Careless Memories

Debugging the heap manager

by Julian Bucknall

I have a confession. Last month's article nearly didn't make it. Not because of the actual text, you understand, but because the code had a bug. I was a little late anyway, but just as I was tidying up the code and making my final passes for grammar, compiles and tests, I got an *Invalid Pointer Operation* exception. The IDE positioned itself at a call to `FreeMem`. Urk.

Now, you may have been lucky and never had this particular error in Delphi, so let me explain in layman's terms what it meant. I'd trashed the heap. I had a memory overwrite or something that stomped values the heap manager was using and now it's all over. Go directly to jail. Do not pass Go. Do not collect £200.

Even if you've never had this particular error, I'm sure you've had that prickly feeling of cold sweat on your forehead as you suddenly realize you're looking at working way into the night to fix a sudden bug. I tried that night, but I was too tired. I sent a message to Our Esteemed Editor apologizing for the delay, saying I'd look at it in the cool calm of my air-conditioned office the next morning.

Well, it was just as bad then. I couldn't see where the problem was for the life of me. OEE had replied with a low jab about how surely TurboPower had lots of debugging tools to track the bug down. OK, so I tried CodeWatch from our new Sleuth QA Suite 2, with the new heap memory checking, but unfortunately it didn't spot the overwrite or whatever it was.

In fact it took me a good couple of hours, off and on, that day to finally nail it down. I'd like to take this article to talk about what I learned about the heap manager in Delphi 5 in my debugging odyssey, what structures and algorithms it uses, and to see what code and structures I used to finally track this particular bug down.

Ordinary World

To the casual programmer, the Delphi 5 heap manager is a black box. You ask for memory with `GetMem` or `New`, or a call to a `Create` constructor, and you get it. When you finish with that memory, you give it back with `FreeMem`, `Dispose` or `Destroy`. If you don't, you get a memory leak (which Sleuth CodeWatch can easily point out to you). Easy peasy, no worries mate.

It's when things go wrong with the heap that you are left with this nagging feeling that maybe you should know more.

The first point to note is that the standard heap manager uses only one exception: the aforementioned *Invalid Pointer Operation*. It's not well known that there are actually 14 separate errors that can cause this exception to be raised and they all occur when memory is being freed. If an error occurs, the standard heap manager merely raises the one exception and merrily discards the error code. Sometimes, however, as in my case, it would be nice to know the actual error code. Maybe it would trigger some 'ah-ha!' thought in my mind.

How do we find this error code? Well, you could use the debugger, but it's a bit of a pain to find the right place to place a breakpoint and there's a much easier way.

In Delphi 2, Borland encapsulated the heap manager in such a way that you can easily replace it. Indeed, they give you a replacement heap manager in the box: the `ShareMem` unit. `ShareMem` is an interface to a DLL that implements a heap manager. If you use Delphi DLLs with your Delphi application then all modules will have their own heap managers. Apart from one major case, if you're careful about making sure you free a memory block in the same module (ie EXE or DLL) that you allocated it in, you can go on your merry way without using `ShareMem`. The exception is if you pass long strings between modules. Here, you have to be *extremely* careful about what you do: the compiler will, at the drop of a hat, reallocate the memory held by a long string and, if the reallocation occurs in a different module to the original, you can say bye-bye to your application. Borland thus provided a heap manager in a DLL (BORLNDMM.DLL) that all modules can use: they ignore their own heap managers and switch to the shared one, using the `ShareMem` unit, and live together happily ever after.

(Actually, we came across another type that can cause disastrous memory problems between DLL and EXE during some debugging at TurboPower. The books and the help all say be wary of long strings, but sometimes the long strings are hidden, in variants, for example. And that's what we had: a variant containing a long string.)

