

# Advanced RArray

Paul Todd

Published by the Symbian Developer Network

Version: 1.0– April 2008

1	INTRODUCTION .....	2
2	ARCHITECTURE OF THE RARRAY CLASSES.....	2
3	COMMON FUNCTIONS.....	3
4	SEARCHING AND SORTING WITH RARRAY CLASSES .....	5
5	ISSUES WHEN USING THE RARRAY CLASSES.....	7
6	TIPS FOR GET THE MOST OUT OF USING THE RARRAY CLASSES.....	10
7	CODE EXAMPLES .....	15
8	FURTHER INFORMATION .....	15

## 1 Introduction

This document covers intermediate and advanced usage of the RArray classes. It gives explanations and examples for some of the most frequently asked questions and encountered problems that have appeared on the Symbian forums.

On embedded systems, understanding arrays is critical to getting the best performance and smallest memory footprint out of applications, and hence to improving the user experience.

RArray and its companion RPointerArray provide the fastest and most efficient way to safely access arrays of structures and objects on Symbian OS. They offer range checking, type safety, and reliable pre-built functions to assist in sorting, searching, and modifying items, whilst at the same time offering superior performance and usability over generic implementations or legacy APIs.

Starting from Symbian OS v9, RArray has been upgraded to make it safer for resource management by leaving if functions fail, adding support so that it can be used in the kernel, and adding functions to reserve memory.

As the RArray classes are highly optimized to use the ARM processor, the items need to be 32-bits wide to make the best use of the processor. Even if the object is less than four bytes it will still consume four bytes. One and two byte arrays should be handled using the descriptor classes (such as TPtr and TDes) as these are more amenable to working with byte and word type structures.

The RArray classes have been designed to look and behave as any Symbian C++ 'R' class would. They provide excellent examples of the design and application of coding standards in Symbian C++. More on Symbian C++ coding standards can be found in the booklets available to download at the end of this article.

## 2 Architecture of the RArray classes

The RArray and RPointerArray classes use both patterns to their fullest extent, in that they provide type-safety at compile time through the use of templates and they also use the resource management pattern common to Symbian C++ classes.

The first pattern is the use of thin-template patterns. Many classes in Symbian C++ provide type-safety through the use of templates, but these templates are then inlined to provide a call down into a base class where the actual implementation may be found.

This design makes the compiler do the work, enforcing type-safety at compile time by using templates, whilst at the same time producing highly efficient code that can be optimized away at compile time – the best of both worlds.

The second pattern is that of resource management. It is a convention in Symbian C++ to declare classes that are lightweight and to work with resources such as handles or memory as 'R' classes (or 'Resource' classes). These classes are also very lightweight, in that they generally hold a handle to a kernel side object and provide simple implementations to a class that works with the handle by calling to the underlying session object.

In the RArray classes, the RArray class forms a wrapper over the RArrayBase and RPointerArrayBase classes. To see how this works look at `e32cmn.inl` in the SDK.

RArray is implemented as a class inherited from RArrayBase, and RPointerArray is implemented as an inherited class of RPointerBaseArray. In essence, the base classes provide a means of manipulating a block of memory very efficiently. The array keeps track of the object size, the size of the block allocated, the number of items to allocate when a resize is needed, as well as a pointer to the block of memory.

The whole architecture underlying the implementation of arrays revolves around the fact that the items in the array are of a fixed size and can be accessed using simple pointer arithmetic and memory copies.

Internally, all the accesses are done using pointers and pointer arithmetic which is highly efficient, both in code size and execution time on the device. Typically, to find an item in the array required the following calculation: Base Pointer of memory block + (size of object \* index). Obviously there are checks in place to ensure the index is not out of range, but the overhead of this is negligible.

To reduce memory fragmentation and memory thrashing, typical array implementations declare a granularity which defines how many additional items will be over-allocated so that, when a new item is added, the array does not need to be reallocated. Reallocation is usually a very expensive operation as it requires a new larger memory block to be allocated if possible (otherwise the memory allocator may attempt to reclaim memory), and then all the data from the original block needs to be copied to the new block. Two enhancements to the RArray classes in Symbian OS v9 are to preallocate memory when the array is created and to resize the array by a granularity or minimum size and factor. The first is provided by the Reserve() function and the second is provided by the granularity, minimum size and factor parameters on some of the constructors. Again these are useful to prevent memory fragmentation and thrashing in cases where there is moderate to heavy addition and removal of items from the array.

Insertion becomes a simple matter of checking to see if there is sufficient space for the new item (if not, the array is expanded), then copying the remaining memory from the index address to the index address plus the size of an item. The newly inserted item is then copied to the empty space at the current position.

