Debugging Memory Problems

You will learn:

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

Topics:

→ **Overview**

Finding Memory Corruption

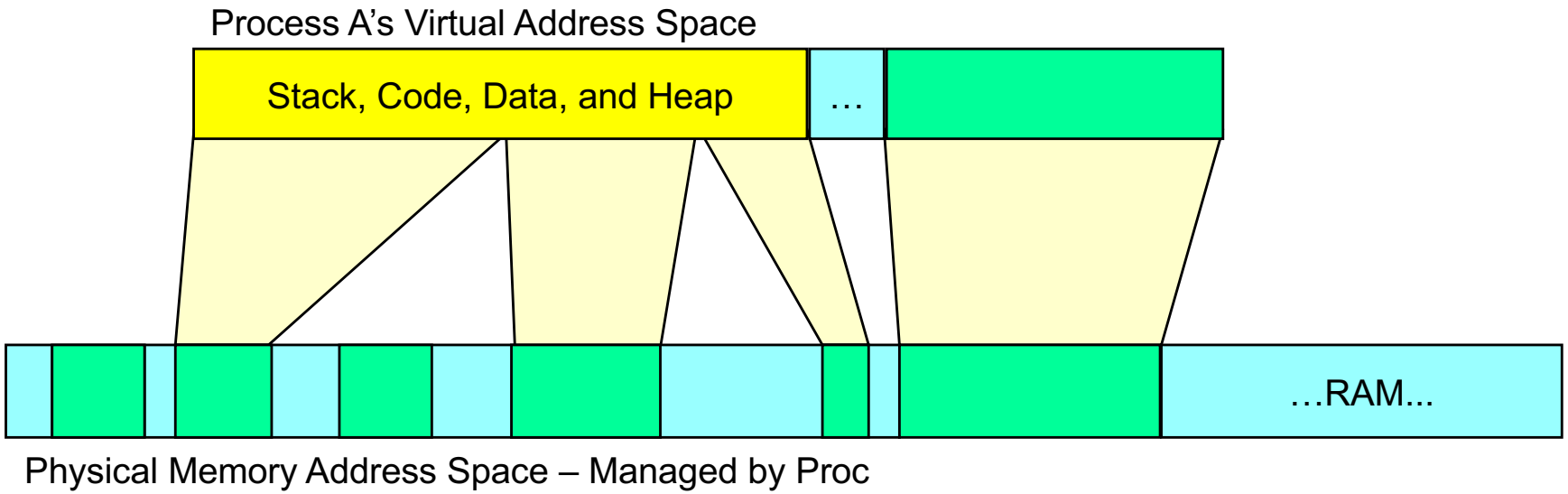
Excessive Memory Usage

Finding Memory Leaks

Importing and Exporting

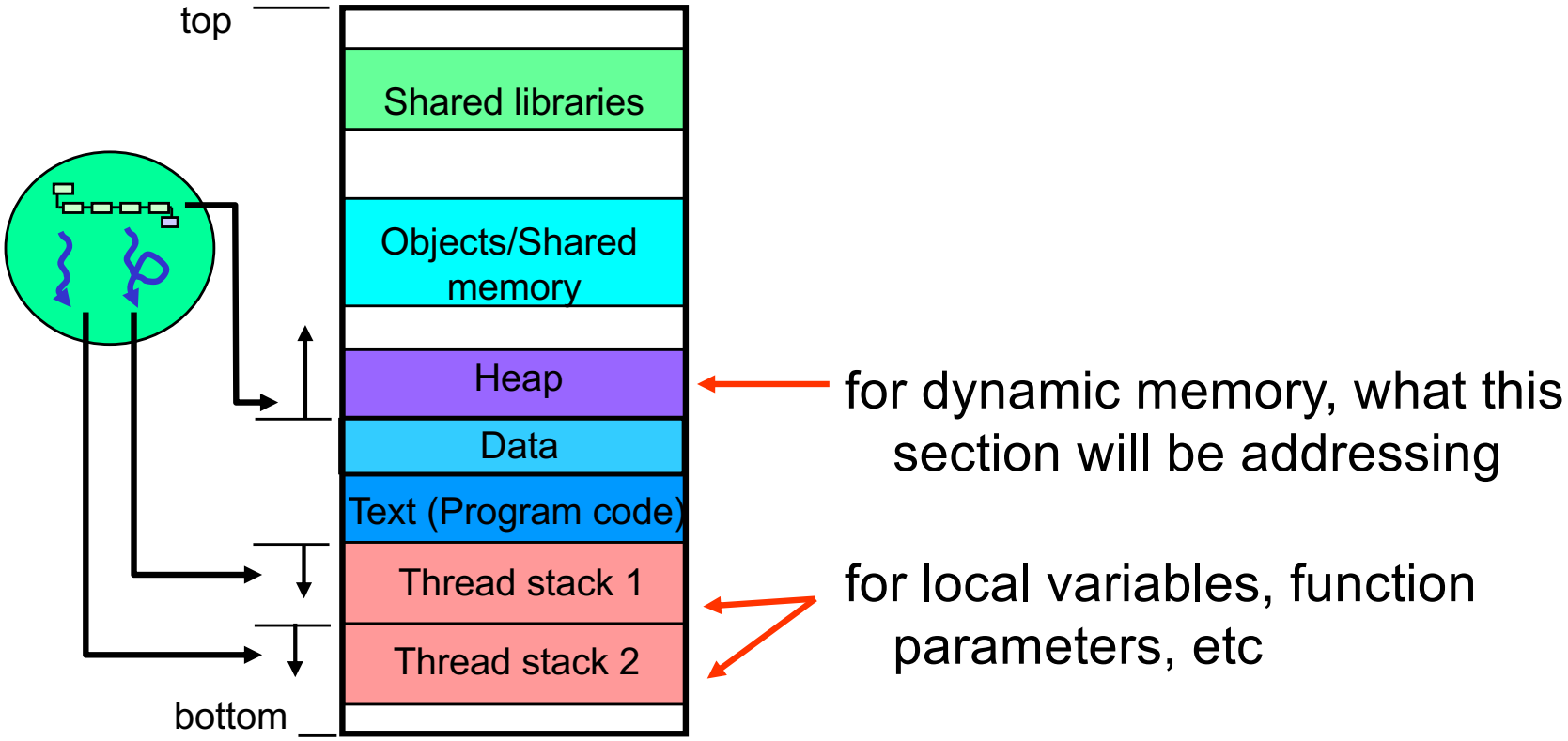
Conclusion

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)

Virtual Address space of a process without ASLR:



