Exposing Memory Corruption and Finding Leaks: Advanced Mechanisms in Ada

Emmanuel Briot[1], Franco Gasperoni[1], Robert Dewar[2], Dirk Craeynest[3], and Philippe Waroquiers[3]

[1] ACT Europe, 8 rue de Milan, 75009 Paris, France
{briot,gasperon}@act-europe.fr
[2] Ada Core Technologies
104 Fifth Avenue, New York, NY 10011, USA
dewar@gnat.com
[3] Eurocontrol/CFMU, Development Division
Rue de la Fusée, 96, B-1130 Brussels, Belgium
{dirk.craeynest,philippe.waroquiers}@eurocontrol.int

Abstract. This article discusses the tools that Ada offers to deal with dynamic memory problems. The article shows how the storage pools mechanism of Ada 95 can be extended to enpower developers when tracking memory leaks and memory corruption in their code. This Ada extension rests on the notion of "checked pools", i.e. storage pools with an additional `Dereference` operation. The paper describes how a particular instance of the checked pool, called the "debug pool", is implemented in the GNAT technology. Performance measurements for the use of debug pools are provided in the context of the Air Traffic Flow Management application at Eurocontrol.

1 Introduction

1.1 About Garbage Collection

Any system confronted with the possibility of memory leaks and memory corruption has to consider why garbage collection (GC) cannot be used as a solution to its potential dynamic memory problems. While it is true that, when practical, GC removes incorrect deallocations and dangling pointer problems, it is not a systematic panacea. For one thing, GC cannot deal completely with the problem of memory leaks. Consider, for instance, the following Java code:

```
global = new Huge (100000000);
```

If `global` is a static class field that is never set to `null` or updated with another reference, the memory allocated in `new Huge (...)` cannot be freed until the class in which `global` is declared is finalized, even though the data allocated in `new Huge (...)` may never be accessed.

Another potential and more serious problem with GC exists for systems requiring a guaranteed (predictable) response time. We will come back to the issue of GC in the conclusion.

1.2 Dynamic Memory Allocation and Ada Safety Nets

Ada provides a number of safety nets to help programmers catch some common dynamic memory handling mistakes. All pointers are, for instance, set to `null` upon creation or deallocation and an exception is raised if a program tries to dereference a `null` pointer. Furthermore, the accessibility rules of access types of Ada 95 are designed to prevent dangling references when the scope of the pointee is inside that of the access type. These rules help to ensure that the most obvious cases of memory corruption are avoided.

When it comes to memory deallocation, Ada makes it possible to allocate on the stack (and hence automatically deallocate) all objects created through an access type local to a subprogram. In GNAT, for instance, the storage allocated for type `String_Access` in the following code excerpt, will be allocated on `P`'s stack and automatically freed upon `P`'s return.

```
procedure P is
   type String_Access is access all String;
   for String_Access'Storage_Size use 100_000;
   A : String_Access := new String (1 .. 1_000);
begin
   ...
end P;
```

1.3 User-Defined Storage Managment in Ada 95

Ada 95 gives the programmer complete freedom for the allocation/deallocation algorithms to use for a given access type through the mechanism of storage pools [1]. A storage pool is an abstract limited controlled type with 3 additional abstract operations: `Allocate`, `Deallocate`, and `Storage_Size`. In a concrete storage pool type `SP_Type`, the programmer must define the bahavior of `Allocate`, `Deallocate`, `Storage_Size`, and possibly `Initialize` and `Finalize`. Then when the programmer writes

```
Pool : SP_Type;
type A_Type is access ...;
for  A_Type'Storage_Pool use Pool;
```

it requests the Ada compiler to use `SP_Type`'s `Allocate` and `Deallocate` every time memory is allocated or deallocated for `A_Type`. See section 13.11 in [1] for details.

Ada implementations are free to provide specialized storage pools in addition to the standard (default) one. GNAT, for instance, provides 2 specialized pools:

- `System.Pool_Size.Stack_Bounded_Pool`, a stack-bounded pool where dynamic memory allocation is done on the stack and memory is globally reclaimed by normal stack management. GNAT uses this pool for access types with a `Storage_Size` representation clause as shown in the example of the previous section.

