02: Leaves and the Cleanup Stack

Exercise Instructions

Goal

This module provides a hands-on experience with leaves and various ways on how the Cleanup Stack can help with handling risky memory situations.

Introduction

Symbian OS has its own mechanisms for handling memory. Especially preventing memory leaks is one of the most important aspects of development for mobile phones, where memory is scarce and phones are expected to run for several weeks without a reboot. While the concepts of Leaves, Traps and the Cleanup Stack might look difficult at first, they give you maximum control over memory handling and allow you to effectively prevent errors in your application.

Structure of this Exercise

The entry point of the application is the `E32Main()`-function, which will catch and output any errors (= leaves) that the code might produce. Also, it checks the heap state of the application to check for memory states. See the detailed descriptions for an explanation of the expected output. No changes have to be made here.

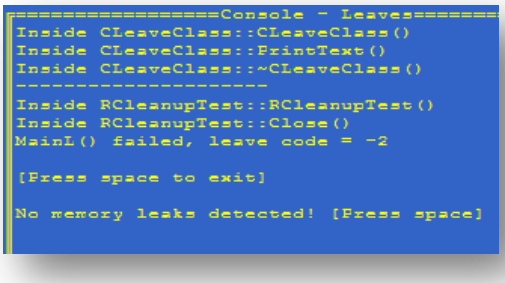
Two very simple classes have been defined:

- ***CLeaveClass:*** A prototype C based class (without two phase construction) that includes a function (`InitializeL()`) that intentionally causes a leave.
- ***RCleanupTest:*** This class just has a constructor and a `Close()`-function and will be used to test the cleanup stack-behavior for R type classes.

The tasks are split up into several sections:

- ***The Framework:*** This part includes very few edits. You should get to know how the pre-made application framework handles leaves and memory leaks and take a look at how these are caught and displayed.
- ***Experiments with Leaves and the Cleanup Stack:*** In the main section of this exercise, you have to create an instance of a class and see in which situations a memory leak can occur and how you can prevent them.
- ***Cleanup Stack for R classes:*** In real-life applications, you will use R type classes quite often. This section shows how to correctly handle the resources in combination with the cleanup stack.

The final output should look like this:



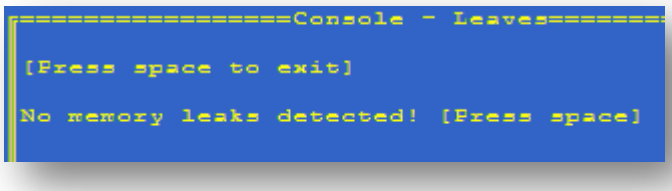
Detailed Descriptions

The Framework

The framework of this exercise includes two important tests, which are executed directly in the `E32Main()`-function:

- ***Leaves:*** First, it traps any leaves that might go through from the `MainL()`-function and displays the error code.
- ***Memory Leaks:*** Secondly, it checks the state of the heap memory before and after `MainL()` is executed. If a memory leak has occurred, the `__UHEAP_MARKEND` macro will raise a Panic. The S60 and UIQ UI-frameworks would handle this and display an error message. However, the console view doesn't do this, as there is no layer above our application. If a memory leak occurs, the application exits when the Panic is raised by the macro.
  Therefore, the sample application requires pressing the space key twice:
  - The first time after execution of the `MainL()`-function, so that you can see its output and any leaves that might have been caught in any case.
  - The second time after the heap check macro. If you don't get to this point and the emulator closes down before you see the second message, you will know that there is a memory leak in your application.

The output of the framework with no memory leaks should look like this:



The edits in the first section of `MainL()` allow you to see how the `TRAP` in `E32Main()` reacts to a leave and also demonstrates that the `__UHEAP_MARK` macro can make you aware of memory leaks in your application.

Experiments with Leaves and the Cleanup Stack

Please take extra care of the order of the steps – the edits are not in sequential order in this part!

Through the edits, you will experience how memory leaks can occur and how they can be prevented in a step-by-step introduction. After instantiating an object of `CLeaveClass`, you will first see that the `new (ELeave)`-operator causes a leave right when there is a problem with memory allocation, instead of having you check the allocation every time.

The next demonstration shows what happens when a leave occurs before the delete-statement is reached. The cleanup stack is a countermeasure against such a situation.

Remember that instance variables of objects have to be handled differently and don't have to be put on the cleanup stack, as they will be deleted by the destructor of the class and must not be deleted twice.

Cleanup Stack for R classes

This part shows how the `CleanupClosePushL()`-function works. It will automatically call the `Close()`-function of the R class object when the object is deleted – either because of a leave or through the normal `CleanupStack::PopAndDestroy()`-function.

This is very important as you will encounter R classes quite often in real-world applications – be it the `RFs` (file server) or parts of the multimedia framework.