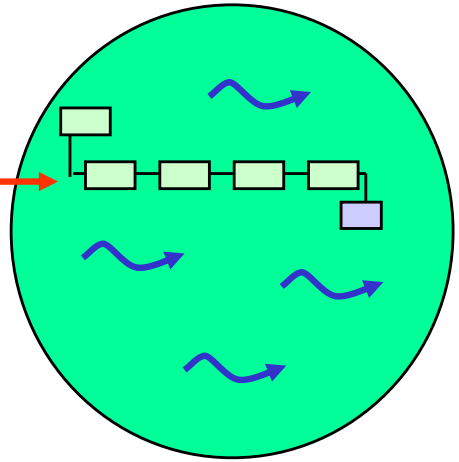
- since each process has its own virtual address space, each process has its own set of these sections
- sizes and addresses for the sections are shown in:

  System Information perspective→Memory Information view

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



some_process

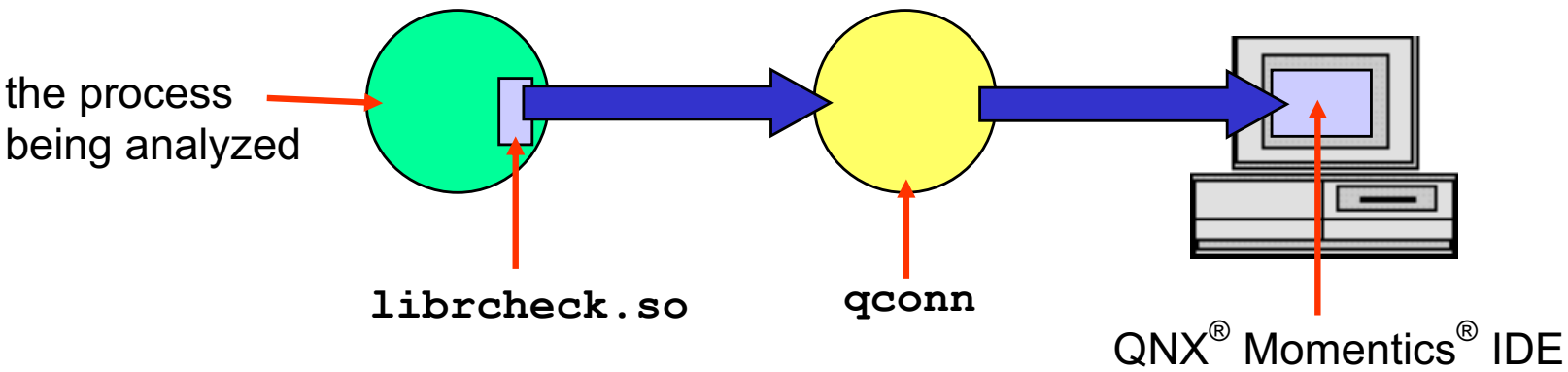
procnto

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

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)

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

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`**

Topics:

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

Debugging Memory Corruption

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

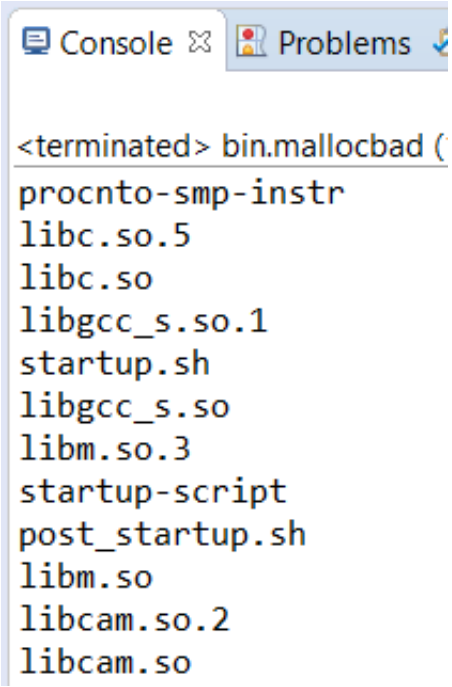
- access past the end
- access before the beginning
- doing a *free()* with an invalid address
- bad values passed into memory related library functions, e.g. *memcpy()* and *strcpy()*

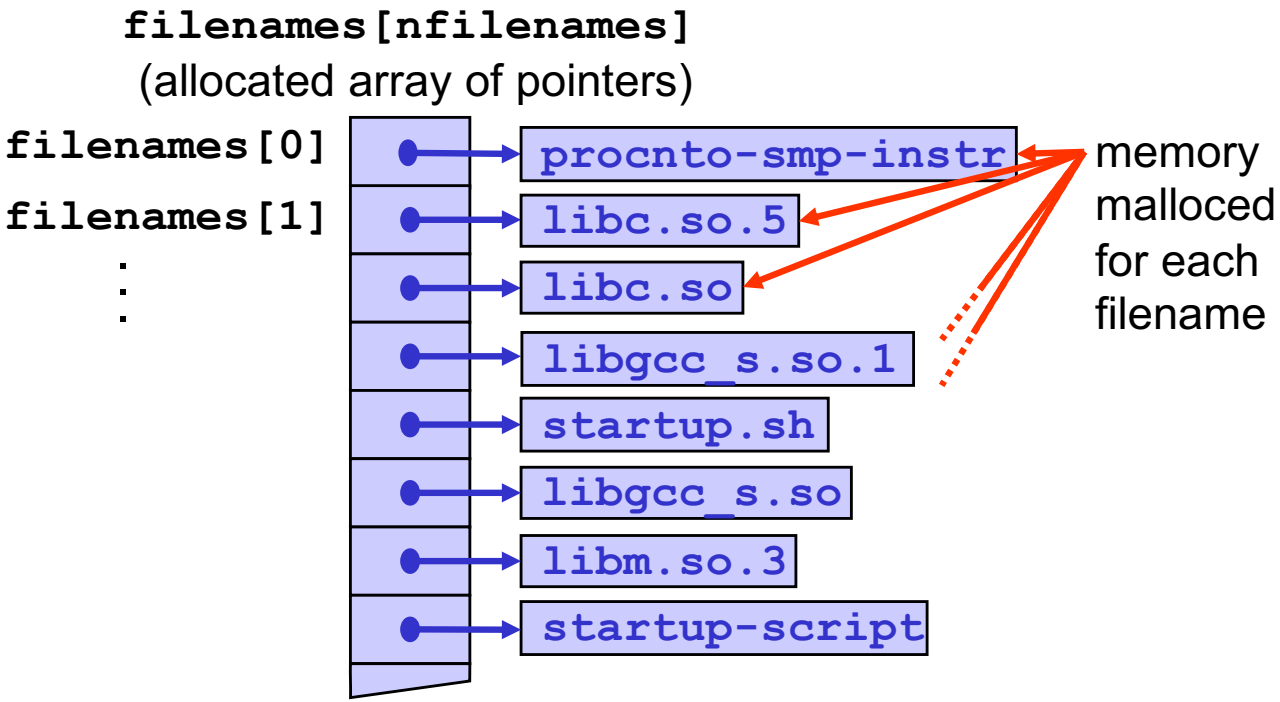
When a memory error occurs:

- 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

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

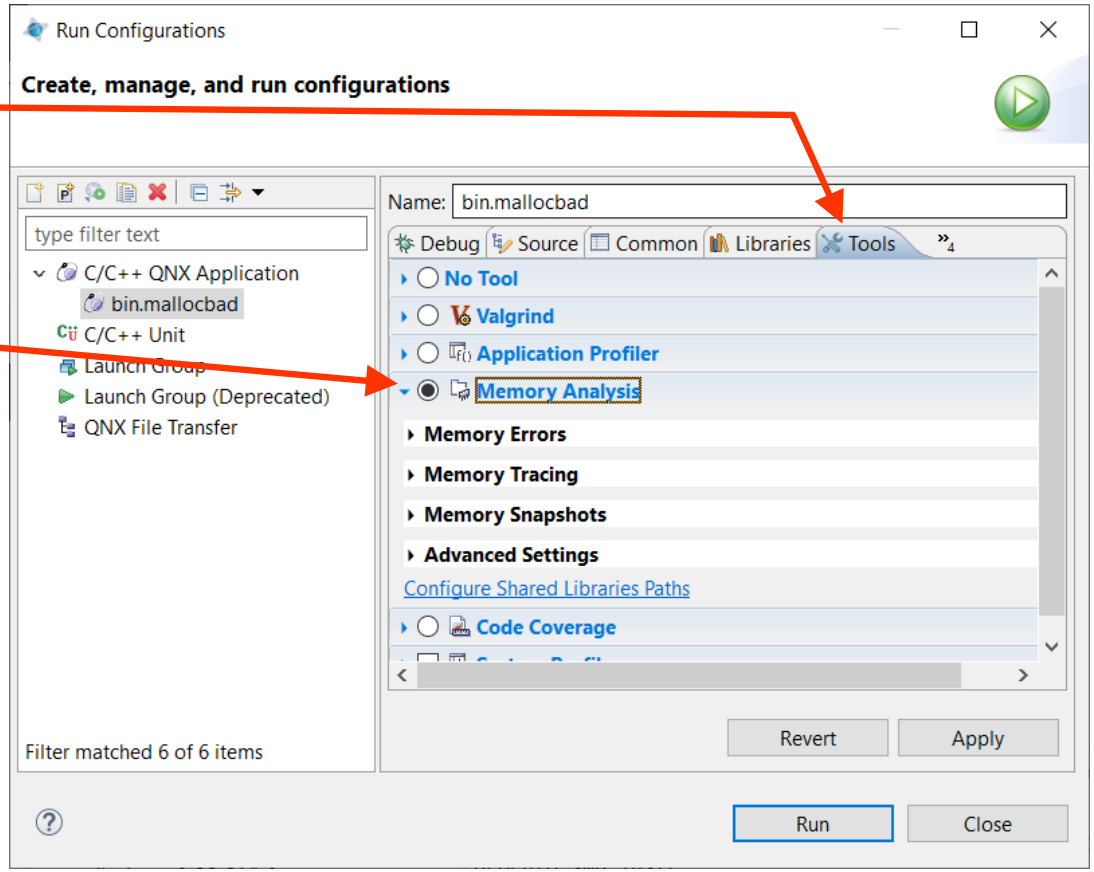




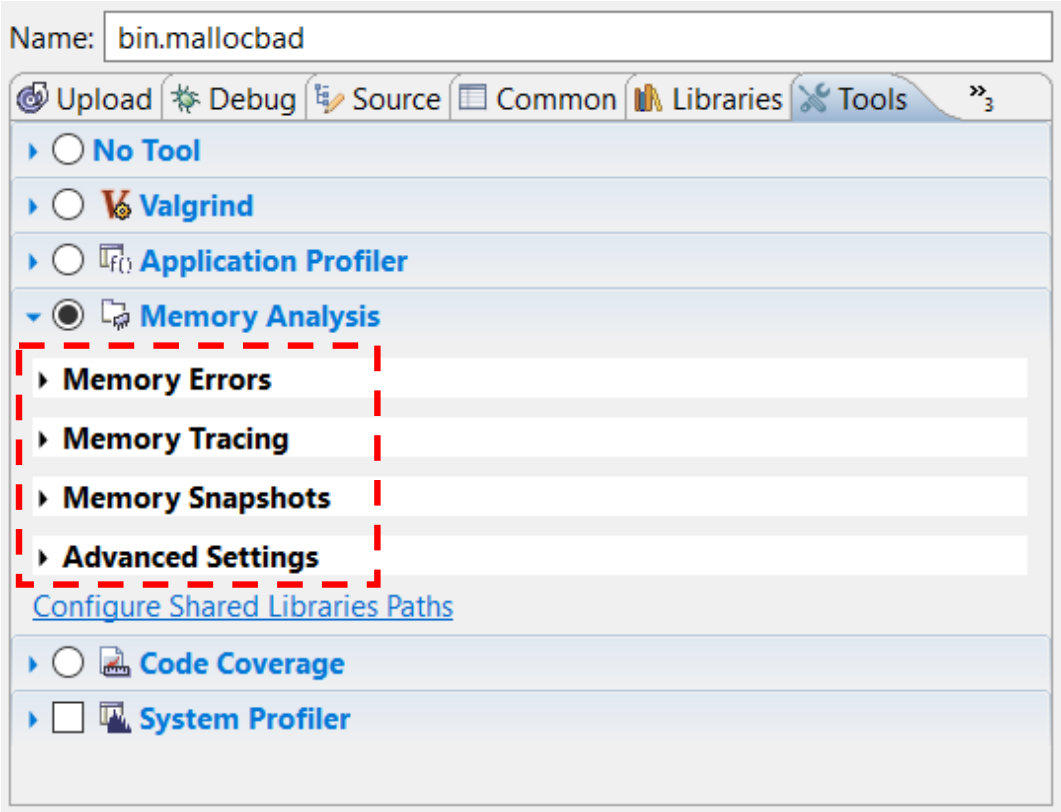
To launch an application for doing Memory Analysis:

– create a normal launch, and…

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



There are various options to configure:

