symbian Academy

Leaves and the Cleanup Stack

Introduction

Symbian OS was designed to perform well on devices with limited memory and uses the cleanup stack to ensure that memory is not leaked, even under error conditions.

Two of the most fundamental programming patterns of Symbian OS are leaves - a 'lightweight' exception on Symbian OS - and the cleanup stack, which is used to manage memory and other resources in the event of a leave.

symbian Academy

Leaves and the Cleanup Stack

Leaves: Lightweight Exceptions for Symbian OS

- Know that, before v9, Symbian OS did not support standard C++ exceptions (**`try/catch/throw`**) but used a lightweight alternative: **`TRAP`** and leave, which is still preferred in Symbian OS v9
- Know that leaves are a fundamental part of Symbian error handling and are used throughout the system
- Understand the similarity between leaves and the **`setjmp`**/**`longjmp`** declarations in C
- Recognize the typical system functions that may cause a leave, including the **`User::LeaveXXX()`** functions and **`new(ELeave)`**
- Be able to list typical circumstances which cause a leave (for example, insufficient memory for a heap allocation)
- Understand that **`new(ELeave)`** guarantees that the pointer return value will always be valid if a leave has not occurred

Why Not Standard C++ Exceptions?

Symbian OS was first designed at a time when exceptions were part of the C++ standard

- Exception-handling support was found to add substantially to the size of compiled code and to run-time RAM overheads, regardless of whether or not exceptions were actually thrown

The emphasis on a compact operating system and client code

- Meant that exceptions presented too much overhead on Symbian OS
- So a simple, lightweight alternative to standard C++ exceptions was provided - leaves
- A leave is used to propagate an error to where it can be handled

Why Not Standard C++ Exceptions?

Symbian OS v9.x

- On Symbian OS versions before v9.x, compilers are explicitly directed to disable C++ exception handling
- Symbian OS v9.x, by taking advantage of compiler improvements, supports C++ standard exceptions and provides a more open environment
- This makes it easier to port existing C++ code onto the Symbian platform

What is a Leave?

A leave suspends code execution

- At the point the leave occurs and resumes execution where the leave is trapped
- The trap harness in Symbian OS is a **`TRAP`** macro
- The leave sets the stack pointer to the context of the **`TRAP`** and jumps to that location - restoring the register values
- A leave does not terminate the flow of execution of the thread

`User::Leave()` or **`User::LeaveIfError()`**

- Are similar to a C++ **`throw`** instruction
- Except for its destruction of stack-based variables (as discussed shortly)

`TRAP` macros

- May be seen as combination of **`try`** and **`catch`** - they are discussed in more detail later

`setjmp()` and `longjmp()` Methods

`TRAP` macros and **`User::Leave()`** are analogous to the standard library **`setjmp()`** and **`longjmp()`** methods

- A call to **`setjmp()`** stores information about the location to be "jumped to" in a jump buffer ...
- ... used by **`longjmp()`** to determine the location to which the point of execution "jumps"

1. **`setjmp`** i.e. stack position stored

4. Execution resumes, **`result = KErrNotFound`**

```
void SomeFunction()
    {
    // do something ...
    TInt result;
    TRAP(result,SomeLeavingFunctionL(...));


    }
```

2. normal C++ call of function

3. **`longjmp`** occurs drops back to the TRAP

```
void SomeLeavingFunctionL (...)
    {
    ...
    User::Leave(KErrNotFound);
    ...
    }
```

Leave Behavior

The leave mechanism simply deallocates objects on the stack

- Unlike a C++ exception, it does not call any destructors on stack objects
- If a stack object owns a resource which must be deallocated or otherwise "released" as part of destruction it will leak that resource in the event of a leave

This is why only T classes, and built-in types may be instantiated and used safely on the stack

- T classes are restricted to ownership of built-in types or other T classes and do not own resources and thus do not need a destructor
- A stack-based T-class object will thus be cleaned up correctly if a leave occurs because, in effect, there is nothing to clean up as the stack unwinds

R classes may also be created on the stack - but they must be made "leave safe"

- The cleanup stack is used for this, as shall be discussed shortly

What Causes a Leave?

