

The Intel logo is displayed in white text on a blue square background in the top left corner of the slide.

Intel[®] Inspector

Memory and Thread Debugger

CERN, March 3rd 2022

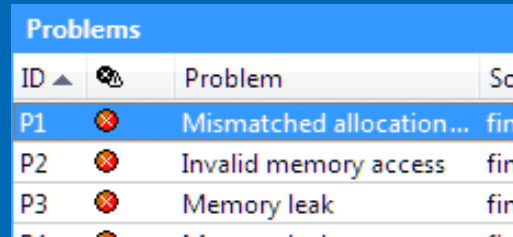
Heinrich Bockhorst, Intel

Agenda

- Intro to Intel® Inspector XE
- Analysis workflow
- Command line interface
- Memory problem analysis
- Threading problem analysis
- Persistent memory analysis
- Documentation

Motivation for Inspector

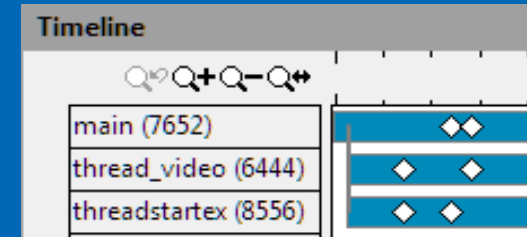
Memory Errors



ID	Problem	So
P1	Mismatched allocation...	fin
P2	Invalid memory access	fin
P3	Memory leak	fin

