symbian

Introducing the RBuf Descriptor

Mark Shackman

Version 1.0

1. Introduction to Descriptors

Descriptors have occupied a fundamental place in Symbian OS since its inception. They serve as a safe, consistent and economical mechanism for accessing and manipulating both strings and general binary data.

Descriptors allow data to be resident in either the device's ROM or its RAM, on either the heap or the stack. All access to the data is through the methods of the descriptor object, which maintains pointer and length information for that data.

All descriptors inherit from the abstract base class `TDesC`. Three other classes, `TDes` , `TBufBase` and `TBufCBase`, are abstract; all the others are concrete.

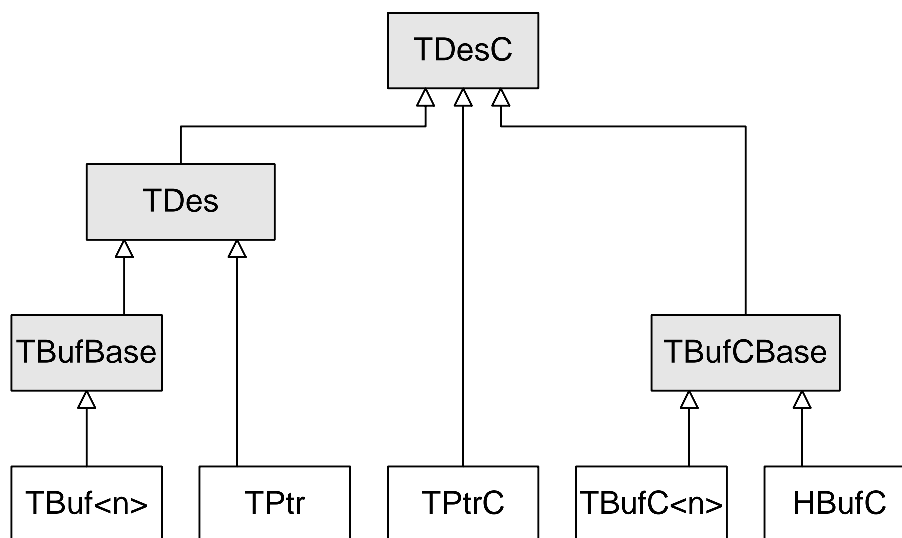

Figure 1: The original descriptors and their hierarchy

Descriptors come in three types:

- Buffer descriptors: these store the data as part of the descriptor object, which typically lives on the program stack. They are used for smaller amounts of data of a fixed maximum size.
  `TBuf<n>` and `TBufC<n>` are buffer descriptors.
- Heap descriptors: these store the data as part of the descriptor object, which lives on the program heap. They are used for larger amounts of data and are resizable.
  `HBufC` is a heap descriptor.
- Pointer descriptors: these contain a pointer to the data, which lives elsewhere in memory.
  `TPtr<n>` and `TPtrC<n>` are pointer descriptors.

Descriptors with a C suffix in their class name are constant and therefore unmodifiable (although the Des() function, where available, can be used to circumvent this).

For further general information on descriptors, please see the Symbian Developer Library [1], books from Symbian Press [2], Forum Nokia's [3] "Descriptors For Text And Binary Data" paper and the Descriptors FAQ & Descriptors top tips at http://www.whoshavesthebarber.com/.

2. Introducing RBuf

The set of descriptors described above are those originally included in Symbian OS. These provide for a wide range of requirements and situations: constant, unmodifiable descriptors for strings embedded in applications, stack-based descriptors for manipulating short strings, heap-based descriptors for strings that may grow or shrink, pointer descriptors for ROM-based resources etc.

Of the five concrete descriptor classes initially available, the only omission was that of a modifiable heap-based descriptor. Thus, just as TBufC has the modifiable TBuf, and similarly TPtrC has TPtr, a companion for HBufC was required. Although there is a way to modify the data in an HBufC, (using the Des() method and maintaining a companion TPtr), the RBuf descriptor, which is derived from TDes as shown below, was introduced in version 8.0 (and documented in the Symbian Developer Library from version 8.1)..

An RBuf is particularly useful as all those temporary or companion TPtrs (required to alter the contents of an HBufC) become completely unnecessary, because RBuf both acts as a TPtr, but also owns the underlying heap object, hence replaces the HBufC*.

