Advanced memory debugging and leak detection

Version 1.0

Non-Confidential

Issue 01

102604_0100_01_en

Advanced memory debugging and leak detection

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	1 January 2021	Non-Confidential	First release

Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Overview

The questions it can answer and problems it can solve include:

- How much memory am I using?
- Which parts of my code are allocating the most memory?
- Are there memory leaks - and where am I failing to deallocate?
- Is a pointer being used after it is deallocated .. or after it has been re-used - and crashing my program?
- For a given pointer, is it still valid, where was it allocated and how large is the block of memory?
- Is my program deallocating or freeing invalid pointers?
- Am I reading or writing beyond the end of an allocation and overwriting memory? If so, where?

Answering these questions solves many unexplained crashes. Ensuring code is clear from the types of issue listed also improves software quality.

2. The Heap

The region of memory that the memory debugging mode helps with is known as the heap. The heap is the area managed by the malloc, free and similar functions in C, the new and delete operator in C++, and the allocate and deallocate primitives in F90, and later Fortran derivatives.

DDT intercepts these functions to provide error detection, to record information, and to measure how much memory is being used.

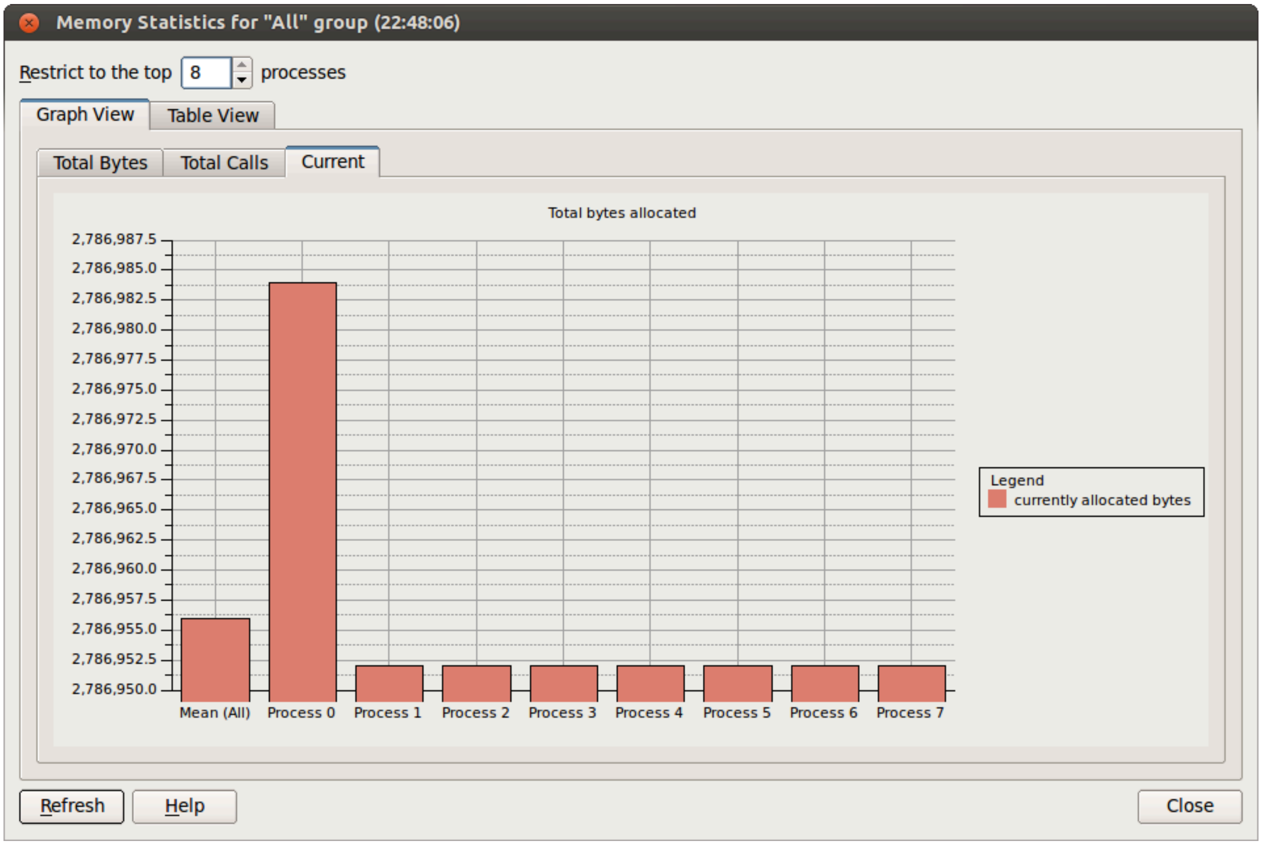
The level of checks is determined by a settings level in the Memory Debugging settings dialog - from basic to full checks. Full mode can slow down codes that perform very large numbers of allocations whereas basic has a usually near-zero time cost.