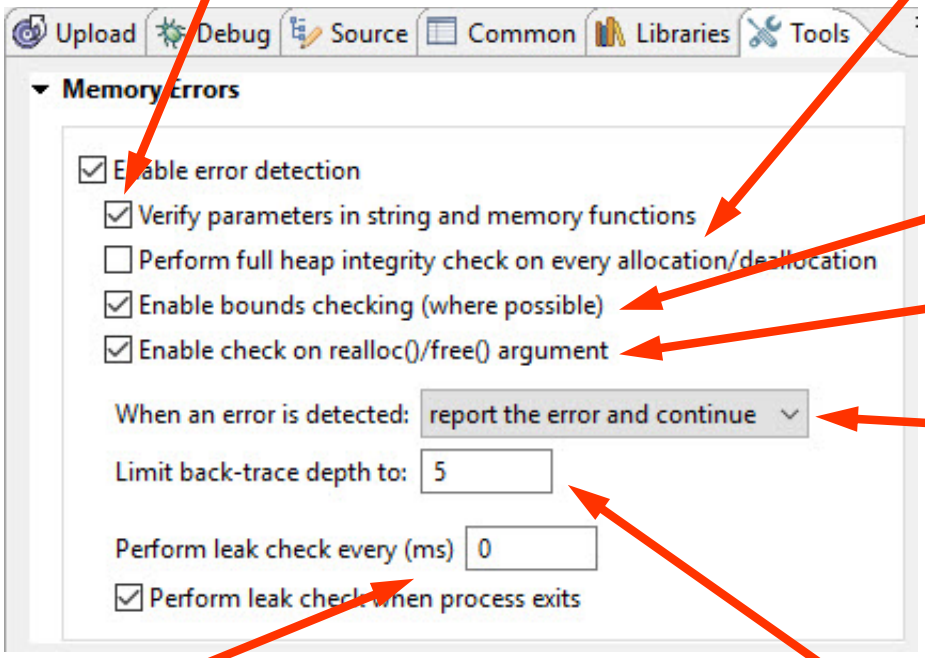


- in this section we'll deal with the Memory Errors configuration
- leak detection will use memory tracing

Choose settings:

Check parameters to memcpy(), strcpy(), etc.

full heap validation is expensive (slow), only enable if needed



was data written past end or before beginning of allocated blocks?

check realloc() and free() parameters

what will happen when a memory event occurs?

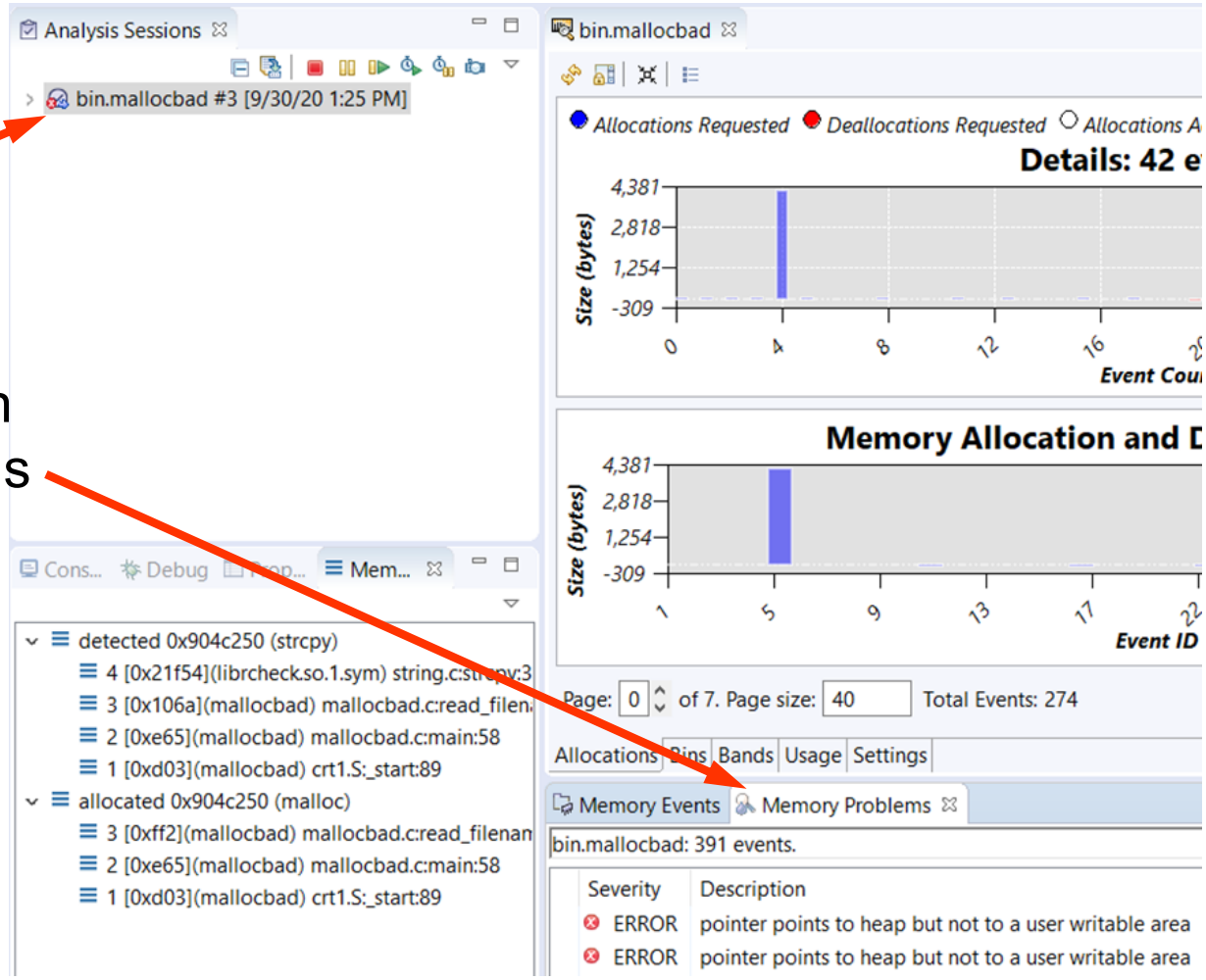
leak checks are expensive (slow), only enable if needed, and with needed frequency

How far back up stack to report on error – more depth is more overhead

Open the QNX Analysis perspective:

double-click on a session to open it

the errors will be shown in the Memory Problems view



Reporting Memory Errors:

Memory Backtrace view

Console | Debug | Properties | Memory B...

- detected 0x904c250 (strcpy)
  - 4 [0x21f54](librcheck.so.1.sym) string.c:strcpy:308
  - 3 [0x106a](mallocbad) mallocbad.c:read_filenames:130
  - 2 [0xe65](mallocbad) mallocbad.c:main:58
  - 1 [0xd03](mallocbad) crt1.S:_start:89
- allocated 0x904c250 (malloc)
  - 3 [0xff2](mallocbad) mallocbad.c:read_filenames:124
  - 2 [0xe65](mallocbad) mallocbad.c:main:58
  - 1 [0xd03](mallocbad) crt1.S:_start:89

Memory Events | Memory Problems

bin.mallocbad: 391 events.

