

Exception Handling in Symbian OS

Ben Morris

Published by the Symbian Developer Network

Version: 1.02 – July 2008

1	INTRODUCTION	2
2	EXCEPTIONS, ERRORS AND DEFECTS.....	2
3	SYMBIAN OS NATIVE EXCEPTION HANDLING	3
3.1	USING LEAVES AND TRAPS	4
3.2	CLEANING UP AFTER A LEAVE.....	6
4	STANDARD C++ EXCEPTIONS IN SYMBIAN OS V9.....	7
4.1	USING TRY/CATCH/THROW	7
5	MIXING LEAVES AND STANDARD C++ EXCEPTIONS.....	8
6	AVOID NESTED EXCEPTIONS.....	8
6.1	CHOOSING THE RIGHT STRATEGY.....	9
6.2	PORTING CODE TO SYMBIAN OS	9
6.3	WRITING CODE FOR MULTIPLE PLATFORMS	9
6.4	ORIGINATING CODE FOR SYMBIAN OS.....	9
7	RELATED MECHANISMS	9
7.1	ASSERTIONS AND PANICS	9
8	WRAPPING UP	10
9	READING	11
9.1	REFERENCES	11
9.2	WEB RESOURCES.....	12
10	AUTHOR PROFILE.....	12

1 Introduction

Until the release of Symbian OS v9, Symbian OS supported only non-standard exception handling for native C++ code, using 'leaving' methods and a runtime cleanup stack. If you have worked with Symbian C++ at all, this will be one of the first differences you encountered compared with standard C++.

From Symbian OS v9 onwards, standard C++ exceptions are also supported. This gives developers a choice between native Symbian C++ and standard C++ exception handling, and allows (with some restrictions) intermixing of the two.

This is good news of course, because it continues to move Symbian C++ towards the mainstream, making it easier for new developers to get up to speed faster, and making it easier to port legacy C++ code to the platform. However, since the native mechanism continues to be supported, developers may be confused about which approach to take and when.

This paper provides an overview of C++ exception handling in Symbian OS, including both native and standard approaches. It discusses when each approach is most appropriate, and how the two approaches can safely be mixed.

2 Exceptions, Errors and Defects

An exception is a response to an error condition that (for whatever reason) a program chooses not to handle locally. Invoking the exception mechanism is a way of passing the error on to code that *is* prepared to handle it.

As Bjarne Stroustrup points out (1994, p.384), exceptions are really a language-level concept. Different languages have different exception philosophies. For example, both Java and Python encourage the use of exceptions to handle *all* errors, making return value checking mostly redundant, which is intended to produce cleaner, more readable and less convoluted code.

In contrast, C++ (as described by Stroustrup) takes a more conservative approach, staying closer to its C language roots and obeying the principle *use exceptions only for exceptional situations* (Kernighan and Pike, 1999). 'Non-exceptional' errors are indicated in C-style by returning error codes or a NULL pointer, and exception handling is used relatively sparingly for 'true' exceptions.

In C++, therefore, when the code that is calling a method is able to pre-empt an error, for example, by checking that a file exists before trying to open it for reading, then it should do so. It is only when it cannot pre-empt a possible error that it should fall back on the exception mechanism, handling any raised exception appropriately.¹

In all systems, throwing an exception is a way of changing the normal flow of control by passing an error back through the call stack, instead of handling it immediately (even though a Java or Python exception may travel no further than an adjacent code block). Language philosophies aside, there are some kinds of errors that a program should never handle itself, for example, programming errors (that is, defects). When a programming error occurs, a program should fail; trying to survive a defect is misguided.

Equally, there are some kinds of errors that should always be handled locally, for example, user input errors; a program should never fail because the user mis-typed a file name, or tried to enter bad data into a dialog box.

¹ This is 'classical' C++ style, and is not mandated by the C++ standard which leaves it for the programmer to decide whether to throw an exception or return an error code.

A very simple error classification follows from those principles:

- Some errors should always halt the program.
- Some errors should never halt the program.
- Some errors are best handled locally/immediately.
- Some errors are best propagated.

Languages (and to some extent systems too) differ in how they fill in the matrix this classification implies, and in what support (if any) they build in for handling the different error categories:

	Do Propagate	Don't Propagate
Do Halt	Fatal error handled at system level	Fatal error handled locally
Don't Halt	Exception	Non-exceptional error

Software exceptions are errors that are always propagated (that is., change the flow of control), and that do not automatically halt the program in which they are raised (although an exception handler still has the freedom to do this if required).