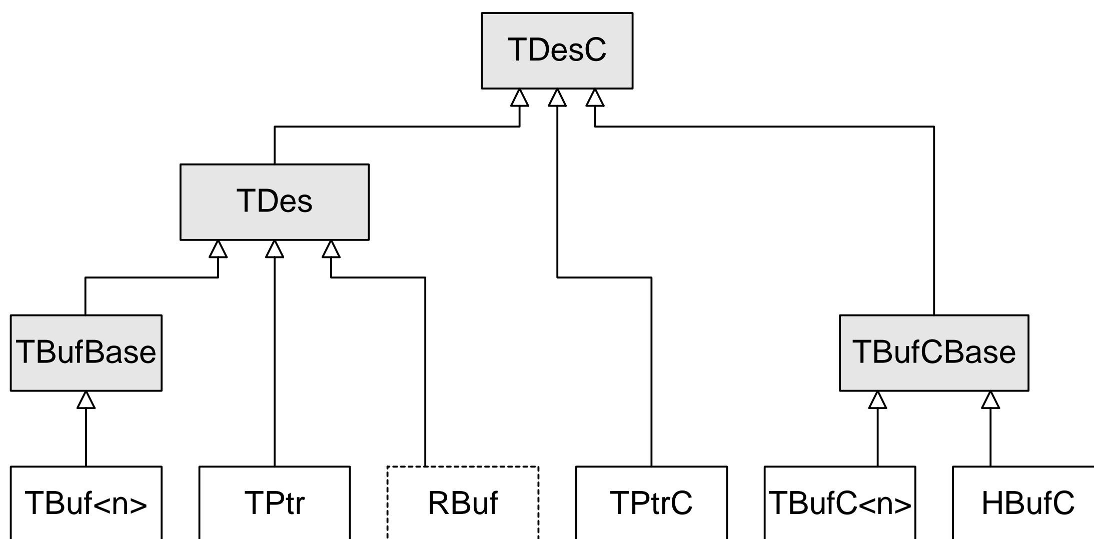

Figure 2: The position of RBuf in the hierarchy

Similarly to an HBufC, an RBuf object can allocate its own buffer on construction. It can also take ownership of a pre-existing section of allocated memory or a pre-existing heap descriptor. Since it's derived from TDes, an RBuf object can easily be modified, and additionally can be passed to any function where a TDes or TDesC parameter is specified.

Internally, RBuf behaves in one of two ways:

- as a TPtr descriptor type, where the buffer just contains data
- as a pointer to a heap descriptor (an HBufC* type) where the buffer contains both descriptor information and the data.

The handling of the distinction is hidden from view. Since RBuf behaves like an HBufC, it's possible to create an RBuf from an existing HBufC, making it easy to move code over to the new class. For examples, see section 3.5 below.

RBuf descriptors can live on the stack, but maintain a pointer to memory in the heap. On the stack, they take up 8 bytes more than an HBufC* (i.e. 3 words rather than 1), but will generally take up 4 bytes less heap space (if not constructed from an existing HBufC).

The new descriptor is not named HBuf because, unlike HBufC, objects of this type are not themselves directly created on the heap (the H prefix stands for heap); choosing an R<class> name carries the implication that the class owns the resources (in this case a Buf) that it manages and is responsible for freeing the memory when it closes.

RBuf descriptors were introduced in Symbian OS v9 and backported to v8.1 and v8.0.

3. Code Examples for RBuf

3.1. Construction

On construction an RBuf object can allocate its own buffer, take ownership of a pre-existing section of allocated memory or take ownership of a pre-existing heap descriptor.

For example, to construct an RBuf with a zero length, resizable buffer that owns no allocated memory, use:

```
RBuf myRBuf;
```

To have an RBuf take ownership of an existing heap descriptor, use:

```
HBufC* myHBuf = HBufC.NewL(20);
RBuf myRBuf (myHBuf);
```

In practice, your code may appear more along the lines of:

```
RBuf resourceString (iEikEnv->AllocReadResourceL(someResourceId));
```

Note that there's no way to construct or set an RBuf from any arbitrary descriptor type. Only buffers that can explicitly be owned and released by the RBuf are supported. On construction, this is limited to just HBufC, as others are ambiguous. Using the Assign() method discussed below, flat heap byte/Uint arrays and existing RBufs are also supported: these are not supported through constructors though as the ownership transfer is not clear enough.