A typical leaving function

- Is one that performs an operation that is not guaranteed to succeed for example:
- Allocation of memory - which may fail under low memory conditions
- Creation of a file - which may fail if there is insufficient disk space

Leaves can also happen if:

- Another leaving function is called, and it leaves
- A system function call explicitly causes a leave (e.g. **`User::Leave()`**)
- Code uses the overloaded form of operator **`new`** which takes **`ELeave`** as a parameter

Why Use Leaves?

Given exceptions presented to much overhead to be used why not simply check everything? For example:

```
CCat* InitializeCat()
    {
    CCat* cat = new CCat();
    if (cat)
      {
      cat->Initialize();
      return (cat);
      }
   else
     return (NULL);
```

- Too much reliance on the developer - it's easy to forget a check
- Even with low-memory testing unchecked code can make it into the wild
- Can create unnecessarily complex code

Heap Allocation using `new(ELeave)`

Symbian OS overloads the global `operator new` to leave

- To provide an option to leave if there is insufficient heap memory for a successful allocation
- Use of this overload allows the pointer returned from the allocation to be used without a further test that the allocation was successful (the allocation would leave if it were not)
- Removes the need for the developer to write an additional check
- Here is a code fragment which illustrates the use of the Symbian OS `operator new` overload:

```
CCat* InitializeCatL()
    {
    CCat* cat = new(ELeave) CCat();
    cat->Initialize();
    return (cat);
    }
```

Functions in Class `User` that cause a Leave

`User::LeaveIfError()`

- Tests an integer parameter passed into it and causes a leave if the value is less than zero
- Uses the integer value as a leave code, for example, one of the `KErrXXX` error constants defined in `e32std.h`
- `User::LeaveIfError()` is useful for turning a non-leaving function which returns a standard Symbian OS error into one which leaves with that value

  e.g. `User::LeaveIfError(FunctionReturningAnError());`

`User::Leave()`

- Does not carry out any value checking but simply leaves with the integer value passed into it as a leave code

Functions in Class `User` that cause a Leave

`User::LeaveNoMemory()`

- Takes no arguments.
- The leave code is hardcoded to be `KErrNoMemory` which makes it, in effect, the same as calling `User::Leave(KErrNoMemory)`

`User::LeaveIfNull()`

- Takes a pointer value and leaves with `KErrNoMemory` if it is `NULL`
- It can sometimes be useful to enclose a call to a non-leaving function which allocates memory and returns a pointer to that memory or `NULL` if it is unsuccessful

symbian Academy

Leaves and the Cleanup Stack

How to Work with Leaves

- Know that leaves are indicated by use of a trailing `L` suffix on functions containing code that may leave (for example, `InitializeL()`)
- Be able to spot functions that are not leave-safe, and those that are
- Understand that leaves are used for error handling; code should very rarely return an error ***and*** be able to leave
- Understand the reason why a leave should not occur in a constructor or destructor

How to Work with Leaves

The problem of unsafe leaves

- Assume that a function has previously allocated memory on the heap and this memory is referenced only by a local pointer variable
- If a leave occurs inside the function, the pointer is destroyed by the leave (as the stack frame is unwound back to the TRAP handler) and the heap memory the pointer references becomes unrecoverable, causing a memory leak

A Memory Leak Anti-Pattern

An example of an unsafe leave ...

```
void SomeFunction()
    {
    // do something ...
    TInt result;
    TRAP (result, UnsafeFunctionL());

    }
```

1. **setjmp** i.e. stack position stored

2. normal C++ call of function

3. Memory allocated on heap

4. Leave occurs (**longjmp**)

*** Never deleted ***

5. Execution resumes at the **TRAP** but heap memory pointed to by **test** is not accessible. In this example at point (3) **CTestClass** is allocated memory but leaves at (4) and **delete** is never called on **test** i.e. so it is "orphaned"

```
void UnsafeFunctionL()
    {
    CTestClass* test = CTestClass::NewL();

    test->FunctionMayLeaveL();

    delete test;
    }
```

Introducing the Cleanup Stack

To make a function leave-safe

- Heap objects referenced only by local variables must be pushed onto the cleanup stack before calling any functions which may leave
- The cleanup stack will delete the heap memory should a leave occur
- The cleanup stack is discussed in more detail later

Where the Cleanup Stack is Used

```
void SomeFunction()
    {
    // do something ...
    TInt result;
    TRAP (result,UnsafeFunctionL());

    }
```

1. **`setjmp`** i.e. stack position stored

6. The TRAP macro detects something has been pushed onto the cleanup stack and destroys it as part of leave processing. Execution continues.

