05: Dynamic Arrays

Tutorial

Please note: This is the second part of the exercise for the “Dynamic Arrays”-module. Please start with the exercise instructions!

Objective

Rewrite the `TStudent`-based dynamic array example:

- Use a `CBase`-derived class (`CStudent`) instead of `TStudent`
- Store the student name in an `RBuf` instead of a `TBuf`
- Use an `RPointerArray` instead of the `RArray`

Organize the project

(*Optional*) In the `TStudent` reference project, the class is fully contained in the `DynamicArrays.cpp`-file. To keep things organized and concise, you should probably give this class its own source- and header-files (`Student.cpp`, `Student.h`).

Array operations

Note that the `RPointerArray` doesn’t support sorting elements based on a key (`TInt`/`TUint` member variable, `_FOFF`-macro). You can remove those parts from the `TStudent` reference project. The only remaining operations are:

- Adding elements
- Sorting using `TLinearOrder`
- Find an element using `TIdentityRelation`

CBase

Remember that all C-type classes have to derive from `CBase`, otherwise the Cleanup stack cannot correctly handle them. When renaming the class, don’t forget to change all references of `TStudent` to `CStudent`. As you will now store pointers in the array, remember that accessing elements will now work with “`->`” instead of “`.`”

Deleting the RPointerArray

As our `RPointerArray` will own the `CStudent`-objects we pass to it, we have to take care of deleting those when deleting the array. The `CleanupClosePushL()`-call suffices to delete an `RArray` in case of a leave and when calling `CleanupStack::PopAndDestroy()`, as it calls `students.Close()`.

However, to delete the objects owned by the array as well, you have to call `students.ResetAndDestroy()`. For some reasons, the corresponding cleanup stack-function is defined in:

```
#include <mmf/common/mmfcontrollerpluginresolver.h>
```

When including this, you can use `CleanupResetAndDestroyPushL(students)` – which will correctly call `students.ResetAndDestroy()` instead of `students.Close()`.

RBuf

In contrast to the `TBuf`, the `RBuf` is heap-based and has to be freed when the object instance is deleted. This is done by calling `Close()` in the destructor of the `CStudent`-class. Also, the `TBuf` had a fixed size that wasted memory in most cases, whereas the `RBuf` can be created based on the source descriptor (`CreateL()`) and will only require as much memory as required.

Two-Phased Construction

Allocating heap-memory through the `RBuf`-object can leave, therefore you should not do this in the constructor. Implement a two-phase construction instead. The standard C++ constructor is now only used to assign the `TInt`-variables, whereas creating the `RBuf` should be done in `ConstructL()`. To make constructing the objects easier, implement `NewL()` and `NewLC()`.

Cleanup Stack – Elements

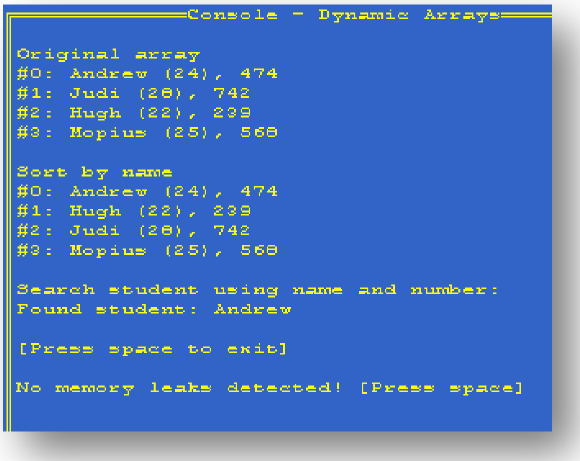
Take care of the cleanup stack when appending elements to the array!

The call to `students.AppendL()` can leave, as indicated by the trailing "`L`". Therefore, make sure the new student object is on the cleanup stack before you add it to the array (→ `NewLC()`). `Pop` (without …`AndDestroy`!) it from the cleanup stack after appending it, as the ownership has been transferred to the array.

Finding an element (`students.Find()`) cannot leave, therefore you don't have to put the reference object on the cleanup stack. Here, you can use `NewL()` and delete it with `delete`, instead of using the cleanup stack to delete it.

Testing

The final output should look somewhat like this:



```
Console - Dynamic Arrays

Original array
#0: Andrew (24), 474
#1: Judi (28), 742
#2: Hugh (22), 239
#3: Mopius (25), 568

Sort by name
#0: Andrew (24), 474
#1: Hugh (22), 239
#2: Judi (28), 742
#3: Mopius (25), 568

Search student using name and number:
Found student: Andrew

[Press space to exit]

No memory leaks detected! [Press space]
```

Now, try to fail individual memory allocations by adding `__UHEAP_FAILNEXT(1);` before them. If your application does not cause any memory leaks, you succeeded!