CleanupStack

CleanupStack class, like a generic stack struct, defines two fundamental methods, PushL and Pop. The main intent of CleanupStack is to address every allocated object that may be "orphaned" when an exception occurs. CleanupStack maintains a stack where its forming elements stores pointers to the objects in the heap.

It is incredibly important that applications do not leak memory, and so you must take steps to ensure that any application resources stored on the heap are cleaned up when an exception occurs. The Cleanup Stack is central to this methodology, as it offers a way to delete data that may otherwise be orphaned.

The cleanup stack addresses the problem of cleaning up objects that have been allocated on the heap, but to which the owning pointer is an automatic variable.

You should never push a member variable to the cleanup stack. Once a pointer to a heap object has been copied into a member of some existing class instance, then you no longer need that pointer on the cleanup stack. This is because the class destructor takes responsibility for the object's destruction.

To ensure you actually deallocate ANY object you create, there is a stack, named the "Cleanup Stack", where you push any object that needs to be deallocated later.

Consider this example:

```
CDemo* demo = new CDemo;
DangerousOperationL();
delete demo;
```

If the DangerousOperationL() leaves, the delete demo will be never reached, leaving the demo object allocated, resulting in a memory leak.

What to push in CleanupStack:

It should never be possible for an object to be cleaned up more than once. If the pointer wer retained in the cleanupstack, then if the function leaves; the leave will destroy it. But if you have stored that pointer to the other object, the object will also try to destroy it(usually in its own destructor). An attempt to delete an object twice on heap causes a system panic.

This is the reason that pointers which are class member variavles should not pushed onto the CleanupStack.

The correct way is to use the cleanup stack as follows:

```
CDemo* demo = new CDemo();
CleanupStack:: PushL(demo);
DangerousOperationL();
CleanupStack:: PopAndDestroy()
```

This way, you push the element that must be deallocated in a deallocation stack. If the execution is performed normally, you pop objects and destroy them, in a single call.

If the execution leaves, the cleanup is also performed in the context of leaving. This is the most important feature of the Cleanup Stack making its use a must in Symbian programming.

What if CleanupStack::PushL() Itself Fails?

Pushing to the cleanup stack may potentially allocate memory, and therefore may itself fail! You don't have to worry about this, because such a failure will be handled properly because there is at least one spare slot on the cleanup stack. When you do a PushL(), the object you are pushing is first placed on to the cleanup stack (which is guaranteed to work, because there was a spare slot). Then, a new slot is allocated. If that fails, then the object you just pushed is popped and destroyed. The cleanup stack actually allocates more than one slot at once, and doesn't throw away slots that have been allocated when they are popped. So pushing and popping from the cleanup stack are very efficient client operations.

Related Links

- CleanupStack class with SDK Help

(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.