Deletion is even simpler as it does not require a memory reallocation, so the memory from the next item to the last item is moved down by the item size. This is, however, done at the expense of not freeing memory at the end of the array until Compress() or GranularCompress() is called.

To further assist developers, the array functions are for the most part hand coded in pure assembly to be just that little bit faster, and they also avoid the overhead of temporary objects that the CArray classes have. So these array functions are almost a straight processor call to shift memory from one area to another and thus are highly efficient.

Internally, memory for the RArray classes blob is allocated using User::Alloc() and freed using the User::Free() function. This is very important to realize as the constructor/destructor logic built into C++ is not used or even known about – the underlying mechanism might as well be C code in the way it works. This also means that adding an object will not call the copy operators, nor will removing an object call its destructor, so care must be taken to ensure that these are not part of any resource management strategy employed by the program – otherwise resources will be lost, be it handles or memory.

There are a number of constructors for RArray, each of which has a very useful purpose in setting up how the RArray will perform when the application is running.

### 3 Common Functions

These are some of the more misunderstood functions of the RArray classes, as well as functions that should be noted as they are often overlooked.

#### 3.1 RArray(TInt aGranularity)

This constructor is used for creating an array where you can control how many items are allocated when an expansion is required. This is helpful to prevent memory thrashing when items are being allocated by providing a trade-off between pre-allocating memory, in the hopes it will not allocate too much memory, and always having to reallocate a block whenever a new item is added. The

difference between a granularity of 1 and of 8 is about three times, and so it's worthwhile spending time profiling highly used arrays to see if a better allocation strategy can be employed.

### 3.2 RArray(TInt aGranularity, TInt aKeyOffset)

This constructor is used to create an array where you control the granularity and the offset of the key in the structure when it is used for searching. When working with sorted data, it is often convenient to search the array by an offset, hence the `XxxSignedKeyOrder()` and `XxxUnsignedKeyOrder()` functions as discussed in Section 4.

### 3.3 RArray(TInt aMinGrowBy, TInt aKeyOffset, TInt aFactor)

This constructor is most useful when there is a need to control how fast the array grows. Instead of using the granularity (as above), this allows the array to be expanded using a different algorithm. Here the creator can specify the minimum number of items to grow the array by when expanding, or the factor (which is how fast to grow the array). Note that the factor tends to cause exponential growth when expanding.

#### 3.3.1 RArray(TInt aEntrySize, T \*aEntries, TInt aCount);

This constructor is used to create an array based on the specified memory supplied. It is up to the caller constructing the array as to whether the array takes ownership of the memory or not. If the array does not own the memory then the array object should not be put on the cleanup stack.

Typical uses for this constructor are to create an array that is mapped onto a compile time structure.

```
TInt32 KInts[] = { 1, 2, 3, 4, 5 };
RArray<TInt32> myArray((TInt32*)&KInts, sizeof(KInts) / sizeof(KInts[0]));
```

In this example the array must not be expanded, put on the cleanup stack, or deleted with `Reset()` as it does not own any memory. However, it does provide a way to take static structures and to map them into a usable `RArray` with minimal overhead.

Using this form of constructor, and if the structures are already in a memory block, it is possible to transfer ownership of the pointer onto the `RArray` with no overhead. Care must be taken to ensure that the block was allocated using `User::Alloc()`. For example:

```
TInt32* data = User::AllocLC(sizeof(TInt32) * 100);
... DoSomething with data - e.g. IPC call
RArray<TInt32> myArray(sizeof(TInt32), data, 100);
CleanupStack::Pop(data);
CleanupClosePushL(myArray);
```

`myArray` can now be added to or removed, etc., as the memory is owned by `myArray`.

### 3.4 Close()

This performs the same functionality as the `Reset()` operation but is here to provide the standard metaphor for working with `RArray` classes.

### 3.5 Reset()

This function will delete the memory owned by the array. It is important to note that it just calls `User::Free()` on the memory block and it does not call any destructors. Care should also be taken to ensure that, even though an array appears to contain no elements (`Count() == 0`), the array may still own memory – if there has been any insertion or deletion then the underlying allocated memory block may still be present and so, by not calling `Reset()`, this block may be orphaned.

### 3.6 ResetAndDestroy()

This is only available to the `RPointerArray` class. It is the only significant function that is templated, as it calls the destructor for each pointer in the array and then calls `reset`.

Unless objects are cleaned up first, calling `Reset()` instead of `ResetAndDestroy()` will orphan items within the array.