Bringing the discussion back specifically to C++, Stroustrup describes a principal design goal of the language as being to support the development of fault tolerant systems (1994, Chapter 16). But that does not mean that every part of every C++ program should be fault-tolerant; not every function should be a firewall. The rationale for C++ exception handling is to allow failure to propagate appropriately. This is in line with one basic design philosophy of the language that 'fault-tolerant systems must be multi-level;' in other words: 'At some point, the unit must give up and leave further cleanup to a "higher" unit' (Stroustrup, 1994, p.385).

3 Symbian OS Native Exception Handling

Since its first releases, Symbian OS has provided an idiomatic native C++ exception handling framework. Exceptions are initiated by invoking a 'leave,' and caught by trapping the leave. When a method leaves, the normal path of program execution is abandoned, and the leave is propagated back through the call stack until it is explicitly trapped by a handler. If the program in which the leave occurred does not trap and handle the leave, the system will do this for you (the exact result may vary between platforms, but typically a panic can be expected).

Alongside the leave/trap framework, Symbian OS provides a cleanup mechanism, called the cleanup stack. This allows heap-allocated objects that would otherwise be orphaned to be cleaned up after a leave has been initiated, thus preventing memory leaks.

In other respects, Symbian C++ follows the C++ principle of using exceptions only for exceptional conditions and makes extensive use of error codes to indicate 'predictable' or non-exceptional errors.

The leave/trap framework was designed as a structured and object-oriented alternative to the setjmp/longjmp exception handling of vanilla C, which at the time was the only approach to exception handling supported by the C++ compiler used to build the operating system.

Until Symbian OS v9, all exception handling in Symbian C++ was provided by the leave/trap framework; mastering efficient and effective use of leaves and the cleanup stack was a prerequisite for developing for the native platform. C++ code for Symbian OS was always compiled with standard C++ exceptions disabled.

While potential performance differences between native and standard frameworks have largely been removed by unifying the implementations (indeed, from Symbian OS v9 the leave/trap framework is implemented in terms of standard C++ catch/throw), the native idiom still has a number of advantages for developers originating C++ code for Symbian OS. In particular, explicit use of the cleanup framework (which is required when using native Symbian OS exceptions) forces a highly disciplined and focused coding style on developers – as John Pagonis has put it, the native framework is always 'in-your-face' (Pagonis, 2006). Harrison also emphasizes the point (Harrison and Shackman, 2007, p.70), that cleanup is a fundamental aspect of Symbian OS programming: 'Every line of code that you write - or read - will be influenced by thinking about cleanup.' It is an essential requirement for writing robust code for Symbian OS that programmers must know when and how to safeguard allocations by using the cleanup stack.

It is also worth recalling the reason why a native exception framework was considered so essential in the first place, and was so carefully integrated with other aspects of the system (the CBase class of the native derivation hierarchy, for example, provides seamless support for the cleanup stack). Symbian OS targets a very specialized device domain: that of communications-centric mobile devices generally, and of mobile phones in particular. On such devices, the possibilities for resource allocations to fail are much greater than in big systems of the kind the C++ language originally targeted (telephone exchanges, for example).

3.1 Using leaves and traps

There are many good Symbian OS programming tutorials that describe how to use leaves and traps (see the references at the end of this paper). As Stichbury (Stichbury, 2004, p.15) summarizes it:

- User::Leave() is analogous to a C++ throw
- TRAP and TRAPD are analogous to a combination of try and catch (or equivalently the pair leave/trap is approximately equal to the pair setjmp/longjmp).

Traps are simple code wrappers around calls, and are generated by use of the TRAP or TRAPD macro.

To trap a leave, call the leaving method from inside a trap harness:

```
TRAPD(error, MyLeavingMethodL());
if (KErrNone == error)
    { // ... do something
    }
else
    { // ... start handling the exception
      // ... and possibly use another Leave to propagate it
    }
```

TRAPD, by the way, is just a convenience form of TRAP that provides a ready-made error variable.

If MyLeavingMethodL(), or any other method called from it, leaves, the leave code will be stored in the error variable, and execution returned to the trap. The code that follows the trap can then test the value of the error, and handle the exception appropriately.

The leave itself is implemented in MyLeavingMethodL() as a call to one of the leaving methods supplied by the user library (EUser). The user library provides a small family of leaving methods, User::Leave(), and three others that are implemented in terms of it that test for particular conditions: User::LeaveIfError(), User::LeaveNoMemory() and User::LeaveIfNull(). In particular, User::LeaveIfError() is provided as a convenient wrapper for converting methods which themselves return errors (for example, calls to the file server) into leaving methods.

