Archived:An Introduction to L Classes

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*}}.

This article provides an introduction to the L-classes, string classes which are simpler to use and which allow you to be more **L**iberal when it comes to Leaving.

Introduction

The EUserHL High Level library introduces a new L class idiom. The 'L class' prefix provides a clear indication of the new leaving behavior of these classes, in order to distinguish them from existing Symbian classes and improve usability and understanding. The L prefix denotes that these classes may be Liberal when it comes to Leaving, while being Leave-safe themselves.

The `LString` class is intended to provide a self-managing, resizable descriptor which bridges the gap between the behavior of standard C++ strings such as `std::string` and descriptors. Symbian C++ descriptors have become a defining attribute of Symbian Platform itself, with the descriptor family being designed with reliability and space efficiency as the primary quality goals. This was achieved by spreading common string behavior across a number of descriptor classes.

Unfortunately, this decision presents a high 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.

`EUserHL` also provides a collection of Symbian C++ class templates, `LCleanedupX` and `LManagedX`, that provide automatic resource management. The technique used is called Resource Acquisition Is Initialization RAII. These are relatively easy to use and provide automatic exception-safe cleanup upon normal or exceptional exit from a scope, in which case you do not need to write any code for invoking the cleanup of resources: the resources are automatically cleaned up by the destructors of the class templates in that case.

Usage

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. Code that uses L classes should be written accordingly, in leave-safe style. This implies that a cleanup stack has been set up in the calling thread and that the L class is used within the scope of a `TRAP` statement.

L classes have characteristics close to standard C++ value and handle classes, including the following:

- constructors, operators and implicit operations may leave
- they are self-managing and do not require auxiliary memory management logic.

L classes may be used like T classes except that they can own resources, they typically rely on guaranteed execution of their destructors, and they cannot necessarily be naively bitwise copied. Any code that manipulates L class instances must be leave-safe. The only exception to this rule is if the particular operations being used are explicitly documented not to leave (for example, the default `LString` constructor and the `LManagedX` constructors).

The example code used below is taken from `euserhl_walkthrough.cpp`, which accompanies this document. It can be built and run via the `bld.inf` file found in the group subdirectory.

Table 1: Exported Files

Header File	Exported To	Exported APIs
`euserhl.h`	`/epoc32/include/`	No APIs; includes all other headers

`estromg.h`	`/epoc32/include/`	`LString16`
		`LString8`
		`typedef LString`
		`typef LData`

`emanaged.h`	`/epoc32/include/`	`LCleanedupHandle`
		`LCleanedupPtr`
		`LCleanedupRef`
		`LCleanedupArray`
		`LCleanedupGuard`
		`LManagedHandle`
		`LManagedPtr`
		`LManagedRef`
		`LManagedArray`
		`LManagedGuard`
		`DEFINE_CLEANUP_FUNCTION macro`
		`DEFINE_CLEANUP_STRATEGY macro`
		`CONSTRUCTORS_MAY_LEAVE macro`

`emisc.h`	`/epoc32/include`	`OR_LEAVE macro`

`typerel.h`
`swp.h`
`isbaseopf.h`
`issame`
	`/epoc32/include/`	No APIs; support files

LString

Introduction

`LString` is a convenient, general-purpose string class derived from `RBuf`. `LString` adds automatic cleanup and on-demand buffer resize facilities. Like an `RBuf`, an `LString` can be passed to any function that is prototyped to take a `TDes` or a `TDesC` reference. Again like an `RBuf`, an `LString` maintains its string data in a heap buffer.

(C) Copyright Nokia 2013. All rights reserved.

Anatomy of LString

LString inherits from RBuf and so provides all of the expected descriptor behavior, but adds self-management and automatic cleanup.

Unlike RBuf, an LString may be used much like a simple T class: LString type variables will automatically clean up after themselves when they go out of scope, and LString type member variables 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. LString replaces standard descriptor APIs with a corresponding leaving method that automatically expands the underlying buffer on-demand. For example, AppendL() will attempt to reallocate the buffer as necessary to fit the new data. The new leaving variants may therefore leave with KErrNoMemory. As well as the leaving API, LString constructors and operators may also leave.

Constructors, operators and methods of LString can all potentially result in creation or re-allocation of the underlying data buffer. This may invalidate any existing raw pointers into the data buffer (for example, those previously returned by Ptr()), and may change the value returned by MaxLength(). If no resources are available for the creation or re-allocation, these constructors, operators and methods can leave with KErrNoMemory.

LString only supports automatic growth when being manipulated directly as an LString. When an LString is passed to a function accepting a TDes, that function will operate on it as if it is a fixed-max-length descriptor. In that case, adequate capacity must be reserved within the LString prior to calling the function. This can be achieved using the appropriate constructor or by calling ReserveFreeCapacityL(), which has similar semantics to the now unsupported method EnsureCapacityL().

LString doesn't support automatic compression, as it is too difficult to predict the future use of an LString object and therefore auto-compression would likely result in further allocations and be inefficient. If compression is desired, this must be done manually via LString::Compress().

Using inherited non-leaving RBuf and TDes methods on an LString

LString derives from RBuf in order to achieve maximum interoperability with existing descriptor-accepting APIs. This derivation forces some unusual API compromises, however, due to the unique characteristics of LString compared with other descriptors. Some of the mutating methods on the base classes, TDes and RBuf8, panic when called with insufficient buffer space. Sufficient space is a precondition of these base classes; LString relaxes this precondition with its capability to start with zero length. LString defines new leaving variants of these methods with auto-growth behavior (for example, AppendL()) but at the same time inherits the original methods (for example, Append()). This makes it too easy for developers to call the wrong method inadvertently. In order to address this, the original non-leaving methods have been made private in LString. Developers should use the leaving LString versions.

Hiding these methods does not remove the problem entirely. The same problem can happen if an LString object of insufficient size is passed into any API accepting a TDes. The advice is that developers should always ensure there is sufficient space before passing LString as a TDes.

LString API

As LString is derived from RBuf its API is broadly equivalent, but any method that panics on buffer overflow is replaced with a leaving variant which attempts to reallocate the underlying buffer and leaves if resources are unavailable. The basic API is not described here but is based on the RBuf API described in the Fundamentals of Symbian C++/Descriptors. In addition to the standard RBuf API, LString adds the following:

SetMaxLengthL()

```
void LStringX::SetMaxLengthL(TInt aMaxLength)
```

This method sets the storage space allocated to this descriptor to the specified value by growing or compressing its buffer size. If the current length of the descriptor is greater than the specified max length, length is truncated to this max length.

ReserveFreeCapacityL()

```
void LStringX::ReserveFreeCapacityL(TInt aExtraSpaceRequired)
```

This method ensures that the remaining unused space is more than the supplied value. It may reallocate a larger storage space to meet the requirement. As a result, MaxLength() and Ptr() may return different values afterwards and any existing raw pointers into the descriptor data may be invalidated.

Typically, you use this method to reserve a known amount of required space in one go instead of relying on the automatic growth pattern.

Compress()

```
void LStringX::Compress()
```

This method reallocates a smaller descriptor buffer space to the current descriptor length. This may cause the string descriptor's heap buffer to be reallocated in order to accommodate the new data. As a result, MaxLength() and Ptr() may return different values afterwards, and any existing raw pointers into the descriptor data may be invalidated.

If there is insufficient memory to reallocate the buffer, then the descriptor is left unchanged.

Reset()

```
void LStringX::Reset()
```

This method re-initializes the descriptor, destroying its contents.

Basic usage patterns

Construction

LString provides a variety of construction options, as shown below:

```
{
// A default constructed LString starts empty, doesn't allocate any memory
// on the heap, and therefore the following cannot leave
LString s;
```


(C) Copyright Nokia 2013. All rights reserved.

```
// You can initialize with a MaxLength value
LString s(KMaxFileName); // This operation may leave

// You can initialize from any descriptor (or literal) and the
// string data is copied into the LString
LString s(L"One"); // From a literal

LString half(s.Left(s.Length() / 2)); // Left returns a TPtrC

// On the other hand, you can initialize from a returned
// HBufC* and the LString automatically takes ownership
LString own(AllocateNameL(L"Testing"));

// LStrings can be allocated on the heap if necessary
LString* s = new(ELeave) LString;
}
```

`LString` can also be used as class member variables, declared in the class declaration and instantiated in `ConstructL()`, in the constructor or in the constructor initialization list.

Manual size management

Although `LString` supports automatic resizing, it is also possible and sometimes necessary to manually change the length of the string. When an `LString` is passed to a function as a `TDes` or `TDesC`, it loses its automatic resizing capabilities and therefore care must be taken to ensure sufficient space is allocated before the call is made.

Extra space can be allocated to the `LString` by calling `ReserveFreeCapacityL()` and the string can be compressed to the minimum length by calling `Compress()`. An example is given below:

```
// Reserve extra free space in preparation for an operation that adds
// characters to the string. You may need to do this when you cannot use
// any of the auto-buffer extending LString methods to achieve your objective.
LString s(L"One");
s.ReserveFreeCapacityL(4);
ASSERT(s.Length() == 4);
ASSERT(s.MaxLength() >= 8);
```

Almost all the methods that may extend the string buffer, including the explicit `ReserveFreeCapacityL()` but excluding `SetMaxLengthL()`, attempt to grow the buffer size exponentially. The exponential growth pattern is expected to give better performance at an amortized complexity of O(n) when adding n characters. If the exponential growth is less than the supplied extra size, then the supplied size is used instead to save time. The exponential growth is used in anticipation of further additions to a string. This trades off speed efficiency for space efficiency. If required, you may be able to swap the oversized buffer for a more compact one, using:

```
s.Compress();
ASSERT (s.MaxLength() >= 4); // note indefinite test
```

Compress attempts to re-allocate a smaller buffer to copy the content into. If the new memory cannot be allocated, then the original string is left unaffected.

When you have finished using the content of a string, you can get its buffer released without destroying the string itself. You may want to do this when using member-declared strings. Automatic strings are destroyed when they go out of scope.

```
s.Reset();
ASSERT(s.Length() == 0);
ASSERT(s.MaxLength() == 0);
```

Passing an LString to a function expecting a TDes& or TDesC&

Because `LString` is derived from `RBuf`, it can be passed directly to any method accepting a `TDes` or `TDesC` parameter. Be aware, though, that once passed as a `TDes` `LString` loses its automatic resizing capability. Care must be taken to ensure that sufficient capacity is reserved by calling `ReserveFreeCapacityL()` or `SetMaxLengthL()` before passing the `LString` as a `TDes`. When passed to a function as a `TDes`, a `USER 11` panic will be raised if the string is modified and the resulting length of this descriptor is greater than its maximum length. An example is given below:

```
void GetCurrentPath(TDes& aDes)
{
aDes = KPath;
}
. . .

{
LString s;
// Calling GetCurrentPath(s) now would panic because LStrings
// are initialized by default to MaxLength 0. Although s is
// an LString, GetCurrentPath takes a TDes& and so inside the function
// s behaves as a TDes and would panic with USER 11 if the resulting
// new length of s is greater than its maximum length.
ASSERT(s.MaxLength() == 0);

// Calling SetMaxLengthL will automatically realloc the
// underlying buffer if required, and is guaranteed to leave
// MaxLength() equal to the specified value
s.SetMaxLengthL(KMaxFileName);
GetCurrentPath(s);
}
```

Note: Note that `LString` instances can be passed and returned by value, but doing so may trigger implicit heap allocation and cause a leave with `KErrNoMemory`. As with other descriptors, passing by reference when possible is more efficient.

Using standard C++ string literals with LString

From the version 1.2 release of `EUserHL`, standard C++ string literals can be used with `LString`, as shown below.

```
{
// A default constructed LString starts empty, doesn't allocate any memory on the
// heap, and therefore the following cannot leave
```


(C) Copyright Nokia 2013. All rights reserved.

```
LString s;

// But it will grow on demand if you assign to it, so it has enough space to hold
// the copied string data, and so assignment may leave
s = L"One ";

// Similarly if you append to it with the leaving variant of Append, AppendL, if
// may grow on demand
s.AppendL(L"Two ");

// The += operator for LString also maps to AppendL
s += L"Three ";
}

...

RTest test(_L("EuserHl Walkthrough"));

{
// An LString can be printed the same way as any descriptor
test.Printf(_L("Value: %S\n"), &s);

// An LString supports all TDesC and TDes methods
// LString findToken(L"Two ");
test(s.Find(L"Two ") == 4);

// LString matchPattern(L"*Two* ");
test(s.Match(L"*Two*") == 4);
test (s.Match(L"*T?o*") == 4);

// LString compare(L"some string");
test(s.Compare(L"One Two Three Testing ") == 0);
test(s.Compare(L"One Two Three Testing! ") < 0);
test(s.Compare(L"One Two Testing ") > 0);

// also LString ==,!=,>,<,<=,>=(L"some string");
test(s == L"One Two Three Testing ");
test(s < L"One Two Three Testing! ");
test(s > L"One Two Testing ");
test(s != L"not equal");
}

...

{
// 8-bit wide null terminated character string support

// A default constructed LString8 starts empty, doesn't allocate any memory on the
// heap, and therefore the following cannot leave
LString8 s;

// But it will grow on demand if you assign to it, so it has enough space to hold
// the copied string data, and so assignment may leave
s ="One ";

// Similarly if you append to it with the leaving variant of
// Append, AppendL, if may grow on demand
s.AppendL("Two ");

// The += operator for LString8 also maps to AppendL
s +="Three ";
s +="Testing ";

// An LString8 can be printed the same way as any descriptor
test.Printf(_L("Value: %S \n"), &s);
}
```

Other issues

Pushing an `LString` onto the cleanup stack is not necessary or recommended, but the effects of doing so are benign.

Whilst being simpler to use than existing descriptors in many cases, `LString`'s use of heap allocation and its resizing variant methods clearly come with associated costs. Their use in performance-critical code should be carefully considered. On the other hand, `LString`'s small stack footprint and ability to better handle inputs of unpredictable size may make them a better choice when the alternative is a large, fixed-max-size `TBuf` or `HBufC`.

LManagedX Classes

Introduction

The `LManagedX` classes provide a means of automatically cleaning up member variables when the containing object is destroyed. Variants are provided for cleaning up pointers, references and handles, as well as generic cleanup items. The user is able to define a cleanup strategy to clean up the resource.

Anatomy of LManagedX

The `LManagedX` automatic resource management classes were added to make it easier for Symbian C++ developers to write robust code. They are designed to reduce the number of lines of required cleanup member variables in the owning classes' destructor. Code quality is increased by improved readability and having fewer lines of code.

Table 2: `LManagedX` classes and their uses

Class ! Use	Default Cleanup Action	
`LManagedPTR`	Automatic management of object pointers held as member data	Delete the managed pointer
`LManagedRef`	Automatic management of references to resource handles held as member data	Call the `Close()` member function of the referenced handle
`LManagedhandle`	Automatic management of resource handles held as member data	Call the `Close()` member funtion of the managed handle
`LManagedArray`	Automatic management of arrays held as member data	Delete the managed array
`LManagedGuard`	Automatic generic cleanup of member data using a `TCleanupItem`	Invoke cleanup as specified by the `TCleanupItem`

Cleanup details

The `LManagedX` classes should only be used for data members. If they are used for locals, they can interact counter-intuitively with cleanup stack-based cleanup (from other functions on the call stack) with respect to cleanup ordering. Locals can be managed using the `LCleanedupX` classes.

The contained object is automatically cleaned up when the `LManagedX` object goes out of scope due to the containing object being destroyed. The destruction order of `LManagedX` members is based on the declaration order – fields are destroyed in reverse order of declaration, as specified by the C++ standard.

(C) Copyright Nokia 2013. All rights reserved.

The default cleanup behavior depends on the specific `LManagedX` variant being used; refer to Table 2 for details.

An alternative cleanup operation can be specified by passing a cleanup strategy class template as the second template parameter to the `LManagedX` constructor. More information on how this is done is provided in #Controlling automatic cleanup.

If you wish to disable automatic cleanup of your managed variable, this can be achieved by calling `LManagedX::Unmanage()`. This relinquishes control of the managed resource and hands over responsibility for cleanup. See the example in Section #Controlling automatic cleanup.

Cleanup of a managed variable can be forced by calling `LManagedX::ReleaseResource()`. This function invokes the cleanup strategy for the given object and disables automatic resource management.

`LManagedGuard` is used to provide automatic cleanup of a generic member object using a `TCleanupOperation` on the destruction of the `LManagedGuard` object.

How the LManagedX classes affect code size

Because the automatic resource management classes have been implemented via class templates, the use of these classes has impact on code size. Table 3 shows example uses of these classes and the associated increase in code size.

Table 3: Impact of RAII APIs on code size

Original code	Code using RAII classes	Code size increase

`class CMyClass: public CBase`
`{`
`public:`
`CMyClass()`
`{`
`iPtr = HBufC16::NewL(10);`
`}`
`~CMyClass()`
`{`
`delete iPtr;`
`}`
`private:`
`HBufC16* iPtr;`
`};`
`{`
`CMyClass* myClass =`
`new(ELeave) CMyClass;`
`// use myClass`
`delete myClass;`
`}`
	`class CMyClass: public CBase`
	`{`
	`public:`
	`CMyClass()`
	`{`
	`iPtr = HBufC16::NewL(10);`
	`}`
	`~CMyClass()`
	`{`
	`}`
	`private:`
	`LManagedPtr<HBufC16> iPtr;`
	`};`
	`{`
	`CMyClass* myClass =`
	`new(ELeave) CMyClass;`
	`// use myClass`
	`delete myClass;`
	`}`
		48 bytes

`class CMyClass: public CBase`
`{`
`public:`
`CMyClass()`
`{`
`iFs.Connect();`
`}`
`~CMyClass()`
`class CMyClass: public CBase`
`{`
`iFs->Connect();`
`}`
`private:`
`RFs iFs;`
`};`
`{`
`CMyClass* myClass =`
`new(ELeave) CMyClass;`
`// use myClass`
`delete myClass;`
`}`
	`class CMyClass: public CBase`
	`{`
	`public:`
	`CMyClass()`
	`{`
	`iFs->Connect();`
	`}`
	`~CMyClass()`
	`{`
	`}`
	`private:`
	`LManagedHandle<RFs> iFs;`
	`};`
	`{`
	`CMyClass* myClass =`
	`new(ELeave) CMyClass;`
	`// use myClass`
	`delete myClass;`
	`}`
		27 bytes

Basic usage patterns

Construction

An `LManagedX` object is declared using one of the class template constructors. The type to be managed is declared as the first template parameter. An optional second template parameter can be declared to implement a non-default cleanup strategy.

Example code for declaring `LManagedX` classes is shown below:

```
class CManagedUserTwoPhase : public CBase
{
. . .
private:
// We have to use LManagedXxx for fields, not LCleanedupXxx
LManagedPtr<CTicker> iTicker;
LManagedHandle<RTimer> iTimer;
};
```

Controlling automatic cleanup

Defining a custom default cleanup strategy

Each resource management class has a default cleanup strategy that is invoked when the managing object goes out of scope. The default cleanup strategy is dependent on the managing type. Although the examples below use `LManagedX` classes, the method for defining a custom cleanup strategy is equally applicable to `LCleanedupX` classes.

Consider the example below:

```
class CTicker : public CBase
{
public:
void Tick() { ++iTicks; }
void Tock() { ++iTocks; }

void Zap() { delete this; }

public:
TInt iTicks;
```


(C) Copyright Nokia 2013. All rights reserved.

```
TInt iTocks;
};

class CManagedUserSinglePhase : public CBase
{
public:
. . .
~CManagedUserSinglePhase()
{
// The iTicker manager will automatically delete the CTicker
// The iTimer manager will automatically Close() the RTimer
}
. . .
private:
// We have to use LManagedXxx for fields, not LCleanedupXxx
LManagedPtr<CTicker> iTicker;
LManagedHandle<RTimer> iTimer;
};
```

The default cleanup strategy for an `LManagedPtr` is to delete the pointer and this is the action taken in the destructor for `CManagedUserSinglePhase`.

A custom cleanup strategy for a given class can be defined using the `DEFINE_CLEANUP_STRATEGY` macro. The macro must be declared in the namespace in which the managed class is defined. Consider the example below:

```
class CTicker : public CBase
{
public:
void Tick() { ++iTicks; }
void Tock() { ++iTocks; }
void Zap() { delete this; }

public:
TInt iTicks;
TInt iTocks;
};

DEFINE_CLEANUP_STRATEGY(CTicker, Zap);
```

The class `CManagedUserSinglePhase` is defined in a different namespace to `CTicker`, defined above.

```
class CManagedUserSinglePhase : public CBase
{
public:
. . .
~CManagedUserSinglePhase()
{
// The iTicker manager will automatically Zap() the CTicker
// The iTimer manager will automatically Close() the RTimer
}

. . .
private:
// We have to use LManagedXxx for fields, not LCleanedupXxx
LManagedPtr<CTicker> iTicker;
LManagedHandle<RTimer> iTimer;
};
```

In the example above, the `DEFINE_CLEANUP_STRATEGY` macro changes the default cleanup strategy for `CTicker`. Now the cleanup strategy for an `LManagedPtr<CTicker>` is to call `CTicker::Zap()` in the destructor for `CManagedUserSinglePhase`. The `DEFINE_CLEANUP_ STRATEGY` macro must be used in the namespace in which the class is defined.

If the class to be managed is in external code, it is also possible to declare a local cleanup function in a namespace in which the class is used, by using the `DEFINE_CLEANUP_FUNCTION` macro as shown below.

```
class CTicker : public CBase
{
public:
void Tick() { ++iTicks; }
void Tock() { ++iTocks; }

void Zap() { delete this; }

public:
TInt iTicks;
TInt iTocks;
};

// CTicker is defined in an external namespace
DEFINE_CLEANUP_FUNCTION(CTicker, Zap);

class CManagedUserSinglePhase : public CBase
{
public:
. . .
~CManagedUserSinglePhase()
{
// The iTicker manager will automatically Zap() the CTicker
// The iTimer manager will automatically Close() the RTimer
}
. . .
private:
// We have to use LManagedXxx for fields, not LCleanedupXxx
LManagedPtr<CTicker> iTicker;
LManagedHandle<RTimer> iTimer;
};
```

Note: Note that `DEFINE_CLEANUP_FUNCTION` has local scope only and any definition of `DEFINE_CLEANUP_STRATEGY` will override the use of `DEFINE_CLEANUP_FUNCTION`.

Defining a custom alternative cleanup strategy Several useful alternative cleanup strategies are pre-defined in emanaged.h. It is also possible to define a custom alternative cleanup strategy that can be passed as the second template parameter to the constructor of an `LManagedX` or `LCleanedupX` object. Although the examples below use `LManagedX` classes, the method for defining an alternative cleanup strategy is equally applicable to `LCleanedupX` classes.

(C) Copyright Nokia 2013. All rights reserved.

Consider the `CTicker` class defined below:

```
class CTicker : public CBase
{
public:
void Tick() { ++iTicks; }
void Tock() { ++iTocks; }

void Zap() { delete this; }

public:
TInt iTicks;
TInt iTocks;
};

DEFINE_CLEANUP_STRATEGY(CTicker, Zap);
```

The default cleanup strategy is to call `CTicker::Zap()`. It is possible, however, to define a custom alternative cleanup strategy, as shown below:

```
// Defines a custom pointer cleanup policy that calls the
// Zap member template <class T>
class TTickerZap
{
public:
static void Cleanup(T* aPtr)
{
// The general template/class scaffolding remains the same
// for all custom cleanups, just this cleanup body varies
aPtr->Zap();
test.Printf(_L("Zapped CTicker\n"));
}
};
```

Once the new cleanup strategy is defined, the strategy can be defined as the second template parameter to the `LManagedX` constructor:

```
class CManagedUserSinglePhase : public CBase
{
public:
. . .
~CManagedUserSinglePhase()
{
// The iTicker manager will automatically Zap() the CTicker
// and print a debug message
// The iTimer manager will automatically Close() the RTimer
}
. . .
private:
// We have to use LManagedXxx for fields, not LCleanedupXxx
// The second template parameter defines the cleanup strategy for the
// managed object
LManagedPtr<CTicker, TTickerZap> iTicker;
LManagedHandle<RTimer> iTimer;
};
```

If the optional second template parameter is not defined, then the default cleanup strategy will be invoked.

```
class CManagedUserTwoPhase : public CBase
{
public:
. . .
~CManagedUserTwoPhase()
{
// The iTicker manager will automatically Zap() the CTicker
// The iTimer manager will automatically Close() the RTimer
}
. . .
private:
// We have to use LManagedXxx for fields, not LCleanedupXxx
LManagedPtr<CTicker> iTicker;
LManagedHandle<RTimer> iTimer;
};
```

Disabling automatic cleanup

An object that is being managed by an `LManagedX` class can be unmanaged by calling the `Unmanage()` method of the `LManagedX` class. After a call to `Unmanage()`, the managed object can no longer be accessed through the managing object. See the example for `LCleanedupX` in Section #Controlling automatic cleanup.

Accessing and using the managed object

The managed object can be accessed using the -> operator. The managing object can be accessed using the . operator. See the example for `LCleanedupX` in Section #Controlling automatic cleanup.

LCleanedupX Classes

Introduction

The `LCleanedupX` classes provide a means of automatically cleaning up local variables on scope exit. Variants are provided for cleaning up pointers, references, handles and arrays, as well as generic cleanup items. The user is able to define a cleanup strategy to clean up the resource on scope exit.

Anatomy of LCleanedupX

The `LCleanedupX` automatic resource management classes were added to make it easier for Symbian C++ developers to write robust code. They are designed to reduce the number of lines required to achieve leave-safe code by hiding away the cleanup stack implementation. This increases developer productivity because the code is more expressive and exception handling is semi-automated. Code quality is increased by improved readability and having fewer lines of code.

The following `LCleanedupX` classes are provided:

(C) Copyright Nokia 2013. All rights reserved.

Table 4: LCleanedupX classes

Class	Use	Default Cleanup Action
LCleanedupPtr	Automatic local-scope management of object pointers	Delete the managed pointer
LCleanedupRef	Automatic management of references to resource handles	Call the Close() member function of the referenced handle
LCleanedupHandle	Automatic local-scope management of resource handles	Call the Close() member funtion of the managed handle
LCleanedupArray	Automatic local-scope management of arrays	Delete the managed array
LCleanedupGuard	Automatic local-scope generic cleanup using a TCleanupItem	Invoke cleanup as specified by the TCleanupItem

The LCleanedupX classes should not be used in the same function as the legacy cleanup stack APIs. They can interact unintuitively and this is best avoided. It is all or nothing at function granularity; however, the two concepts can be safely used within the same thread.

Consider the example below:

```
// Create an HBufC and leave it on the cleanup stack
HBufC* buf = HBufC::NewLC(5);

LCleanedupHandle<RFile> file;

. . . // Do something with the file and the descriptor
// Cleanup the buffer
CleanupStack::PopAndDestroy(buf); // PANIC! – The LCleanedupHandle item
// comes off the stack first!
```

If an object is created and pushed on to the cleanup stack and at a later point in the same function an LCleanedupX object is declared, there will be problems when trying to pop the original object off the cleanup stack. Because the LCleanedupX classes are implemented internally in terms of the cleanup stack, their declaration results in a cleanup item being pushed onto the stack. This can lead to stack imbalance when popping, as the LCleanedupX object is pushed on to the stack after the object that was explicitly pushed.

Cleanup details

The LCleanedupX classes are for managing locals only and may not be used to manage data members. This is because the LCleanedupX classes are implemented in terms of the classic cleanup stack. Member variables may be managed using the LManagedX classes.

The contained object is automatically cleaned up when the LCleanedupX object goes out of scope, either through normal scope exit or due to a leave. The default clean up behavior depends on the specific LCleanedupX variant being used; refer to Table 4 for details.

An alternative cleanup operation can be specified by passing a cleanup strategy class template as the second template parameter to the LCleanedupX constructor. More information on how this is done is provided in Section #Controlling automatic cleanup.

Table 5: Predefined common cleanup strategies

Strategy	Cleanup Behavior
TClose	Calls the Close member function of the managed class
TRelease	Calls the Release member function of the managed class
TDestroy	Calls the Destroy member function of the managed class
TResetAndDestroy	Calls the ResetAndDestroy member function of the managed class
TPointerDelete	Deletes the managed pointer
TPointerFree	Calls user:: Free() with the managed pointer
TArrayDelete	Deallocates the array using array delete

Cleanup of a managed variable can be forced by calling LCleanedupX::ReleaseResource(). This function invokes the cleanup strategy for the given object and disables automatic resource management.

If you wish to disable automatic cleanup of your managed variable, this can be achieved by calling LCleanedupX::Unmanage(). This relinquishes control of the managed resource and hands over responsibility for cleanup.

Automatic cleanup of a generic local object is achieved using LCleanedupGuard. This uses a TCleanupOperation on the destruction of the LCleanedupGuard. This is analogous to creating a TCleanupItem and pushing it onto the cleanup stack.

General use of the templates

Class templates can be complex to implement but they're not particularly difficult to understand or use in the way they're employed here. Symbian programmers are already familiar with the basics from RArray<T> and RPointerArray<T>. And to anyone coming from standard C++ and STL, it's second nature, as in fact is this style of declaration to desktop Java and C# programmers used to generics.

In the end, it's just a concise, declarative annotation that takes leaky code and helps turn it into robust code.

Leaky:

```
{
CFoo* foo = CFoo::NewL();
foo->ExecuteL(); // the CFoo leaks if this leaves
if (foo->IsAsync())

{
// ...
return; // the CFoo leaks if we return
}
// ...
} // the CFoo leaks if we exit scope normally
```

Safe (new style):

```
{
LCleanedupPtr<CFoo> foo(CFoo::NewL());
foo->ExecuteL(); // the CFoo is automatically deleted if we leave
if (foo->IsAsync())
{
// ...
return; // the CFoo is automatically deleted if we return
}
// ...
} // the CFoo is automatically deleted if we exit scope normally
```

Safe (classic style):

(C) Copyright Nokia 2013. All rights reserved.

```
{
CFoo* foo = CFoo::NewL();
CleanupStack::PushL(foo);
foo->ExecuteL(); // the CFoo is automatically deleted if we leave
if (foo->IsAsync())
{
// ...
CleanupStack::PopAndDestroy(foo); // the CFoo is deleted manually
return;
}
// ...
CleanupStack::PopAndDestroy(foo); // the CFoo is deleted manually
}
```

Looking at these examples, it could be thought that the template APIs simply trade depth for width, as there may be less lines of code but the statements are longer and more complex.

Placing as much information at point of declaration as possible guards against separate lines of code becoming inconsistent with one another, particularly over time in the face of later code changes. Where cleanup code is duplicated, for example, in the case of a function with multiple exit points, or where the cleanup point is separated by tens of lines of code from the creation point, these risks increase further.

Another way to look at this is that it is far easier in a real, non-trivial piece of code to take an unprotected local (a raw pointer to an object) and to apply cleanup protection to it (for example, if a call to a leaving function has been added) once at its point of creation, than it is to study what might be a quite complex flow of control in order to determine where all the matching calls to `CleanupStack::PopAndDestroy()` need to go.

In some cases, faced with the cleanup stack, you might actually choose to rework an existing function to have a single exit point to simplify the process of leave-protecting its code. The risks inherent in that are not necessary when using the mode of protection offered by these new APIs.

Table 6: Impact of RAII APIs on code size

Original Code	Code using RAII classses	Increase in code size (bytes)
RFs fs; // use fs fs.Close();	LCleanedupHandle<RFs> fs; // use fs	104
HBufC* buf = HBufC::NewL(10); // use buf delete buf;	LCleanedupPtr<HBufC> buf(HBufC::NewL(10)); // use buf	90

templates, the use of these classes has impact on code size. Table 6 shows example uses of these classes and the associated increase in code size.

Basic usage patterns

Construction

An `LCleanedupX` object is declared using one of the class template constructors. The type to be managed is declared as the first template parameter. An optional second template parameter can be declared to implement a non-default cleanup strategy.

Example code for declaring `LCleanedupX` classes is shown below:

```
{
// Trivially exercise the manager using classes defined above
CTicker* ticker1 = new(ELeave) CTicker;
LCleanedupPtr<CManagedUserTwoPhase> one(CManagedUserTwoPhase::NewL(ticker1));

CTicker* ticker2 = new(ELeave) CTicker;
LCleanedupPtr<CManagedUserSinglePhase> two(CManagedUserSinglePhase::NewL(ticker2));

// Both instances are automatically deleted as we go out of scope
}
```

When instantiating an `LCleanedupPtr`, the managed object should always be instantiated via the `NewL()` method rather than `NewLC()`.

Using `NewLC()` when instantiating an `LCleanedupPtr` object can lead to cleanup stack mismatch as `NewLC()` leaves an object on the cleanup stack. Consider the example declaration below:

```
LCleanedupPtr<HBufC> buf(HBufC::NewLC(5));
```

The order of creation is as follows:

1. the `HBufC` is created and pushed onto the cleanup stack
2. `buf` is created and pushed onto the cleanup stack.

The problem occurs when it is time to pop the `HBufC` off the cleanup stack. Because `buf` was pushed on second, it is not possible to pop the `HBufC` off the stack without first popping off `buf`.

It's possible that no `NewL()` method is supplied, only a `NewLC()`. APIs should always offer pure L variants (after all, what if you want to store the result directly in a data member?) and this scenario should be rare. If it does arise, however, you have a few options.

Note: Note that code checking tools like `LeaveScan` can flag dangerous combinations of LC method calls and the `LCleanedupX` classes within the same function.

```
// If a badly-behaved API were to offer only an LC variant,
// you would have to use it as follows
HBufC* raw = HBufC::NewLC(5);

// Must pop immediately to balance the cleanup stack, before
// instantiating the manager
CleanupStack::Pop();
LCleanedupPtr<HBufC> wrapped(raw);

// Never do this:
// LCleanedupPtr<HBufC> buf(HBufC::NewLC(5));
// CleanupStack::Pop();
```


(C) Copyright Nokia 2013. All rights reserved.

```
// because the manager will be popped (having been pushed
// last), not the raw buf pointer as you might have hoped
```

A cleaner alternative may be to write your own L function wrapper around the LC function supplied. Luckily, this situation (an LC method without a corresponding L method) is rare in practice.

Controlling automatic cleanup

Each resource management class has a default cleanup strategy that is invoked when the managing object goes out of scope. It is, however, possible to define an alternative cleanup strategy when constructing the managing object. It is also possible to define a custom cleanup strategy using the `DEFINE_CLEANUP_FUNCTION` macro. Although the examples below use `LCleanedupX` classes, the method for defining how the object is cleaned up is equally applicable to `LManagedX` classes.

Consider the example below:

```
// Class definition of trivial R-Class
class RSimple
{
public:
RSimple(){iData = NULL;}

void Open(TInt aValue)
{
iData = new(ELeave) TInt(aValue);
}

// Cleanup function – frees resource
void Close()
{
delete iData;
iData = NULL;
}

// Cleanup function – frees resource
void Free()
{
delete iData;
iData = NULL;
}

// Cleanup function – frees resource
void ReleaseData()
{
delete iData;
iData = NULL;
}

// static cleanup function – frees aRSimple resources
static void Cleanup(TAny* aRSimple)
{
static_cast<RSimple*>(aRSimple)->Close();
}
private:
TInt* iData;
};
```

The above `RSimple` class has three cleanup member functions: `Close()`, `Free()` and `ReleaseData()`. The default cleanup strategy for an `LCleanedupHandle` is to call the `Close()` member function of the managed handle, as below:

```
{
LCleanedupHandle<RSimple> simple;
Simple->Open(23);
. . .
// Default cleanup strategy is to call RSimple::Close() on scope exit
}
```

As well as the default cleanup strategy, several other common cleanup strategies are predefined as class templates. An alternative strategy can be defined by passing one of the pre-defined class templates as the second template parameter to the `LCleanedupHandle` constructor as shown below:

```
{
LCleanedupHandle<RSimple, TFree> simple;
Simple>Open(23);
. . .
// Specified cleanup strategy is to call RSimple::Free() on scope exit
}
```

A custom strategy can be defined by using the `DEFINE_CLEANUP_FUNCTION` macro. This macro allows any appropriate member function of the given class to be used as the cleanup function as seen below:

```
// Because the DEFINE_CLEANUP_FUNCTION is defined below, the default cleanup
// function for RSimple is RSimple::ReleaseData() rather than RSimple::Close()
DEFINE_CLEANUP_FUNCTION(RSimple, ReleaseData);

. . .
{
LCleanedupHandle<RSimple> simple;
simple>Open(23);
. . .
// Custom cleanup strategy is to call RSimple::ReleaseData() on scope exit
}
```

It is also possible to define a custom default cleanup strategy for a class and to define a custom alternative cleanup strategy. These topics are covered for `LManagedX` in Section 3.3.2 and the same approaches can be applied to `LCleanedupX` classes.

Automatic cleanup using LCleanedupGuard

`LCleanedupGuard` is used to provide automatic cleanup for a generic object. To use `LCleanedupGuard`, a static cleanup function is defined which accepts a pointer to the object to be cleaned up. This cleanup function should call an appropriate member function on the pointer to free resources. The constructor

(C) Copyright Nokia 2013. All rights reserved.

of LCleanedupGuard takes a pointer to this cleanup function and a pointer to the object to be cleaned up. On scope exit, the cleanup function is called passing the pointer supplied to the LCleanedupGuard constructor.

```
{
RSimple simple(10);

// The RSimple class above defines a static cleanup function
// RSimple::Cleanup.
LCleanedupGuard(RSimple::Cleanup, &simple);
. . .
// On scope exit RSimple::Cleanup() is called passing &simple
}
```

Disabling automatic cleanup An object that is being managed by an LCleanedupX class can be unmanaged by calling the Unmanage() method of the LCleanedupX class. An example is given below:

```
static CStringUserTwoPhase* NewL(const TDesC& aName)
{
// We can use the resource management utility classes in
// two-phases if we want to
LCleanedupPtr<CStringUserTwoPhase> self(new(ELeave) CStringUserTwoPhase);
self->ConstructL(aName);

// Calling Unmanage() disables cleanup and yields the
// previously managed pointer so that it can be safely
// returned
return self.Unmanage();
}
```

Unmanage() disables automatic cleanup and returns a pointer to the previously managed object. Responsibility for cleaning up the object is handed over to the caller. After a call to Unmanage(), the managed object can no longer be accessed through the managing object.

Accessing and using the managed object

The managed object can be accessed using the -> operator, as in the call to ConstructL() in the example above:

```
self->ConstructL(aName);
```

The managing object can also be accessed using the . operator, as in the call to Unmanage() in the example above:

```
return self.Unmanage();
```

The managed object can be passed to a function by dereferencing the managing object. Consider the example below:

```
void RegisterTicker(CTicker& aTicker)
{
(void)aTicker;
}
```

The RegisterTicker() function defined above takes a CTicker& argument. If we have a LCleanedupPtr<CTicker> as defined below, we can pass the CTicker object to the function by dereferencing as shown:

```
LCleanedupPtr<CTicker> t(new(ELeave) CTicker);
// We can get at a reference to the managed object using *
// when we need to, e.g. if we need to pass it to a function
RegisterTicker(*t); // Takes a CTicker&
```

LCleanedupX and LManagedX Common API

This section covers the common RAII API shared by LCleanedupX and LManagedX, and examines why two separate classes are required.

Common API

The LCleanedupX and LManagedX class templates share a common API, as described below:

ReleaseResource()

```
void ReleaseResource()
```

This method forces cleanup of a managed resource. If automatic resource management is enabled, the specified cleanup strategy is invoked for the managed reference and the automatic resource management is then disabled.

Unmanage()

```
T& Unmanage()
```

This method disables automatic cleanup of a managed resource and returns a reference to the object of type T.

IsEnabled()

```
TBool IsEnabled()
```

This method returns ETrue if automatic resource management is enabled, otherwise returns EFalse.

Get()

```
T& Get()
```


(C) Copyright Nokia 2013. All rights reserved.

This method returns a reference to the managed object of type T for Ref, Handle and Guard variants of `LCleanedupX` and `LManagedX`.

```
T* Get()
```

This method returns a pointer to the managed object of type T for Ptr and Array variants of `LCleanedupX` and `LManagedX`.

Operator .
The . operator gives access to the public members of the managing class.

Operator ->
The -> operator gives access to the public members of the managed class.

Operator *
The * operator is used to get a reference or pointer to the underlying managed object.

Why the different classes?
It would be greatly preferable to be able to use the same classes to protect objects referenced from local variables, as we do to protect objects referenced from data members. In an ideal world, the `LManagedX` classes would be used in both scenarios (just like `auto_ptr` is elsewhere); `LCleanedupX` would not be necessary.

In fact, the `LManagedX` classes can be used to protect locals if you wish:

```
{
LManagedPtr<CQuery> query(CQuery::NewL());
query->BuildQueryL();
query->ExecuteQueryL();
} // query deleted on normal scope exit or leave
```

This works fine in isolation. Unfortunately, cleanup order becomes counter-intuitive if calls to functions that use LManagedX are mixed with functions that use the cleanup stack with the same call stack.

Here is an example. Call order is top-to-bottom.

```
TInt CFramework::DoStartMainLoop()
{
TRAPD(err, StartMainLoopL);
return err;
}

void CFramework::StartMainLoopL()
{

CFramework* fw = CFramework::NewL();
CleanupStack::PushL(fw); // #1
// ...
CleanupStack::PushL(event); // #2
fw->DispatchL(*event);
CleanupStack::PopAndDestroy(event);
// ...
CleanupStack::PopAndDestroy(fw);
}

void CFramework::DispatchL(CEvent& aEvent)
{
this->MyUserCallback()(*this, aEvent);
}
void MyUserCallback(CFramework& aFw, CEvent& aEvent)
{
LManagedPtr<CMyEventWrapper> myEvent(CMyEventWrapper::NewL(aEvent)); // #3
// ...
aFw->LookupL(arg);
// ...
}

void CFramework::LookupL(const TDesC& aArg)
{
//
CleanupStack::PushL(tmp); // #4
// ...
User::LeaveIfError(status);
// ...
CleanupStack::PopAndDestroy(tmp);
}
```

If the `User::LeaveIfError()` expression triggers an unhandled leave in the above, the cleanup order would be: `#4, #2, #1, #3`. If out-of-order execution of `CMyEventWrapper`'s destructor can never cause problems, then there is not an issue. However, if `CMyEventWrapper`'s destructor were to, for example, decrement a reference count on the (deleted) `CEvent`, this would lead to a panic.

The `cleanup` order is this way because of the way `User::Leave()` processing works. It locates all pushed `CleanupStack` items up to the point of the handling `TRAP` and executes their cleanups. Then, finally, it uses C++ throw to unwind the C stack to the point of the TRAP. It is during C stack unwinding that the destructors for objects stored in locals get run, and LManagedPtr (like auto_ptr) relies exclusively on destructor execution to trigger cleanup.

On the other hand, if `MyUserCallback()` uses `LCleanedupPtr` instead of `LManagedPtr`, then its associated cleanup is triggered as part of `CleanupStack` processing and in the more intuitive sequence: `#4, #3, #2, #1`.

> **Note:** Note that although it also has a destructor, `LCleanedupPtr`'s cleanup action is only ever run once; execution of the cleanup action via the destructor is disabled if the cleanup action has already been run as part of `CleanupStack` processing.

This is why the `LCleanedupX` classes exist. Situations where use of `LManagedX` for locals (or more precisely the resulting out-of-order cleanup) may cause problems are subtle to characterize. We therefore guide conservatively that `LCleanedupX` should always be used to protect locals.

(C) Copyright Nokia 2013. All rights reserved.

Conclusion

L classes are designed to make Symbian C++ programming easier by bridging the gap between standard C++ strings and Symbian C++ descriptors, and by providing class templates that automate resource management, thereby improving usability and understanding. The classes are compatible with all versions of Symbian OS from v9.1 onwards, and are freely available as an installable SIS file from File:EUserHL.zip. They are also in SDKs from Symbian^3, including the Qt SDK

Further Information

- Archived:EUserHL Core Idiom Library

See also

- C class
- R class
- M class
- T class

© 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 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.