

# Leaves and Exceptions

Jason Morley

Published by the Symbian Developer Network

Version: 1.3.1 – September 2007

<b>1 INTRODUCTION.....</b>	<b>2</b>
<b>2 MOTIVATIONS .....</b>	<b>2</b>
<b>3 IMPLEMENTATION DETAILS.....</b>	<b>2</b>
3.1 USING LEAVES AND TRAPS.....	3
3.2 USING EXCEPTIONS NATIVELY .....	3
<b>4 BEST PRACTICE .....</b>	<b>4</b>
<b>5 MIGRATION DETAILS .....</b>	<b>5</b>
<b>6 FAQs .....</b>	<b>5</b>

## 1 Introduction

Symbian OS v9.x now explicitly supports standard C++ exceptions and these are used to implement the Leave/TRAP architecture.

This document is intended to provide a brief discussion of the motivations for changing the implementation of Leave to use C++ exceptions, some technical discussion and details of any resulting usage limitations.

## 2 Motivations

There are a number of reasons why exceptions are now used to implement TRAP. Some of these are as follows:

- Exceptions use significantly less CPU cycles to set-up compared to TRAPs which were costly even if the containing code did not leave.
- TRAP was incompatible with standard C++ exceptions. This meant that it was impossible for standard C++ code to co-exist with Symbian-style code. With TRAPs implemented as exceptions, it is possible (with care) for these to co-exist within the same binary, easing porting effort.
- C++ exceptions represent the industry standard and it is logical for Symbian to move towards this.
- Since exceptions are standardised, there is much greater native hardware support (e.g. EABI). This serves to make exceptions faster and would quite likely make an architecture port easier.

## 3 Implementation Details

Following standard Symbian OS behaviour standards, the implementation of exceptions has been designed in such a way to ensure that it is both deterministic and safe in Out of Memory (OOM) situations. This introduces some limitations over standard implementations.

When an exception object is thrown, memory must be allocated for the exception object. Supporting nested exceptions requires dynamic heap allocation. This is because, when an exception is thrown while another exception is already being handled, a further allocation is necessary; the memory requirement would therefore be unbounded. Such unbounded memory requirements are unacceptable when addressing OOM situations.

Therefore, in Symbian's implementation, nested exceptions are not supported and sufficient memory is pre-allocated to ensure that a single exception object can always be created. **On target, Symbian OS code will always call abort() if a nested exception occurs. Specifically, this occurs if an exception is thrown while unwinding the stack (within a destructor).**

N.B. - Nested exceptions **are** supported on WINS, as exceptions are implemented using win32 structured exception handling. Therefore code which appears correct on emulator may not function as intended on target.

### 3.1 Using Leaves and TRAPs

The implementation of `User::Leave()` is as follows:

1. `User::Leave()` is called.
2. The Cleanup Stack is un-wound.
3. The corresponding exception object is created and 'thrown'.  
*If there is enough space on the heap, this object is allocated on the heap; if not, the object is created using pre-allocated space on the stack. This is to ensure that the exception object can be created even if the exception is being thrown as the result of an OOM error.*
4. The normal stack is unwound as part of the exception handling.
5. The "catch" block (or equivalent) is executed.

There are a number of points which should be noted here:

- Step 2 deals with the destruction of any heap-based object on the cleanup stack.

*The exception associated with the Leave has not yet been created.*

*It is safe to throw an exception at this point as we will not introduce any nesting of exceptions. Exceptions in the destructors of objects on the cleanup stack will be 'completed' before this exception is even created.*

- Step 4 deals with the destruction of stack based objects.

*The exception associated with the Leave has now been created.*

*Throwing an exception at this stage would require nested exception support as the previous exception object has now been created and thrown. While this will work on WINS, it is explicitly forbidden on target and `abort()` will be called.*

- Step 5 executes the 'recovery' code.

*At this point, we have exited the Leave / exception handling and are running normal code again, so it is now safe to throw a new exception.*

Therefore, it is safe for heap-bound objects (whose destruction is handled by the Cleanup Stack) to use TRAP within their destructors. It is not safe to do this for stack-based objects.

Since any `CBase`-derived object should exist on the heap and should be pushed onto the Cleanup Stack, it is safe to assume that TRAPs can be used within the destructors of `CBase`-derived objects.

It is not safe for any object whose destruction occurs via the stack-unwind (i.e. Step 4 above) to Leave or TRAP within its destructor. This is because in Step 4 no exception can safely occur.

### 3.2 Using Exceptions Natively

It is possible to use Exceptions directly without the Leave, TRAP framework. In this case the restrictions are more straightforward.

The Cleanup Stack is a Symbian convention and is not used within Native code (i.e. standard C++ code compiled for a Symbian device.) Therefore in Native code only stack based objects need be considered. Here the same restrictions apply; it is not safe to use exceptions within destructors of objects on the stack.

It should also be noted that Standard C++ features such as `auto_ptr` (which ensures cleanup of heap based objects) are handled as part of the stack unwinding. Therefore, objects whose destruction is performed in this way would have the same restrictions as pure stack-based objects; it is not safe to use exceptions in the destructors of these objects.