2. Normal C++ call of function

3. Memory allocated on heap

4. **`test`** (address of the memory allocated) is placed on the cleanup stack

5. Leave occurs (**`longjmp`**)

If no leave occurs it is important to pop the item off the stack and destroy it normally.

```
void UnsafeFunctionL()
    {
    CTestClass* test = CTestClass::NewL();
    CleanupStack::PushL(test);

    test->FunctionMayLeaveL();

    CleanupStack::Pop(test);
    delete test;
    }
```

Don't worry about the exact details - these will be covered later

More Symbian OS Naming Conventions

How is it possible to know whether a function may leave?

- Symbian OS has a naming convention in place to indicate this
- If a function may leave, its name must end with a trailing "**`L`**" to identify as much
- This is important: If a leaving function is not named to indicate its potential to leave, callers of that function may not defend themselves against a leave and may accidentally leak memory
- Symbian OS provides a tool, **`LeaveScan`**, that checks code for incorrectly named leaving functions

More Symbian OS Conventions!

Generally

- Leaving functions should return **`void`**
- Unless they use the return value for a pointer or reference to a resource allocated by the function
- Since leaving functions by definition leave with an error code - a "leave code" - they do not also need to return error values

Any error that occurs

- In a leaving function should be passed out as a leave
- If the function does not leave it is deemed to have succeeded and will return normally

Examples of Typical Leaves

Four possible leaves:

```
TInt UseCat(CCat* aCat); // Forward declaration
CCat* InitializeCatL()
    {
    CCat* cat = new(ELeave) CCat();  // (1)
    CleanupStack::PushL(cat);        // (2)
    cat->InitializeL();              // (3)
    User::LeaveIfError(UseCat(cat)); // (4)
    CleanupStack::Pop(cat);
    return (cat);
    }
```

1. Using overloaded **`operator new`** to allocate memory
2. Pushing something up to the cleanup stack (see later)
3. Calling a leaving method
4. Calling a non-leaving method that may return an error code surrounded with a system function to cause a leave

When a Leave should not Occur: constructors & destructors

If a leave occurs in a constructor

- It places the object in an indeterminate state
- If a constructor can fail, through lack of the resources necessary to create or initialize the object, it is possible that memory would be leaked.
- The two-phase construction paradigm must be used to prevent this (discussed next lecture)

A leave should never occur in a destructor or in cleanup code

- A leave part-way through a destructor will leave the object destruction incomplete - which may leak its resources
- A destructor could itself be called as part of cleanup following a leave and a further leave at this point would be undesirable...

  ...if nothing else because it would mask the initial reason for the leave

Member Variables are Leave-Safe

Heap variables

- Referenced only by local variables may be orphaned if a leave occurs

Member variables

- Will not suffer a similar fate - unless their destructor neglects to destroy them when it is called at some later point

```
void CTestClass::SafeFunctionL()
    {
    iMember = CCatClass::NewL(); // Allocates a heap member
    FunctionMayLeaveL();         // Safe for iMember
    }
```

symbian Academy

Leaves and the Cleanup Stack

Comparing Leaves and Panics

- Understand the difference between a leave and a panic
- Recognize that panics come about through assertion failures, which should be used to flag programming errors during development
- Recognize that a leave should not be used to direct normal code logic

Comparing Leaves and Panics

Leaves are for graceful handling of problems

- Leaves occur under exceptional conditions such as out-of-memory or out-of-disk-space and are used in place of returning an error
- Leaves should only be used to propagate an error or exception to a point in the code which can handle it gracefully
- They should not be used to direct the normal flow of program logic
- Leaves should always be caught and handled — they do not terminate the flow of execution

Comparing Leaves and Panics

Panics - a 'stop everything' mechanism for programming errors

- Panics cannot be caught and handled
- A panic terminates the thread in which it occurs - usually the entire application

Panics should only be used

- In assertion statements to check code logic and fix programming errors during development (bad user experience)
- If a panic occurs from system or application code during development, it's necessary to find the cause and fix it

Symbian OS panics

- Are documented in the Symbian Developer Library
- Panics and assertions are covered in more depth in a later lecture

symbian Academy

Leaves and the Cleanup Stack

What Is a TRAP?