– `System.Pool_Local.Unbounded_Reclaim_Pool`, a scope-bounded pool where dynamic memory allocation is done using the system `malloc()` routine but which is automatically freed when the scope where the storage pool object is declared is exited. A programmer can use this pool for access types that are locally declared and for which he is not in a position to provide a maximum `Storage_Size`, or whose maximum `Storage_Size` would exceed the overall stask size.

When writing user-defined storage pools the main catch is the issue of alignment, i.e. how to ensure that the portion of memory returned by `Allocate` is aligned on a given byte boundary [2]. Note that the requested byte boundary could be greater than the maximum memory alignement for the underlying processor. If the programmer is trying to allocate data to be placed in a data cache, for instance, he may wish to specify the data cache line size as the alignment.

2 Extending Ada's Storage Pools

Despite the help and safety checks that Ada 95 provides, there are three type of programming problems that Ada does not address in full generality:

– Memory leaks (i.e. forgetting to deallocate dynamically allocated storage);
– Incorrect deallocation (i.e. deallocating unallocated memory);
– Dangling references (i.e. accessing deallocated or unallocated memory).

Because storage pools provide no means to check dereferences, GNAT offers a special type of storage pool, called a "checked pool", with an additional abstract primitive operation called `Dereference`. `Dereference` has the same parameter profile as the subprograms `Allocate` and `Deallocate` and is passed the same information. `Dereference` is invoked before dereferencing an access type using a checked pool. Its intended role is to do some checking on the reference, e.g. check that the reference is valid.

Checked pools are abstract types. They only act as a framework which other tagged types must extend. A concrete implementation is provided in the GNAT library package `GNAT.Debug_Pools`. This package was developed in collaboration with Eurocontrol.

3 Debug Pools

The goal of a debug pool is to detect incorrect uses of memory, specifically: incorrect deallocations, access to invalid memory, and memory leaks. To use a debug pool developers need to instrument their code for each access type they want to monitor, as shown below in the lines marked with the special comment `-- Add`. The debug pool reports errors in one of two ways: either by immediately raising an exception, or by logging a message that can be printed on standard output, which is what we have decided to do in the following example. Thye example contains a number of typical errors that the debug pool will point out.

```
File p.ads
----------
 1. with GNAT.Debug_Pools; use GNAT.Debug_Pools;          --  Add
 2. with Ada.Unchecked_Deallocation;
 3. with Ada.Unchecked_Conversion; use Ada;
 4.
 5. procedure P is
 6.    D_Pool : GNAT.Debug_Pools.Debug_Pool;              --  Add
 7.
 8.    type IA is access Integer;
 9.    for  IA'Storage_Pool use D_Pool;                   --  Add
10.
11.    procedure Free is new Unchecked_Deallocation (Integer, IA);
12.    function Convert is new Unchecked_Conversion (Integer, IA);
13.
14.    Bogus : IA := Convert(16#0040_97AA#);
15.    A, B  : IA;
16.    K     : Integer;
17.
18.    procedure Nasty is
19.    begin
20.       A := new Integer;
21.       B := new Integer;
22.       B := A;         --  Error: Memory leak
23.       Free (A);
24.       K := B.all;     --  Error: Accessing deallocated memory
25.       K := Bogus.all; --  Error: Accessing unallocated memory
26.       Free (B);       --  Error: Freeing deallocated memory
27.       Free (Bogus);   --  Error: Freeing unallocated memory
28.    end Nasty;
29.
30. begin
31.    Configure (D_Pool, Raise_Exceptions => False);     --  Add
32.    Nasty;
33.    Print_Info_Stdout (D_Pool, Display_Leaks => True); --  Add
34. end P;
```

For each faulty memory use the debug pool will print several lines of information as shown below[1]:

```
Accessing deallocated storage, at p.adb:24 p.adb:32
   First deallocation at p.adb:23 p.adb:32
Accessing not allocated storage, at p.adb:25 p.adb:32
Freeing already deallocated storage, at p.adb:26 p.adb:32
```

[1] The actual output shows full backtraces in hexadecimal format that we have post-processed with the tool `addr2line` to display the information in symbolic format.

```
   Memory already deallocated at p.adb:23 p.adb:32
Freeing not allocated storage, at p.adb:27 p.adb:32

Total allocated bytes:  8
Total logically deallocated bytes:  4
Total physically deallocated bytes:  0
Current Water Mark:  4
High Water Mark:  8

List of not deallocated blocks:
Size:  4 at: p.adb:21 p.adb:32
```

The debug pool reports an error in the following four cases:

1. Accessing deallocated storage
2. Accessing not allocated storage
3. Freeing already deallocated storage
4. Freeing not allocated storage

As the reader can see the debug pool displays the traceback for each faulty memory access, memory free, and potential memory leaks. The depth of the traceback is programmer-configurable. Note how the information reported when accessing deallocated storage or when freeing already deallocated storage is much richer than what one can get with a debugger since the debug pool indicates the backtrace of the program location where the original deallocation occurred.

