Eliminating Memory Leaks in Symbian OS C++ Projects

John Pagonis, Symbian Developer Network, January 2005

In Symbian OS, the programming framework, in debug mode, makes sure memory allocations are tracked and that during development if a memory leak exists on its termination, an application will be panicked. Panicking that application is a direct and immediate way of bringing to the developer's attention that a memory leak exists and that it needs to be fixed. There is a lot to be said about this philosophy as it forces one to fix bugs as soon as they happen, where the effort and cost is at its lowest.

Moreover, there are facilities at your disposal to ensure that such memory checking can be performed manually during development, within the scope of your choosing. You can set marks around some code block, to make sure that it doesn't leak any memory. In fact that is what the framework does; it sets these at the beginning and end of an application. Subsequently if one is developing software outside the Symbian OS C++ application framework, these manual mechanisms have to be employed.

These facilities, which are present in debug builds, are calls to the memory allocation framework and have been wrapped for ease within macros (defined in e32def.h), such as __UHEAP_MARK and __UHEAP_MARKEND. What the combination of these two calls does is to verify the consistency of allocations and de-allocations for a block of code. If any inconsistencies are found, they panic the offending process. When that happens the developer is called to track the offending unallocated cell(s). For a more detailed treatment of these facilities see [1] and [2].

1. What to do when faced with a memory leak

Before you start with WINS memory leak debugging and in order to have as much symbolic debugging information at your disposal, place the UIQ 2.1 PDB files supplied by the SDN to your \epoc32\release\wins\udeb path. For WINSCW UIQ users this is not necessary as the debug binaries contain that information embedded. Developers that use Series 60/80 SDKs can build them from the SDK libs and dlls [3].

Note that for the following discussion, Microsoft Developer Studio terminology and examples will be used. CodeWarrior users can apply similar techniques in their environment and workflow.

1.1. Tracking the offending cell

So, you tried to exit an application when an exception occurred and that exception was due to a memory leak. In that case the system complained about a cell (or cells) not deleted. This can be identified by looking into the call stack and finding something like 'User::__DbgMarkEnd()'.

From the 'Call Stack' window, click on

User::__DbgMarkEnd(RHeap::TDbgHeapType Euser, int 0x00000000)

In most cases, unless you have licensed the particular Symbian OS code, you will be presented with the disassembly window for that method, in which case just 'hover' with the mouse over the badCell and note down the address presented in the 'tool tip' - or by hovering over and pressing Shift+F9. Alternatively you will be presented with the source code for that method (in \e32\UCDT\UC_KUSR.CPP); from there do a 'Quickwatch' on the badCell (on the line: info.AppendFormat(_L("%x\n"),badCell)), by pressing the right mouse button.

Now, the address shown from the 'Quickwatch' is the address of the orphaned cell.

An alternative way of obtaining the badCell address is to allow after the panic the emulator to continue the debugging session (by pressing F5) and noting down the message at the output window that looks like: 'Thread panic ALLOC: 16497110'. The number printed there is actually the decimal representation of the address of the badCell.

1.2. Discovering the identity of the orphaned object

Discovering the identity of the orphaned object needs a bit more work. If it happens to be a CBase-derived object it is actually much easier, therefore the simplest thing to do is to check if the offending cell is an instance of a C class.

1.2.1. If it is a C-class object

Use the 'Watch' or 'Quickwatch' functions to view it as a pointer to a CBase object. Note that you must put "0x" before the hex value pasted from the clipboard (and if it is represented in decimal, then you need to convert it first). If it is indeed a C object, the debugger will tell you the class name, which is in most cases sufficient to solve the problem.

In the case when the debugger cannot detect a virtual function table pointer, it isn't a C object and this simple approach has failed and we will have to try other techniques.

There are two ways to narrow the search for the object, one more quick and rough, the other more involved and precise.

1.2.2. Use a quick data breakpoint

Put a breakpoint somewhere in the beginning of the suspect component, but close enough to the suspect code - which can be tricky:-).

Stop the debugging session and re-run it. As soon as the breakpoint is reached add a data breakpoint, by keying 'Ctrl-b'. This will present you with a property sheet where you should enter, in the data tab, the address of the badCell and a number of bytes, usually 4, (starting from that address), that you're interested in observing for any change.