Playing With Uranium

So how do you replace the heap manager? (I know I seem to have gone off the beaten track with respect to my original problem, but bear with me.) You have to write three routines: one that gets a memory block, one that frees a memory block and one that reallocates a memory block. Once you've written those to follow the correct interface, you save the current heap manager by calling `GetMemoryManager`, and replace it with yours by calling `SetMemoryManager`. These two routines use a simple record of routine pointers known as `TMemoryManager`. (Roberto De Marini talked about this briefly in Issue 19, in *Tips & Tricks*.)

```
unit AAHpMin;
{WARNING: this unit *must* appear first in your project's
 uses list.}
interface
implementation
uses
  Windows; // OK to use Windows unit: it allocates no memory
var
  OrigHeap : TMemoryManager;
  OurHeap  : TMemoryManager;
function OurGetMem(Size : integer) : pointer;
begin
  Result := OrigHeap.GetMem(Size);
end;
function OurFreeMem(P : pointer) : integer;
begin
  Result := OrigHeap.FreeMem(P);
end;
function OurReallocMem(P : pointer; Size : integer) :
  pointer;
begin
  Result := OrigHeap.ReallocMem(P, Size)
end;
procedure InitializeUnit;
begin
  GetMemoryManager(OrigHeap);  {get the original manager}
  {set up our heap manager}
  OurHeap.GetMem := OurGetMem;
  OurHeap.FreeMem := OurFreeMem;
  OurHeap.ReallocMem := OurReallocMem;
  SetMemoryManager(OurHeap);   {replace heap mgr with ours}
end;
procedure FinalizeUnit;
begin
  SetMemoryManager(OrigHeap);  {restore original manager}
end;
initialization
  InitializeUnit;
finalization
  FinalizeUnit;
end.
```

➤ *Listing 1: The minimal heap manager replacement.*

The get memory routine you write is a function that accepts an integer `Size` parameter and returns a pointer to a block of bytes that is that size, or if it can't allocate such a block, a nil pointer. The free memory routine is a function that accepts a pointer to a memory block and returns an integer error code to say whether the block was freed properly (error code 0) or not (any other value). It is this error code that the standard heap manager throws away and that we want. (Actually it's a bit of a stretch to say the heap manager 'throws it away', since it does store it in an internal variable, albeit inaccessible to the outside world.) The re-allocate memory routine is a function that takes a pointer parameter and an integer `Size` parameter and returns a pointer to the newly resized memory block (or nil if it couldn't do it).

Let's write the minimal heap manager replacement. In fact, it will be so minimal that it can act as a template for much of the other work we're going to do. Listing 1 has the complete code for the `AAHpMin` unit. If you look at it, all it does is to implement the three replacement routines such that they merely call the original heap manager. If you write a small application and add `AAHpMin` as the *first* unit in the *project's* `uses` clause (it should never be used anywhere else) you can run it without any problems whatsoever: it acts just as if the `AAHpMin` unit wasn't even there.

```
function HeapErrorMsg(aErrorCode : integer) : shortstring;
begin
  case aErrorCode of
     1 : Result := 'operating system returned an error on release';
     2 : Result := 'operating system returned an error on decommit';
     3 : Result := 'list of committed blocks looks bad';
     4,5,6 : Result := 'filler block is bad';
     7 : Result := 'current allocation zone is bad';
     8 : Result := 'couldn''t initialize';
     9 : Result := 'used block looks bad (invalid pointer? double free?)';
    10 : Result := 'prev block before a used block is bad';
    11 : Result := 'next block after a used block is bad';
    12 : Result := 'free list is bad';
    13 : Result := 'free block is bad';
    14 : Result := 'free list doesn''t correspond to blocks marked free';
  else
    Result := 'unknown error message';
  end;
end;
function OurFreeMem(P : pointer) : integer;
begin
  Result := OrigHeap.FreeMem(P);
  if (Result <> 0) then
    WriteLogFreeErr(HeapErrorMsg(Result));
end;
```

➤ *Listing 2: Getting the error code from a failed FreeMem.*

Now, though, let's make a copy of the `AAHpMin` unit and call it `AAHpDbug`. We'll be constantly building up this `AAHpDbug` unit in this article so that it can act as a complete heap debugger. We'll make one change to the `AAFreeMem` function: if the return value from the call to the original function is non-zero, we'll log it to a heap debug log file.

Why not raise an exception? I can hear at least one of my readers ask. Why not indeed? Well, the problem is that raising an exception will cause at least one memory allocation, maybe more (the parameter to `Exception.Create` is a long string, remember), and maybe even a memory block deallocation. Since we're reporting a problem in the heap, we may get ourselves into a fairly nasty recursion here as we try and report some heap trashing when we are trying to report some heap trashing as we are trying... Well, you get the picture. Much safer to assume that the heap is highly unstable at this point and not make any allocations at all. Hence, I've been very careful about making sure that I don't make any allocations whilst I'm reporting a heap problem. In fact, as you'll see, we shall have to be very careful about using the heap manager when we're debugging the heap manager: one of my early experiments resulted in a long string being allocated to report an allocation (I was logging all gets and frees of memory). Massive recursion then set in, with the usual results.

