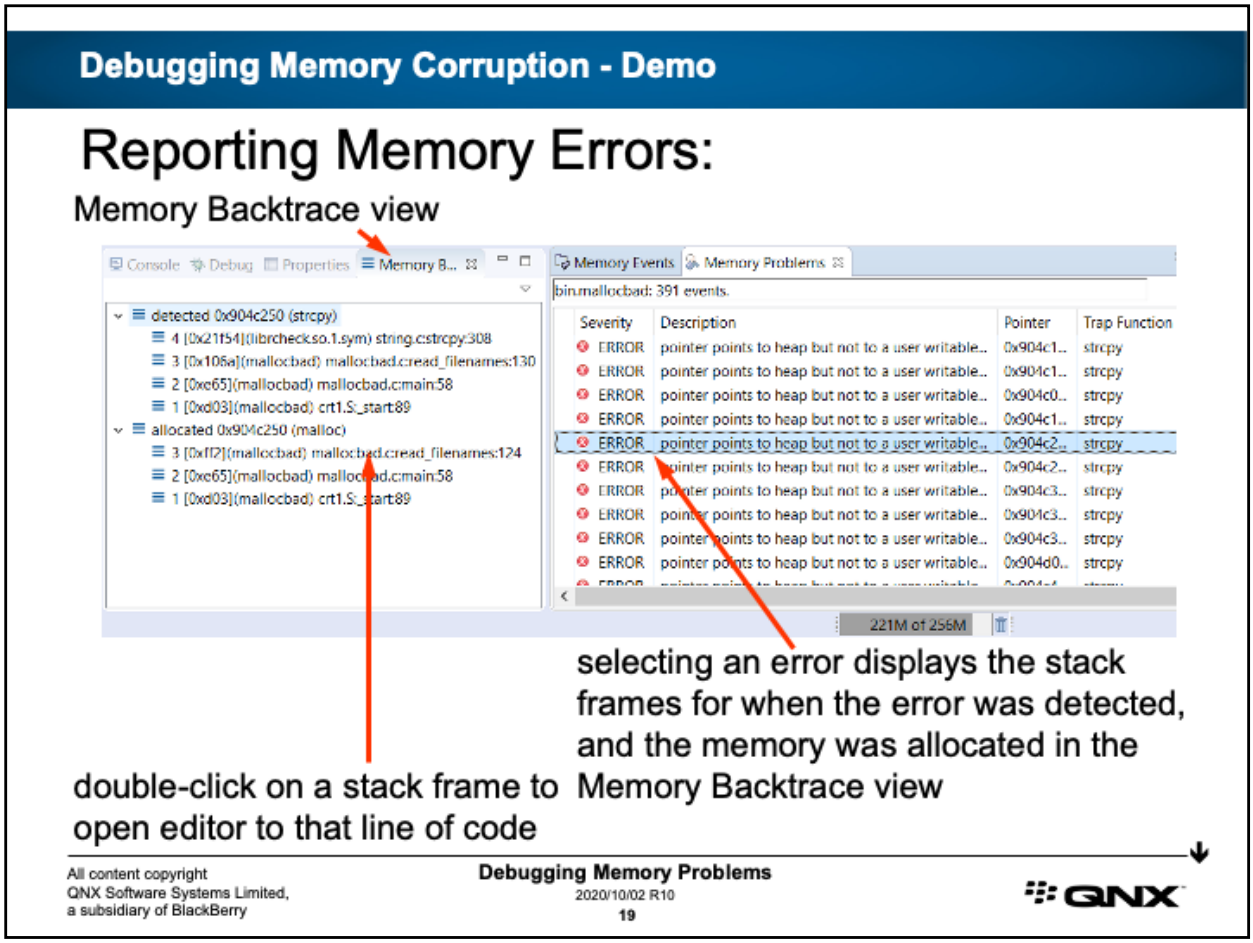
Open the QNX Analysis perspective:

double-click on a session to open it

the errors will be shown in the Memory Problems view



NOTES:



NOTES:

For many errors the Memory Backtrace will have two entries. The “detected” entry is where the error itself was trapped. The “allocated” backtrace is where the block that triggered the error was allocated (e.g. the *malloc()* call that got that block).