In addition to the above, the debug pool prints out the traceback for all memory allocations that have not been deallocated. These are potential memory leaks. The debug pool also displays the following memory usage information :

1. High water mark: The maximum amount of memory that the application has used at the point where `Print_Info_Stdout` is called.
2. Current water mark: The current amount of memory that the application is using at the point where `Print_Info_Stdout` is called.
3. The total number of bytes allocated and deallocated by the application at the point where `Print_Info_Stdout` is called.
4. Optionally the tracebacks of all the locations in the application where allocation and deallocation takes place (this information is not displayed in the previous output). This can be used to detect places where a lot of allocations are taking place.

It is worth noting that debug pools can be used in several important situations:

– One does not have to wait for the program to terminate to collect memory corruption information, since the debug pool can log problems in a file and developers can look at this file periodically to ensure that no problems have been detected so far by the debug pool mechanism.

– Various hooks are provided so that the debug pool information is available in the debugger. A developper can, for instance, interactively ask the status of a given memory reference: is the reference currently allocated, where has it been allocated, is the reference logically deallocated, where has it been deallocated, etc.

4 Debug Pool Implementation

The debug pools package was designed to be as efficient as possible, but has an impact on the code performance. This depends on the number of allocations, deallocations and, somewhat less, dereferences that the application performs.

4.1 Debug Pool Memory Release Strategy

Physical allocations and deallocations are done through the usual system calls. However, in order to provide proper checks, the debug pool will not immediately release a memory block when asked to. The debug pool marks the memory block as "logically deallocated" but keeps the released memory around (the amount kept around is configurable) so that it can distinguish between memory that has not been allocated and memory that has been allocated but freed. This allows detection of dangling references to freed memory, which would not be possible if memory blocks were immediately released as this memory could be reused by a subsequent call to the system `malloc()`.

Retaining memory could be a problem for long-lived applications or applications that do a lot of allocations and deallocations. To address this the following parameters were added to the debug pool:

1. **Maximum Logically Freed Memory**: This parameter sets the limit of the amount of memory that can be logically deallocated, but not released to the system. When this limit is reached, the debug pool will start freeing memory.
2. **Minimum to Free**: This parameter indicates how much memory the debug pool should try to release to the system at once. For performance reasons it is better to free several blocks of memory at the same time.

The debug pool can use one of two algorithms to select the memory blocks to hand back to system memory:

1. **First deallocated - First released**: The first block that was deallocated by the application is the first to be released to the system.
2. **Advanced block scanning**: This more expensive algorithm parses all the blocks currently allocated, and finds all values that look like pointers. If these values match a currently deallocated block, that block will not be physically released. This ensures that dangling pointers are properly detected when the access type is dereferenced. This algorithm is only a good approximation: it is not guaranteed to detect all dangling pointers since it doesn't check task stacks or CPU registers.

4.2 Data Structures

The debug pool will respect all alignments specified in the user code by aligning all objects using the maximum machine alignment. This limits the performance impact of using the debug pool and, as we will show below, allows to quickly compute the validity of a memory reference.

Global Structures The debug pool contains a packed boolean array. Each entry in this array matches a location in memory to indicate whether the corresponding address is under control of the debug pool (1 bit per address). Because all the addresses returned by the debug pool are aligned on the `Maximum_Alignment` of the underlying machine, the array index of each memory address `Addr` can be quickly computed as follows:

```
array index = (Addr - Heap_Addr) / Maximum_Alignment
```

where `Heap_Addr` is the address of the beginning of the application's heap.

The initial size of the global array is small. During program execution the array grows, doubling its size every time more room is needed.