3.2. Allocation and Modification

3.2.1. Create

Having constructed an RBuf that owns no data (see the first code snipped above), data and/or space for data can be associated with the RBuf using the Create() method. A number of variations are possible, as outlined in the following discussion.

To simply allocate space for data, without specifying the data itself, use:

```
RBuf myRBuf;
TInt maxSizeOfData = 20;
if (myRBuf.Create(maxSizeOfData)) != KErrNone)
	{
	...	// error handling
		// (omitted from subsequent code examples for clarity)
	}
```

The current size of the descriptor data is set to zero. To set the size of the data to the maximum (i.e. equal to the `maxSizeOfData` parameter), use the `CreateMax()` method instead.

It's also possible to create an `RBuf` with its data initialised. For example, to copy the contents of another descriptor into an `RBuf`, specify the other descriptor as the parameter to the `Create()` method:

```
RBuf myRBuf;
TBufC<20> myTBufC (_L("Descriptor data"));
myRBuf.Create(myTBufC);
```

This sets the maximum size of the `RBuf` to the length of the source descriptor. To specify a maximum size larger or smaller than this, pass the maximum size as the second parameter to the `Create()` method:

```
RBuf myRBuf;
_LIT(KGenesis, "In principio creavit Deus caelum et terram.");
TInt maxSizeOfData = 30;	// trims text to 30 characters
myRBuf.Create(KGenesis(), maxSizeOfData);
```

A final option allows the contents of an `RBuf`'s data to be set from a stream which contains the length of the data followed by the data itself. Both the stream and the maximum permitted size of the stream are specified as parameters to the `CreateL()` method – there is no `Create()` method available:

```
RBuf myRBuf;
RReadStream myStream;
TInt maxSizeOfData = 30;
myRBuf.CreateL(myStream, maxSizeOfData);
```

The `Create()` methods return a standard error code, usually `KErrNone` if successful or `KErrNoMemory` if there is insufficient memory. For all the `Create()` methods (including `CreateMax()`) there are corresponding `CreateL()` methods which leave on failure.

Note that `Create()` orphans any data already owned by the `RBuf`, so `Close()` should be called where appropriate to avoid memory leaks.

3.2.2. Assign

An `RBuf` can take ownership of a pre-existing section of memory using the `Assign()` method.

For an `RBuf` to take ownership of a preallocated heap descriptor, use:

```
HBufC* myHBufC = HBufC::NewL (20);
RBuf myRBuf.Assign(myHBufC);
```

To transfer ownership of another `RBuf`, the `RBuf` is specified as the parameter to the `Assign()` method:

```
RBuf myRBuf1;
RBuf myRBuf2;
```

```
HBufC* myHBufC = HBufC::NewL(20);
myRBuf1.Assign(myHBufC);      // take ownership of heap memory
myRBuf2.Assign(myRBuf1);      // take ownership of another RBuf
myRBuf2.Close();
```

Incidentally, the method `Swap()` allows the contents of two `RBuf`s to be exchanged.

Finally, the `RBuf` can take ownership of preallocated memory on the heap by specifying the heap cell and the maximum size of the data:

```
TInt maxSizeOfData = 20;
RBuf  myRBuf;
TUint16* pointer = static_cast<TUint16*>(User::AllocL(maxSizeOfData * 2));
myRBuf.Assign(pointer, maxSizeOfData);
```

For this call, the current size of the descriptor is set to zero; to specify a different size, insert this value as a new parameter between the existing two. As an example, the last line of the code snippet above could be replaced by:

```
TInt currentSizeOfData = maxSizeOfData / 2;
myRBuf.Assign(pointer, currentSizeOfData, maxSizeOfData);
```

Note that `Assign()` (similarly to `Create()`) orphans any data already owned by the `RBuf`, so `Close()` should be called where appropriate to avoid memory leaks.