Listing 2 shows the first iteration of the heap debug unit. As you can see, it has a set of hard-coded short strings describing each of the possible heap corruption errors (I lifted this directly out of GETMEM.INC, which is shipped with Delphi 5 Enterprise).

Using `AAHpDbug` with my problem app gave me a log file saying that the previous block before a block I was freeing is bad. Ohhh-kayyy...

Something I Should Know?

To understand this we need to delve some more into how Delphi manages blocks. Astute readers might have noticed something peculiar. Although we specify a `Size` parameter when we allocate a block of memory, we don't specify one when we deallocate a block. How does the heap manager 'know' how big a memory block is?

Enter memory alignment, heap granularity and the Size/Flags value. Time for some more theory. The first point to take in is that the Delphi heap manager *always* returns a memory block that is aligned on a 4-byte boundary. What that means is that when you call `GetMem`, or any other primitive routine that allocates memory, the address returned to you is divisible by 4 (if you look at the address in binary the lower two bits are clear, or if you look at it in hex the least significant digit is 0, 4, 8, or C). So what? Well, alignment plays a great part in efficiency: if you have four-byte variables (`longints`, `pointers` etc) aligned on a four-byte boundary using them is much more efficient than not. This is a consequence of the PC hardware.

And heap granularity? Well, if memory blocks are returned on a four-byte boundary then it makes sense to make memory blocks multiples of four bytes. This makes it much easier to maintain the alignment. And so that's what the heap manager does: it rounds up each memory allocation to the next four bytes. Hence, if you allocate from one to four bytes, you'll get four bytes, from five to eight bytes, eight bytes, from nine to twelve bytes, twelve bytes, and so on. (Those of you who've investigated the heap manager before may be jumping up and down at this point, but bear with me.)

This is all very well, but again the problem occurs in that the heap manager doesn't know the size of a memory block. So it makes sure it can work out the memory block size: it adds an extra four bytes to your allocation and uses it as a long integer value to store the size of the memory block. This value is placed just before the memory block that's returned by the call to `GetMem`. In other words, if you allocate a memory block and get the pointer value `X`, the `longint` at address `X-4` is the size of the block.

Well, nearly. If you actually look at this `longint` value for an allocation, you may see a value that seems wrong: it may be odd, or it may not be a multiple of four. What's going on? Is Julian losing it? Think of it like this: the size value is going to be a positive number that's also a multiple of four. Considering it as a binary value, three bits will always be clear: the most significant bit (otherwise the `longint` value will be negative) and the two least significant bits (otherwise the value won't be a multiple of four). The heap manager makes use of these three bits as flags for certain conditions. Hence the `longint` value is known as the Size/Flags value.

Bit 0 (the least significant bit), when set, signifies that the previous memory block to this one has been freed and so, if this one is freed, the two blocks can be coalesced into one at that time. Bit 1, when set, signifies that this block is in use (ie it has been allocated but not freed). This bit is used to check that a memory block is not freed twice (which, if not checked for, could cause a major heap corruption problem). Bit 31 (the most significant bit), when set, is used internally to designate a filler block that, due to the flow of allocations and frees, is too small to be used.

The interesting bits are bit 0 and 1. Bit 1 is set when you allocate a memory block and is cleared when you free it. This helps track down double-frees: when you manage to write code that frees a pointer twice. Bit 0 will come on and off as you use your memory block, depending on the ebb and flow of allocations in the heap manager. If the previous block to this one is freed, the heap manager sets the flag on for this next block, if the previous block is allocated again, the heap manager makes sure that the flag is cleared.

Assume the heap manager is about to free a block. The flag in the Size/Flags value for this block indicates that the previous block is free, therefore the two can be coalesced. But how does the heap manager *find* the previous block? It may be 20 bytes in size or 200 bytes: the memory block we're freeing doesn't know.