- Recognize the characteristics of a **`TRAP`** handler
- Understand that, for efficiency, use of **`TRAP`** s should be kept to a minimum
- Understand the meaning of the Symbian OS function suffixes **`C`** and **`D`**

What is a **TRAP** ?

Symbian OS provides two trap harness macros **TRAP** and **TRAPD**:

- Both are used to trap leaves and allow them to be handled

  TRAPD declares the variable in which the leave code is returned

  TRAP must declare a **TInt** variable itself first

```
TRAPD(result, MayLeaveL());
if (KErrNone!=result)
    ...
```

is equivalent to:

```
TInt result;
TRAP(result, MayLeaveL());
if (KErrNone!=result)
    ...
```

- If a leave occurs inside the **MayLeaveL()** function, which is executed inside the harness, the program control will return immediately to the TRAP harness macro
- The variable result will contain the error code associated with the leave or will be **KErrNone** if no leave occurred

Traps Incur Overheads

Each `TRAP`

- Has an impact on executable size and execution speed
- The entry to and exit from a `TRAP` macro results in kernel executive calls `TTrap::Trap()` and `TTrap::UnTrap()`
- Kernel calls switch the user-side code into processor-privileged mode in order to access kernel resources
- Kernel calls are quite expensive in terms of execution speed.
- In addition, a structure is allocated at run-time to hold the current contents of the thread's stack in order to return to that state should a leave occur

These factors

- Combined with the inline code generated by the `TRAP` macro, add up to a fairly significant overhead.
- The number of `TRAP`s should be minimized where possible
- Use intelligently, code review with peers, and refactor where necessary

A Note about the D Suffix Naming Convention

In UI programming

- The D suffix means something completely different!
- A function whose name ends in D will take responsibility for destroying the object on which it is called
- Since the function will **`delete`** the object when it is finished with it, any calling code should not attempt to do so
- A good example of such a function is **`CEikDialog::ExecuteLD()`**

symbian Academy

Leaves and the Cleanup Stack

The Cleanup Stack

- Know how to use the cleanup stack to make code leave-safe, so memory is not leaked in the event of a leave
- Understand that **`CleanupStack::PushL()`** will not leak memory even if it leaves
- Know the order in which to remove items from the cleanup stack, and how to use **`CleanupStack::PopAndDestroy()`** and **`CleanupStack::Pop()`**
- Recognize correct and incorrect use of the cleanup stack
- Understand the consequences of putting a C class on the cleanup stack if it does not derive from **`CBase`**
- Know how to use **`CleanupStack::PushL()`** and **`CleanupXXXPushL()`** for objects of **`C`**, **`R`**, **`M`** and **`T`** classes and **`CleanupArrayDeletePushL()`** for C++ arrays
- Understand the meaning of the Symbian OS function suffixes **`C`** and **`D`**

TRAPS, leaves and the cleanup stack

To Recap

- A memory leak can occur as a result of a leave when there are heap objects accessible only through pointers local to the function that leaves
- Pointers to objects that are not otherwise leave-safe should be placed on the cleanup stack before calling code that may leave.

This ensures

- the objects pointed to are destroyed correctly if a leave occurs as part of the **`TRAP`** the cleanup stack manages the deallocation of all objects which have been placed upon it

Recap: The Cleanup Stack in Use

```
void SomeFunction()
    {
    // do something ...
    TInt result;
    TRAP (result,UnsafeFunctionL());

    }
```

1. **`setjmp`** i.e. stack position stored

6. The TRAP macro detects something has been pushed onto the cleanup stack and destroys it as part of leave processing. Execution continues.

2. Normal C++ call of function

```
void UnsafeFunctionL()
    {
    CTestClass* test = CTestClass::NewL();
    CleanupStack::PushL(test);

    test->FunctionMayLeaveL();

    CleanupStack::Pop(test);
    delete test;
    }
```

3. Memory allocated on heap

4. **test** (address of the memory allocated) is placed on the cleanup stack

5. Leave occurs (**`longjmp`**)

If no leave occurs it is important to pop the item off the stack and destroy it normally.