This action tells the debugger to notify you about any change or access to that memory location.

Press 'F5' to continue, now the debugger should automatically break into the code that tries to modify this memory, when it happens.

This should probably be either a NewL, NewLC or ConstructL method, if you have put the breakpoint early on, or some code that tries to alter it.

So you have now discovered which object is not getting deleted or leaks memory!

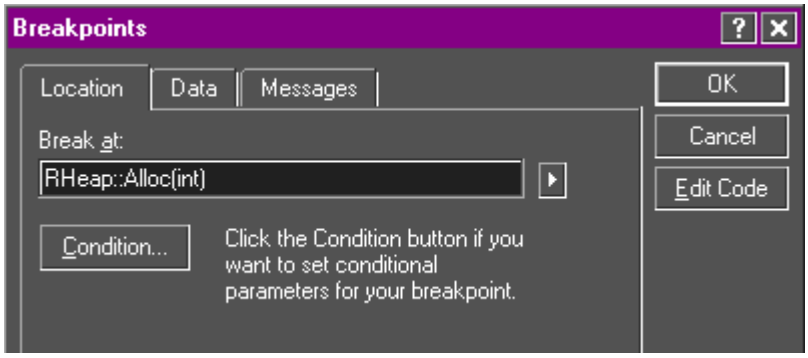
Remember that if you haven't narrowed down, your search, close to the offending code, then you may falsely be led to believe that a correctly deleted object wasn't destroyed correctly. This is due to the fact that this address may be reused many times.

In that case you have to patiently step through the code to discover the last allocated object to that address, or try the following technique.

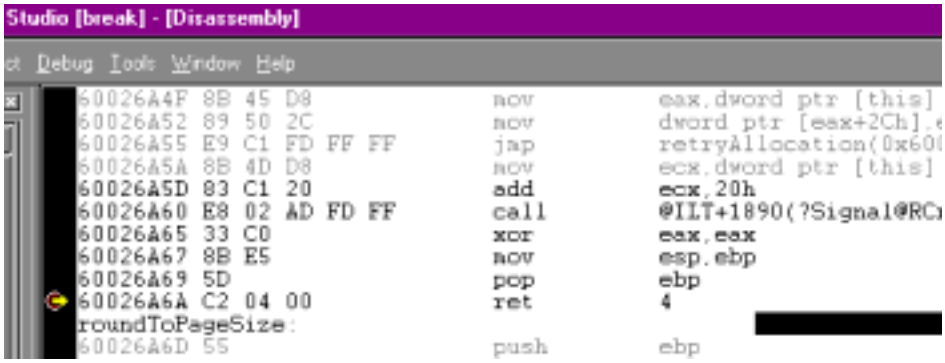
1.2.3. Alloc breakpoint

The alternative to the quick data break point is to set a breakpoint to occur when the offending cell is allocated. All heap memory allocation goes through the function RHeap::Alloc(int).

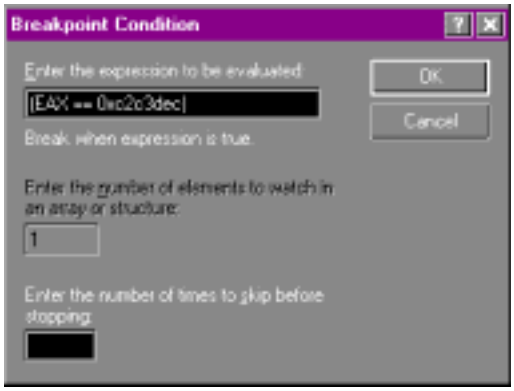
Start by putting a breakpoint there. Symbian doesn't supply the source code for this function, but you can set the breakpoint explicitly using 'Edit-Breakpoints-BreakAt'



Again, for WINS users, having the PDB files makes this easy. Use 'Debug-Go' (F5) to continue until the system's first allocation. You won't be able to view the source code, but you can see the disassembled code. Scroll down through the disassembled code, passing the label retryAllocation, until the line before the start of the next function, roundToPageSize. Put a breakpoint on the RET line before it.