To Whom It May Concern

Let's discuss what happens when a block is freed and both the previous and next blocks are being used. Ideally, we would like to be able to reuse this block in the future (we don't want to *not* reuse it, otherwise we'd run out of memory pretty quickly). The heap manager therefore adds the block to a *free list*, a doubly linked list of blocks that can be reused. When the user wants another memory block, the heap manager can quickly walk the free list looking for a block of the right size. Since it's a doubly linked list, the heap manager must use part of the memory block to store a forward and backward link. The heap manager must also store the length of the block. This means the heap manager requires the memory block to be at least 12 bytes in size (two pointers for the links and a `longint` size). When a block is deleted, the heap manager sets up these twelve bytes at the beginning of the block. Listing 3 shows the definition of this record structure: the `TFree` record. The rest of the memory block is assumed to follow after this structure.

Thus, not only does the heap manager allocate memory blocks such that they are aligned on a four-byte boundary, and that they are a multiple of four bytes in size, it also makes sure the memory blocks are at least 12 bytes in size, with four bytes used for the Size/Flags value. So you can see if you want to allocate from one to eight bytes, you actually get an eight-byte block, with the previous four bytes being the Size/Flags value. Allocating such a small amount will always reduce the amount of usable memory by 12 bytes.

However, even with the `TFree` record being overlaid on the first twelve bytes of our memory block, we certainly haven't helped the

```
type
  PFree = ^TFree;
  TFree = packed record
    prev: PFree;
    next: PFree;
    size: Integer;
  end;
```

➤ *Listing 3: The TFree record.*

next block along to find this `TFree` record when it needs to. This is where it gets clever. Let's free a memory block and suppose the flag for it states that the previous block is free, so we need to coalesce. Let's assume the previous memory block was exactly 12 bytes in size. This means that the `longint` value before our Size/Flags value is the size of the previous memory block (that is, 12). The previous memory block can't be any smaller than this, of course, but it could be larger, say 16 bytes or bigger. What the heap manager does in that case is use the final four bytes of the memory block as a `longint` size value. So, no matter how big the previous freed block may be, the `longint` prior to our memory block is the size of this previous block.

Here's what the memory looks like with a previous block of 12 bytes:

```
Previous block:
  Next pointer
  Prev pointer
  Size (=12)
Our block:
  Size/Flags
  ..memory..
```

And here it is if the previous block is larger than 12 bytes:

```
Previous block:
  Next pointer
  Prev pointer
  Size (>=16)
  ..memory..
  Size (>=16)
Our block:
  Size/Flags
  ..memory..
```

So, in all cases, we can, if required, find the previous block. Note that it goes without saying that we know where the next block is because we know the size of our own block.

By the way: the Delphi heap manager doesn't have just one free list, as I intimated just now. It has a free list to store blocks of 12 bytes, a free list for blocks of 16 bytes, for 20 bytes, and so on up to 4,096 bytes. It then has a free list for larger blocks. The reason for this is that the majority of the objects we create and memory we allocate are less than 4Kb in size (we'll see how to show this for your applications in a moment). By having separate free lists for different block sizes, the heap manager makes some impressive speed improvements for these small blocks. Of course, if the heap manager coalesces memory blocks on freeing, it will have to move blocks out of one free list into another.

Lonely In Your Nightmare

So, knowing all this, what does my error code (*Previous Free Block is Bad*) mean? The heap manager reads the size of the previous block (it's the `longint` before our block). If this is less than 12 (the minimum size of a memory block, remember) or it's not a multiple of four, then the heap manager assumes that the previous block is bad (something has trashed that `longint`). Otherwise it finds and reads the `TFree` record for the previous block. If the size as given by the `TFree` is different from the size just read, then obviously something is corrupted again (either the first or second size read is corrupt).

The heap manager was therefore telling me that I'd trashed the previous block somehow. Or I'd trashed my own Size/Flags value so that it indicated that the previous block was freed when it wasn't. To me it looked like a memory overwrite before the beginning of the block I was freeing. An off-by-minus-one error, perhaps.

Memory overwrites are the bane of any professional programmer. They have this very annoying habit of occurring at point X and having an effect at point Y, many lines of code later. Debugging them can be a nightmare.

How can we help this debugging process? Professional debugging products that test for memory overwrites use the principle of guard bytes. When the user allocates X bytes, by all means give him X bytes, but in fact allocate X+2Y bytes with Y bytes before the user's memory block and Y bytes after. These two chunks of Y bytes are called guard blocks and are initialized to some value (say, zero, but more often a value like `$CC`) and their only purpose is to draw out the memory overwrites so the bug doesn't trash the internal heap structures. When the block is freed, the debugger checks to see if the guard bytes have been altered.