The `CleanupStack` Class

All methods are static

- Defined in **`e32base.h`**

```
class CleanupStack
    {
public:
    IMPORT_C static void PushL(TAny* aPtr);
    IMPORT_C static void PushL(CBase* aPtr);
    IMPORT_C static void PushL(TCleanupItem anItem);
    IMPORT_C static void Pop();
    IMPORT_C static void Pop(TInt aCount);
    IMPORT_C static void PopAndDestroy();
    IMPORT_C static void PopAndDestroy(TInt aCount);
    IMPORT_C static void Check(TAny* aExpectedItem);
    inline static void Pop(TAny* aExpectedItem);
    inline static void Pop(TInt aCount,TAny* aLastExpectedItem);
    inline static void PopAndDestroy(TAny* aExpectedItem);
    inline static void PopAndDestroy(TInt aCount,TAny* aLastExpectedItem);
    };
```

The `CleanupStack` Class

It is a stack - Last In First Off ("LIFO")

- Pointers are pushed onto and popped off the cleanup stack in strict order
- A series of `Pop()` calls must occur in the reverse order of the `PushL()` calls (LIFO)

`Pop()` and `PopAndDestroy()`

- Pops the last item off the stack
- Pops the last item off the stack and deletes it

`Pop(TInt aCount)` and `PopAndDestroy(TInt aCount)`

- Pops the last `aCount` number of items off
- Pops and deletes the last `aCount` number of items

The `CleanupStack` Class

`Pop(TAny* aExpectedItem)`

`Pop(TInt aCount, TAny* aLastExpectedItem)`

`PopAndDestroy(TAny* aExpectedItem)`

`PopAndDestroy(TInt aCount,TAny* aLastExpectedItem)`

- As with the previous methods pops off a single item or a number of items
- The last item is named: **`aExpectedItem`**
- In debug builds, the cleanup stack will panic if the item being popped off is not the same as the one passed in
- It is good practice to use these methods

`Check(TAny* aExpectedItem)`

- Checks the item on top of the stack is the expected item without popping it
- Panics if it is not
- Used as a debugging tool

PushL()

Why does **`CleanupStack::PushL()`** leave?

- It may need to allocate memory for pointer storage and thus may fail in low-memory situations
- The object passed into the **`PushL()`** method will be safe
- It will not be orphaned because when the cleanup stack is created it has at least one spare slot
- **`PushL()`** adds the pointer to the next vacant slot and then checks to see if there are slots free for next time **`PushL()`** is called

If there are no remaining slots available

- The cleanup stack implementation attempts to allocate more slots for future usage
- If this allocation fails only then does a leave occur
- The pointer passed in has already been stored safely thus the object it refers to will be safely cleaned up

PushL()

There are three overloads of the **PushL()** method used to place items onto the cleanup stack:

```
IMPORT_C static void PushL(CBase* aPtr);
IMPORT_C static void PushL(TAny* aPtr);
IMPORT_C static void PushL(TCleanupItem anItem);
```

- Each overload determines how the item is later destroyed when it is cleaned up when a leave occurs or through a call to **CleanupStack::PopAndDestroy()**

PushL() and PopAndDestroy()

```
IMPORT_C static void PushL(CBase* aPtr)
```

- Takes a pointer to a **CBase**-derived object
- It will be configured to be destroyed by invoking **delete** on the pointer
- The virtual destructor of the **CBase**-derived object is called
- This is the reason that the **CBase** class has a virtual destructor and must be used as a base class for all C classes. It means that C-class objects can be placed on the cleanup stack and destroyed safely if a leave occurs

PushL() and PopAndDestroy()

```
IMPORT_C static void PushL(TAny* aPtr)
```

- Used whenever any heap-based object that does not derive from **CBase** is pushed onto the cleanup stack
- T-class objects and **structs** that have been allocated on the heap
- The object's heap memory is deallocated by invoking **User::Free()**
- **delete** is not called - so no destructor is called
- T classes do not have destructors, as previously discussed

PushL() and PopAndDestroy()

```
IMPORT_C static void PushL(TCleanupItem anItem)
```