3. Memory usage

The total memory usage is an important number to watch. Allocating too much memory results in the operating system killing your process.

The running total for a process is shown on the Tools / Overall Memory Stats menu item. When debugging more than one process, they (or the top N processes by use) display in this view.

Figure 3-1: Memory Statistics Graph View



Memory Statistics for "All" group (22:48:06)

Restrict to the top 8 processes

This view is available whenever Memory Debugging is enabled: if the total creeps up, then this is an indication of a memory leak.

Arm Forge, the tool suite that includes DDT, also has memory profiling in its MAP performance profiler. MAP graphically profiles the usage of memory as it varies over time which helps to narrow down when and where memory usage is increasing.

4. Memory leak detection

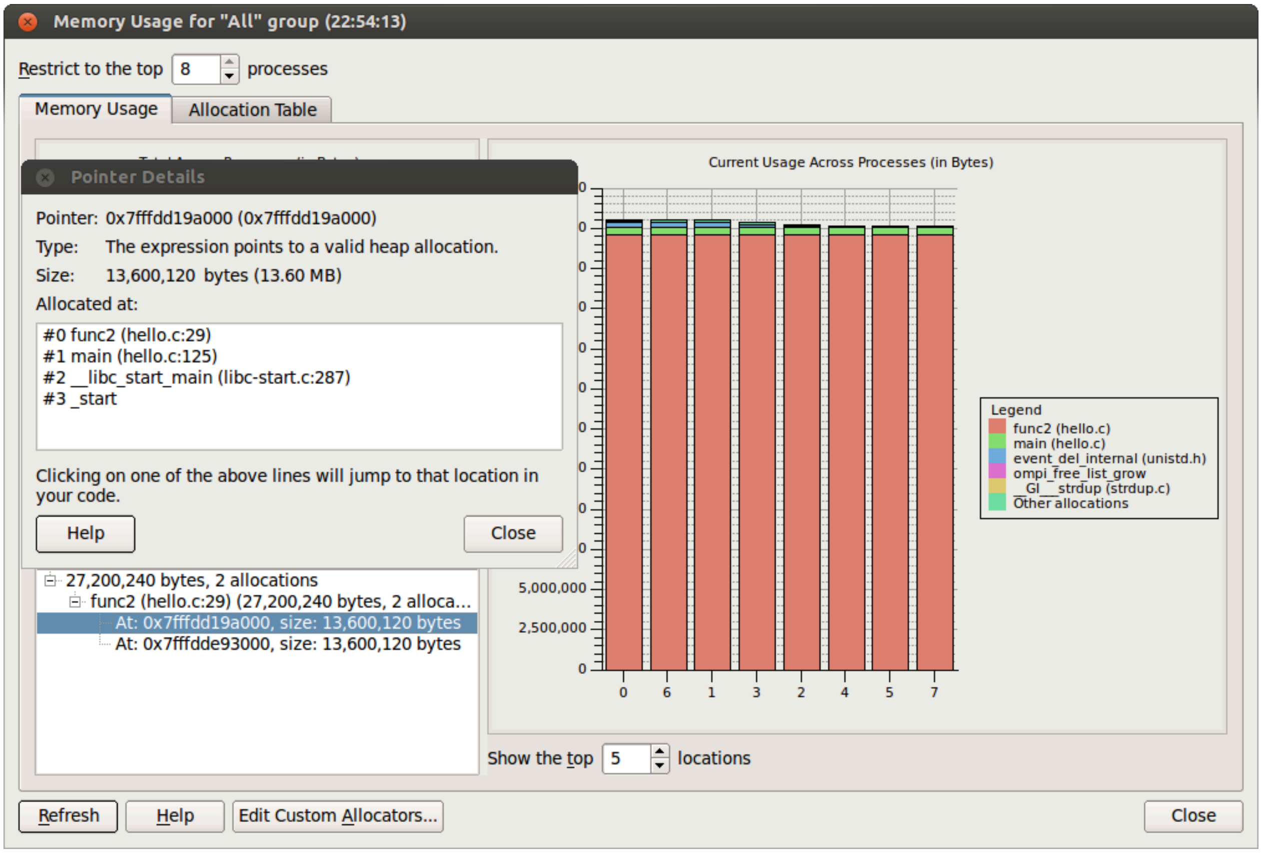
If memory is allocated but not deallocated, then this eventually leads to memory exhaustion and abrupt termination.