- Invalid Accesses
- Memory Leaks
- Uninitialized Memory Accesses

Threading Errors



- Data Races
- Deadlocks
- Cross Stack References

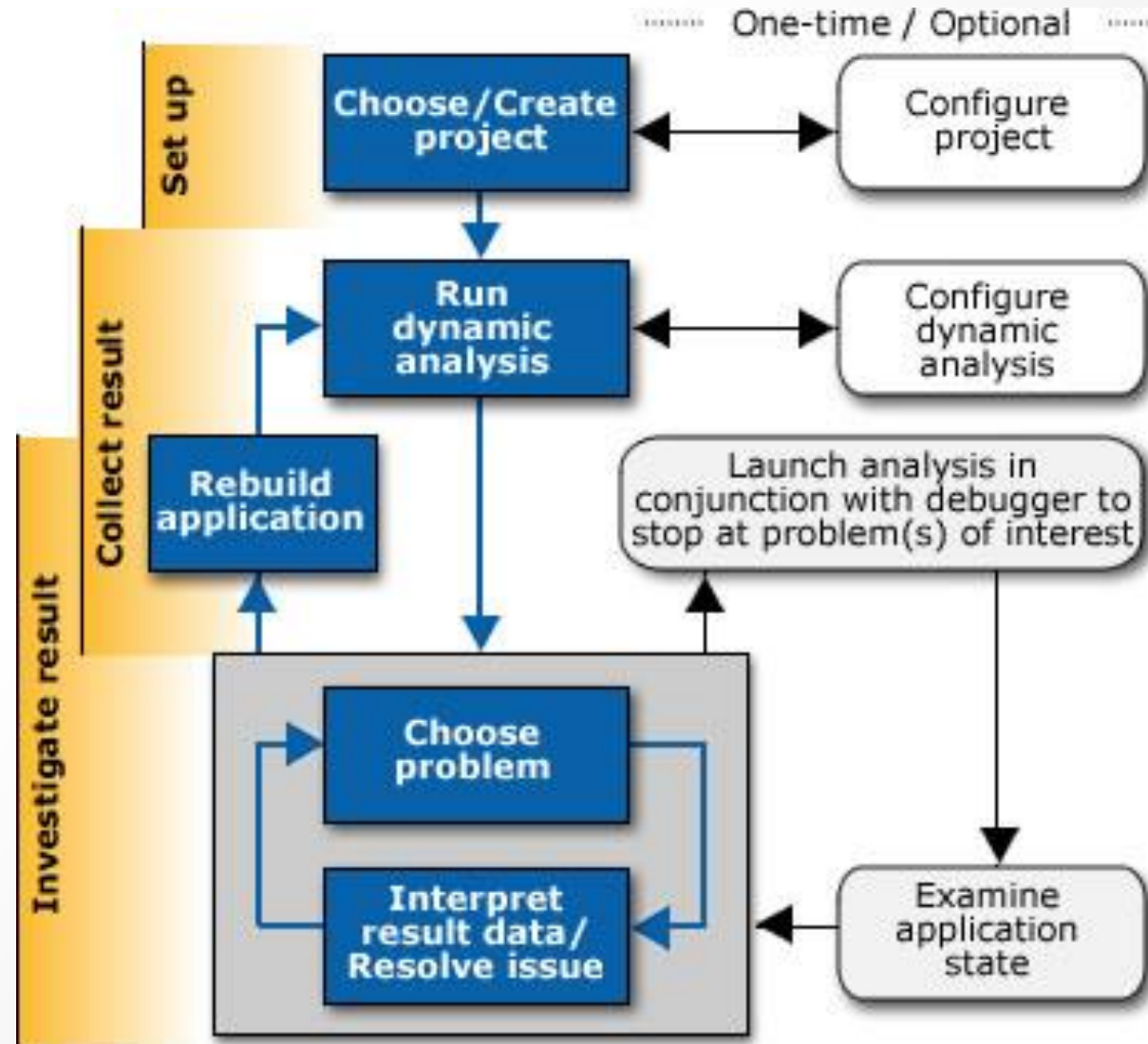
Multi-threading problems

- Hard to reproduce,
- Difficult to debug
- Expensive to fix



Let the tool do it for you

Workflow: Dynamic Analysis



Select analysis and Start

Configure Analysis Type

Analysis Type

Memory Error Analysis

2x-20x Detect Leaks

10x-40x Detect Memory Problems

20x-80x Locate Memory Problems

Analysis Time Overhead

Memory Overhead

Locate Memory Problems

Widest scope memory error analysis type. Maximizes the load on the system and the time and resources required to perform analysis; however, detects the widest set of errors and provides context and maximum detail for those errors. Press F1 for more details.

- Detect invalid memory accesses
- Analyze stack accesses
- Detect memory leaks upon application exit
- Enable interactive memory growth detection
- Enable on-demand memory leak detection

Start

Stop

Copy

Measure Growth

Reset Leak Tracking

Find Leaks

5

1. Select Analysis Type

2. Click Start

User Interface Overview

Locate Deadlocks and Data Races

Target Analysis Type Collection Log Summary

ID	Type	Sources	Modules	State
P1	Data race	memissues_omp.cpp	8ecee7e10398e29e; memissues_omp_exe	New

Filters

Type	Count
Data race	1 item(s)

Source

Source	Count
memissues_omp.cpp	1 item(s)

Module

Module	Count
8ecee7e10398e29e	1 item(s)
memissues_omp_exe	1 item(s)

State

State	Count
New	1 item(s)

Code Locations: Data race

Description	Source	Function	Module	Variable
Write	memissues_omp.cpp:13	_Z12parallel_sumPjmPm.extracted	8ecee7e10398e29e	0x3842800
Read	memissues_omp.cpp:13	_Z12parallel_sumPjmPm.extracted	8ecee7e10398e29e	0x3842800

```
11 // #pragma omp atomic
12 // #pragma omp critical
13 *output += input[i]; // This is a race
14
15 // std::cout << "Hi from thread " << omp_get_thread_num() << "\n"
```

Select a problem set

Code snippets displayed for selected problem

Filters let you focus on a module, or error type, or just the new errors or...

Problem States: New, Not Fixed, Fixed, Confirmed, Not a problem, Deferred, Regression

Source & Call Stack details

Source code locations displayed for selected problem

Data race

Target Analysis Type Collection Log Summary Sources

Write - Thread TBB Worker Thread (80030) (8ecce7e10398e29e!_Z12parallel_sumPjmPm.extracted - memissues_omp.cpp:13)

```
memissues_omp.cpp Disassembly (8ecce7e10398e29e!0x193) Call Stack
8 #pragma omp parallel for
9 for(size_t i=0;i<size;i++)
10 {
11 //#pragma omp atomic
12 //#pragma omp critical
13 *output += input[i]; //This is a race
14
15 //std::cout << "Hi from thread " << omp_get_thread_num() << "\n";
16 }
17 }
```

