Archived:EUserHL Core Idiom Library

Archived: This article is **archived** because it is not considered relevant for third-party developers creating commercial solutions today. If you think this article is still relevant, let us know by adding the template {{ReviewForRemovalFromArchive|user=~~~~|*write your reason here*}}.

Core Idioms libraries deliver improved APIs to reduce the number of coding activities associated with a given task, and where possible make core Symbian idioms more accessible to non-Symbian programmers. The initial delivery provided simpler mechanisms for string handling and semi-automated memory management - see Archived:An Introduction to L Classes for further documentation about the `LString` class. Core Idioms consists of a the EUserHL.dll along with supporting header files. These Core Idiom classes are included in the Symbian^3 platform, including the Symbian^3 targets in the Qt SDK.

EUserHL can also be used in earlier phones/SDKs running Symbian OS v9.x (including Symbian^1 and Symbian^2). The File:EUserHL.zip package delivers the header, and an installation file that is compatible with all phones running - see #Issues with Embedding EUserHL.sis in an Application for associated information.

Overview

The new `EUserHL` Core Idioms library delivers:

- `LString`, a string class that handles its own buffer management and cleanup and whose parameters take natural C++ string literals
- `LCleanedupX` and `LManagedX`, a set of cleanup management helper templates
- `CONSTRUCTORS_MAY_LEAVE`, a helper macro that enables single-phase construction
- `OR_LEAVE`, a helper macro to cleanly convert error-returning code into leaving code.

The necessary header files for exploiting the new code idioms library are supplied in a header `EUserHL.h`, supported by a DLL, `EUserHL.DLL`, and an `EUserHL.SIS` installable package, available for all Symbian OS v9-based devices.

This makes Symbian platform easier for programmers by:

- Making it much easier and cleaner to write correct cleanup-safe code, with fewer lines of code than before.
- Making an elegant, leave-safe implementation of the widely-used C++ RAII [1] idiom available for Symbian C++ programming.
- Making it much easier and cleaner to write code involving arbitrary-length strings without choosing magic numbers for their length, and variable-length strings without performing manual memory management.

Using the Core Idioms library has a pervasive impact on line-of-code count and on simplicity and cleanness. That's great when you write the code, and awesome when you come to maintain it.

The Core Idioms library delivers these improvements by exploiting the mapping of Symbian Platform `User::Leave()` onto C++ throw, introduced in Symbian Platform v9 and by relieving the application programmer of much explicit responsibility for memory management for strings and cleanup.

Idioms define the style by which programmers use an OS, and therefore have a pervasive ease-of-use impact in normal Symbian programming by programmers working at all levels of the software stack.

Use of this library is recommended for all new code. Only if you know you can do better by managing your own memory with traditional descriptor APIs, cleanup stack idioms, and two-phase construction, should you continue to use the traditional Symbian C++ features instead of Core Idioms. Before and after

Here's how the Core Idioms library makes a difference in real-life situations:

Without Core Idioms Library	With Core Idioms Library

`{`
`TBuf<KMaxQuery> query; // fixed worst-`
`// case max`
`query.Format(KQueryFormat, param);`
`ExecuteQueryL(query);`
`}`
	`{`
	`LString query;  // can grow its heap`
	`// buffer on demand`
	`query.FormatL(KQueryFormat, param);`
	`ExecuteQueryL(query);`
	`} // query buffer released on`
	`// normal scope exit or leave`

(C) Copyright Nokia 2014. All rights reserved.