## 4 Searching and Sorting with RArray classes

The `RArray` classes contain functions for searching and sorting. These functions provide ways of keeping an array sorted, as well as for searching an array. Both these provide methods to search using linear or binary searching.

The `RArray` classes provide a method to sort the arrays by using either the offset key or by using a comparison function. The method used to sort the array is the `heapsort` function, rather than the more common `quicksort` algorithm, though for general cases there is very little performance difference. `Heapsort` has the advantage over `quicksort` in that the worst case performance is constant and the additional memory requirements are constant, and so runtime performance is always predictable.

The `InsertXxx()` functions search for the nearest item in the array and then insert items in the correct position based on the results of the search. The insertion functions allow for both sequential and sorted insertions. The most useful are the sorted insertion functions, which ensure that the array always remains sorted and therefore easily searchable. The trade-off, however, is that the insertion will be slightly slower due to the need to find the correct insertion point.

The `SearchXxx()` functions allow an item to be searched for in a number of ways. The most obvious is just a plain linear search by searching for a specific item, or by supplying an equality function that compares an item and the items in the array. This is very inefficient, so a much improved solution is to ensure that the array is sorted and then a binary search can be used to find an item. These are the 'InOrderXxx()' functions.

The arrays also support searching for duplicate items using the `SpecificXxx()` variants of the functions. These will allow the first item, the last item, or any matching item to be returned when a duplicate is found. So for example, if there are three items in a sorted array all with the same key, then the `SpecificXxx()` variant in the `RArray` class can be used to select the first item in the matching key set, the last item in the matching key set, or any item in the key set.

There are a number of common methods when using the searching and sorting functions.

- Class offset:

The first is to use an offset into the object that is used to compare and decide the matching. When searching or storing items this offset is used as the key, which keeps items in order.

This offset is typically used as follows:

```
struct TMyStruct
{
    TInt x;
    TInt y;
}

RArray<TMyStruct> myArray(4, _FOFF(TMyStruct, y));
```

- Callback function:

The second method is to use a function that will be called back each time a comparison is required.

There are two types of function: one to just return information whether the item matches the item being searched for, and the second to provide an ordering relationship.

The `TIdentityRelation` provides for a function that returns whether or not the two items are identical, rather than providing a relative view of the item. This is only useful when attempting to determine whether an item exists in an array.

The `TLinearOrder` class provides for a function that can be used to provide the relative ordering of an item by returning 0 if the items are identical, -1 if the first item is less than the second, and 1 if the first item is greater than the second.

Both of these functions are rendered type-safe through the use of templates. For this reason, it is recommended that the comparison functions are static functions within the class or friends so that they can access private and protected member variables. Whilst this breaks encapsulation it also results in cleaner and more legible code.

For example:

```
// Equality operation
TBool IsEqual (const TStruct& aLeft, const TMyStruct& aRight)
{
    return aLeft.x == aRight.x;
}

// relative operation
TInt Compare (const TStruct& aLeft, const TStruct& aRight)
{
    if (aLeft.x == aRight.x)
        return 0;
    if (aLeft.x < aRight.x)
        return -1;
    return 1;
}

TLinearOrder<TMyStruct> order(CompareMyStruct);
myArray.Sort(order);

TIdentityRelation<TMyStruct> matchEqual (EqualMyStruct);
TInt index = myArray.Find(3, matchEqual);
User::LeaveIfError(index);

// do something with the index
```

The class `offset` method is of most use when the key is 32-bits wide. In this case, all the functions with `'SignedXxx()'` or `'UnsignedXxx()'` in them will work with signed or unsigned versions of the offset key. Searching for a 32-bit number in an array is a very general use case and it is for this reason that the offset functionality is provided, as it means that relational functions do not need to be written to cater for this use case.

One point to note is that the searching functions require a temporary object to compare it with. In many cases creating this temporary object imposes an overhead, so a new static comparison function that takes a key can be supplied to search. This functionality is undocumented in the help files and SDK, however it does provide a useful means of searching for a key in a manner more akin to the C++ standard template library.

Here is an example:

```

struct TMyStruct
{
    TBuf<32> iItem;
    TInt    iKey;
};

// the comparison function
LOCAL_C TInt CompareFunction(const TInt* aKey, const TMyStruct& aItem)
{
    return (*aKey) - aItem.iKey;
}

// an array RArray<TMyStruct> array;

... insert some items into the array in order of iKey

// now find the key
TInt index = array.FindInOrder<TInt>(1, CompareFunction);

```

## 5 Issues when using the RArray classes

### 5.1 Assuming Reset() calls object destructors

Reset() does not explicitly call the destructors for the contained objects. Should any object in the array contain resources then these will be orphaned and, when the application exits, leaks will be reported in debug mode. It is critical that proper Symbian design be used to prevent these types of handle and memory leaks.