At that point, register EAX will contain the return value from the RHeap::Alloc function. Use 'Edit-Breakpoints' to set up a breakpoint when it's equal to your offending cell. First disable the breakpoint at RHeap::Alloc, then select the breakpoint you just set and use 'Condition' to set the condition that the return value be the cell you're tracking



Click OK to both dialogs, and then use Debug-Go to continue. Run the application as before; when execution stops at the breakpoint, examine the stack to see where the offending cell was allocated.

Note that sometimes the same cell may be allocated and de-allocated a few times; it's only the last allocation you're interested in. If the cell doesn't get allocated, it may be because this run was a bit different from the first one,

and the leaking cell was at a different location; check the virtual address space presented to your application, to discover if that is the reason.

Continue until application exit, and find a new offending cell address.

Then set that as an additional Breakpoint at the same location as the other but with Condition to catch the new offending cell, and use 'Debug-Restart' to restart. You may get the error message "Cannot restore all the breakpoints"; this is because the EUSER DLL isn't loaded immediately when EPOC.EXE first starts; one solution in this case is to re-enable the RHeap::Alloc(int) breakpoint, run until that is called, and then restore the other, conditional, breakpoints.

2. Being Proactive

Of course you don't have to wait for bad things to happen, even if that wait is only until the application exits. It is even easier and cheaper to be proactive about it and catch memory leaks as soon as they happen.

2.1. Be Proactive with the OOM Loop

A technique *for tracking down memory leaks and exception handling cleanup defects* that has been used since the first days of Symbian OS, is that of the so called *OOM Loop*. This is an algorithm that incrementally and deterministically forces out of memory conditions around a block of code.

What this techniques achieves, is to *exercise all the possible exception handling paths due to OOM condition of the code in encompasses*. This is very powerful and amazingly, together with code reviewing, the cheapest technique for finding such bugs.

What the algorithm does is to set heap checks around the code to be tested, in order to ensure that no allocated cells remain without de-allocation. Then it makes allocations fail on every run, while sequentially incrementing the number of allocations that will succeed. Thus on each run one more allocation makes it and one more cleanup path is exercised.

The simplistic code for this techniques looks like this:

```
for (TInt k=1;;++k)
  {
   __UHEAP_SETFAIL(RHeap::EDeterministic,k);
   __UHEAP_MARK;
   TInt err = TestBlock();  //This internally TRAPs
   __UHEAP_MARKEND;
   User::Heap().Check();// paranoid check for internal heap corruption
   if (err==KErrNone)
                  break;
  }
```

Of course in the TestBlock() one needs to Trap any Leave and return the error code for the loop to be able to continue or break. In the case where this involves a top level allocation it will most likely look like this (taking the TRAP block out of that method for simplicity):

```
for (TInt k=1;;++k)
  {
  RDebug::Print(_L("loop number - %d"),k);
  __UHEAP_SETFAIL(RHeap::EDeterministic,k);
  __UHEAP_MARK;
```

```
  TRAPD(err,iModel=CMobInfoAppModel::NewL())
  if(err==KErrNone)
     {            //last run where everything went well
           delete iModel;
           }

   __UHEAP_MARKEND;
   User::Heap().Check();
   if (err==KErrNone)
       break;
  }
```

It is very important to consider the EDeterministic option above in __UHEAP_SETFAIL macro above. The combination of this option and the increment of the loop counter is what makes this technique exercise all cleanup paths. Every time an OOM is forced, all objects on the cleanup stack have to be cleaned up and all TCleanupItems will need to be called to clean any resources. Should there exist any problem with the exception handling logic or a resource persists after a run, it will be flagged at the checkpoint. This checkpoint is set by marking the end of the checking (__UHEAP_MARKEND) and actually checking that at that point there are as many cells allocated as before the run began.
The OOM Loop can be used as a stand-alone technique or can be embedded into code during development and testing. In which case and for completeness, it will look like:

```
//be defensive since PushL may leave, thus pre-alloc enough for the test
for (TInt j=0;j<1000;++j)
  CleanupStack::PushL(&j);

CleanupStack::Pop(1000);

//Extra paranoia marks in case the framework has a memory leak
__UHEAP_MARK;

for (TInt k=1;;++k)
  {
  RDebug::Print(_L("loop number - %d"),k);
  __UHEAP_SETFAIL(RHeap::EDeterministic,k);
  __UHEAP_MARK;

  TRAPD(err,iModel=CMobInfoAppModel::NewL())
  if(err==KErrNone)
     {
     delete iModel;                    }
           }
  __UHEAP_MARKEND;
  User::Heap().Check();
  if (err==KErrNone)
     break;
  }

  __UHEAP_MARKEND;     //end of paranoia checks
  __UHEAP_RESET;       //reset failure tool
  User::Heap().Check();
```

2.2. Where to place the OOM Loop ?

OOM Loop testing can go anywhere really, while it is most useful if it can wrap around most code. Thus in the case of the UIQ framework for example it can wrap around the creation of the 'model' and the AppUi.