Severity	Description	Pointer	Trap Function
ERROR	pointer points to heap but not to a user writable...	0x904c1...	strcpy
ERROR	pointer points to heap but not to a user writable...	0x904c1...	strcpy
ERROR	pointer points to heap but not to a user writable...	0x904c0...	strcpy
ERROR	pointer points to heap but not to a user writable...	0x904c1...	strcpy
ERROR	pointer points to heap but not to a user writable...	0x904c2...	strcpy
ERROR	pointer points to heap but not to a user writable...	0x904c2...	strcpy
ERROR	pointer points to heap but not to a user writable...	0x904c3...	strcpy
ERROR	pointer points to heap but not to a user writable...	0x904c3...	strcpy
ERROR	pointer points to heap but not to a user writable...	0x904c3...	strcpy
ERROR	pointer points to heap but not to a user writable...	0x904d0...	strcpy

221M of 256M

selecting an error displays the stack frames for when the error was detected, and the memory was allocated in the Memory Backtrace view

double-click on a stack frame to open editor to that line of code

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 );
```

EXERCISE

Finding memory corruption:

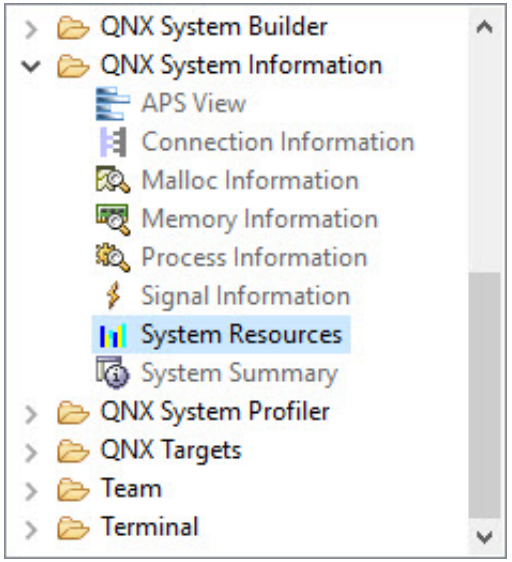
- run **`mallocbad`** (in your **`memory_problems`** project) using Memory Analysis, as shown in the preceding slides
- use the tool to find the offending (error triggering) lines of code
  - fix the problems and run again
  - have the errors gone away?

Topics:

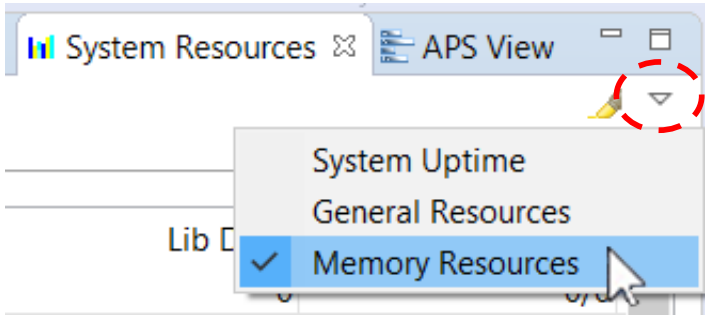
- **Overview**
- **Finding Memory Corruption**
- → **Excessive Memory Usage**
- **Finding Memory Leaks**
- **Importing and Exporting**
- **Conclusion**

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':



Excessive Memory Consumption

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

dynamic, e.g. *malloc()*, *new*

executable code

global data

click to highlight changes on information refresh

System Su... | Process Inf... | Memory Inf... | Malloc Info... | Target File ... | System Re... | APS View

local_VM - Last Updated:Wed Sep 30 13:55:42 EDT 2020

Process Name	Heap	Code	Data	Lib Code	Lib Data	Stack
procnto-smp-instr (1)	0	824K	194K	0	0	0/0