If the Memory Backtrace View is not open by default, open it from the Window -> Show View -> Memory Backtrace drop-down menu

Debugging Memory Corruption - Demo

Examples of memory problems/errors:

- "pointer points to heap, but not to a user writable area"
  - attempt to write outside the memory that was allocated e.g.:

```
void *ptr;
ptr = malloc( 4 );
memcpy( ptr, &object, 8 );
```

- "data has been written outside allocated memory block"
  - similar to above, but caught at free() time, e.g.:

```
int* ptr;
ptr = malloc(8 * sizeof(int));
for (i = 0; i <= 8; i++ )
  ptr[i] = i;
free( ptr );
```


NOTES:

EXERCISE

Finding memory corruption:

- run **`mallocbad`** (in your **`memory_problems`** project) using Memory Analysis, as shown in the preceding slides
- use the tool to find the offending (error triggering) lines of code
  - fix the problems and run again
  - have the errors gone away?

NOTES:

Debugging Memory Problems

Topics:

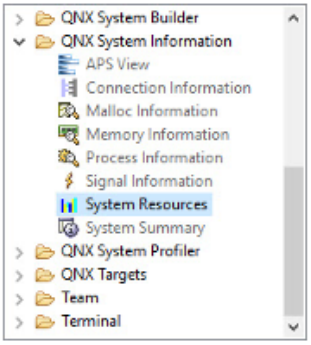
- Overview
- Finding Memory Corruption
- → Excessive Memory Usage
- Finding Memory Leaks
- Importing and Exporting
- Conclusion

NOTES:

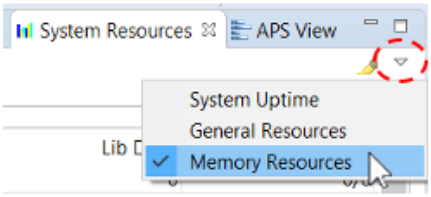
Excessive Memory Consumption

System is running low on RAM unexpectedly:

- who is the culprit?
- System Resources view can help:
  - usually used from System Information perspective
  - shown by default
  - open by: Window→Show View→System Resources



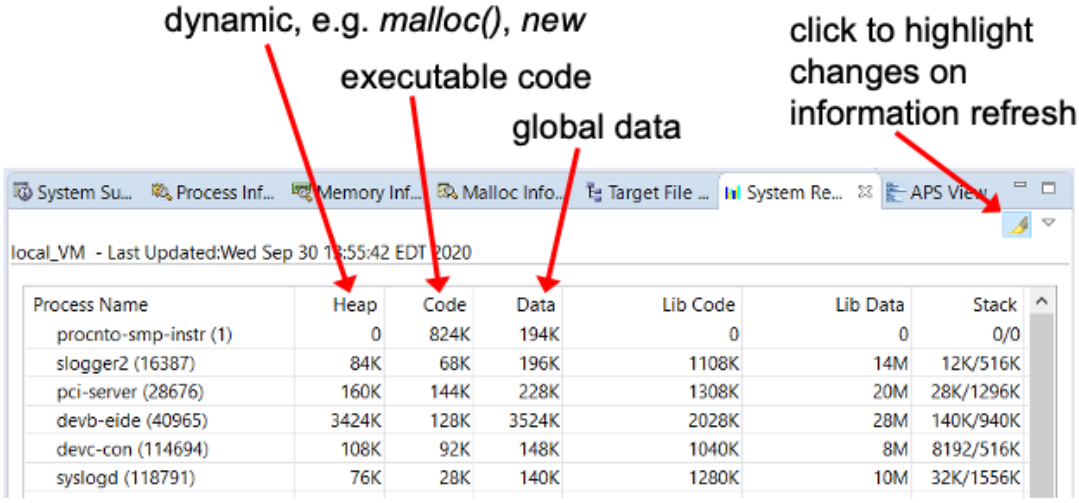
switch to 'Memory Resources':



NOTES:

Excessive Memory Consumption

If a process is using excessive memory, it'll probably be dynamic, which comes from the heap:



NOTES:

EXERCISE

Memory usage:

- try out the System Resources view (Memory Resources section)
- which processes are using the most memory?

