A Comparison of Leaves and Exceptions

Original Author: Jason Morely

Early versions of Symbian OS predate C++ exceptions. Symbian developed its own system of **Leave** and **TRAP**. In Symbian OS v9 support for try-catch exceptions was added and the Leave/TRAP architecture was re-implemented.

This document discusses the motivation for changing the implementation of Leave and TRAP to use C++ exceptions. It also describes some of the technical issues and the resulting usage limitations.

Motivation

There are several reasons why exceptions are now used to implement TRAP:

- Exceptions use significantly fewer CPU cycles than old-style Symbian `TRAP`s, 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 C++ code. With TRAPs implemented as exceptions, it is possible (with care) for standard and Symbian C++ to co-exist within the same binary.
- C++ exceptions are the industry standard. It is logical for Symbian to move towards this.
- Since exceptions are standardised, there is much greater native hardware support (e.g. EABI). This makes exceptions faster and would probably make an architecture port easier.

Implementation Details

In accordance with Symbian requirements, exceptions are implemented in a way that it is both deterministic and safe in Out of Memory (OOM) situations. This introduces a significant limitation over standard implementations: namely that **nested exceptions are not supported.**

When an exception object is thrown, memory must be allocated to create an exception object. The Symbian implementation pre-allocates sufficient memory to ensure that a single exception object can always be created. However, if that exception were to contain a nested exception the pre-allocated memory would not be enough and, in an OOM situation, it would not be possible to allocate more.

On ARM devices Symbian calls `abort()` in the event of a nested exception. This is most likely to occur if an exception is thrown within a destructor while the stack is being unwound.

Note: Nested exceptions are supported on WINS, as exceptions are implemented using win32 structured exception handling. Therefore code which appears correct on an emulator may not function as intended on target.

Using Leave and TRAP

The implementation of `User::Leave()` in Symbian OS v9 is as follows:

1. `User::Leave()` is called.
2. The cleanup stack is unwound.
3. An `XLeaveException` exception object is created and ‘thrown’. If there is enough space on the heap, the object is allocated on the heap; if not, it is created using pre-allocated space on the stack. The pre-allocated space on the stack ensures that an 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 to note:

- 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. Though this will work on WINS, it is explicitly forbidden on ARM.

(C) Copyright Nokia 2014. All rights reserved.

- 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.

It is safe, therefore, for heap-bound objects (whose destruction is handled by the cleanup stack) to use TRAP within their destructors. It is not safe for stack-based objects to do so. Since any CBase-derived object should exist on the heap and should be pushed onto the cleanup stack, it is safe to use TRAPs in the destructor of a CBase-derived object. 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.

Using exceptions in standard C++ code

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 C++ convention and is not used within standard C++ code compiled for a Symbian device. Therefore in such 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.

Standard C++ features such as `auto_ptr` (which ensures cleanup of heap based objects) are handled as part of the stack unwinding. Objects whose destruction is performed in this way have the same restrictions as pure stack-based objects; it is not safe to use exceptions in their destructors.

Best Practice

As described above, you can use TRAPs within the destructors of `CBase`-derived objects. However, we advise that you keep destructors in both heap-based and stack-based objects 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 and should, instead, simply return a `TInt`.

One approach for avoiding Leaving functions within the destructor is to have 'two-phase destruction' where some form of `ShutdownL()` function is called prior to deleting the object.

If you are concerned that this might introduce additional complexity (and risk) into your API you can use guards. One method might be to store the current state of the object internally and then use an ASSERT to check it 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 destruction which may leave.
class CObject::ShutdownL()
  {
  if ( iStateActive == ETrue )
    {
    // Some destruction which may leave.
    iStateActive = EFalse ;
    }
  }

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

If it is impossible to avoid calling a leaving function within the destructor, such a call must 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 the Symbian platform.*

Migration Details

The use of C++ exceptions to implement Leave and `TRAP` creates some additional restrictions in their use. These restrictions only affect use within destructors and the following rules apply:

(C) Copyright Nokia 2014. All rights reserved.

- Leave and `TRAP` must **not be used** in the destructor of any object which is destroyed during stack-unwinding.
- Leave and TRAP **may 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.
- A destructor **must not** leave or throw an exception. Leave must always be `TRAP`ed and exceptions must always be caught.

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

FAQs

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?

Yes.

Is it forbidden to call `delete` directly?

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 un-winding. 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 a destructor **is 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".

Note: In this situation, the order of the `delete` and `CleanupStack::Pop()` calls is not significant as long no code leaves between the two calls.

2. Calling `delete` from the destructor of a heap based object (C-Class) **is sometimes safe**

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


(C) Copyright Nokia 2014. All rights reserved.

```
  ~CBar()
    {
    delete iFoo ;
    }

  CFoo* iFoo ;
  }
```

This is safe ***only if*** the destruction of `CBar` is handled by the cleanup stack. This is because an exception cannot have been 'thrown' before the destructor of CBar is called.

3. Calling `delete` from the destructor of a stack based object (T-Class) **is not safe**

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

  ~TYetAnother()
    {
    delete iFoo ;
    }
  CFoo* iFoo ;
  }
```

In this example `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.

- Before leaves-as-exceptions were introduced, stack-based objects were simply de-allocated and their destructors were not called - so `iFoo` would not have been deleted.
- With leaves-as-exceptions, destructors *are* called as part of the stack-unwinding process, but it is possible for an exception to have been thrown before the delete is called leading to a forbidden nested exception.

If you wished to achieve something similar, you must use the cleanup stack.

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

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

Yes this is correct, assuming that `CFoo` is derived from `CBase`, that the object exists on the heap and has that it is pushed to the cleanup stack.

Further Information

- Fundamentals of Symbian C++/Leaves & The Cleanup Stack
- Explanation of nested exceptions

© 2010 Symbian Foundation Limited. This document is licensed under the Creative Commons Attribution-Share Alike 2.0 license. See http://creativecommons.org/licenses/by-sa/2.0/legalcode for the full terms of the license.

Note that this content was originally hosted on the Symbian Foundation developer wiki.

(C) Copyright Nokia 2014. All rights reserved.

(C) Copyright Nokia 2014. All rights reserved.