- Used to allow objects with types of cleanup processing other than **CBase** deletion or simple deallocation to be made leave-safe
  - Such as R or M classes, or classes with customized cleanup routines
- A **TCleanupItem** object encapsulates a pointer to the object and a pointer to a function that provides cleanup for that object
- The cleanup function can be a local function or a static method of a class
- A leave or a call to **PopAndDestroy()** removes the object from the cleanup stack and calls the cleanup function provided by the **TCleanupItem**
- Symbian OS also provides a set of template utility functions to generate an object of type **TCleanupItem** and pushes it onto the cleanup stack

`CleanupXXXPushL()` Template Functions

`CleanupReleasePushL()`

- The cleanup method calls Release() on the object
- Typically used to make leave-safe an object referenced through an M-class (mixin) pointer

`CleanupDeletePushL()`

- Calls delete on the pointer passed into the function.
- Typically used for M-class objects, which should not be pushed onto the cleanup stack using the `CleanupStack::PushL(TAny*)` overload

  Objects referred to by M-class pointer usually cannot simply be deallocated by a call to User::Free()

`CleanupXXXPushL()` Template Functions

`CleanupClosePushL()`

- The cleanup method calls `Close()` on the object in question
- Typically used to make stack-based R-class objects leave-safe

```
RFs theFs;
User::LeaveIfError(theFs.Connect());
CleanupClosePushL(theFs);
... // Call functions which may leave
CleanupStack::PopAndDestroy(&theFs);
```

`CleanupXXXPushL()` Template Functions

`CleanupArrayDeletePushL()`

- Used to push a pointer to a heap-based C++ array of T-class objects (or built-in types) on to the cleanup stack
- When `PopAndDestroy()` is called, the memory allocated for the array is cleaned up using `delete[]`
- No destructor is called on the elements of the array

When to Remove an Item from the Cleanup Stack

Ownership Transfer

- It should never be possible for an object to be cleaned up more than once
- If a pointer to an object on the cleanup stack is later stored elsewhere, say as a member variable of another object which is accessible after a leave, the pointer should then be popped from the cleanup stack.

1. The stack variable **`ptr`** points to a chuck of memory allocated on the heap

2. The next function may leave so placing the pointer on the cleanup stack means the heap memory shall be freed if **`AppendL()`** leaves

3. **`iItemPtrArray`** does not copy the **`CItem`** object but takes ownership of it by allocating a new slot to store the pointer i.e. the address of the **`CItem`** object. The allocation could fail hence it is a leaving function, which is ptr has been placed on the cleanup stack in advance

4. **`iItemPtrArray`** now owns the heap object pointed to by **`ptr`** so **`ptr`** may safely popped off the stack (but it should not be destroyed as **`iItemPtrArray`** owns it, and it would become invalid)

```
void TransferOwnershipExampleL
    {
    CItem* ptr = new(ELeave) CItem();

    CleanupStack::PushL(ptr);

    iItemPtrArray->AppendL(ptr);

    CleanupStack::Pop(ptr);
    }
```

Pointer Member Variables

Pointers which are class member variables

- Should not be pushed onto the cleanup stack
- The object may be accessed through the owning object which destroys it when appropriate
  - Typically in its destructor
- So it does not need to be made leave-safe through use of the cleanup stack
- The next slide shows the consequences of this mistake in more detail...

Coding Error Example

1. **`CSimple`** is created and pushed onto the clean up stack as the next function may leave

```
CSimple* simple = new (ELeave) CSimple();

CleanupStack::PushL(simple);
```

2. A leaving method is called on simple

```
TRAPD(res,simple->MayLeaveFuncL());
...
```

```
class CSimple : CBase
    {
public:
        ~CSimple();
         void MayLeaveFuncL();
private:
        PrivateMayLeaveL();
        CItem* iItem;
    };
```

```
void CSimple::MayLeaveFuncL()
     {
      iItem = new (ELeave) CItem();
      CleanupStack::PushL(iItem);

      PrivateMayLeaveL();
      CleanupStack::Pop(iItem);
      }
```

3. The member variable is pushed onto the clean up stack (oops!)

4. What happens if a leaves occurs?

5 The TRAP does the right thing and clears the clean up stack i.e. **`CSimple::iItem`** is deleted

```
...
```

6. The code logic completes with the popping and deleting of the **`simple`** object.

```
CleanupStack::PopAndDestroy(simple);
```