Well, *Algorithms Alfresco* is nothing if not professional, so let's alter our debugging heap manager to use this principle. What we'll do is construct a memory block that looks like this:

```
Size/Flags
User block size
16 bytes of $CC
..user memory..
16 bytes of $CC
```

Why the extra user block size? Well, the Size/Flags value contains the size of the block rounded up to the nearest 4 bytes. We could, for instance, allocate thirteen bytes and write to the fourteenth without any error at all, since the heap manager actually gives us sixteen bytes in our block. So, to be ultra-professional, we should check that the user of the block doesn't write to these hidden bytes at the end of his allocation. Hence we save the size of the requested block, rather than using the rounded size.

This debugging change requires us to write some extra routines. Although some of these routines are in the `SysUtils` unit, we cannot use them because `SysUtils` allocates some memory in its initialization section prior to us gaining control. This is one of the intriguing problems of writing heap managers and debuggers: you can't use other units that may cause `SysUtils` to be loaded.

Listing 4 shows the heap debugger unit `AAHpDbug` with these changes. Note how we increase the allocation in the `GetMem` to allow for

our guard blocks, and in the FreeMem, how we check the guard blocks to be unaltered. To make the whole thing more flexible, we read the number of bytes in the guard blocks from the registry. That way we can test with 4 extra bytes either side, or go to a maximum of 32 bytes (the routine that reads this information can't really pop up a dialog box if the number is in error, so it takes the view that the value should be any multiple of 4 between 4 and 32 and for any other value it'll use the default 16).

Notice also that at this stage, I decided to write the ReallocMem routine as a call to GetMem, copy the data over, and then a call to FreeMem. This gives the heap debugger the greatest control over the allocation/free process, and although we could get away with not doing it this time, in the next iteration of the debugger we will be forced to do it this way.

Actually, despite the fact that this all seems very logical and reasonable, there was another problem that hit me between the eyes. The user can trip the debugger up by passing a bad pointer (maybe it was corrupted). As soon as the debugger tries to dereference it to look at the guard bytes, for example, the debugger will cause an Access Violation. This led me to add pointer tracking: for every successful GetMem, I saved the pointer returned in an array; for every FreeMem I checked that the pointer being freed existed in the allocated pointer array and, if so, removed it. This solved a bunch of problems.

So, what did this amendment show in practice? Well, the bug didn't show up for one (hooray!) but no guard blocks were overwritten (rats!). All I'd done was hide the bug: it's still there but doesn't have the effect I assumed.

Too Much Information

What now? All I've shown so far is that it's not a memory overwrite. There's one more possibility: perhaps I'm freeing a pointer and then writing to it afterwards. On looking through my code, it certainly was not obvious. So, how can we check for this? As soon as we free a block, the heap manager writes stuff all over it for its own purposes.

The answer is to use a technique called delayed freeing. We create a queue of pointers in the heap debugger. When the user wants to free a memory block, instead of deallocating it we merely add the pointer to the queue. Once the number of items in the queue grows to a preset limit, we start deallocating the memory blocks at the head of the queue as we add new ones to the tail. Memory will get recycled, sure, but it takes a while: blocks get put in limbo for a time. To check for writing to a block after freeing it, all we do is fill the block with $CC bytes when the user frees the block and we add it to the delay queue, and then verify that the block still contains $CC bytes when we actually free it once it reaches the head of the queue.

It sounds simple enough, but there are some ramifications we should address. We would like the length of the queue to be defined through the registry for a start. But then that would imply that the queue is a dynamic structure allocated from the heap. Uh, no, we can't do that: we *are* the heap manager. Instead let's use the Windows heap of the process to allocate the queue. Every process has a Windows heap allocated to it by the system when the process