The user library also provides an overloaded `new` operator, which is used to convert a failed allocation directly into a leave, invoked as:

```
// Construct a new instance of MyObject
MyObject* obj = new (ELeave) MyObject;
```

In this overloaded form, a leaving version of the heap allocator is used, which leaves if there is insufficient memory to allocate the requested object.

Some things to notice are:

- Essential in Symbian C++ programming is the use of the 'trailing L' naming convention to identify methods that may leave, for example, `MyLeavingMethodL()`. All system code adopts this convention, and you should too, making it immediately clear to a caller that a method may leave.
- A leave may be invoked explicitly or may be propagated from code that has been called within a method.
- When a method leaves, it does so with a leave code; you do not therefore need to return an error code in addition. Typically, methods that leave should return `void`, and all errors that occur in the method should be passed out as leaves.
- A leave causes the current path of execution to be abandoned and then resumed at the first trap handler encountered when the stack is unwound.

Note that it is not necessary to trap every leave, and in some cases a leave may be left untrapped, in which case it will propagate to system level where it will be trapped (typically resulting in a panic or termination of the program). The over zealous use of traps can turn out to be the surprise culprit in cases of code bloat, when the macro invocations are expanded during compilation.

3.1.1 Summary

To summarize:

- A method is leaving if it invokes a leave, for example, by invoking `User::Leave()`, directly or indirectly through one of the related system calls, or if it calls a method which may leave, including calls to `NewL()`, the overloaded operator `new`.
- A leave aborts its method and any methods that called it, and unwinds the call stack, up to but excluding the first method that contains a trap harness (Harrison, 2003, p.138).
- Methods that leave should not return error values since leaves include a leave code.
- Wrap calls to methods that only return errors, to convert them into leaving methods (for example, by using `User::LeaveIfError()`).
- Leaves are for actions that are not guaranteed to succeed, but especially for any resource allocation at all, or any operation that immediately (or even ultimately) depends on resource allocation. As Harrison (2003) puts it, these can be thought of as environment errors, such as a lack of the resources your program needs when it runs. Always leave if allocation fails.
- If you don't trap a leave, it will propagate.
- Not every leave should be trapped, and in fact trapping should be the exception and not the rule; allow exceptions to propagate to an appropriate level before handling them.
- If TRAPDs are nested, the error variable is reused, hence code may test the wrong error variable; use TRAP instead (Stichbury, 2004, p.21).

- Traps can be used to trap and handle an error without leaving, for example, in an application Draw() method (which must not leave); in other words, trap and handle in order not to leave (Harrison, 2003, p.153).
- The cleanup framework can be used to catch and handle user input errors too, for example, incorrect input into user dialogs (Harrison, 2003, p.172). This is covered in the next section.

3.2 Cleaning up after a leave

When a leave occurs, execution is transferred to the first trap handler encountered in a traversal back through the call stack, i.e., as the stack is unwound. The current execution context is thrown away, and is replaced with the context saved by the code to which the TRAP macro expands. Thus a leave abandons the stack frame of the function in which it occurs, and replaces it with the stack frame of the nearest enclosing trap harness.

This means that any objects allocated on the heap by code executed between the leave and the trap, and whose pointers are only locally scoped by variables on the stack (automatics), will have been orphaned. The basic C++ stack unwinding which occurs when traversing back to the trap does not perform any heap deallocation - only the local variables are deallocated, and not the heap objects they point to. In such a scenario, heap memory may be leaked.

On Symbian OS, the cleanup stack is provided to enable developers to explicitly mark heap objects for cleanup in the event of a leave. The first action the leave framework performs when it is invoked is to unwind the cleanup stack and call the destructors of all objects which have been pushed to it by code executing between the leave and the trap.

Therefore, allocations from within methods that may leave should be guarded by pushing pointers to the newed objects onto the cleanup stack, and popping them off the cleanup stack either as part of their destruction, or after the potentially leaving action has successfully completed.

Thus, to the trap handler code example from the previous section we would add the following guards:

```

CleanupStack::PushL(something); // will stay on the cleanup stack in the
                                // event of a leave from the following code...
//
// ... Since here the leave is not trapped
MyLeavingMethodL();
//
// Danger over, MyLeavingMethodL() did not leave
CleanupStack::Pop(something);

```

Just as a number of different leaving methods are provided, the cleanup stack provides a number of alternative methods for pushing and popping pointers to heap-based objects including CleanupStack::PopAndDestroy(), which does as it says when a local object is finished with. Again, there are many good Symbian OS programming tutorials showing how to use the cleanup stack; see the references at the end of this paper.