7. PANIC!

BUT this calls the **`CSimple`** destructor, which deletes the **`iItem`** which has already been deleted by the TRAP

```
CSimple::~CSimple
      {
      delete iItem;
      }
```

When to leave a Pointer on the Cleanup Stack

(More naming conventions)

- In a function, if a pointer to an object is pushed onto the cleanup stack and remains on it when that function returns, the Symbian OS naming convention is to append a C to the function name
- This indicates to the caller that, when the function returns successfully, the cleanup stack has additional pointers on it

```
/*static*/ CSiamese* CSiamese::NewLC(TPointColor aPointColour)
    {
    CSiamese* me = new(ELeave) CSiamese(aPointColour);
    CleanupStack::PushL(me); // Make this leave-safe...
    me->ConstructL();
    return (me); // me remains on the cleanup stack
    }
```

- This type of function is useful because the caller can instantiate **CSiamese** and immediately call a leaving function without needing to push the pointer onto the cleanup stack

When to leave a Pointer on the Cleanup Stack

Functions that leave objects

- On the cleanup stack must not be called from immediately inside a **TRAP** harness
- If objects are pushed onto the cleanup stack inside a **TRAP** and a leave does not occur,
- They must be popped off again before exiting the **TRAP** macro otherwise a panic occurs

This is because the cleanup stack stores objects in nested levels

- Each level is confined within a **TRAP**, and must be empty when the code inside it returns
- The following code panics with **E32USER-CBASE 71** when it returns to the **TRAPD** macro

```
CSiamese* MakeSiamese(TPointColor aPointColour)
    {// The next line will cause a panic (E32User-CBase 71)
    CSiamese* pCat = TRAPD(r, CSiamese::NewLC(aPointColour));
    return (pCat);
    }
```

Creating the Cleanup Stack

The cleanup stack is created as follows:

```
CTrapCleanup* theCleanupStack = CTrapCleanup::New();

... // Code that uses the cleanup stack within a TRAP macro

delete theCleanupStack;
```

- Once created, any leaving code which uses it must be called within a base-level TRAP harness
- It is not necessary to create a cleanup stack for a GUI application, since the application framework creates one.
- A cleanup stack must be created if:
  - writing a server
  - a simple console-test application
  - any code which creates an additional thread that uses the cleanup stack (or calls code that does so)

symbian Academy

Leaves and the Cleanup Stack

Detecting Memory Leaks

- Recognize the use of the **`__UHEAP_MARK`** and **`__UHEAP_MARKEND`** macros to detect memory leaks

Detecting Memory Leaks

Memory is a limited resource on Symbian OS

- It must be managed carefully to ensure it is not wasted by memory leaks
- Applications must gracefully handle any exceptional conditions arising when memory resources are exhausted
- Symbian OS provides a set of debug-only macros that can be added directly to code to check that memory is not leaked
- There are a number of macros available, but the most commonly used are defined as follows:

```
#define __UHEAP_MARK User::__DbgMarkStart(RHeap::EUser)

#define __UHEAP_MARKEND User::__DbgMarkEnd(RHeap::EUser,0)
```

Detecting Memory Leaks

The **__UHEAP_MARK** and **__UHEAP_MARKEND** macros

- Verify that the default user heap is consistent
- The check is started by using **__UHEAP_MARK**
- A subsequent call to **__UHEAP_MARKEND** performs the verification
- Checks to see if any heap cells were allocated after the call to **__UHEAP_MARK** that have not been freed before the call to **__UHEAP_MARKEND**

A panic will occur

- In debug builds to indicate that the heap is inconsistent
- The panic raised is **ALLOC nnnnnnnn**
- **nnnnnnnn** is a hexadecimal pointer to the first orphaned heap cell
- The heap-checking macros can be nested inside each other and used anywhere in code
- They are ignored by release builds of Symbian OS so they can be left in production code without any impact on the code size or speed

symbian Academy

Leaves and the Cleanup Stack

- ✓ Leaves: Lightweight Exceptions for Symbian OS
- ✓ How to Work with Leaves
- ✓ Comparing Leaves and Panics
- ✓ What Is a TRAP?
- ✓ The Cleanup Stack
- ✓ Detecting Memory Leaks