3.2.3. ReAlloc

Having created an `RBuf`, a significant benefit over other descriptors is that, should the data exceed the size of the descriptor, the data space in the descriptor can be resized to accommodate the data. This is achieved by using the `ReAlloc()` method:

```
myRBuf.CleanupClosePushL();
...
const TInt newLength = myRBuf.Length() + appendBuf.Length();
if (myRBuf.MaxLength() <  newLength)
      {
      myRBuf.ReAlloc(newLength);
      }
myRBuf.Append(appendBuf);
...
CleanupStack::PopAndDestroy();    // calls myRBuf.Close();
```

Using an `RBuf` is preferable to using an `HBufC` in that, if the `ReAlloc()` method is used on the `HBufC` and causes the heap cell to move, any associated `HBufC*` and `TPtr` variables need to be updated. This update isn't required for `RBuf` objects.

A corresponding `ReAllocL()` method is available that acts similarly but leaves on failure.

3.3. Destruction

Regardless of the way in which the buffer has been allocated, the `RBuf` object is responsible for freeing memory when the object itself is closed. Both the `Close()` and the `CleanupClosePushL()` methods are available and should be called appropriately. See the example for `ReAlloc()` above.

3.4. Other Methods

This paper has discussed the methods introduced in the RBuf class. Since RBuf inherits from TDes, all the methods from TDes and TDesC are available to RBuf and can be used as described in the Symbian Developer Library documentation.

```
_LIT(KTextHello, "Hello");
_LIT(KTextWorld, " World");
RBuf myRBuf;
myRBuf.CleanupClosePushL();
myRBuf.CreateL(KHello());
imyRBuf.ReAllocL(KHello().Length() + KWorld().Length());
myRBuf.Append(KWorld);
CleanupStack::PopAndDestroy() // calls myRBuf.Close();
```

3.5. Migration

Migrating code that uses HBufC* and TPtr to using just RBuf is desirable on three counts:

- it makes the code easier to read & understand and hence to maintain
- it reduces the possibility of errors
- the object code is slightly smaller

Since it is possible to create an RBuf from an existing HBufC, it's easy to move code over to this new class. For example, whilst migrating code internally, our changes were mainly from code previously of the form:

```
HBufC* mySocketName;
...
if(mySocketName==NULL)
    {
    mySocketName = HBufC::NewL(KMaxName);
    }
TPtr socketNamePtr (mySocketName->Des());
aMessage.ReadL(aMessage.Ptr0(),socketNamePtr);
```

which was converted to code like this:

```
RBuf mySocketName;
...
if(mySocketName.IsNull())
    {
    mySocketName.CreateL(KMaxName);
    }
aMessage.ReadL(aMessage.Ptr0(), mySocketName);
```

An RBuf can directly replace an HBufC, so you use them to call predefined APIs that already return a new HBufC. For example:

```
HBufC* resString = iEikonEnv->AllocReadResourceLC(someResourceId);

// increase size of label, copy old text and append new text
labelLength += 4;
HBufC* label = HBufC::NewLC(labelLength);
TPtr labelPtr(label->Des());
labelPtr.Copy(*resString);
labelPtr.Append(_L("-Foo"));
//...etc...
SetLabelL(ELabel, *label);
CleanupStack::PopAndDestroy(2);
```

converts to:

```
RBuf resString (iEikonEnv->AllocReadResourceL(someResourceId));
resString.CleanupClosePushL();
 // Use modifiable descriptor to append new text...
resString.ReAllocL(resString.Length() + 4);
resString.Append(_L("-Foo"));
//...etc...
SetLabelL(ELabel, resString);
CleanupStack::PopAndDestroy() // calls resString.Close();
```

4. Conclusion

RBuf makes a welcome and integral addition to Symbian OS descriptors, as it provides a flexible, modifiable heap-based buffer. Little overhead is required to convert existing code, as RBuf uses the hierarchy of descriptors already familiar to Symbian OS programmers. With the use of RBuf instead of HBufC, code becomes slightly simpler and more understandable & maintainable.

References

[1] Symbian Developer Library http://www.symbian.com/developer/techlib/sdl.html

[2] Symbian Press http://www.symbian.com/books/index.htm

[3] Forum Nokia http://www.forum.nokia.com

Thanks to reviewers JohnP & JonathanD and also to ColinT.

Back to Developer Area

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.