NOTES:

Debugging Memory Problems

Topics:

- Overview
- Finding Memory Corruption
- Excessive Memory Usage
- → Finding Memory Leaks
- Importing and Exporting
- Conclusion

NOTES:

Finding Memory Leaks

The IDE can help you find memory leaks:

- caused by never freeing or deleting memory that was allocated using:
- *malloc()*, *calloc()*, *realloc()* functions
- *new* operator
- two types of memory leaks:
  - pointer still exists, but should have been freed (e.g. entry in a client list for a client that has disconnected)
  - no pointer exists to the memory at all (e.g. entry removed from linked list, but memory not freed)
- the following tools can help find leaks:
  - Malloc Information (System Information perspective)
  - QNX Analysis perspective (Memory Analysis sessions)

NOTES:

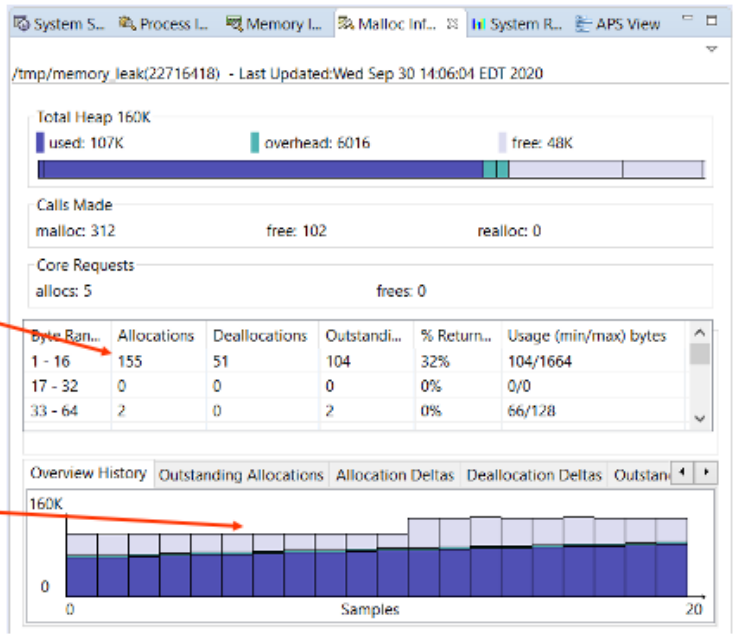
Finding Memory Leaks - Malloc Information view

Using the Malloc Information view:

in the Target Navigator view, select the process you want to examine, and then look in the Malloc Information view...

the number of mallocs keep increasing each time the view refreshes but the free count will not (or not as quickly)

this will give you a clear indication that there's a problem

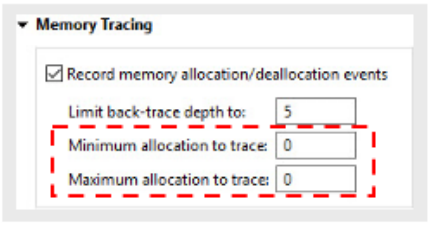


NOTES:

Finding Memory Leaks

The Memory Analysis editor has a pane that:

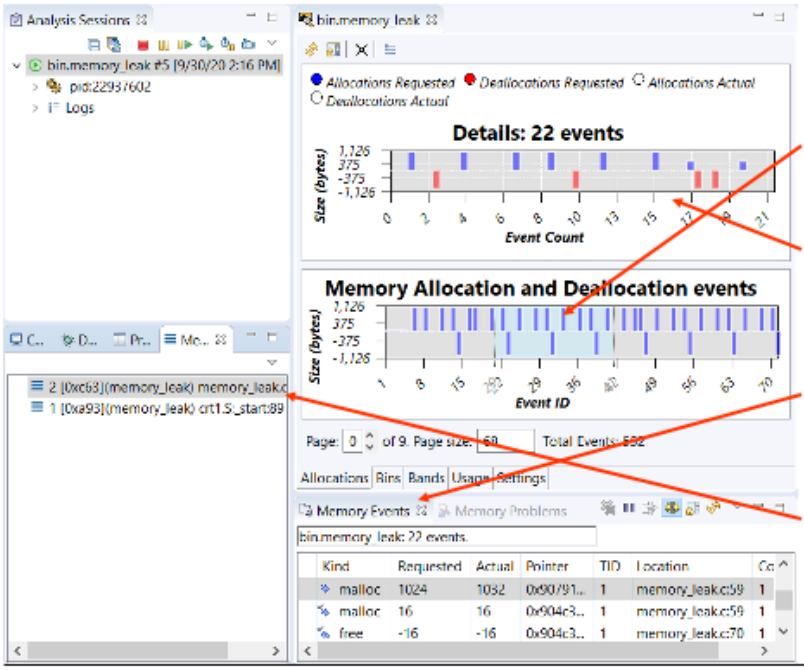
- shows a log of every allocation and de-allocation
- this is every *malloc(), calloc(), realloc()* and *free()*
- *new* and *delete* show up as *malloc()* and *free()*
- from the size data in the Malloc Information view you may wish to filter your memory tracing data collection:
  - less overhead
  - collect data for longer



NOTES:

Finding Memory Leaks

The Memory Analysis editor – Allocations:



Select an area in the Overview

Selected Area will appear in graphic details and Memory Events view

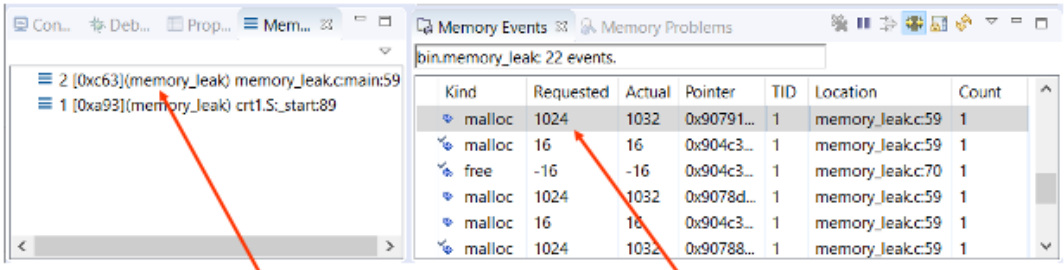
Selecting an event will show the back trace.

NOTES:

Finding Memory Leaks

The Memory Events view:

- log of every allocation and de-allocation



double-click on a stack frame to open editor

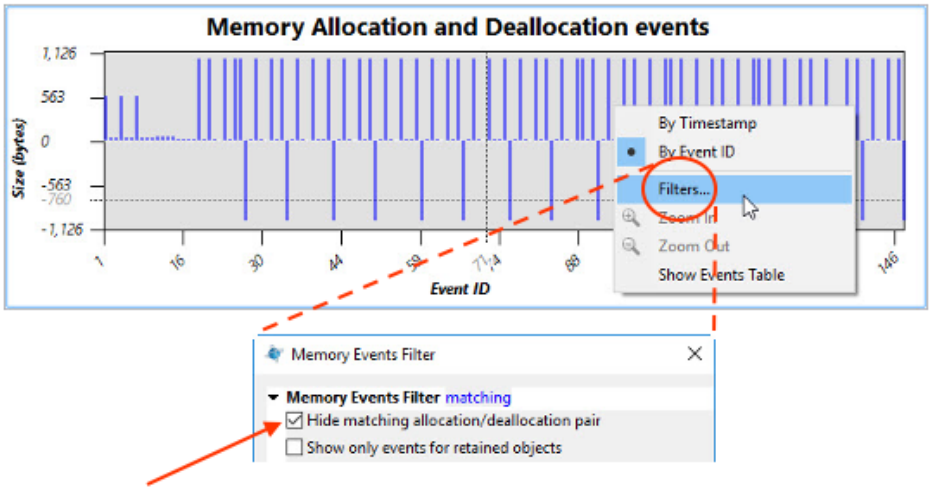
selecting an item gives a stack trace

- allocation
- de-allocation
- allocation that has a matching de-allocation
- de-allocation that has a matching allocation

NOTES:

Finding Memory Leaks – Memory Analysis editor

To see the allocations that haven't been freed:

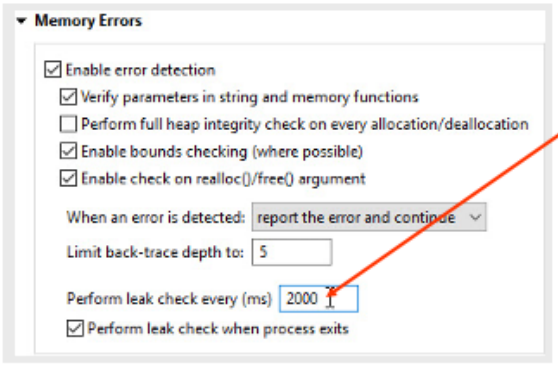


– checking 'Hide matching allocation/deallocation pair' will more clearly show allocations that don't have a matching free

NOTES:

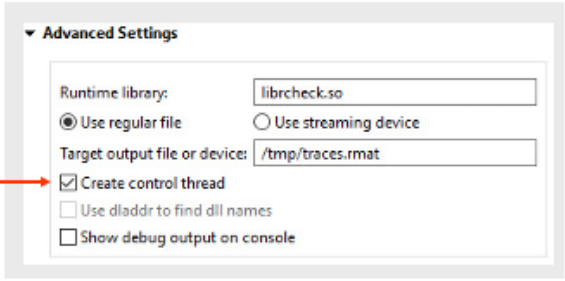
Finding Memory Leaks – Leak Checking

You can enable more leak checking:



add leak checking interval

required for doing regular leak checks at runtime



NOTES:

Finding Memory Leaks – Leak checking

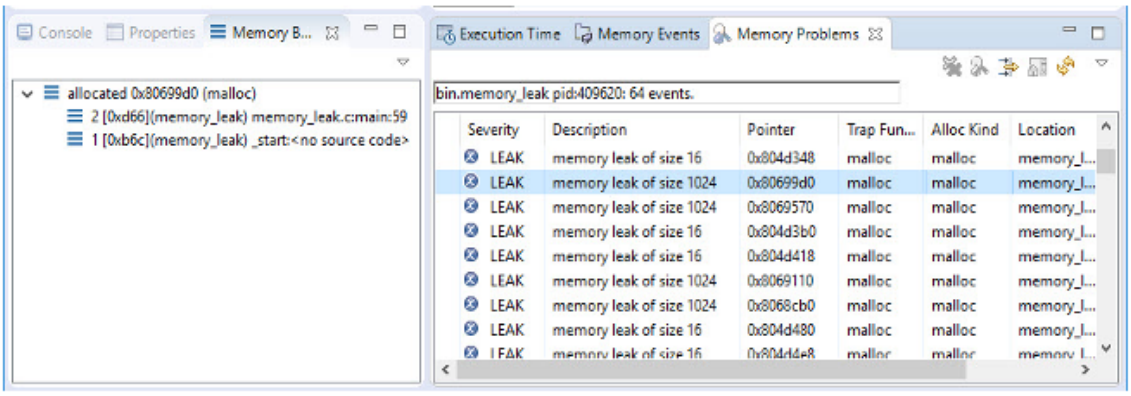
Leak checking is expensive:

- this does a full scan of the process address space, checking for each allocated block to make sure there is a pointer to it
  - like a “garbage collection” scan
  - expensive
    - default is to only do on exit
    - don’t do too frequently

NOTES:

Finding Memory Leaks - Leaks

Memory leaks appear in the Memory Problems view:



NOTES:

EXERCISE

Finding memory leaks:

- run the **`memory_leak`** executable (it's in your **`memory_problems`** project)
- go to the Malloc Information view
- select the leaky process in the Target Navigator view and watch **`memory_leak`** leak memory

continued...