slogger2 (16387)	84K	68K	196K	1108K	14M	12K/516K
pci-server (28676)	160K	144K	228K	1308K	20M	28K/1296K
devb-eide (40965)	3424K	128K	3524K	2028K	28M	140K/940K
devc-con (114694)	108K	92K	148K	1040K	8M	8192/516K
syslogd (118791)	76K	28K	140K	1280K	10M	32K/1556K

EXERCISE

Memory usage:

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

Topics:

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

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)

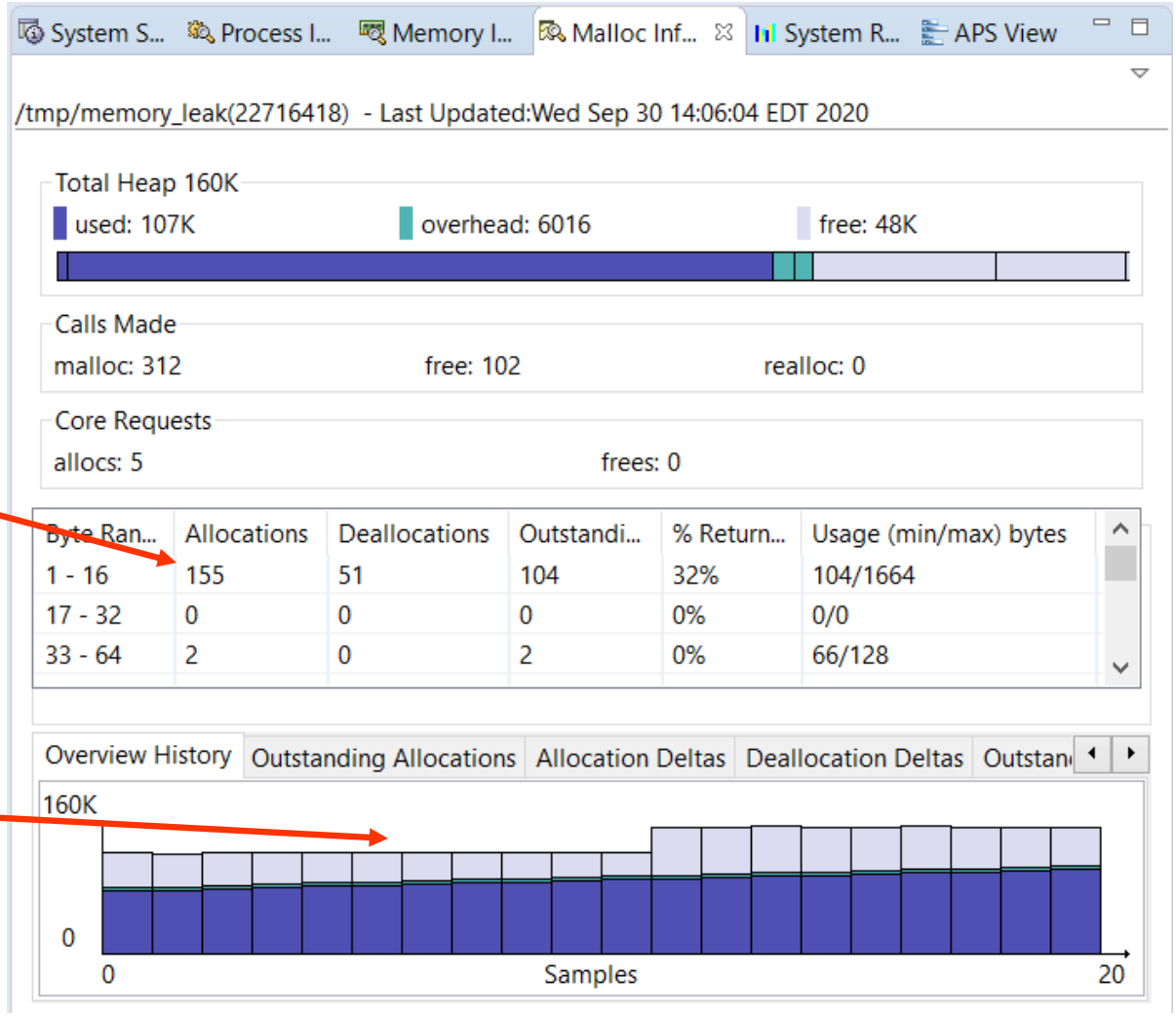
- the following tools can help find leaks:
  - Malloc Information (System Information perspective)
  - QNX Analysis perspective (Memory Analysis sessions)

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...

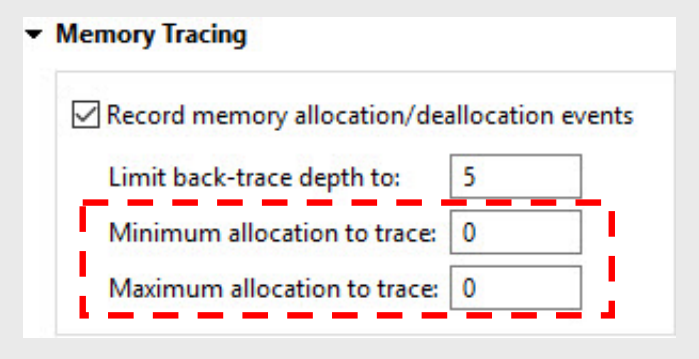
the number of mallocs keep increasing each time the view refreshes but the free count will not (or not as quickly)

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



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



The Memory Analysis editor – Allocations:

Analysis Sessions

bin.memory_leak #5 [9/30/20 2:16 PM]

pid:22937602

Logs

C... D... Pr... Me...

2 [0xc63](memory_leak) memory_leak.c

1 [0xa93](memory_leak) crt1.S:_start:89

bin.memory_leak

Allocations Requested, Deallocations Requested, Allocations Actual, Deallocations Actual

Details: 22 events

Size (bytes): 1,126, 375, -375, -1,126

Event Count: 0, 2, 4, 6, 8, 10, 13, 15, 17, 19, 21

Memory Allocation and Deallocation events

Size (bytes): 1,126, 375, -375, -1,126

Event ID: 1, 8, 15, 22, 29, 36, 42, 49, 56, 63, 70

Page: 0 of 9. Page size: 68 Total Events: 552

Allocations | Bins | Bands | Usage | Settings

Memory Events | Memory Problems

bin.memory_leak: 22 events.

Kind	Requested	Actual	Pointer	TID	Location	Co
malloc	1024	1032	0x90791...	1	memory_leak.c:59	1
malloc	16	16	0x904c3...	1	memory_leak.c:59	1
free	-16	-16	0x904c3...	1	memory_leak.c:70	1

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.

The Memory Events view:

– log of every allocation and de-allocation

Con... Deb... Prop... Mem...

2 [0xc63](memory_leak) memory_leak.c:main:59
1 [0xa93](memory_leak) crt1.S:_start:89