Local Structures For each allocated memory block, the debug pool stores the following data in a header, located just before the memory block returned by the debug pool. The overall size of this header is a total of 16 bytes on 32-bit machines and includes:

1. The size of the allocated memory block. This is needed for the advanced block scanning algorithm described in the previous section. This value is negated when the block of memory has been logically freed by the application but has not yet been physically released.
2. A pointer to the next allocated or logically deallocated memory block.
3. A pointer to the allocation traceback, i.e. the traceback of the program location where this block was allocated.
4. A pointer to the first deallocation traceback. For memory blocks that are still allocated this pointer is used to point back to the previously allocated block for algorithmic convenience.

To save memory, the tracebacks are not stored in the header itself, but in a separate hash table. That way, only one instance of the traceback is stored no matter how many allocation are done at that program location.

All the allocated blocks are stored in a double-linked list, so that the advanced block scanning algorithm can find all of them and look for possible dangling pointers. This list is also used to report potential memory leaks.

When a block is deallocated by the application code, it is removed from the allocated blocks linked list and moved to the deallocated blocks list. This is the list from which, if needed, memory blocks will be returned to system memory.

The debug pool must be usable in a multi-tasking application, and has therefore been made thread-safe. Any time a new memory block is allocated or an existing block deallocated, the GNAT runtime is locked for concurrent accesses.

5 Debug Pool and General Access Types

A debug pools is a powerful mechanism to help debugging memory problems in an application. There is, currently, a limitation with general access types. As shown in the following example access to local variables can not be properly handled by debug pools:

```
with GNAT.Debug_Pools; use GNAT.Debug_Pools;
procedure Q is
   D_Pool : GNAT.Debug_Pools.Debug_Pool;
   type IA is access all Integer;
   for  IA'Storage_Pool use D_Pool;

   Ptr : IA;
   K   : aliased Integer;
begin
   Configure (D_Pool);
   Ptr := K'Access;
   Ptr.all := 4;
   --  Exception GNAT.Debug_Pools.Accessing_Not_Allocated_Storage raised
end Q;
```

Because the memory pointed by `Ptr` wasn't allocated on the heap, the debug pool will consider this as an invalid dereference, and will report an error.

6 Partition-Wide Storage Pool

The intention of the storage pool design is that an access type without a storage pool clause use a default storage pool with appropriate characteristics. An obvious idea is to provide a facility for changing this default, and indeed given the usefulness of debug pools in finding memory corruption problems, wanting to use a partition-wide debug pool by default would be sensible. There are, however, a number of difficulties in providing this feature.

The first difficulty is that this requires the entire run-time and third party libraries to be recompiled for consistency. This is because an object allocated with a given allocator must be deallocated with the matching deallocator. This can only be guaranteed if all the application libraries are compiled in the same configuration, e.g. using a debug pool as the default. In the case of third party libraries this may not be possible since the sources of such libraries may not be available.

Another issue is that certain `new` operations may malfunction when the default pool is changed. A most obvious and notable example is that the body of the debug pool itself contains allocators, so if the wish is to change the default storage pool to be this debug pool there will be an infinite recursion. This can be fixed by making the pool to be used for each of these types within the implementation of the debug pool explicit.

However, again the issue of third party libraries arises in an even fiercer form. It may be quite impractical to analyze a third party library to find those cases (e.g. performance requirements) where the use of the debug pool would disrupt the correct operation of the third party library, even if sources were available.

The design of the GNAT runtime has introduced the `System.Memory` abstraction partly to allow a completely different approach to replacing the default storage pool, which is to essentially replace the interface to the system `malloc()` and `free()` by a set of user-supplied routines.

7 Debug Pools in a Real Application

The development of the debug pool was sponsored by Eurocontrol. The CFMU (Central Flow Management Unit) is in charge of the European flight plan processing and air traffic flow management.

The total down-time for the CFMU has to be kept to a strict minimum, and therefore it is important to eliminate as many memory leaks in the application as possible.

In addition, access to invalid blocks of memory has proved to be difficult to find in more than 1.5 million lines of code. For example, several years ago such a bug took approximately 3 person-weeks of work to isolate; another one early last year required roughly 3 person-days.

To detect such problems as early as possible, a specialized build environment is now set up by CFMU using a common debug pool for all access types.