3.2.1 Summary

Effective and correct use of the cleanup stack imposes a substantial learning curve on developers. The following points are worth remembering:

- The cleanup stack maintains at least one free slot, so that a push is always guaranteed to succeed. For further discussion, see [Harrison 2003 p.14] or [Stichbury 2004, p.35] for how the cleanup stack works.

- CBase includes a virtual C++ destructor that enables any derived class to be cleaned up from the cleanup stack.
- Each thread requires a cleanup stack. This is created for you (for the first program's main thread) by the UI framework. However, if you create further threads, you will need to construct a cleanup stack for each. Console-based applications must also explicitly create their own cleanup stack (Harrison, 2003, p.11).
- The pattern for using the cleanup stack is to push a heap-based object to the cleanup stack, perform a potentially leaving action, and pop the cleanup stack when the potentially leaving action has successfully completed; in case of a leave, all objects pushed to the cleanup stack will be cleaned up, i.e., deleted. A variety of push and pop methods are provided, including, for example, `PopAndDestroy()`, which pops an object and then calls its destructor.
- Always use the 'trailing L' naming convention for methods that can leave, thus warning callers of the method that their own method may therefore leave.
- Use the LeaveScan tool (installed by SDKs into `\epoc32\tools`) to check source files for any un-suffixed leaving methods.

4 Standard C++ Exceptions in Symbian OS v9

Compared with standard C++ exceptions, the original Symbian OS leave implementation was designed to be deliberately lightweight (Stichbury, 2004, p27), while performing the same job of propagating an error from the location in which it occurs to higher level code which can handle it. From v9, Symbian OS adds support for standard C++ exception handling and C++ code for Symbian OS is compiled with exception handling switched on. In fact, behind the scenes the entire exception handling framework has been re-engineered to integrate the two different approaches. While usage of the native leave/trap framework is unchanged (for obvious reasons of compatibility), from v9 both `TRAP` and `User::Leave()` are implemented internally in terms of the standard C++ catch and throw. For details, see the short paper by Jason Morley (2007).

4.1 Using try/catch/throw

Instead of leave/trap, C++ code can now use the standard try/catch/throw:

```
// some code block
{
    ...
    // try block
    try {
        // perform some action that might fail, for example
        // memory allocation, or file system or network operation
    }
    catch ( exception ) {
        // handle the failure
        return failure;
    }
    ...
    return success;
}
```

The 'try' block guards code that could throw an exception, which is a typed object (Stroustrup, 1994, p.387); the 'catch' clause catches exceptions which match the exception-type (or its derivations) and performs some suitable response.

The exception is raised from the code which fails, responding to the failure with:

```
... // it failed!
throw ( exception );
```

Execution is terminated at this point, and the value `exception` is returned to the caller. For more details, see any good C++ programming primer.

Standard C++ exceptions do not use a cleanup stack, therefore code that uses standard exceptions should manually ensure that memory leaks do not occur when an exception is raised ('smart pointers' provide one well-documented approach (Meyers, 1996)). Indeed, when porting standard C++ code to Symbian OS, this is an essential step toward ensuring that code written for other platforms will have robust behavior on a typical Symbian OS device.

5 Mixing Leaves and Standard C++ Exceptions

Because the leave/trap framework (from Symbian OS v9 onwards) is implemented in terms of the standard C++ exception mechanism, it is possible to use both leaves and standard C++ exceptions on Symbian OS, although it is not recommended to do so within a single program binary.

The critical rules for mixing the two approaches are:

1. Leaving methods must not throw, unless they also catch internally. The Symbian OS trap can only handle a specific exception, not all classes of standard C++ exception. These must therefore be caught before they can reach the trap. While you can throw and catch exceptions within your code, and you can leave and trap, you should never leave and catch, nor throw and trap. Code that leaves and code that throws should not be mixed within each other's scope.
2. Code that throws should not use the cleanup stack and should not depend on code that uses the cleanup stack, because the cleanup stack will not be unwound in the case of a standard exception.
3. Code that leaves must not call code that throws.
4. But code that throws (e.g., ported third-party code) may call code that leaves (Symbian OS system code).

Destructors should never leave and should never throw – see the next section.