Thus in the derived (from CQikDocument) document class it can wrap around the iModel and CEikAppUI creations, like in the following examples (taken from the Mobinfo API test app)

```
void CMobInfoAppDocument::ConstructL()
   {

   __UHEAP_MARK;
   for (TInt k=1;;++k)
     {
          RDebug::Print(_L("loop number - %d"),k);
          __UHEAP_SETFAIL(RHeap::EDeterministic,k);
          __UHEAP_MARK;

          TRAPD(err,iModel=CMobInfoAppModel::NewL())
          if(err==KErrNone)
            {
            delete iModel;
            }

           __UHEAP_MARKEND;
           User::Heap().Check();
           if (err==KErrNone)
              break;
           }
   __UHEAP_MARKEND;
   __UHEAP_RESET;
   User::Heap().Check();


   iModel = CMobInfoAppModel::NewL();
   ResetModel();
   }
```

Note that one may be tempted to do as above in the CreateAppUiL() method from within the AppDocument. This would actually be useful for the application framework creators only, since it would exercise the path from the AppUi constructor and below. Developers should note that this is of limited value when building an application, since their code should actually be instantiated in the *second phase construction* found in AppUi's ConstructL().

Symbian OS's view switching architecture employs *two phase view construction* (and in some products even three phase) so that it can minimise the application (and view) bootstrap time, as well as the memory used since allocations for unused views are minimised. Subsequently, it is of benefit to place an OOM loop around the blocks found in the view's ConstructL() and ViewConstructL() methods. To do so, as a minor side effect, one would need to refactor these blocks into new private methods to keep the code cogent.

If you do as above during development, every time the framework starts your application as well when it activates the views, the OOM loops will do a fine job at finding bugs immediately and before you even close down the app :-).

Remember that although memory leak bugs will always be caught at application exit, the exception safety of your code will not be stressed unless you force it as the OOM Loop does.

2.3. Tools of the trade

As already described, the Symbian OS C++ framework has a lot of support for tracking and proactively making sure resources are not leaked. A manifestation of such support is in the embedded 'Heap and File Failure Tool'. This tool is embedded in the (Uikon) application framework on all Symbian OS platforms; for debug builds.

Using the Uikon debug key sequence Ctrl+Alt+Shift+P, the tool comes up, allowing one to set deterministic or random failure points for both fileserver resources and memory allocations. Note that such treatment, from the tool, applies only to the application context from which it was called.

There are two main ways to make use of the tool (as far as OOM testing is concerned), one is to set the tool to randomly fail memory allocations and the other is to set it to do it deterministically; on every predetermined number of allocations.

A third way to make use of this tool is to use it as a shortcut to force memory allocation failure at any point while running the application (in debug mode). This is achieved by setting the 'App heap failure type' to deterministic and the 'failure rate' equal to 1. This shortcut is quite handy for testing 'what would happen if it failed here' scenarios as the application is being used.

In doing so, one can make use of another debug key sequence, Ctrl+Alt+Shift+A, to present on screen the number of cell allocations at each point of testing. Therefore if the above sequence is used before and after a forced OOM failure, one can make immediately sure if a particular area of functionality behaves correctly under OOM conditions, without embedding any macros in the code.

2.4. Epilogue

With the above techniques under your belt, you should be well on the way to producing Symbian OS code which is free of memory leaks!

References

[1] ‘Symbian OS C++ for Mobile Phones’, Richard Harrison, 2003, Wiley
[2] ‘Symbian OS Explained’, Jo Stichbury, 2004, Wiley
[3] ‘Generating debug symbols for the emulator’, Mika Raento, http://www.cs.helsinki.fi/u/mraento/symbian/symbols.html

Back to Developer Library