7.1 Use of Debug Pools

To give an idea of the CFMU development environment: at the time of writing, the set of Ada sources for TACT, one of the CFMU applications, consists of over 4,354 files: 1,993 specifications and 2,362 bodies and subunits. The total size of these sources is roughly 1.25 million lines of code. In these units, 675 access types are defined in 427 different files.

As the use of the GNAT debug pools has a performance impact, we obviously want to make it optional when building the TACT application. An emacs script was developed to automatically insert the appropriate code snippets to activate the debug pool.

7.2 Special Cases

For a small subset of all access types in the TACT code, the use of GNAT debug pools is not appropriate. This is because the access type is used for:

1. access to parts of untyped memory chunks (5 access types);
2. conversion to untyped memory (1 type);
3. access to memory allocated by C code in the OS or by system calls (13 types);

4. access to shared memory (4 types);
5. access to objects allocated in block via an array (1 type);
6. test code of a reference counting package (1 type), which was created to enable the detection of dangling pointers before the debug pool mechanism was available.

In retrospect, only 25 of the 675 access types in our application (less than 4%) cannot be associated to the debug pool.

7.3 Impact on Executable Size

One of the build options in the TACT development environment, is to create one executable for the whole system instead of creating separate executables for each logical processing. This configuration contains all code of the complete system and gives us a good measure of the impact of debug pools on the size of executables. The impact on the size of executables and on the image of processes is only in the order of 5 percent and hence negligible.

Table 1. Size of Executables (sizes are in bytes, bss = uninitialized data)

	file size	section sizes - 'size'			
	'ls'	text	data	bss	total
no debug pool	195,904,512	76,192,889	12,199,776	70,560	88,463,225
debug pool	201,839,848	80,988,283	12,343,312	70,560	93,402,155
increase	+3.0%	+6.3%	+1.2%	0.00%	+5.6%

7.4 Run-Time Performance Impact

To get an indication of the run-time performance impact of debug pools, a realistic test is used. The test consists of running the TACT application and setting up its normal environment including meteo forecast data for the complete pan-European airspace. Furthermore, 853 flight plans are injected in the system, and 55 regulations are created all over Europe to force several of these flights being delayed.

Then, the arrival of a large number of radar reports is simulated over time, who, just as in real operations, provide real-time corrections to the predicted positions of these flights. These radar reports were generated to induce different kinds of shifts, such as in position, flight level or time of overflying a point. Each of these reports implies a new calculation of the remaining flight profile of that specific flight, which could mean a change of the load distribution in the different airspaces and eventually a reallocation of flight slots for several flights.

This is a rather heavy test which exercises a reasonably large part of the TACT system. The Unix "time" system call is used to measure the performance: "real" is the elapsed clock time in seconds, "user" is the CPU time spent

executing user code and "system" is the CPU time spent in the OS itself (I/O operations, memory allocations, etc.). Typically, user + system is a good measure of the time needed for a job, regardless of the system load (within reason).

With the default CFMU debug pool configuration, the test runs roughly 4 times slower, hence the impact of extensively using debug pools on the performance is quite large.

Table 2. Execution times averaged over multiple runs

	real	user	system	user+system
no debug pool	1689.49	1213.90	17.54	1231.44
debug pool	5670.13	3766.97	1400.99	5167.96
increase	∗3.36	∗3.10	∗79.87	∗4.20

Depth of Stack Traces An important element in this slow-down is the computation of backtraces in the debug pool implementation. This is controllable with the `Stack_Trace_Depth` parameter, described as:

```
-- Stack_Trace_Depth. This parameter controls the maximum depth
-- of stack traces that are output to indicate locations of
-- actions for error conditions such as bad allocations. If set
-- to zero, the debug pool will not try to compute backtraces.
-- This is more efficient but gives less information on problem
-- locations.
```

The CFMU default value for this parameter is 10. When set to 0 or to the GNAT default of 20, respectively, the timing results of the test are given in the following table.

Table 3. Impact of Stack Trace Depth

	real	user	system	user+system
no debug pool	1689.49	1213.90	17.54	1231.44
debug pool (stack=0)	1898.30	1635.51	17.20	1652.71
increase	∗1.12	∗1.35	∗0.98	∗1.34
debug pool (stack=20)	8575.56	5442.96	2553.74	7996.70
increase	∗5.08	∗4.48	∗145.60	∗6.49

These results clearly show the majority of the slow-down is due to the computation of backtraces. When backtraces are disabled in the debug pool implementation, execution time only goes up one third. On the other hand, when the

maximum depth of backtraces is set to the GNAT default of 20, execution time increases with a factor of more than six!