6 Avoid Nested Exceptions

The Symbian OS implementation of standard C++ exception handling pre-allocates sufficient memory to ensure that it will always be possible to create at least one exception object when normal memory is exhausted. However, nested exceptions, which require memory for multiple exception objects, have a memory requirement that exceeds the pre-allocation. In out-of-memory (OOM) conditions, it would not be possible to guarantee more than one exception object could be allocated.

For this reason, Symbian OS forbids nesting standard C++ exceptions. Although a nested exception will succeed on an emulator target, on an ARM target device it will result in a system abort.

Nested exceptions will occur if, during the handling of an exception, a further exception is raised. For example, if a destructor which is called during the handling of an exception itself raises an exception then we find ourselves nesting exceptions. Destructors, and any methods called from them, should therefore *never* raise exceptions.

Because leaves are implemented using the standard C++ throw method, similar restrictions apply to nesting leaves, although in this case the restriction is not quite universal. For full details, see the discussion in Morley (2007).

6.1 Choosing the right strategy

The choice between Symbian native exceptions and standard C++ exceptions is best made depending on whether new code is being written or existing code is being ported.

Whichever approach is taken, the three most critical rules for error handling are:

1. always be ready to handle errors
2. handle errors without losing data
3. handle errors without leaking memory/resources.

6.2 Porting code to Symbian OS

Re-writing existing C++ code that uses standard C++ exceptions to use Symbian's native exception handling framework is time consuming and error prone and offers no advantages. Beware, however, of any potential nested exceptions (see above). Also, be aware that this code cannot use the cleanup stack, and therefore be alert to possible memory leaks arising from raised exceptions, or consider using an alternative strategy like smart pointers. And again, it is important to emphasize that writing native code for mobile devices is different from writing for desktop machines, therefore care needs to be taken to ensure that the ported code will be robust and efficient on small devices.

6.3 Writing code for multiple platforms

When writing new C++ code intended for multiple platforms, use standard C++ exceptions.

6.4 Originating code for Symbian OS

The leave/trap framework is effective, lightweight, fast and robust. It is domain-specialized and proven, and has been significantly optimized in v9. Using the leave/trap architecture is therefore strongly recommended wherever possible. In particular, it is both used and supported across all native APIs and frameworks; is targeted specifically at embedded devices and provides explicit support for OOM situations; used with the cleanup stack it offers a much lighter-weight approach to heap-based memory management than standard C++ exceptions, and reduces the risk of memory leaks due to coding errors.

These advantages outweigh the disadvantage that it is non-standard, and therefore represents a barrier to entry for new developers.

7 Related Mechanisms

7.1 Assertions and panics

Assertions are fatal errors that are not propagated but cause an immediate halt at the error location. Symbian OS provides the macros `__ASSERT_DEBUG`, `ASSERT` and `__ASSERT_ALWAYS`. Assertions are probably the most effective way to check for programming errors in user code, by forcing the program to fail at the point of the error. Typically they are used to define pre- and post-

conditions for code blocks to catch 'impossible' conditions in your code, and other errors that 'can't happen' (Hanson, p.59). When executed, the assertion macros halt a program and display diagnostic information to show which condition failed.

The appropriate response to an assertion is to debug your code and to identify and fix the error, rather than to remove the assertion.

Assertions are implemented using panics. Panics are raised with a category and a panic number; the category identifies the system component which has raised the panic, and the number identifies the reason for the panic. An un-trapped exception in user code will also be converted into a panic when the system is unable to find a handler.

See the Panic reference in the Symbian Developer Library for the lists of categories and reasons.

Symbian OS also provides the ability to log panics to file for later analysis, as described in the Symbian Developer Library.

8 Wrapping Up

Work began on the precursor to Symbian OS some years before the ISO standard mechanism was standardized. Vanilla C++ provided only unstructured `setjmp/longjmp` methods (respectively, instantiate the handler and raise the exception), in effect a non-local `goto`. Any additional support for exception handling, and in fact even the `setjmp/longjmp` implementation, was incompatible between different vendors. Even when the C++ standard did emerge, it was inconsistently supported by different vendors; for example, as late as mid-2003 Eric Raymond could still bemoan the fact that no compiler yet fully supported the C++99 ISO standard for the language, though GCC C++ came closest (Raymond, 2004, p.410).

Stanley Lippman provides interesting compiler comparison data on the code size and speed impacts when exception handling is turned on (regardless of whether it is ever activated). He also tells some interesting anecdotes. The original C-generating `cfront` implementation of C++ was, he claims, killed off because the problems of implementing exception handling in a universal and portable way were perceived to be insurmountable. And then there is the story of Junior, a chess program that grew so much when recompiled (the night before a tournament) with a new exception handling C++ compiler that it would no longer fit on its host machine. All's well that ends well; reunited with an old compiler, Junior went on to tie with IBM's Deep Blue.