Read - Thread TBB Worker Thread (80080) (8ecce7e10398e29e!_Z12parallel_sumPjmPm.extracted - memissues_omp.cpp:13)

```
memissues_omp.cpp Disassembly (8ecce7e10398e29e!0x193) Call Stack
9 #pragma omp parallel for
10 for(size_t i=0;i<size;i++)
11 {
12 //#pragma omp atomic
13 //#pragma omp critical
14 *output += input[i]; //This is a race
15
16 //std::cout << "Hi from thread " << omp_get_thread_num() << "\n";
17 }
```

Call Stacks

Command Line Interface I

- Start analysis

- *Memory:* `inspxe-cl -c mi3 -- <app> [app_args]`
- *Threading:* `inspxe-cl -c ti3 -- <app> [app_args]`

- View results

- `inspxe-cl -report=problems -report-all`
- To open result in GUI, type:
`inspxe-gui <result folder>`

Command Line Interface II

- Help menu:

```
$ inspxe-cl -help
```

- Export results: create an archive with cached source

```
$ inspxe-cl -help export
```

```
$ inspxe-cl -export -include-source -archive-name <arch_name> -r <result>
```

- Move archive `arch_name.inspxez` to local machine with inspector installation.

Inspector and MPI

- To run Inspector in an Intel MPI job you may use the “-gtool” flag
- More convenient is the I_MPI_GTOOL environment variable:

```
$ export I_MPI_GTOOL= "inspxe-cl -c ti3 -r TI3:0"
```

run your program, as usual, under MPI. The setting will collect data on rank #0. Use a list of ranks or :all for multi rank analysis.

- More information:

<https://www.intel.com/content/www/us/en/develop/documentation/mpi-developer-reference-linux/top/command-reference/mpiexec-hydra/gtool-options.html>

Memory problems

Memory leak

- a block of memory is allocated
- never deallocated
- not reachable (there is no pointer available to deallocate the block)
- Severity level = (Error)

Memory not deallocated

- a block of memory is allocated
- never deallocated
- still reachable at application exit (there is a pointer available to deallocate the block).
- Severity level = (Warning)

Memory growth

- a block of memory is allocated
- not deallocated, within a specific time segment during application execution.
- Severity level = (Warning)

```
// Memory leak
```

```
char *pStr = (char*) malloc(512);  
return;
```

```
// Memory not deallocated
```

```
static char *pStr = malloc(512);  
return;
```

```
// Memory growth
```

```
// Start measuring growth  
static char *pStr = malloc(512);  
// Stop measuring growth
```

Threading Issues: Data race

```
CRITICAL_SECTION cs; // Preparation
int *p = malloc(sizeof(int)); // Allocation Site
*p = 0;
InitializeCriticalSection(&cs);
```

Write -> Write Data Race

Thread #1

```
*p = 1; // First Write
```

Thread #2

```
EnterCriticalSection(&cs);
*p = 2; // Second Write
LeaveCriticalSection(&cs);
```

Read -> Write Data Race

Thread #1

```
int x;
x = *p; // Read
```

Thread #2

```
EnterCriticalSection(&cs);
*p = 2; // Write
LeaveCriticalSection(&cs);
```

Deadlock

```
CRITICAL_SECTION cs1;  
CRITICAL_SECTION cs2;  
int x = 0;  
int y = 0;  
InitializeCriticalSection(&cs1); // Allocation Site (cs1)  
InitializeCriticalSection(&cs2); // Allocation Site (cs2)
```

Thread #1

```
EnterCriticalSection(&cs1);  
x++;  
    EnterCriticalSection(&cs2);  
    y++;  
    LeaveCriticalSection(&cs2);  
LeaveCriticalSection(&cs1);
```

Thread #2

```
EnterCriticalSection(&cs2);  
y++;  
    EnterCriticalSection(&cs1);  
    x++;  
    LeaveCriticalSection(&cs1);  
LeaveCriticalSection(&cs2);
```