Memory leaks are detected by using the Tools > Current Memory Usage menu item, which is available whenever Memory Debugging is enabled.

For each memory allocation, DDT records the stack trace and the requested allocation size. This allows it to know where in the code allocations occur, and how much is being used.

The Current Memory Usage dialog uses this information to plot the most commonly calling locations, in terms of bytes used. If a leak is happening, it shows in this bar chart. Clicking through elements on the bar chart lists the pointers, and selecting a pointer shows the stack and the size of that allocation.

Figure 4-1: Memory Usage showing the Pointer Details popup for the selected pointer



Pointer: 0x7fffdd19a000 (0x7fffdd19a000)
Type: The expression points to a valid heap allocation.
Size: 13,600,120 bytes (13.60 MB)
Allocated at:

```
#0 func2 (hello.c:29)
#1 main (hello.c:125)
#2 __libc_start_main (libc-start.c:287)
#3 _start
```

Clicking on one of the above lines will jump to that location in your code.

```
⊟ 27,200,240 bytes, 2 allocations
  ⊟ func2 (hello.c:29) (27,200,240 bytes, 2 alloca...
      At: 0x7fffdd19a000, size: 13,600,120 bytes
      At: 0x7fffdde93000, size: 13,600,120 bytes
```

Custom Allocators and Class Constructors

Many codes often channel allocations through a small number of entry points.

For example, the constructor of a C++ class constructor would often be used to allocate memory. The most useful classes can be invoked throughout a program. In these cases, it is more helpful to group allocations by the line of code that called the constructor, so that the calls to the constructor are not lumped together in one unstructured form.

To group by calls to the constructor, right-click in a block in this bar chart to add that represented function as a Custom Allocator. Calls to that function are then grouped separately by call location.

Automating Leak Detection and Regression Testing

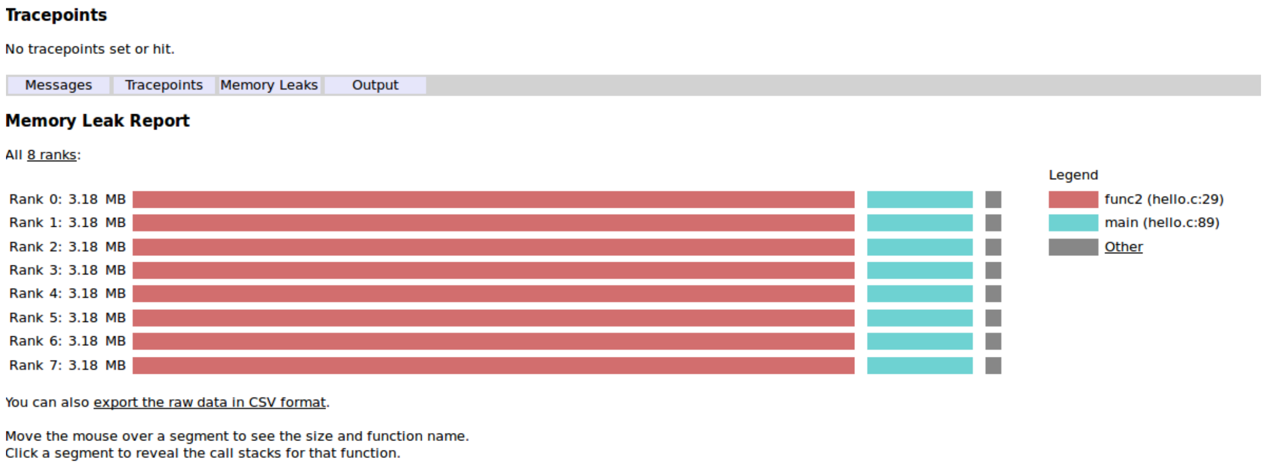
DDT has a non-interactive memory debugging mode which replicates the interactive information of the Current Memory Usage tool described above. This mode is often used during overnight tests or in continuous integration servers to measure memory usage and automatically ensure that leaks do not enter production code.

This mode creates an HTML file containing the memory allocations that remain after a process (or processes) terminates.

```
ddt --offline offline-log.html --mem-debug ... application.exe ....
```

This creates an annotated log file of the non-interactive debugging session which contains a leak report with the top leaks identified along with significant debugging events that are logged throughout the execution of the program.

Figure 4-2: An example log file Memory Leak report



Tracepoints

No tracepoints set or hit.

Memory Leak Report

All 8 ranks:

You can also export the raw data in CSV format.

Move the mouse over a segment to see the size and function name.
Click a segment to reveal the call stacks for that function.

5. Deallocating invalid pointers

Crashes can also occur when trying to deallocate an allocation that has previously been deallocated, or a bogus address (either not allocated, or part way through a legitimate range of heap memory).

This leads to either immediate termination, or heap corruption, and a crash at some future point.

In DDT, this kind of problem is prevented as invalid pointers immediately trigger an error message, and stop the process exactly where the error occurs.

6. Dangling pointers

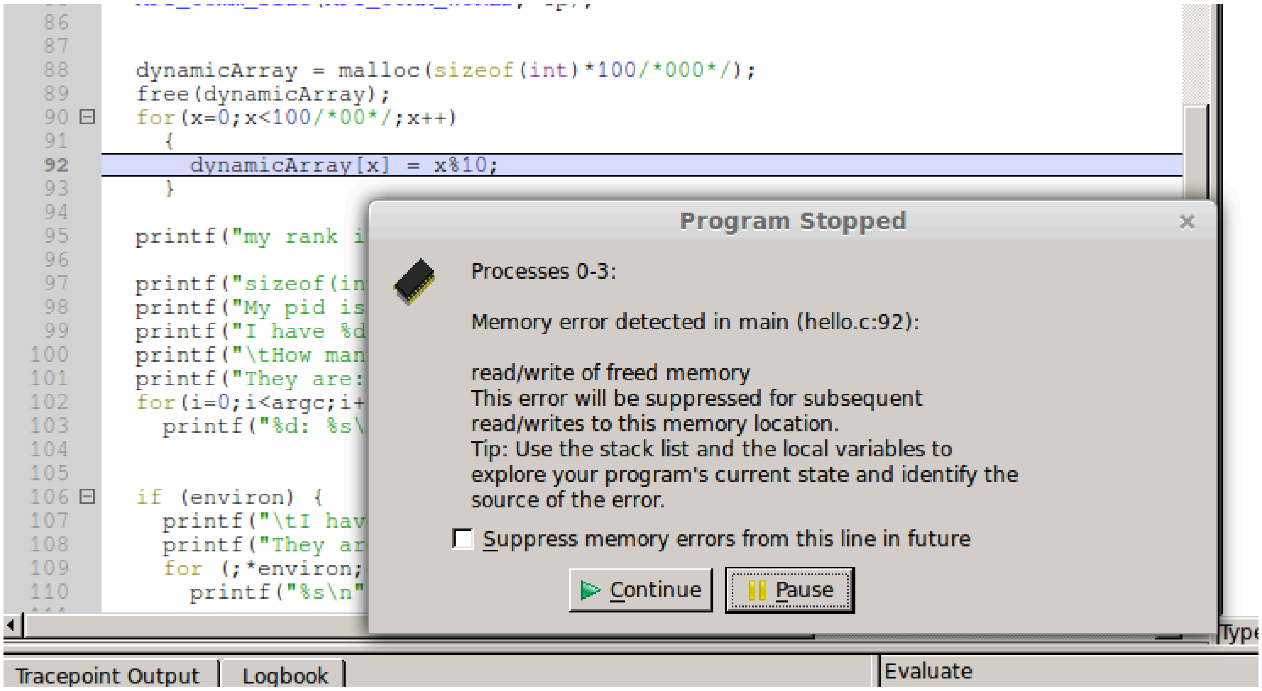
Dangling pointers are pointers that have had memory freed but which have not been set to null. It is often possible for subsequent code to keep on using the dangling pointer and to keep on getting valid-looking data until that memory is suddenly repurposed for something else.

This leads to unpredictable behavior, silent corruption and program crashes.

To enable detection of dangling pointers, the memory debugging settings must be set to one level higher than Fast. The term free-protect appears in the Enabled Checks window.

This level of memory debugging is usually all you need to find dangling pointer problems. When a dangling pointer is reused, DDT stops your program at the exact line of code that reuses it, with an error like this one:

Figure 6-1: Program Stopped - Memory error detected in main



```
 86
 87
 88  dynamicArray = malloc(sizeof(int)*100/*000*/);
 89  free(dynamicArray);
 90  for(x=0;x<100/*00*/;x++)
 91    {
 92      dynamicArray[x] = x%10;
 93    }
 94
 95  printf("my rank i
 96
 97  printf("sizeof(in
 98  printf("My pid is
 99  printf("I have %d
100  printf("\tHow man
101  printf("They are:
102  for(i=0;i<argc;i+
103    printf("%d: %s\
104
105
106  if (environ) {
107    printf("\tI hav
108    printf("They ar
109    for (;*environ;
110      printf("%s\n"
```

Processes 0-3:

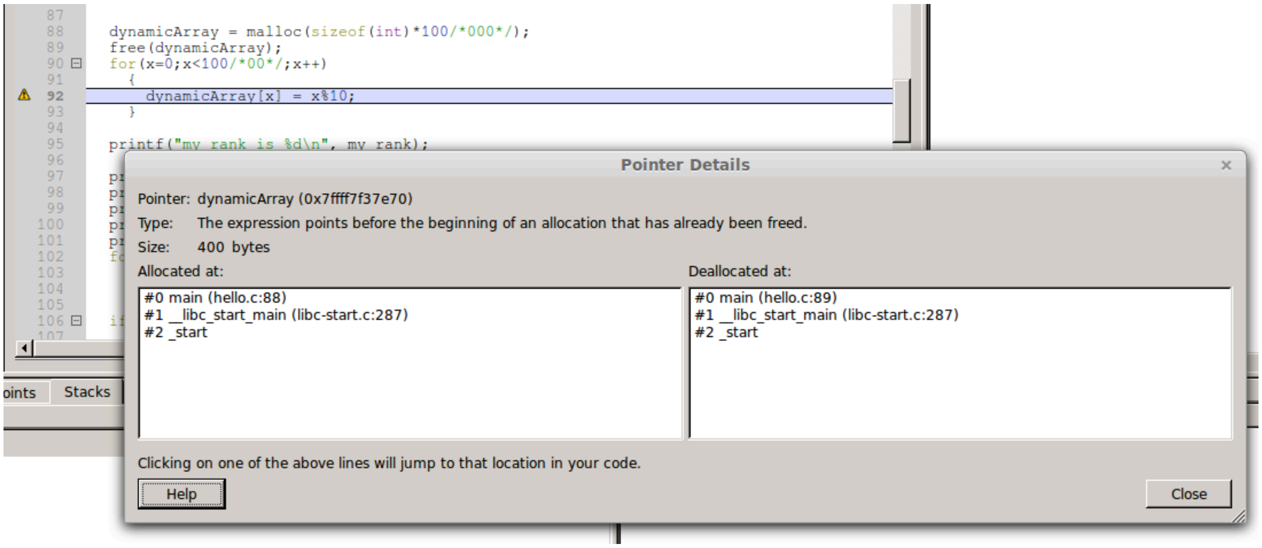
Memory error detected in main (hello.c:92):