➤ *Listing 4: Debugging with guard blocks.*

```
procedure CheckGuardBlocks(P : pointer);
var
  Mem  : PChar;
  Size : integer;
  RoundedSize : integer;
  SecondSize  : integer;
begin
  {get the address of the first guard block, and verify that
    it hasn't been changed by an overwrite}
  Mem := P;
  dec(Mem, GuardSize);
  if not aaCompareMem(Mem, GuardSize, $CC) then
    WriteLogOverwrite(P, 1, Mem, GuardSize);
  {get size of user's memory block and work out address and
   size of 2nd guard block; verify it hasn't been changed}
  dec(Mem, sizeof(integer));
  Size := PInteger(Mem)^;
  inc(Mem, sizeof(integer) + GuardSize + Size);
  RoundedSize := (Size + 3) and $7FFFFFFC;
  SecondSize := GuardSize + (RoundedSize - Size);
  if not aaCompareMem(Mem, SecondSize, $CC) then
    WriteLogOverwrite(P, 0, Mem, GuardSize);
end;
function OurGetMem(Size : integer) : pointer;
type PInteger = ^integer;
var  RoundedSize : integer;
begin
  {we have to add size of our guard blocks and an extra size
   value to size to allocate; round up to nearest 4 bytes}
  RoundedSize := (Size + (2 * GuardSize) + sizeof(integer) +
    3) and $7FFFFFFC;
  Result := OrigHeap.GetMem(RoundedSize);  {get the memory}
  {providing some memory was allocated...}
  if (Result <> nil) then begin
    {save the original size at the start of the block}
    PInteger(Result)^ := Size;
    {advance the result pointer over this size value}
    inc(PChar(Result), sizeof(integer));
    {fill remainder of memory block with $CC}
    FillChar(Result^, RoundedSize - sizeof(integer), $CC);
    {return the address of the memory block in between the
      two guard blocks}
    inc(PChar(Result), GuardSize);
  end;
end;
function OurFreeMem(P : pointer) : integer;
var BlockSize : integer;
begin
  {check that the user hasn't overwritten the guard bytes}
  CheckGuardBlocks(P);
  {check that the memory block itself wasn't overwritten}
  BlockSize := PInteger(PChar(P) - GuardSize -
    sizeof(integer))^;
  if not aaCompareMem(P, BlockSize, $CC) then
    WriteLogOverwrite(P, 2, P, BlockSize);
  {move to the size value stored in the block: it is this
    pointer that will get freed}
  dec(PChar(P), GuardSize + sizeof(integer));
  Result := OrigHeap.FreeMem(P);  {free the memory}
  if (Result <> 0) then
    WriteLogFreeErr(HeapErrorMsg(Result));
end;
function OurReallocMem(P: pointer; Size: integer): pointer;
var OrigSize : integer;
begin
  if (P = nil) then begin
    if (Size <= 0) then
      Result := nil
    else
      Result := OurGetMem(Size);
  end else begin
    if (Size = 0) then
      Result := nil
    else begin
      Result := OurGetMem(Size);
      OrigSize := PInteger(PChar(P) - GuardSize -
        sizeof(integer))^;
      if (OrigSize < Size) then
        Move(P^, Result^, OrigSize)
      else
        Move(P^, Result^, Size);
    end;
    OurFreeMem(P);
  end;
end;
```

```
function OurFreeMem(P : pointer) : integer;
var BlockSize : integer;
begin
  {check to see if the pointer exists in our list}
  if (P <> nil) and (not RemovePointer(P)) then begin
    WriteLogFreeErr(HeapErrorMsg(99)); {not a valid pointer}
    Result := 99;
  end else begin
    if (P <> nil) then begin  {add pointer to delay queue}
      BlockSize := PInteger(PChar(P) - GuardSize -
        sizeof(integer))^;
      FillChar(P^, BlockSize, $CC);
    end;
    DelayQueue[QTail] := P;
    QTail := (QTail + 1) mod DelaySize;
    {check to see whether we can actually free a pointer}
    if (QHead <> QTail) then Result := 0
    else begin
      P := DelayQueue[QHead]; {pointer at head of queue}
      QHead := (QHead + 1) mod DelaySize;
      {check that user hasn't overwritten guard bytes}
      CheckGuardBlocks(P);
      {check that memory block itself wasn't overwritten}
      BlockSize := PInteger(PChar(P) - GuardSize -
        sizeof(integer))^;
      if not aaCompareMem(P, BlockSize, $CC) then
        WriteLogOverwrite(P, 2, P, BlockSize);
      {move to the size value stored in the block: it is
        this pointer that will get freed}
      dec(PChar(P), GuardSize + sizeof(integer));
      Result := OrigHeap.FreeMem(P);      {free the memory}
      if (Result <> 0) then
        WriteLogFreeErr(HeapErrorMsg(Result));
    end;
  end;
end;
```

➤ *Listing 5: Using a delay queue.*

starts. The heap handle is obtained by calling `GetProcessHeap`. So we can allocate an array of pointers from that heap to hold the queue. At the end of the application, we free everything in the queue and then free the queue back to the process' heap.