## 4 Best Practice

Section 3.1 explains how it is possible to use TRAPs within the destructors of CBase-derived objects. However, we still advise that destructors in both heap-based **and** stack based objects are kept simple and avoid calling leaving code wherever possible.

Calling a leaving function from within a destructor implies that part of the destruction might fail, potentially leading to memory or handle leaks.

Ideally, APIs which might be used in destructors should be designed to avoid using the Leave mechanism to return error codes and should, instead, simply return a `TInt`.

One possible approach to avoid Leaving functions within the destructor would be to have what might be referred to as 'two-phase destruction' whereby some form of 'ShutdownL()' function is called prior to deleting the object.

If there are concerns that this may introduce additional complexities in API-usage (and greater risk) suitable guards can be introduced. One method might be to store the current state of the object internally and then use an ASSERT to check this in the destructor. This should ensure that any usage-errors are discovered by very simple run-time testing. Consider the following example:

```
// Shutdown function which performs any destructi on which may leave.
class CObject::ShutdownL() {
    if (iStateActive == ETrue)
    {
        // Some destructi on which may leave.
        iStateActive = EFalse;
    }
}

class Cfoo: ~foo()
{
    // Assert to ensure that ShutdownL has already been called.
    ASSERT(iStateActive == EFalse);
}
```

If it is impossible to avoid calling leaving code within the destructor, this code should **always** be handled within a TRAP as an un-trapped leave within a destructor will terminate the entire process. *This does not represent any functional change over previous versions of Symbian OS.*

## 5 Migration Details

Symbian's move to using C++ exceptions to implement Leaves and TRAPs has created some additional restrictions in their use. These restrictions only affect use within destructors and the following rules apply:

- Leaves and TRAPs **cannot be used** in the destructor of **any** object which is destroyed during stack-unwinding.
- Leaves and TRAPs **can be used** in the destructor of any object which is destroyed by the cleanup stack (i.e. CBase-derived objects). However, this is not recommended.
- Destructors **should never** Leave or throw an exception. It should always be trapped or caught.

N.B. These rules apply to the destructor and **any** function called from within that destructor.

## 6 FAQs

A number of questions have been asked in relation to this document. These have been collected here along with the respective responses, to offer further clarification.

**Q: So TRAPs can be used in destructors under the condition that the object is allocated on the heap and its cleanup is handled by the Cleanup Stack?**

**A:** Yes.

**Q: Is it forbidden to call 'delete' directly?**

**A:** Not always.

It is safe to call 'delete' directly in almost all code. It is only dangerous to call 'delete' on an object which uses a TRAP in its destructor where the 'delete' may be executed during stack unwinding. This is because there may already be an active exception at this stage and the TRAP could lead to a subsequent exception.

Consider the following three examples using the following class:

```
class CFoo : public CBase
{
    ~CFoo()
    {
        TRAP(err, /* Leaving Code */);
    }
}
```

### 1. Calling 'delete' from code outside of destructors – **ALWAYS SAFE**

```
class CAnother
{
    void CMiscFunction()
    {
        CFoo foo = new(ELeave) CFoo();
        CleanupClosePushL(foo);
    }
}
```

```

        foo->ConstructL();
        ...
        delete foo;
        CleanupStack::Pop(foo);
    }
}

```

*This is always safe because there is no possibility of this code being called while an exception is being "thrown".*

N.B. In this situation, any ordering of the "delete" and "CleanupStack::Pop()" may be used as long no code leaves between these two calls.

## 2. Calling "delete" from destructor of heap based object (C-Class) – **SOMETIMES SAFE**

```

class CBar : public CBase
{
    CBar(CBase* aFoo)
        : iFoo(aFoo)
    {
    }

    ~CBar()
    {
        delete iFoo;
    }

    CFoo* iFoo;
}

```

*This is safe if and only if the destruction of CBar is handled by the Cleanup Stack. This is because an exception will not have been 'thrown' before the destructor of CBar is called.*

## 3. Calling "delete" from destructor of stack based object (T-Class) – **NOT SAFE**

```

class TYetAnother
{
    TYetAnother(CBase* aFoo)
        : iFoo(aFoo)
    {
    }

    ~TYetAnother()
    {
        delete iFoo;
    }

    CFoo* iFoo;
}

```

*Here, TYetAnother has been designed as an attempt to automatically destruct the heap-based object iFoo. However, this is not safe on any version of Symbian OS. Before leaves-as-exceptions, stack-based objects were just de-allocated and the destructors were not called so iFoo would not be deleted. After leaves-as-exceptions, destructors are called as part of the stack-*

*unwinding, but it is possible for an exception to have been thrown before the delete is called leading to a nested exception as described in the document.*

*If you wished to achieve something similar, you should be using the Cleanup Stack.*

**Q: According to the three rules shown in "5 Migration Details", the following code is safe but is not recommended. Is this correct?**

```
CFoo: : ~CFoo()  
{  
...  
TRAPD(err, iMember->Delete());  
...  
}
```

**A:** Yes this is correct, assuming that CFoo is derived from CBase and the object exists on the heap and has been pushed to the Cleanup Stack.

Want to be kept informed of new articles being made available on the Symbian Developer Network?

[Subscribe to the Symbian Community Newsletter.](#)

The Symbian Community Newsletter brings you, every month, the latest news and resources for Symbian OS.