NOTES:

EXERCISE

Finding memory leaks (continued):

- kill **`memory_leak`** and this time launch it with the Memory Analysis tool
  - enable leak checking
- look at the results, and find where it is leaking using the Memory Analysis editor

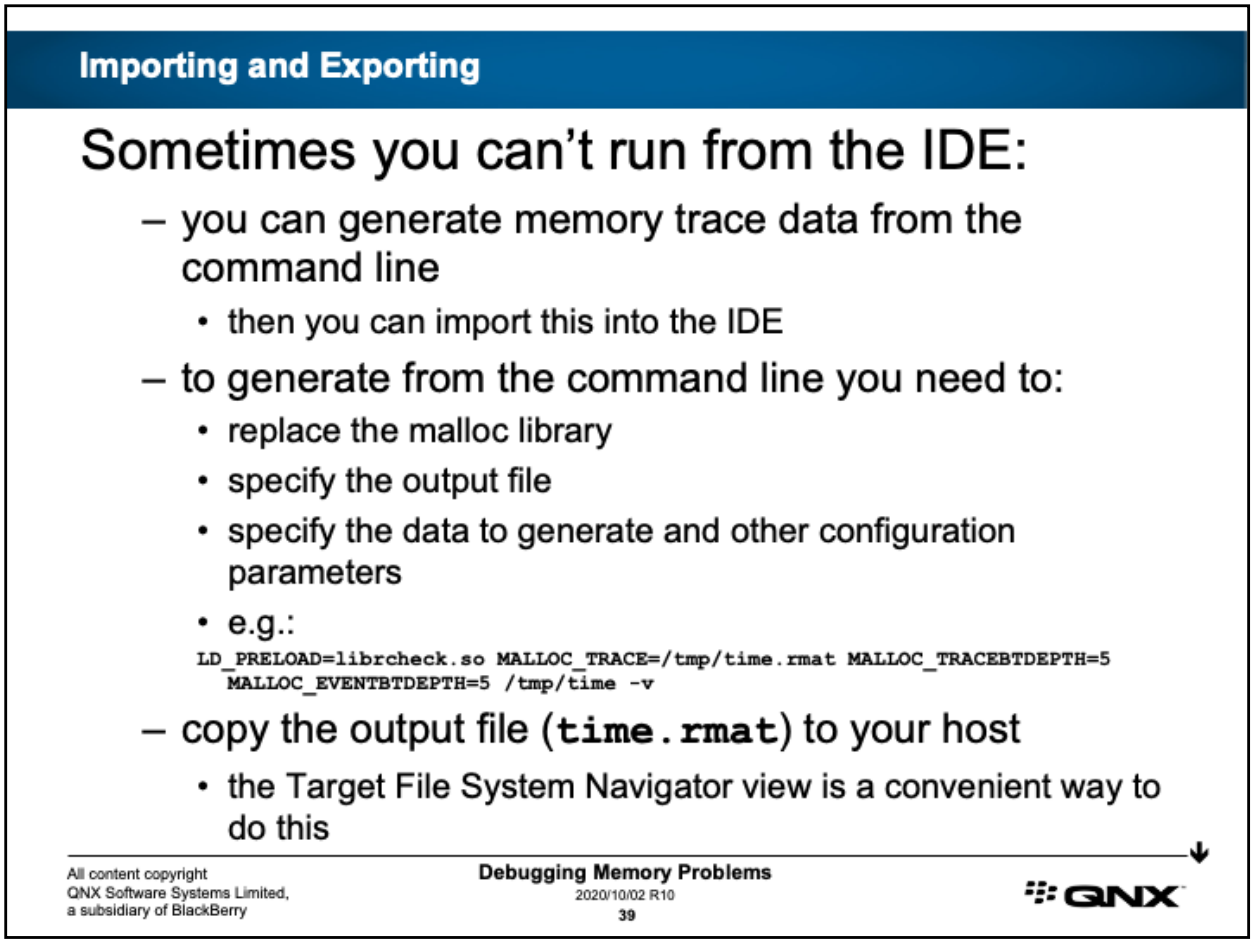
NOTES:

Debugging Memory Problems

Topics:

- Overview
- Finding Memory Corruption
- Excessive Memory Usage
- Finding Memory Leaks
- → Importing and Exporting
- Conclusion

NOTES:



NOTES:

One trick for figuring out the correct environment variables to pass is to launch a simple program for doing memory analysis from the IDE. While that program is running, go to the System Information perspective and select the it in the Target Navigator view. Then look in the Process Information view for the section containing the program's environment variables. Copy the environment variables that are related to memory analysis from there (most have **`MALLOC_`** in their name) and paste them into a text file which you'll turn into a script for running the program later without the IDE.