Listing 5 shows the `FreeMem` for this heap debugger. It's rather more involved than before but still fairly understandable. Notice also that we use an array of pointers to hold the queue, but that we make it a circular queue with head and tail pointers. This is vastly more efficient than the method used by the `TQueue` class (`TQueue` is one of the container classes in Delphi's `CONTNRS` unit). If we find a corrupted freed block, the contents of the block are written to the log.

So, did it work this time? Yes! There was one memory block I was freeing that I was writing to afterwards. This would trash the heap manager's structures that maintain the free list and allow for memory block coalescing. Once I'd confirmed and seen the problem, finding the piece of code in error was much easier. I ran the application again and placed a data breakpoint on the `longint` being altered. From that point, it was a matter of moments to find the problem.

Of Crime And Passion

However, that happy ending wasn't the real end of the story. I noticed that in using the final version of the heap debugger that I was getting an Access Violation right at the end of the program. Worse still, it was happening after most, if not all, of the exception handling was closed down. I groaned inwardly, thinking that my delay queue code was broken or my pointer-tracking array was hosed, and started debugging. This took some tracking down. For writing to my heap debug log file I was using the basic `Assign`, `Append`, `writeln` and `Close` routines from the `System` unit. They'd been there for years, all the way from Turbo Pascal, and since I remembered how they worked from those halcyon days, I naturally assumed they didn't allocate any memory.

Well, I was wrong. Even though the `Assign` procedure took a `shortstring` parameter for the file name (I'd already checked that, of course) the compiler was inserting some hidden code to allocate a long string on the heap, to convert the short string to a long string, and then to copy the long string as a `PChar` to the relevant field in the `TTextRec` record for the text file variable. Why, for heaven's sake? It's a sheer waste of time and effort, why not just copy the `shortstring` over to the field and append a null? Why this complexity? There was nothing for it, of course, but to write my own text file device driver so that I could have my own `Assign` procedure that did it properly. Except that then I noticed that the `TTextRec` record and the magic file mode constants were now defined in `SysUtils`...

New Religion

Having seen how to create a heap debugger for helping us find heap memory problems, I'm sure you will be able to write your own heap managers. Listing 6, as an example, ignores the standard Delphi heap manager completely and instead uses the Windows heap routines for memory allocation.

Another possible use for customized heap managers is to generate the distribution of heap allocations for your application. How many allocations of 8 bytes are you making? Of 12 bytes? Of 16 bytes? And so on? The results tend

➤ *Listing 6: A heap manager using a Windows heap.*

```
var HeapHandle : THandle;
function OurGetMem(Size : integer) : pointer;
begin
  Result := HeapAlloc(HeapHandle, 0, Size);
end;
function OurFreeMem(P : pointer) : integer;
begin
  if HeapFree(HeapHandle, 0, P) then Result := 0
  else Result := 1;
end;
function OurReallocMem(P : pointer; Size : integer) :
  pointer;
begin
  Result := HeapRealloc(HeapHandle, 0, P, Size);
end;
procedure InitializeUnit;
begin
  GetMemoryManager(OrigHeap);  {get the original manager}
  {set up our heap manager}
  OurHeap.GetMem := OurGetMem;
  OurHeap.FreeMem := OurFreeMem;
  OurHeap.ReallocMem := OurReallocMem;
  {create a Windows heap}
  HeapHandle := HeapCreate(0, 1024*1024, 0);
  if (longint(HeapHandle) <> 0) then
    SetMemoryManager(OurHeap);  {replace heap mgr with ours}
end;
procedure FinalizeUnit;
begin
  SetMemoryManager(OrigHeap);  {restore original manager}
  if (longint(HeapHandle) <> 0) then
    HeapDestroy(HeapHandle);  {dispose of the Win32 heap}
end;
```

```
procedure UpdateBin(Size : integer);
var
  RoundedSize : integer;
begin
  {calculate the rounded size of the requested allocation...
   -actual size rounded up to nearest aaHeapAlign bytes (4)}
  RoundedSize := (Size + aaHeapAlign - 1) and
    (not (integer(aaHeapAlign) - 1));
  {-if result is less than minimum round up to minimum}
  if (RoundedSize < aaHeapMinAlloc) then
    RoundedSize := aaHeapMinAlloc
  {-if greater than maximum round down to maximum plus 4 (in
    other words, this allocation is for the 'other' bin)}
  else if (RoundedSize > aaHeapMaxAlloc) then
    RoundedSize := aaHeapMaxAlloc + aaHeapAlign;
  {increment the count in the relevant bin}
  InterlockedIncrement(
    aaHeapBins[RoundedSize div aaHeapAlign]);
end;

function OurGetMem(Size : integer) : pointer;
begin
  UpdateBin(Size);  {update the relevant bin}
  Result := OrigHeap.GetMem(Size);  {allocate the memory}
end;

function OurReallocMem(P: pointer; Size: integer): pointer;
begin
  {update relevant bin; NB: Size=0 is same as a FreeMem}
  if (Size <> 0) then
    UpdateBin(Size);
  Result := OrigHeap.ReallocMem(P, Size)  {do the work}
end;

procedure FinalizeUnit;
var
  Log : System.Text;
  i   : integer;
  LogNameZ : array [0..255] of char;
  LogName  : shortstring;
begin
  SetMemoryManager(OrigHeap);  {restore original manager}
  {get the log name}
  aaReadRegistryString(LogNameZ, 256,
    'software\AlgorithmsAlfresco\AAHpDist', 'LogName',
    'C:\HEAPDIST.LOG');
  LogName := aaStrPas(LogNameZ);
  {write out data to log}
  aaLogOpen(Log, LogName);
  try
    writeln(Log, 'Heap Allocation Distribution');
    writeln(Log, '----------------------------');
    writeln(Log);
    writeln(Log, 'Size':5, 'Count':10);
    for i := low(aaHeapBins) to pred(high(aaHeapBins)) do
      writeln(Log, (i * aaHeapAlign):5, aaHeapBins[i]:10);
    writeln(Log, 'Other':5,
      aaHeapBins[high(aaHeapBins)]:10);
  finally
    aaLogClose(Log);
  end;
end;
```

➤ *Listing 7: Calculating the allocation size distribution.*

to be very interesting. For a standard, fairly simple, UI application, the vast majority of heap allocations are all less than about 128 bytes. You can therefore easily optimize your heap manager replacement by making sure that allocations of less than 128 bytes come off simple free lists, one per allocation size. Ignore the coalescing action of the standard Delphi heap manager: on a free, you add the block to its free list, on an allocate you get the top free block off the relevant free list. If there are no free blocks in that list, then allocate off the Delphi heap manager.

Listing 7 shows a heap manager that shows the distribution of memory blocks being allocated in your application. It's also fairly easy to modify it to log every memory allocation and free.

Another good debugging heap manager is one that stresses your application by generating out of memory errors. Generally the only

time you would get an out of memory error on your own machine would be when you have runaway recursion allocating memory. This would be a race between an out of stack error and an out of memory error. On customers' machines (they always seem to be running Windows 95 in 16Mb of RAM) it may occur more often. Actually, this heap debugger is simple to implement: at some signal from the program (a flag, a certain number of allocations, a maximum total size of allocations) your replacement `GetMem` routine returns nil. That's all. Sit back and see how your code behaves...

For those of you who are using the `ShareMem` unit, the good news is that you can still debug and replace the heap. It takes a little more doing because you have to create a DLL to put your heap manager in (this will replace BORLNDMM.DLL) and then write an interface unit for it (to replace `ShareMem`) but it's fairly simple rote programming.

Come Undone

With that I must close this particular article. We've come a long way, although the algorithmic and data structure parts were all pretty easy. We've seen how to replace the heap manager, and used this technique to develop a debugging heap manager to help us pinpoint heap errors. We've also seen a few ideas on how to write our own heap managers with customized, specialized behaviors. Have fun tracking your memory overwrites, generating out of memory exceptions and checking your memory allocation distributions!

Julian Bucknall has heaps of memories of Roger Vadim's film Barbarella starring Jane Fonda. One of the other characters was of course Duran. Email Julian at julianb@turbopower.com. The code that accompanies this article is freeware and can be used as-is in your own applications.