read/write of freed memory
This error will be suppressed for subsequent read/writes to this memory location.
Tip: Use the stack list and the local variables to explore your program's current state and identify the source of the error.

☐ Suppress memory errors from this line in future

The debugger also shows which pointer is dangling and exactly where it was originally allocated. Right-click on any pointers or dynamically-allocated arrays and choose View Pointer Details from the menu:

Figure 6-2: The Pointer Details window



```
dynamicArray = malloc(sizeof(int)*100/*000*/);
free(dynamicArray);
for(x=0;x<100/*00*/;x++)
  {
    dynamicArray[x] = x%10;
  }

printf("my rank is %d\n", my_rank);
```

Pointer Details

Pointer: dynamicArray (0x7ffff7f37e70)
Type: The expression points before the beginning of an allocation that has already been freed.
Size: 400 bytes

Allocated at:

```
#0 main (hello.c:88)
#1 __libc_start_main (libc-start.c:287)
#2 _start
```

Deallocated at:

```
#0 main (hello.c:89)
#1 __libc_start_main (libc-start.c:287)
#2 _start
```

Clicking on one of the above lines will jump to that location in your code.

Arm DDT immediately indicates that this pointer is dangling (it indicates that the pointer points to an allocation that has already been freed) and shows the full stack of function calls that led to its allocation.

7. Checking pointer information

As described in Dangling pointers, the Pointer Details window provides a lot of information about any pointer:

- Whether the pointer is valid, not yet allocated or dangling.
- What size of allocation is or was made.
- The exact position and stack in the code that the pointer was allocated at, even if it is a dangling pointer that has since been freed.
- The exact position and stack in the code that the pointer was deallocated again, if it is indeed a dangling pointer.

In particular, the image in Dangling pointers shows that it was allocated at `hello.c:88`. Clicking on this immediately jumps to that location in the source code viewer. It also indicates where this memory was deallocated.

8. Reading or writing beyond array bounds or allocation ends

Reading a value from the start or end of the range of an array or other allocation is difficult.

Much of the time, it can go unnoticed, but it can also cause intermittent crashes, where the likelihood of a crash occurring or not depends on the tiniest of changes in runtime environment.

- Reading a value means polluting a calculation, as the resulting value or code path now depends on something unreliable and uncertain.
- Writing to such a location can cause uncertain behavior in other areas of the code that then re-use this now corrupted location.
- Both reading and writing can cause a crash if the address is outside of the program's allocated pages (usually 4096 bytes but some systems use much larger pages).

DDT prevents these kinds of error by working with the operating system to create a page after, or before, each allocation and makes it read and write protected. As soon as the protected memory is touched for a read or write, the Linux O/S notifies the debugger. This is known as Guard Pages, also known as Red Zones.

For codes with a relatively small number of large allocations, such as most scientific codes, and most F90 codes, the number of pages used as guard pages is small.

C++ codes often use significantly many small allocations and can exhaust process limits. For these types of codes DDT, offers an alternative setting known as fence checking or fence painting. This periodically validates a few bytes above and below an allocation to check for unexpected writes.

Note

This mode only checks writes, it cannot detect erroneous reads, and therefore Guard Pages mode is generally preferred where possible for a code.