Deadlock

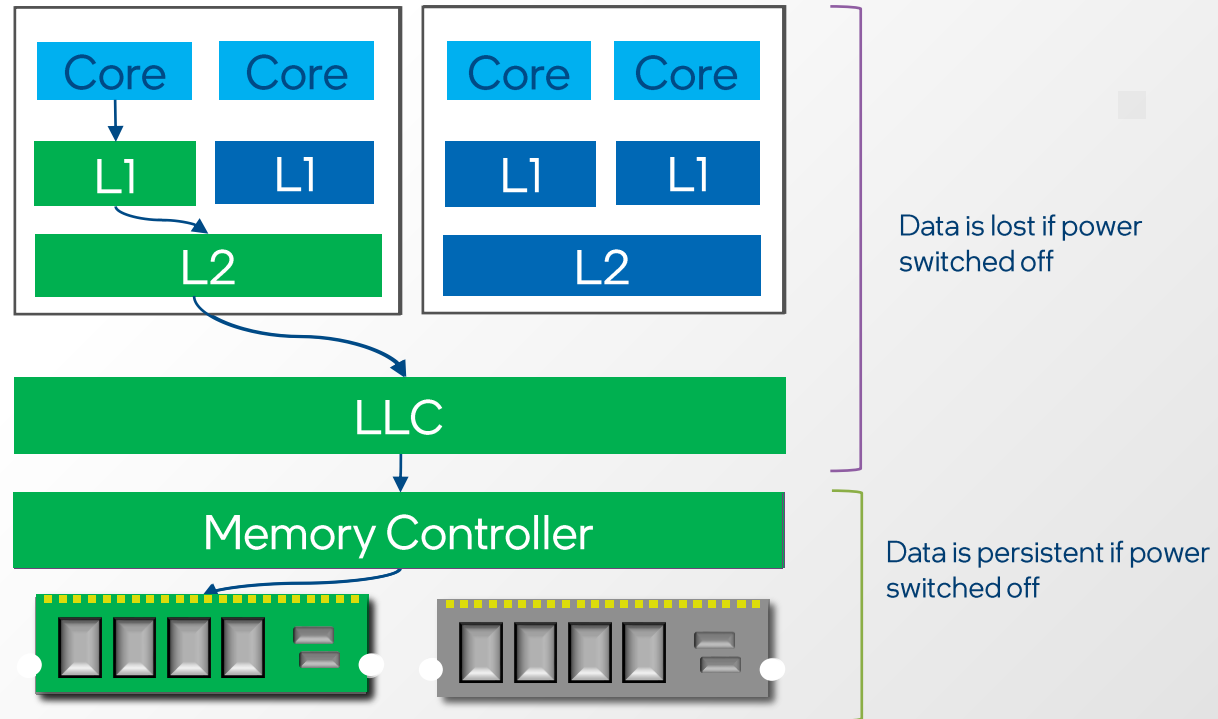
1. EnterCriticalSection(&cs1); in thread #1
2. EnterCriticalSection(&cs2); in thread #2

Lock Hierarchy Violation

1. EnterCriticalSection(&cs1); in thread #1
2. EnterCriticalSection(&cs2); in thread #1
3. EnterCriticalSection(&cs2); in thread #2
4. EnterCriticalSection(&cs1); in thread #2