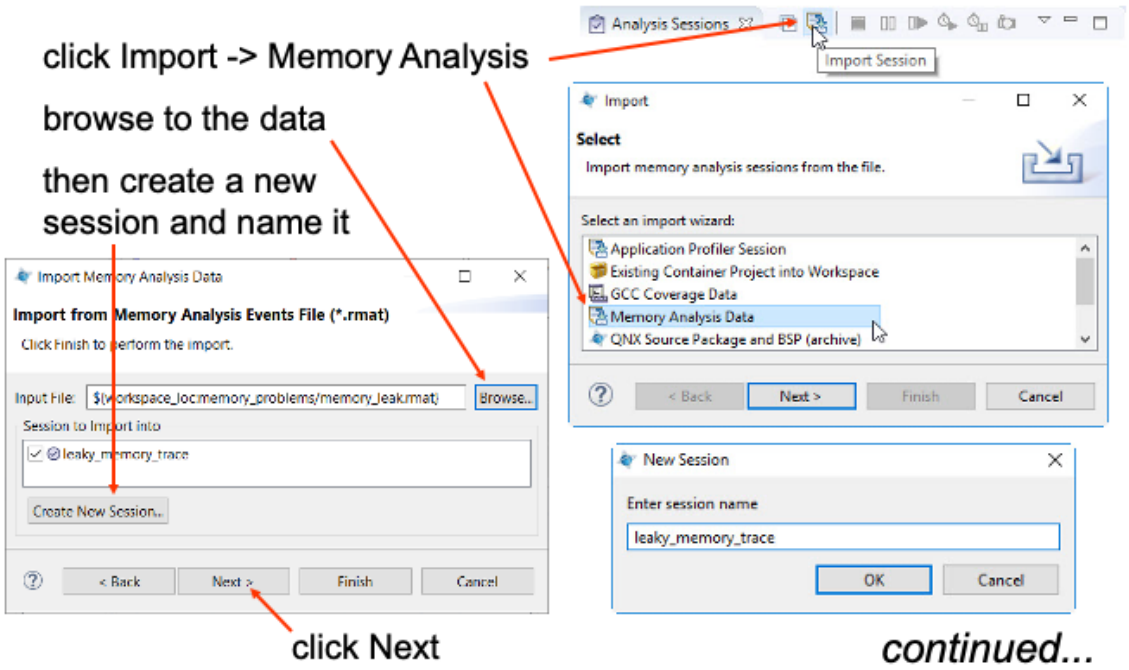
As an alternative to using the System Information perspective, you can get the environment variables by running:

`pidin -p` *program_name* **`env >> my_script_file`**

on your target and redirecting its output to your script file.

Importing

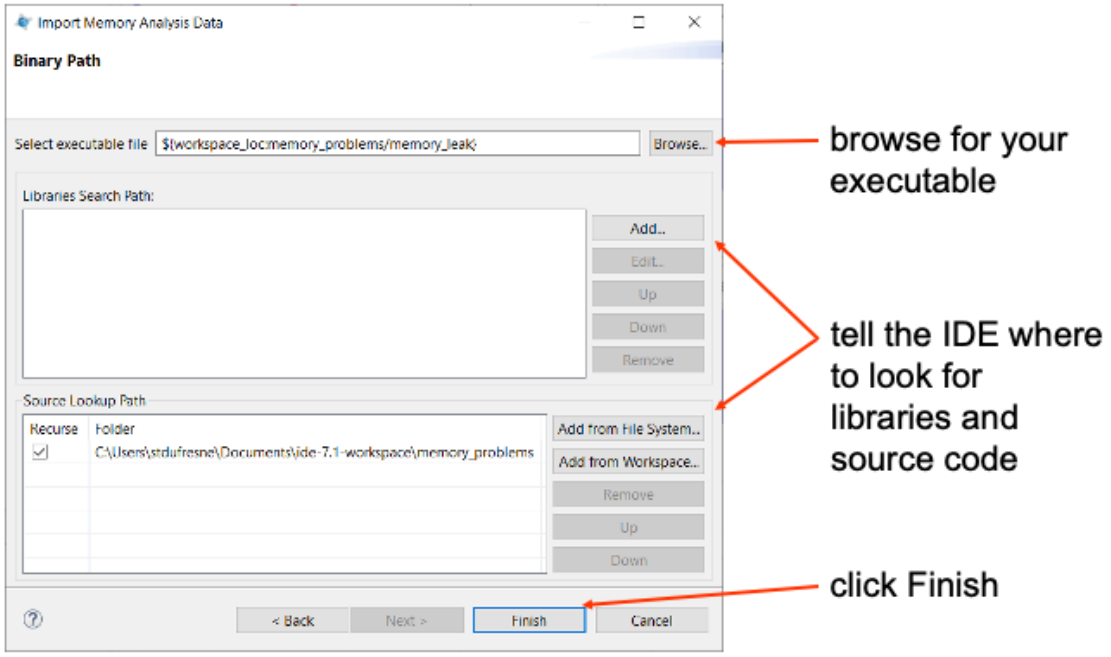
In the Analysis Sessions view:



NOTES:

Importing

Importing (continued):



browse for your executable

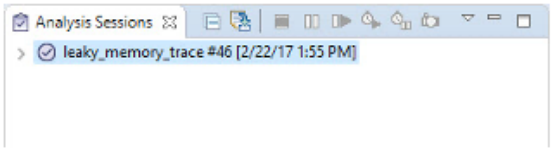
tell the IDE where to look for libraries and source code

click Finish

NOTES:

Importing

Your new session will appear in the sessions view:



- you can open it just like any other session
  - the default data generated is different
  - the library can give you some help:
    `LD_PRELOAD=librcheck.so MALLOC_HELP=1 /tmp/time`
    or look at *mallopt()* in the Library Reference manual
  - or try configuring an IDE session for what you want, then run your program and look in the System Information perspective's Process Information view for the environment it generates

NOTES:

One trick for figuring out the correct environment variables to pass is to launch a simple program for doing memory analysis from the IDE. While that program is running, go to the System Information perspective and select the it in the Target Navigator view. Then look in the Process Information view for the section containing the program's environment variables. Copy the environment variables that are related to memory analysis from there (most have **`MALLOC_`** in their name) and paste them into a text file which you'll turn into a script for running the program later without the IDE.

As an alternative to using the System Information perspective, you can get the environment variables by running:

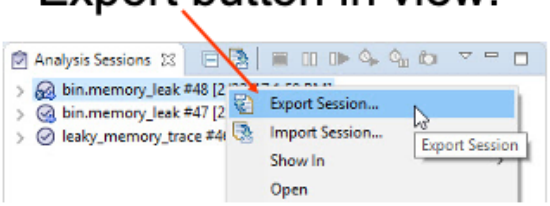
`pidin -p` *program_name* **`env >> my_script_file`**

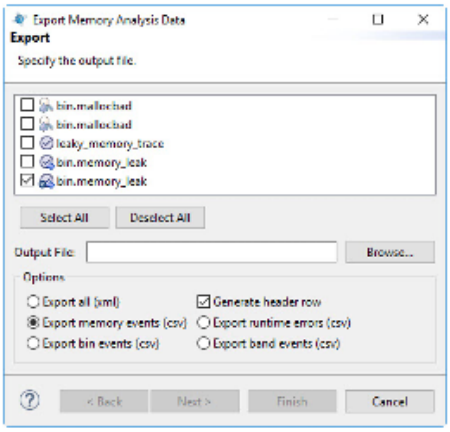
on your target and redirecting its output to your script file.

Exporting

You can export the memory analysis data:

- generates a CSV (comma separated value) file
  - standard form for importing into Excel or other spreadsheet software
  - can be used for further analysis
- Right-click, then Export button in view:





NOTES:

Debugging Memory Problems

Topics:

- Overview
- Finding Memory Corruption
- Excessive Memory Usage
- Finding Memory Leaks
- Importing and Exporting
- → Conclusion

NOTES:

Conclusion

You learned how to find:

- memory corruption
- processes that may be using excessive memory
- memory leaks

NOTES: