

Contents

- [1 Introduction](#)
- [2 The L prefix](#)
- [3 The Package](#)
- [4 LString class](#)
 - ◆ [4.1 Key Points](#)
 - ◆ [4.2 Basic Usage](#)
- [5 LManagedX class](#)
- [6 LCleanedupX class](#)
 - ◆ [6.1 Key Points](#)
 - ◆ [6.2 Basic Usage](#)
- [7 CONSTRUCTORS MAY LEAVE](#)
- [8 OR LEAVE](#)
- [9 Download Package](#)
- [10 See also](#)
- [11 More Reading External Link](#)

Introduction

EUserHL stands for User High Level library. This library introduces the L classes? idioms. This new idiom provides a clear indication of the new leaving behavior of these classes. The package still contains a set of template classes that provides automatic resource management and some macros that easy single phase construction and leave launch. In the EUserHL package you will find:

- LString, a string class that handles its own buffer management and cleanup
- LCleanedupX a cleanup template class for local variables
- LManagedX, cleanup template class for member variables
- 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 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 them.

The Package

The library is available [here](#)

In this package you will find:

- Windows installer file (.msi)
- The documentation in a help file (.chm)

The installation is quite simple. You will need to choose in which SDK you will install the package (can be installed in multiples SDKs). The library is compatible with Symbian v9.1 onwards. The installation procedure basically unpacks some files to the SDK folder:

- The headers files and libraries files in the epoc32 folder
- An example to the example folder
- Installation package (.sis) file to the SDK folder. This file should be installed on the device to run applications that uses EUserHL library

LString class

LString is a general-purpose class for string manipulation. It inherits from RBuf class and adds cleanup and on-demand resize of the allocated memory. This class, as a RBuf type, can be passed as argument of functions that accept TDes and TDesC references types.

Key Points

- As an L class, LString instances will automatically cleanup themselves when going out of scope or when the object is destroyed. Also, they automatically manages the memory allocated resizing the buffer when necessary ? for example, a call to AppendL may increase the buffer size launching a KErrNoMemory if there is not enough memory for this action.
- As a direct consequence of the automatic memory resize, a raw pointer to the data (obtained with a call to Ptr() function) may become invalid easily, as well as the value obtained from MaxLenght() function, so avoid to use then extensively.
- The automatic resize of the buffer only happens when the LString class is manipulated directly. So, if you pass a LString to a function that receives a TDes that function will operate as if the descriptor has only the max-length. This may cause errors. See bellow.

```
LString string;
addDatatoString(string);
?
void addDatatoString(TDes& aDes)
{
    aDes.Append(_L(?Data?)); //this will PANIC
```

```
}

```

This happens because LString default constructor sets the buffer to zero (0) size. The programmer is responsible for ensuring the size of the buffer is enough to fit the data. To do this the API provides the ReserveFreeCapacityL() function. This function must be called when you cannot use any of the auto-buffer extending LString methods to achieve your objective. This often happens inside functions that receives a TDes& as parameter.

```
LString string;//zero size
string.ReserveFreeCapacityL(4);
addDatatoString(string);
?
void addDatatoString(TDes& aDes)
{
    aDes.Append(_L(?Data?)); //OK!
}

```

- Is important to notice 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**.
- The use of LString class increase productivity by easing the development of application because of the automatic allocation methods. These methods incur in cost of performance because more instructions must be executed. So in performance-critical code, the use of LString class is discouraged. The good side of the LString is the huge improvement on the readability of the code, as it can be seen bellow.

Code without LString:

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

```

Code WITH LString:

```
{
    LString query;
    BuildQueryL(query);
    ExecuteQueryL(query);
}

```

Basic Usage

The common cases of construction of the LString can be seen bellow.

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

```

EUserHL

```
// 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;
}
```

LManagedX class

LCleanedupX class

LCleanedupX are templated classes that avoid explicit calls to cleanup stack functions automatically cleaning local objects in case of leaves or scope exit. This kind of class put the object in the cleanup stack automatically. The provided LCleanedupX classes and their use are shown bellow:

- LCleanedupPtr. Automatic local-scope management of object pointers.
- LCleanedupRef. Automatic local-scope management of references to resource handles.
- LCleanedupHandle. Automatic local-scope management of resource handles.
- LCleanedupArray. Automatic local-scope management of arrays.
- LCleanedupGuard. Automatic local-scope generic cleanup using a TCleanupItem

Key Points

- The LCleanedupX classes **SHOULD NOT** be used in the same function as the legacy cleanup stack APIs. The creation of a LCleanedupX object pushes an object onto the stack. As this is not explicitly stated on the code it may cause confusion or misinterpretation of the order of the objects in the stack. Due to this, it is recommended that inside a function scope only one idiom is used (or pure legacy idiom with cleanupstack calls or LCleanedupX classes). See the example bellow:

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

LCleanedupHandle<RFile> file; //putting file onto the cleanupstack

. . . // Do something with the file and the descriptor

// Cleanup the buffer
CleanupStack::PopAndDestroy(buf); // PANIC! ? The LCleanedupHandle item comes off the stack first
```

EUserHL

- The LCleanupX classes are for managing locals only and may not be used to manage data members. This is because the LCleanupX classes are implemented in terms of the classic cleanup stack. Member variables may be managed using the LManagedX classes.
- Be careful with the use of NewLC with LCleanupX. See the example below:

```
// 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();

LCleanupPtr<HBufC> wrapped(raw);
// Never do this:
//     LCleanupPtr<HBufC> buf(HBufC::NewLC(5));
//     CleanupStack::Pop();
// because the manager will be popped (having been pushed last), not the raw buf pointer as you m
```

- The contained object is automatically cleaned up when the LCleanupX object goes out of scope, either through normal scope exit or due to a leave. The default clean up behavior depends on the specific LCleanupX variant being used.
- Cleanup of a managed variable can be forced by calling LCleanupX::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 LCleanupX::Unmanage(). This relinquishes control of the managed resource and hands over responsibility for cleanup.
- Due to the use of templates, the size of the code increase. See the comparisons below:

1) Code increase: 104 bytes

```
//Without
RFs fs;
// use fs
fs.Close();
//-----
//With
LCleanupHandle<RFs> fs;
// use fs
```

2) Code increase: 90 bytes

```
//Without
HBufC* buf = HBufC::NewL(10);
// use buf
delete buf;
//-----
//With
LCleanupPtr<HBufC>
buf(HBufC::NewL(10));
// use buf
```

Basic Usage

Bellow you'll see more example of the basic usage of the LCleanupX classes.

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

CTicker* ticker2 = new(ELeave) CTicker;
LCleanupPtr<CManagedUserSinglePhase> two(CManagedUserSinglePhase::NewL(ticker2));
// Both instances are automatically deleted as we go out of scope
}
```

CONSTRUCTORS_MAY_LEAVE

OR_LEAVE

Download Package

- [EUserHL Package](#)

See also

- [C class](#)
- [R class](#)
- [M class](#)
- [T class](#)

More Reading External Link

- [L Classes](#)
- [L Classes PDF](#)