When standard C++ exceptions were first switched on in Symbian OS, the ROM image size jumped immediately by ten percent. While that may not seem intolerable for current generations of high end phones, bill of materials costs and performance overheads remain critical success factors in the mobile market, and much work was required to bring the image size back down again.

The drive to support standard C++ exceptions in Symbian OS has therefore mostly come from the desire to lower barriers to entry for developers with C++ but not Symbian OS specific skills, and to open the platform to the large codebase of existing, standard C++ programs.

Those goals are particularly relevant in the enterprise market, with its legacy base of in-house software, and for operators looking to roll out phone-based platforms across all devices on their networks. But they hold equally in the jostling marketplace of third-party developed applications. Standardization is compelling.

9 Reading

Stichbury (2004, Chapter 2) includes best practice and guidance, including example code for using the Symbian OS leave/trap framework and the cleanup stack.

For a pre-v9 perspective on porting to Symbian OS, see the (archived) Simkin discussion paper on the Symbian Developer Network at developer.symbian.com/main/oslibrary/archived_papers.

The technical paper by Jason Morley (2007) is an excellent summary of the new exception framework implementation.

Many of the official Symbian (including Nokia, UIQ, and Sony Ericsson) developer forums contain useful discussions of the v9 exception handling options.

For the standard C++ perspective, Stanley Lippman describes C++ exception handling in some detail in his book, *Inside the C++ Object Model* (1996, Chapter 7). Koenig and Moo (1996, p.284) describe what happens when operator new fails.

Smart pointers are described in Meyers (1996). For a Symbian OS slant, see the paper by Sander van der Wal (2002).

If you like your discussions classical, Kernighan and Pike (1999) contains an excellent general discussion of errors and exceptions.

For an interesting discussion of the merits of pre- and post-condition checking using assertions, see the article by Jezequel and Meyer (1997) on the analysis of the Ariane rocket failure at the Eiffel language website.

The survey article by Ryder and Soffa (2003) provides interesting historical background on languages and exceptions.

9.1 References

Hanson, David R. (1997) *C Interfaces and Implementations*, Addison-Wesley

Harrison, Richard. (2003) [Symbian C++ for Mobile Phones Volume 1](#), John Wiley/Symbian Press

Harrison, Richard and Shackman, Mark. (2007) [Symbian C++ for Mobile Phones Volume 3](#), John Wiley/Symbian Press

Jezequel and Meyer. (1997) *Design by Contract: The Lessons of Ariane*

Kernighan, Brian and Pike, Robert. (1999) *The Practice of Programming*, Addison-Wesley

Koenig, Andrew and Moo, Barbara. (1996) Andrew Koenig and Barbara Moo, *Ruminations on C++*, Addison-Wesley

Lippman, Stanley B. (1996) *Inside the C++ Object Model*, Addison-Wesley

Meyers, Scott. (1996) *More Effective C++*, Addison-Wesley

Morley, Jason. (1997) *Leaves and Exceptions*, at developer.symbian.com/main/downloads/papers/Leaves%20and%20Exceptions.pdf

Pagonis, John. (2006) Presentation, Wireless Developer Forum, Cambridge UK, December 2006

Raymond, Eric S. (2004) Eric S Raymond, *The Art of Unix Programming*, Addison-Wesley

Ryder, Barbara G. and Soffa, Mary Lou. (2003) *Influences on the Design of Exception Handling*, SIG, www.sigsoft.org/impact/docs/p29-ryder.pdf

Stichbury, Jo. (2004) [Symbian OS Explained](#), John Wiley/Symbian Press

Stroustrup, Bjarne. (1994) *The Design and Evolution of C++*, Addison-Wesley

van der Wal, Sander. (2002) *Creating the C++ auto_ptr<> utility for Symbian OS*, at developer.symbian.com/main/downloads/papers/auto_ptr/auto_ptr.pdf

9.2 Web resources

The Symbian Developer Network, or SDN, is at developer.symbian.com.



10 Author Profile

Ben Morris freelances as a writer and software architect specializing in Symbian OS. He is the author of *The Symbian OS Architecture Sourcebook: Design and Evolution of a Mobile Phone OS*, published by Symbian Press. He can be contacted through www.wordmatter.co.uk.

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.