Memory Events | Memory Problems

bin.memory_leak: 22 events.

Kind	Requested	Actual	Pointer	TID	Location	Count
malloc	1024	1032	0x90791...	1	memory_leak.c:59	1
malloc	16	16	0x904c3...	1	memory_leak.c:59	1
free	-16	-16	0x904c3...	1	memory_leak.c:70	1
malloc	1024	1032	0x9078d...	1	memory_leak.c:59	1
malloc	16	16	0x904c3...	1	memory_leak.c:59	1
malloc	1024	1032	0x90788...	1	memory_leak.c:59	1

double-click on a stack frame to open editor

selecting an item gives a stack trace

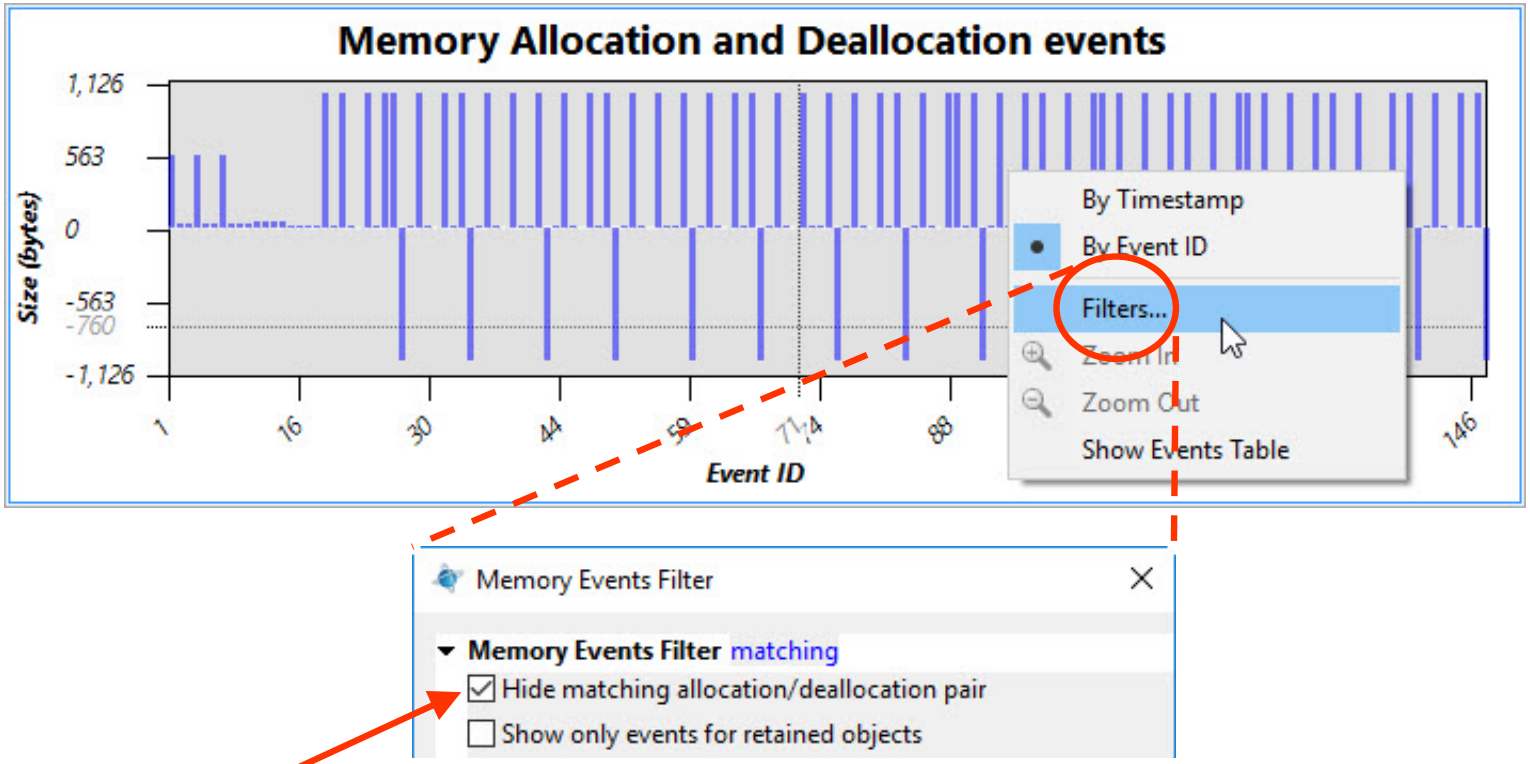
allocation

de-allocation

allocation that has a matching de-allocation

de-allocation that has a matching allocation

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

You can enable more leak checking:

Memory Errors

- [x] Enable error detection
  - [x] Verify parameters in string and memory functions
  - [ ] Perform full heap integrity check on every allocation/deallocation
  - [x] Enable bounds checking (where possible)
  - [x] Enable check on realloc()/free() argument

When an error is detected: report the error and continue

Limit back-trace depth to: 5

Perform leak check every (ms) 2000

- [x] Perform leak check when process exits

add leak checking interval

required for doing regular leak checks at runtime

Advanced Settings

Runtime library: librcheck.so

(•) Use regular file ( ) Use streaming device

Target output file or device: /tmp/traces.rmat

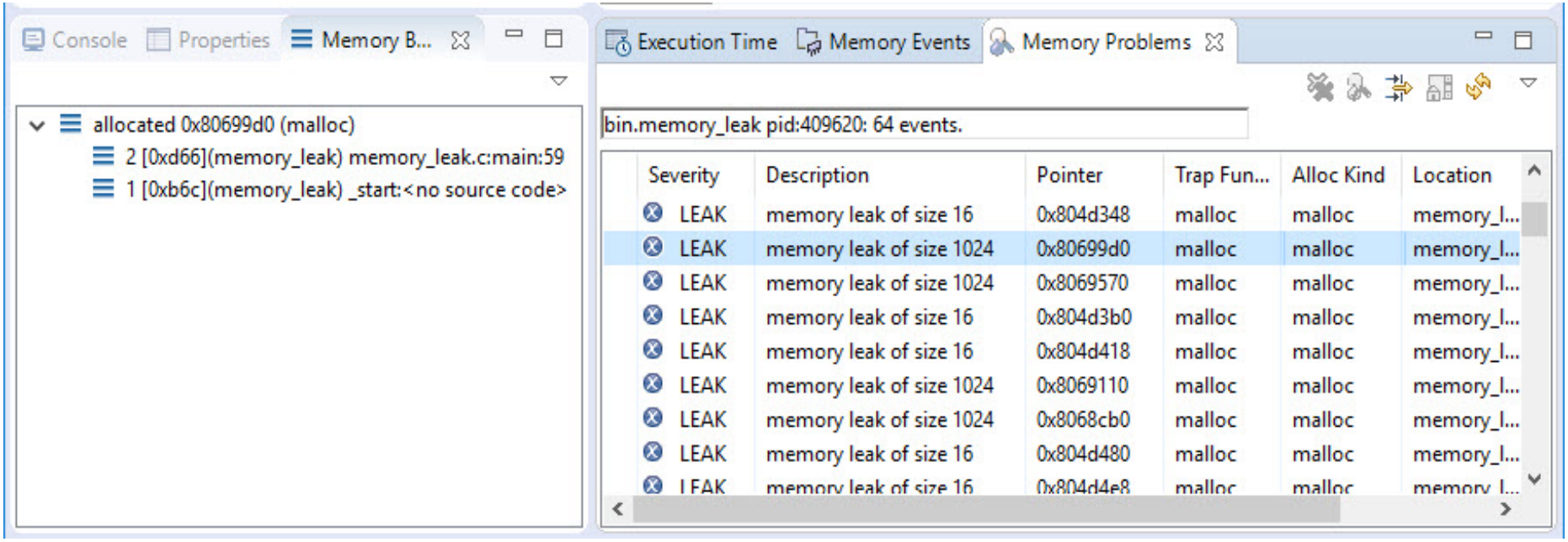
- [x] Create control thread
- [ ] Use dladdr to find dll names
- [ ] Show debug output on console

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

Memory leaks appear in the Memory Problems view:



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...

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

Topics:

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

Sometimes you can't run from the IDE:

- you can generate memory trace data from the command line
  - then you can import this into the IDE
- to generate from the command line you need to:
  - replace the malloc library
  - specify the output file
  - specify the data to generate and other configuration parameters
  - e.g.:

```
LD_PRELOAD=librcheck.so MALLOC_TRACE=/tmp/time.rmat MALLOC_TRACEBTDEPTH=5
  MALLOC_EVENTBTDEPTH=5 /tmp/time -v
```

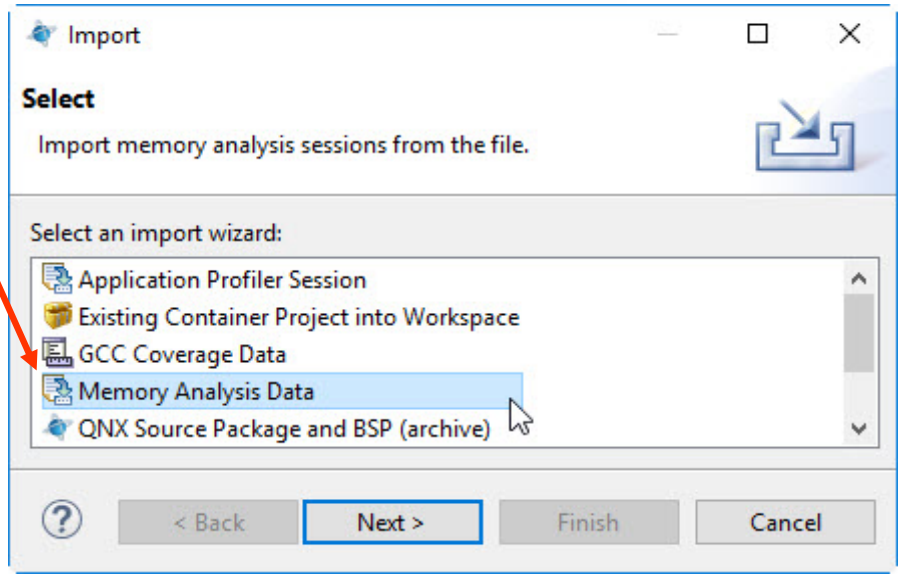
- copy the output file (**`time.rmat`**) to your host
  - the Target File System Navigator view is a convenient way to do this

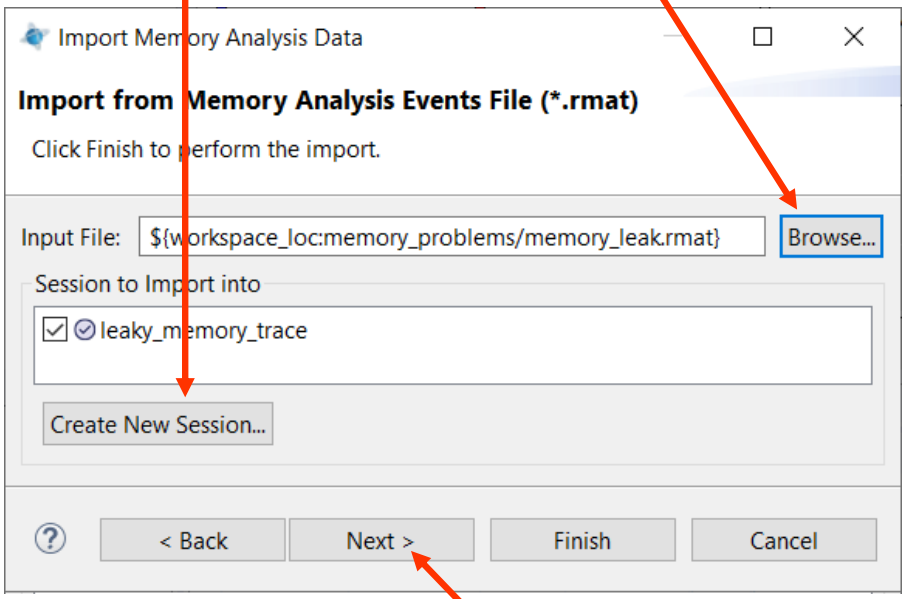
In the Analysis Sessions view:

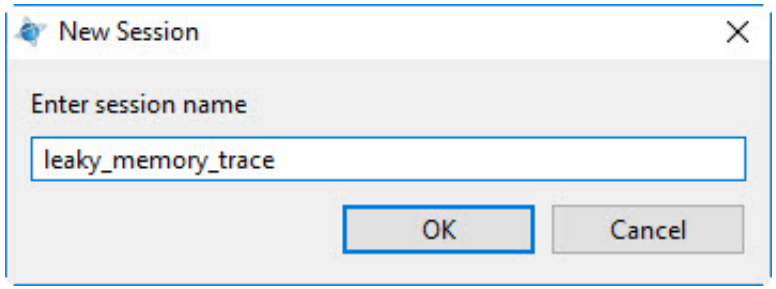
click Import -> Memory Analysis

browse to the data

then create a new session and name it







click Next

continued...

Importing (continued):

Import Memory Analysis Data

Binary Path

Select executable file ${workspace_loc:memory_problems/memory_leak} Browse...

Libraries Search Path:

Add... Edit... Up Down Remove

Source Lookup Path

Recurse	Folder
☑	C:\Users\stdufresne\Documents\ide-7.1-workspace\memory_problems

Add from File System... Add from Workspace... Remove Up Down

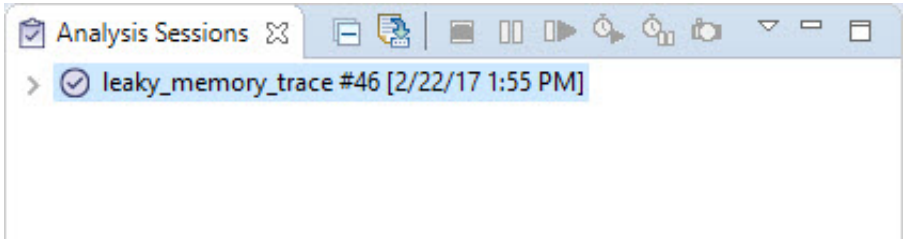
< Back Next > Finish Cancel

browse for your executable

tell the IDE where to look for libraries and source code

click Finish

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:

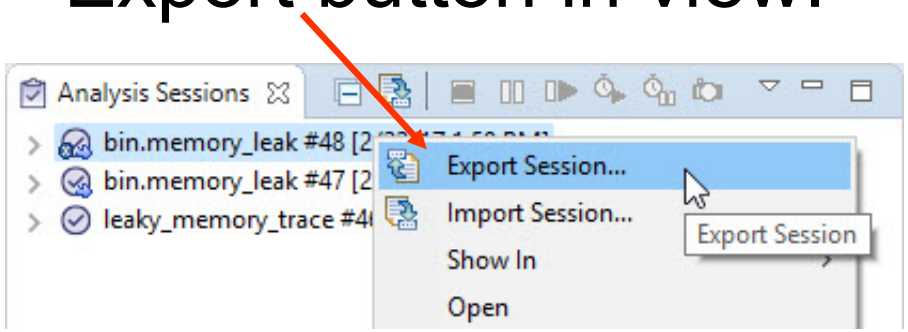
    `LD_PRELOAD=librcheck.so MALLOC_HELP=1 /tmp/time`

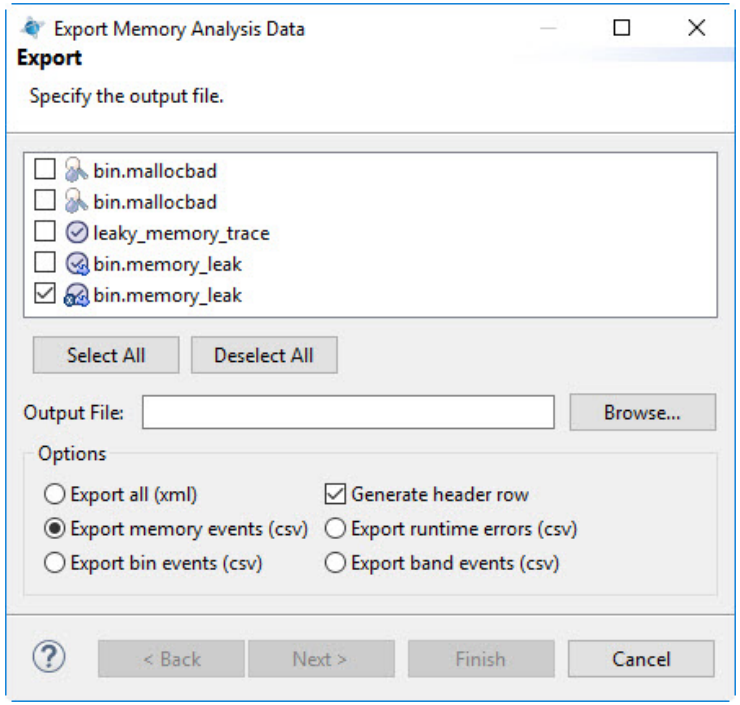
    or look at *mallopt()* in the Library Reference manual
  - or try configuring an IDE session for what you want, then run your program and look in the System Information perspective's Process Information view for the environment it generates

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:





Topics:

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

Conclusion

You learned how to find:

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