So a compromise needs to be found. Depending on the performance of an application without debug pools and on the available resources, it can be interesting to have regular tests with debug pools enabled but without backtrace computation. If these indicate a heap problem, the test can then be rerun with a large value for the `Stack_Trace_Depth` parameter.

Scanning Memory before Releasing Another useful configuration parameter is `Advanced_Scanning`, described as:

```
-- Advanced_Scanning: If true, the pool will check the contents
-- of all allocated blocks before physically releasing
-- memory. Any possible reference to a logically free block will
-- prevent its deallocation.
```

CFMU sets this parameter to the non-default value of True. To get an idea of the overhead this entails, the same tests are run but now with the GNAT-default value of False. The timing results are given in the following table.

Table 4. Impact of No Advanced Scanning

	real	user	system	user+system
debug pool (stack=0)	1825.70	1507.33	17.32	1524.65
vs. with scanning	−72.60	−128.18	+0.12	−128.06
	96.2%	92.2%	100.7%	92.3%
debug pool (stack=10)	5507.90	3621.19	1397.66	5018.85
vs. with scanning	−118.18	−142.73	−3.60	−146.34
	97.9%	96.2%	99.7%	97.7%
debug pool (stack=20)	8385.11	5301.28	2530.61	7831.89
vs. with scanning	−190.45	−141.68	−23.13	−164.81
	97.8%	97.4%	99.1%	97.9%

This shows that the performance cost of "Advanced Scanning" is quite small: it only requires between 2% of the user plus system time, if the `Stack_Trace_Depth` parameter is set to the default GNAT value of 20, and less than 8% if the computation of backtraces is disabled completely. For these test runs, this accounts for roughly 2-3 minutes of CPU time on a total between 25 and 130 minutes.

7.5 Results Obtained

One of the important results obtained through the use of debug pools, is that its extensive reporting has indicated some bizarre heap usage in our application, which caused a serious performance drain.

All processes on the TACT server and all MMIs on various workstations need access to a large amount of changing environment data (definitions of aerodromes, points, routes, etc.). Each of these processes maintains a local cache of the data it needs, and these caches are kept synchronised by inter-process communication via encoded buffer transfers.

Closer examination showed that due to an incorrectly set boolean variable, the encoding and decoding of these buffers was not done in the intended compact binary format but in the bulky textual format (intended for easy human interpretation, though approx. two orders of magnitude larger). This not only implied a lot of unneeded heap usage, detected through the debug pool reporting, but also very inefficient inter-process communication.

Another important result of running our large suite of regression tests with the debug pool enabled, is that our code now is shown to be free of heap corruptions.

And as our system builds always include the systematic execution of the full regression test suite, regular builds with the debug pool enabled offer a protection against the introduction of new heap corruptions in our code.

There are some limitations. Allocations in non-Ada code fall outside the scope of the GNAT debug pools. Nor are allocations on the stack through general access types taken into account: a possible future enhancement?

8 Conclusion

This paper has shown how the notion of checked pools, an extension of the storage pools concept of Ada 95, can be put to profit to implement debug pools to track down memory corruptions and possible memory leaks in a user application. Are debug pools the ultimate solution to dynamic memory problems in Ada code?

Probably not: like for all things if all you have is a hammer all your problems look like a nail and it is important to offer developers a choice of tools and approaches when tackling memory managment issues.

Ada was designed to allow garbage collection. Currently no native Ada implementation offers it. Wouldn't it be nice to combine the power of debug pools with the flexibility of a real-time garbage collecting implementation?

References

1. Taft, S.T., Duff, R.A., Brukardt, R.L. and Pldereder, E.; *Consolidated Ada Reference Manual. Language and Standard Libraries*, ISO/IEC 8652:1995(E) with COR.1:2000, Lecture Notes in Computer Science, vol. 2219, Springer-Verlag, 2001.
2. Barnes, J.; *Storage Pool Alignment*, Ada User Journal, pp. 182-187, vol. 19, number 3, October 2001.