Persistent memory analysis

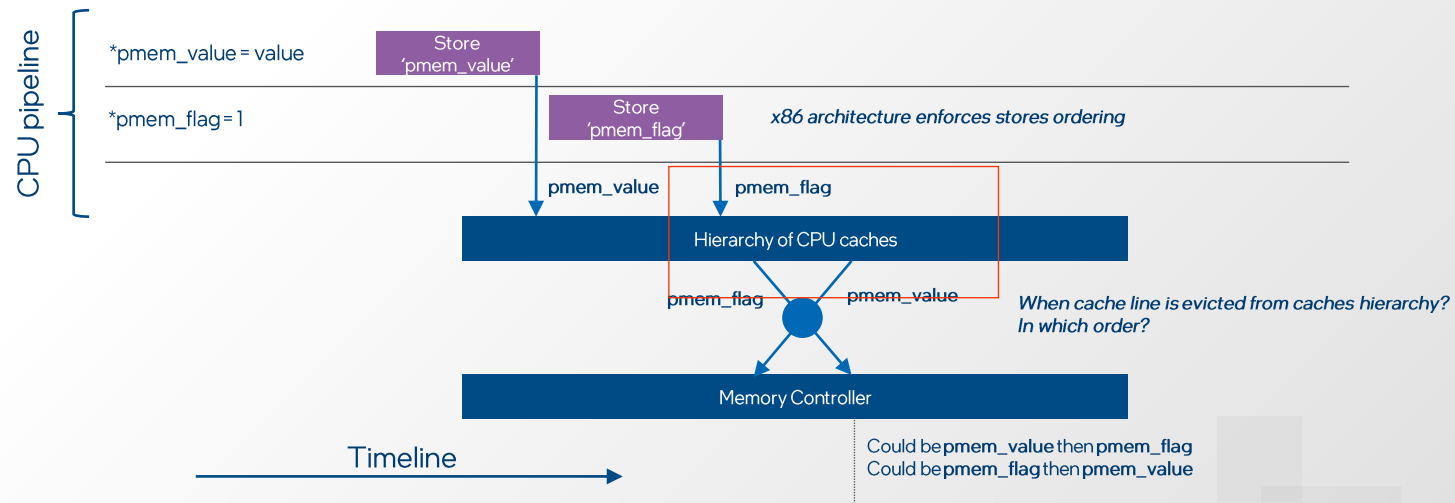
Memory Store != Memory Persistence



CPU cache hierarchy could affect persistence order

```
1 void save_value(uint32_t value, uint8_t* pmem_region)
2 {
3     uint8_t* pmem_flag = pmem_region;
4     uint32_t* pmem_value = (uint32_t*)(pmem_region + 256);
5
6     *pmem_value = value;
7     *pmem_flag = 1;
8 }
```

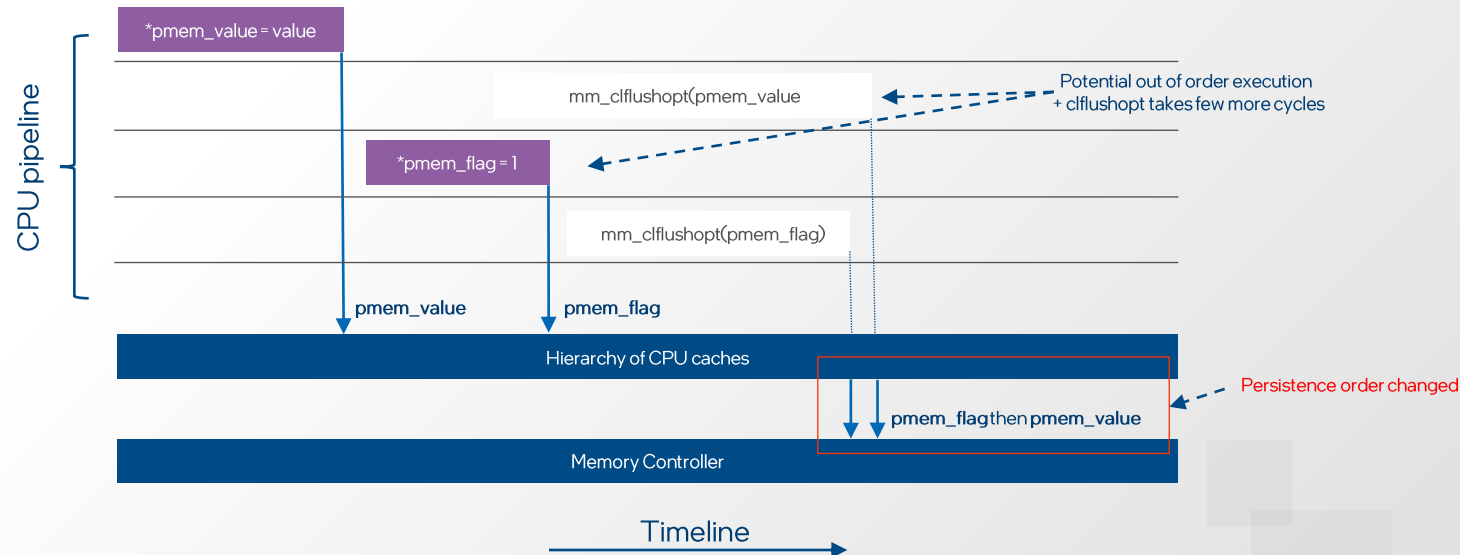
```
1 void load_value(uint32_t value, uint8_t* pmem_region)
2 {
3     uint8_t* pmem_flag = pmem_region;
4     uint32_t* pmem_value = (uint32_t*)(pmem_region + 256);
5
6     if(*pmem_flag)
7         printf("Value = %u", *pmem_value);
8 }
```