{
HBufC* queryc =
 	
HBufC::NewLC(KTooBigForStackMaxQuery);
TPtr query(queryc->Des());
BuildQueryL(query);
ExecuteQueryL(query);
CleanupStack::PopAndDestroy();
}
	{
	LString query;
	BuildQueryL(query);
	ExecuteQueryL(query);
	}

{
RBuf query;
query.CleanupClosePushL();
query.CreateL(TooBigForStackMaxQuery);
BuildQueryL(query);
ExecuteQueryL(query);
CleanupStack::PopAndDestroy();
}
	{
	LString query;
	BuildQueryL(query);
	ExecuteQueryL(query);
	}

{
CQuery* query = CQuery::NewL();
CleanupStack::PushL(query);
query->BuildQueryL();
query->ExecuteQueryL();
CleanupStack::PopAndDestroy();
}
	{
	LCleanedupPtr<CQuery>
	query(CQuery::NewL());
	query->BuildQueryL();
	query->ExecuteQueryL();
	} // query deleted on normal scope
	// exit or leave

{
CQuery* query = CQuery::NewL();
CleanupStack::PushL(query);
query.BuildQueryL();
CleanupStack::Pop();
return query;
}
	{
	LCleanedupPtr<CQuery>
	query(CQuery::NewL());
	query->BuildQueryL();
	return query.Unmanage();
	// query was protected until Unmanage()
	} // was called

{
RQuery query;
CleanupClosePushL(query);
query.BuildQueryL();
query.ExecuteQueryL();
CleanupStack::PopAndDestroy();
}
	{
	LCleanedupHandle<RQuery>;
	query->BuildQueryL();
	query->ExecuteQueryL();
	} // query is closed on normal scope
	// exit or leave

As this shows, the code with Core Idioms is usually one or two lines shorter than pre-Core Idioms code, and is never longer.

Cleanup stack operations don't appear explicitly in the Core Idioms code, even though the cleanup stack is in fact being used, and all of the code shown here – both with and without Core Idioms – is totally leave-safe.

(C) Copyright Nokia 2014. All rights reserved.

The code without Core Idioms often uses more memory than is really necessary, either on the stack or the heap, and the only way to avoid this is to code awkward extra lines of code. The code with Core Idioms uses only as much space as necessary and is brief.

If you want to use literals with `LString`, you can just use wide literals from pure C++ (rather than specifying string literals with `_LIT`). This featured was introduced in the Core Idioms v1.2 zip file attached.

Core Idioms Prior to v1.2	Core Idioms v1.2

`{`
`// bunch of literal descriptors`
`_LIT(KStringOne, "One ");`
`_LIT(KStringTwo, "Two ");`
`_LIT(KStringThree, "Three");`
`// somewhat later, use those literals`
`LString s;`
`s = KStringOne;`
`s.AppendL(KStringTwo);`
`s += KStringThree;`
`}`
	`{`
	`// just use wide literals from pure C++`
	`LString s;`
	`s = L"One ";`
	`s.AppendL(L"Two ");`
	`s += L"Three";`
	`}`

This feature exploits the fact that expressions such as L"Three" produce a `wchar_t[]`, and that in all ABIs of interest to Symbian platform, `wchar_t` is a 16-bit wide character, compatible with `LString16`.

The L prefix

The L prefix denotes that construction, copying, passing and returning by value, assignment and manipulation via operators should all be considered potentially leaving operations unless otherwise explicitly documented. So, this means that all the code that manipulates L instances **MUST** be leave-safe. To be able to create leave-safe code the programmer will need to set up the cleanup stack the calling thread and use the L class within the scope of a TRAP statement. The main characteristics of L classes are:

- Constructors, operators and implicit operations may leave
- They are self-managing and do not require auxiliary memory management logic

Also, they are as simple to use as T classes. So they can be used as built-in types. The main difference is that they own resources that are managed automatically, so the programmer doesn't need to care about then.

Using Core Idioms

The `EUserHL` Core Idioms library provides new APIs for simpler, exception-safe programming, a new string class and single-phase object construction.

Self-managing string class

The `LString` class provides a self managing, resizable descriptor that bridges the gap between the behavior of standard C++ strings, such as std::string, and Symbian descriptors. `LString` is recommended instead of traditional `TBuf`, `HBufC` and `RBuf` descriptors. `LStrings` can be passed into Symbian platform APIs which take `TDes&` and `const TDesC&` parameters.

The Symbian C++ descriptors have been a defining attribute of Symbian platform. The descriptor family was designed with reliability and space efficiency as primary goals. This was achieved by spreading common string behavior across a number of descriptor classes. Unfortunately, this decision has a usability cost due to the number of classes that must be directly understood by developers to achieve common objectives with strings. `LString` is intended to provide a self-managing alternative to several of the standard descriptor types.

An `LString`may be used much like a simple T class. `LString`-typed variables will automatically clean up after themselves when

(C) Copyright Nokia 2014. All rights reserved.

they go out of scope, and `LString`-typed fields will similarly automatically release all owned resources when their containing object is destroyed.

In addition to the value-type semantics described above, `LString` also supports automatic in-place resizing. All standard descriptor methods are available but for any non-leaving descriptor method that may panic due to buffer overflow, `LString` adds a corresponding leaving method that automatically expands the underlying buffer on-demand. For example, `Append()` will panic if the new data overflows available buffer space, while `ppendL()` will attempt to `realloc()` the buffer as necessary. The new leaving variants may therefore leave with `KErrNoMemory`, may invalidate any existing raw pointers into the data buffer (e.g., those previously returned by `Ptr()`), and may change the value returned by `MaxLength()`. To protect developers from inadvertently calling the non-leaving methods on `LString`, these have been privatized.

`LString` is compatible with existing APIs that accept const `TDesC&` and `TDes&` arguments. Both 8 and 16 bit versions are provided.

Finally, to allow `LString` to be allocated on the heap and deleted, the class defines implementations for various overloads of the operator `new()` and operator `delete()` methods.

For more information about the new string class, see Archived:An Introduction to L Classes.

Class templates to automate cleanup

The `LCleanedupX` and `LManagedX` class templates from Core Idioms are Symbian-specific analogues to C++ smart pointers:

- The `LCleanedupX` templates allow leave-safe objects to be declared, which are placed automatically on the Symbian C++ cleanup stack. These objects are then cleaned up automatically, whether processing terminates normally or leaves.
- The `LManagedX` templates are intended for exception-safe objects to be declared as members of classes. These objects are then cleaned up automatically, whether processing terminates normally or leaves. A key benefit is that this permits single-phase construction of CBase-derived objects.

The `LCleanedupX` and `LManagedX` class templates are:

LCleanupX template	LManagedX template	What it manages	how it cleas up by default
`LCleanedupPtr`	`LManagedPtr`	a pointer	deletes the pointer
`LCleandupRef`	`LManagedRef`	a handle, by reference	calls `Close()` on the handle
`LCleanedupArray`	`LManagedArray`	a C array	deletes the array
`LCleandupGuard`	`LManagedGuard`	anthing protected with a `TCleanupItem`	calls `TCleanupItem`'s cleanup operation

The cleanup strategy can be changed to one of the predefined cleanup strategies provided, or to a user-defined custom cleanup strategy. For more information about the cleanup management helper templates, see Archived:An Introduction to L Classes.

LCleanedupX templates

Use `LCleanedUpX` templates instead of the classic `CleanupStack::PushL()` and `CleanupStack::PopAndDestroy()` approach to write leave-safe code more succinctly and elegantly than was previously possible, as shown in the before-and-after examples above. Using:

```
{
LCleanedupPtr<CQuery> query(CQuery::NewL());
// ...
if (condition)
    return;
// ...
}
```

has a similar behaviour to:

```
{
CQuery* query =  CQuery::NewL();
CleanupStack::PushL(query);
//...
if (condition)
```


(C) Copyright Nokia 2014. All rights reserved.

```
    {
    // Pop and destroy the object (also deletes the pointer)
    CleanupStack::PopAndDestroy(query);
    return;
    }
// ...
// Pop and destroy the object (also deletes the pointer)
CleanupStack::PopAndDestroy(query);
}
```

LManagedX classes

Use `LManagedX` templates for member variables in classes which implement single-phase construction. See the example below.

Do not use `LManagedX` for any other purpose, without being aware of the differences between C++ exception-handling semantics and Symbian C++ leave semantics: [2]

- `LManagedX` templates, and conventional C++ exception safety, rely on standard unwinding of the C++ program stack when an exception, including a Symbian C++ leave, occurs. This unwinding of the program stack causes destructors to be invoked in a sequence specified by the C++ language.
- `LCleanedupX` templates, and classic Symbian C++ cleanup stack operations, rely on the unwinding of the Symbian C++ cleanup stack when a `User::Leave()` occurs. Although `User::Leave()` generates a C++ exception, the cleanup stack is not unwound when a C++ exception is thrown any other way. Also, `User::Leave()` processing unwinds the Symbian C++ cleanup stack first, and then throws the C++ exception which unwinds the C++ program stack.

These differences can create counter-intuitive effects if `LManagedX` objects are used other than as recommended.

Single-phase construction for CBase-derived classes

The Core Idioms library supports single-phase construction of CBase-derived classes. To implement single-phase construction,

- specify the `CONSTRUCTORS_MAY_LEAVE` macro in your class declaration
- encapsulate members which must be cleaned up if a constructor leaves, in `LManagedX` templates

Here's how:

```
#include <e32std.h>
#include <f32file.h>
#include <euserhl.h> // Core Idioms

class CFinder : public CBase
    {
public:
    // We have opted to use single-phase construction here, and some of
    // our constructor's initialization actions may leave. In order to
    // guarantee full cleanup in all cases,  we have to declare this fact.
    CONSTRUCTORS_MAY_LEAVE
    static CFinder* NewL(const TDesC&  aPattern);
    ~CFinder();
    void GetNextMatchL(TDes& aMatch);
protected:
    CFinder(const TDesC& aPattern);
protected:
    LString iPattern; // looks after its own cleanup
    LManagedHandle<RFs> iFs; // will be closed as required
    // ...
    };

CFinder* CFinder::NewL(const TDesC& aPattern)
    {
```


(C) Copyright Nokia 2014. All rights reserved.

```
    return new(ELeave) CFinder(aPattern);
    }

CFinder::CFinder(const TDesC&  aPattern)
    // This initializer may leave, since the LString will allocate a
    // heap buffer large enough to contain a copy of aPattern's data
    : iPattern(aPattern)
    {
    // If connection fails and we leave here, iPattern's destructor
    // will be called automatically, and the string's resources will
    // be released
    iFs->Connect() OR_LEAVE;
    }

CFinder::~CFinder()
    {
    // Automatic destruction of each of the data members does all
    // of the work for us: iPattern's heap buffer if freed, while
    // Close() is called on the managed RFs in iFs.
    // Even though this destructor is textually empty, it should
    // still be exported; the compiler is generating destruction
    // logic for us in this case
    }
```

Compared to two-phase construction, `NewL()` is not implemented in terms of `NewLC()` followed by `CleanupStack::Pop()`; and there's no `ConstructL()`.

In fact, the `NewL()` function really isn't necessary at all. Its sole benefit is the possibility to change the implementation to old-fashioned two-phase construction without breaking compatibility. Without a `NewL()`, the implementation would be even briefer and clients would just call `new(ELeave) CFinder(pattern)`.

Note: Note the use of the pointer operator rather than the member-selection operator. If `iFs` had been an unmanaged RFs, we'd have used `iFs.Connect()`. But since `iFs` is an `LManagedHandle<RFs>`, we use `iFs->Connect()`. Macro to cleanly convert error-returning code into leaving code.

In the example above, the line

```
        iFs->Connect() OR_LEAVE;
```

shows the Core Idioms `OR_LEAVE` macro, which uses C++ operator overloading to generate code which is exactly equivalent to

```
        User::LeaveIfError(iFs->Connect());
```

With `OR_LEAVE`, it's just as efficient, and much more readable.

Roadmap

The Core Idioms will become part of the platform in Symbian^3. The sections below list the work that has been done so far, in order of most recent first.

Note that the current File:EUserHL.zip is #EUserHL 1.2.

Backlog

Available from Symbian^3.

EUserHL 2.0 (Not yet available)

(C) Copyright Nokia 2014. All rights reserved.

feature	use case	before	after
active callbacks	single async request	10 coding activities, one new class	5 coding activities
	adding a time-out	7 coding activities	1 coding activity
	single activity under active scheduler	3 coding activities	1 coding activity
	long-running active object	4 coding activities	2 coding activities
	time one-shot task	3 coding activities	1 coding activity

EUserHL 1.2 (Current Deliverable - June 2009)

feature	use case	before	after
natural literals	string literals in LString API	2 coding activities	1 coding activity

EUserHL 1.2 includes an updated version of EUserHL.sis that supports backup/restore and which installs without any warnings.

EUserHL 1.1 (Superseded)

- EUserHL 1.1 fixes a problem with a header file present in EUserHL 1.0 and corrects the LString class's documentation.
- EUserHL 1.1 uses EUserHL.sis 1.0 (the same version EUserHL 1.0 uses), with the same bugs.

EUserHL 1.0 (Superseded)

feature	use case	before	after
LString	most low-level string use cases	3-5 lines of code	2-3 lines of code
LCleanedupX	stack-based variables needing cleanup	3 coding activities	2 coding activities
single-phase construction	boilerplate for C class with leaving construction	4 coding activities	1 coding activity
OR_LEAVE macro	converting error codes into leaves	1 opaque line of code	1 transparent line of code

Known issues:

- Documentation for the LString class is incorrect.
- Header file issue that will cause some applications to fail to build.
- EUserHL.sis 1.0 does not support backup/restore.
- EUserHL.sis 1.0 is missing the Machine ID, so the user will get a "Application is not compatible with the phone" warning during installation.

Issues with Embedding EUserHL.sis in an Application

Applications that use the LString class will have a dependency on `EUserHL.dll`, which must be installed by embedding the `EUserHL.sis` package in the application's SIS package. The package UID of EUserHL, required for embedding the sis file with your application, is: **0x2001B440**

The Nokia Developer article Archived:The dependency option is not supported with embedded SIS packages (Known Issue) is also important for applications that need to support S60 3rd Edition and S60 3rd Edition FP1. Applications that need to support those devices may want to avoid depending on EUserHL.dll at all, to avoid this issue.

Note that the issue described in Archived:The dependency option is not supported with embedded SIS packages (Known Issue) is not specific to EUserHL. It applies to any embedded package that is delivered by multiple applications and/or multiple vendors. TODO: There should be a wiki page here that discusses these embedded installation issues in more detail and explains how to deal with them effectively.

As noted above, EUserHL.sis 1.0 (shipped in EUserHL 1.0 and EUserHL 1.1) does not support backup/restore and causes warnings during installation. Applications should embed EUserHL.sis 1.2 or later to avoid these two issues.

Conclusion

The EUserHL Core Idioms library allows:

- Experienced Symbian C++ developers to write robust and compact string-handling code with semi-automated exception handling.
- New Symbian C++ developers to use Symbian OS exceptions, the cleanup stack and descriptors more easily, with fewer programming errors and more rapid application development.

References

1. ↑ Resource Acquisition Is Initialization: a design pattern popular in many object oriented languages, which combines the

(C) Copyright Nokia 2014. All rights reserved.

acquisition and release of resources with initialization and uninitialization of objects.

2. ↑ For further details, see A Comparison of Leaves and Exceptions or the Application Reference Guide.

© 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.