Execution reordering affects data correctness

```
1 void save_value(uint32_t value, uint8_t* pmem_region)
2 {
3     uint8_t* pmem_flag = pmem_region;
4     uint32_t* pmem_value = (uint32_t*)(pmem_region + 256);
5
6     *pmem_value = value;
7     _mm_clflushopt(pmem_value);
8     *pmem_flag = 1;
9     _mm_clflushopt(pmem_flag);
10 }
```

- All stores are in order on x86 architecture
- clflushopt is ordered with respect to stores to that cache line



Potential PM problem detected by the tool

- Missing or redundant cache flushes
- Missing store fences
- Stores not added into a transaction
- Redundant transactions
- Overlapping regions registered in different transactions
- Out-of-order stores
- Memory leaks (unused memory)

Documentation

- Collection of documentation (some links do not work!)
<https://software.intel.com/en-us/inspector>
- Intel Developer Zone (open forum)
<https://software.intel.com/en-us/forums/intel-inspector>
- User Guide:
<https://www.intel.com/content/www/us/en/develop/documentation/inspector-user-guide-linux/top.html>
- Persistent Memory:
<https://www.intel.com/content/www/us/en/developer/articles/technical/detect-persistent-memory-programming-errors-with-intel-inspector-persistence-inspector.html>

Notices & Disclaimers

Performance varies by use, configuration, and other factors. Learn more at www.Intel.com/PerformanceIndex.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details.

Your costs and results may vary.

Intel technologies may require enabled hardware, software or service activation.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.



intel®

Integration with debugger

Debugger integration

Break into debugger

- Analysis can stop when it detects a problem
- User is switched into a standard debugging session

Windows*

- Microsoft* Visual Studio Debugger (vs2017 – vs2019)

Linux*

- gdb

2x-20x | Detect Leaks
10x-40x | Detect Memory Problems
20x-80x | Locate Memory Problems

Analysis Time Overhead

Detect Memory Problems Copy

Medium scope memory error analysis type. Increases the load on the system and the time and resources required to perform analysis. Press F1 for more details.

- Analyze without debugger
Run an analysis and report all detected problems. Use to view correctness issues without stopping in the debugger to examine them.
- Enable debugger when problem detected
Run an analysis under the debugger and stop every time a problem is detected. Use to allow investigation of every problem detected.
- Select analysis start location with debugger
Run target application under the debugger with analysis disabled until you choose to turn on analysis. Before starting, set a code breakpoint to stop execution prior to where you want analysis to begin. Sele...

Debug this problem

The screenshot shows the Intel Inspector XE 2015 interface. At the top, there are tabs for 'r010mi2', 'r009mi2', 'mc.cpp', and 'simple_dll.cpp'. Below the tabs is the 'Detect Memory Problems' section with sub-tabs for 'Target', 'Analysis Type', 'Collection Log', and 'Summary'. The 'Problems' table lists two issues: 'P1 Memory leak' and 'P2 Invalid memory access'. A purple callout bubble points to the 'P2' row with the text 'Right click on a problem'. A context menu is open over the 'P2' row, with 'Debug This Problem' highlighted. Another purple callout bubble points to this menu item with the text 'Inspector XE will set breakpoint, and launch debug session at the place of the problem occurrence'. Below the problems table is a 'Code Location' table with columns for 'Description', 'Source', 'Function', 'Module', and 'Object'. The code snippet below shows a loop where a memory leak occurs at line 150.

ID	Type	State
P1	Memory leak	New
P2	Invalid memory access	Not f

Description	Source	Function	Module	Object
Write	mc.cpp:150	main	mc.exe	

```
148  
149     for (unsigned int i = 0;  
150         local_mbox[i] = 0;  
151  
152     return 0;
```

Context menu options:

- View Source
- Edit Source
- Copy to Clipboard
- Explain Problem
- Create Problem Report...
- Debug This Problem**
- Change State
- Merge S