### 5.2 Incorrect object size

The RArray classes expect all the objects it contains to be between four and 640 bytes. If the size is less than four bytes then this will be rounded up to four bytes. If the size is over 640 bytes or less than or equal to zero then a panic EBadArrayGranularity will result.

Aside from the overhead of using three extra bytes to represent a single byte in an RArray, alignment problems are a much more common issue for developers used to working on the more forgiving Intel platforms. ARM processors work best with 32-bits as this helps its internal processor cache and memory bus throughput. The sources of further information listed at the end contain a more comprehensive overview of why this is.

This is most often seen when extracting objects that have come in over the wire (via TCPIP, for example) and where alignment is not a problem.

Typically, the server has had its structures wrapped around a #pragma pack(1) directive which has packed the structure to remove all padding for it.

For example, the following is one way to raise a panic in emulator builds and on a real device.

```
TUint8 array[4] = {0x11, 0x12, 0x13, 0x14};
TUint8* p = &array[3];
RArray<TUint8> myArray;
myArray.Append(*p);
```

In the example above, `p` will be aligned on a four byte boundary by the compiler. However, `p + 3` is not aligned on a four byte boundary and so the ARM processor will raise a panic and stop the program, as the compiler is unable to correctly determine the type, and it will generate the correct code to avoid a fault.

In the case where the objects are one or two bytes wide, consider using a descriptor which provides most of the functionality of an array without the alignment issues.

It is also recommend when reading blocks of data to use an `RReadStream` or one of its derivate classes, rather than trying to map the memory structure directly.

### 5.3 Polymorphic structure issues

The `RArray` classes assume the items contained in it are of a fixed size, and so you should never add structures that are polymorphic as the smallest size will be chosen by the compiler and unexpected results will occur. The item you put into the array will come out as the base class declared in template, or what amounts to the lowest common denominator of the structures.

For example, do not do the following:

```
struct TAnimal
{
    TBuf<32> iType;
};

struct TDog : public TAnimal
{
    TDogBreed iDogBreed;
};

struct TCat : public TAnimal
{
    TCatBreed iCatBreed;
};
```

It is acceptable to declare an array of `RArray<TDog>` or of `RArray<TCat>` but never `RArray<TAnimal>` when you want to insert structures of `TCat` or `TDog`. If you need polymorphism, you must use the `RPointerArray` classes and ensure your item's base class is derived from `CBase`.

### 5.4 Pointer ownership in RPointerArray

The other array class is the `RPointerArray`, which will hold a pointer so that when it is deleted it will be cleaned up correctly. It is up to the developer how they handle the ownership of pointers within the array as there are many uses cases for handling objects. The general requirement is that when the object is added to the array the array takes ownership of it, and when it is removed the array loses ownership.

```

// Block 1
RPointerArray<CMyClass> myOwnngArray;
// the array both owns the block holding the pointers
// and the pointers themselves.
CleanupResetAndDestroyPushL(myOwnngArray);

// now add a object
CMyClass* object = new (ELeave) CMyClass;
CleanupStack::PushL(object);
User::LeaveIfError(myOwnngArray.Append(object));
CleanupStack::Pop(object);

// a another object
object = new (ELeave) CMyClass;
CleanupStack::PushL(object);
User::LeaveIfError(myOwnngArray.Append(object));
CleanupStack::Pop(object);

....
// Block 2
// we now create a shallow array that has pointers that
// are not owned by the array but the array still has
// a block of memory for the contents
RPointerArray<CMyObject> myArray;
// Note how we push this, the array does not delete the pointers
CleanupClosePushL(myArray);
myArray.AppendL(myOwnngArray[0]);
myArray.AppendL(myOwnngArray[1]);

CleanupStack::PopAndDestroy(&myArray);
CleanupStack::PopAndDestroy(&myOwnngArray);

```

It is important to note in Block 1 that, as `myOwnngArray` both owns and contains a list of pointers allocated on the heap, it is not sufficient to just call `Reset()` or `Close()`. This will just clear out the block of memory holding the pointers – it will not call the destructors for the objects contained in the array. For this reason, the array is pushed onto the cleanup stack using the `CleanupResetAndDestroyPushL()` function, rather than `CleanupClosePushL()`. Should the array leaves (for example, if the new call fails), then the array will have all its objects deleted and then the memory freed.

The implementation for `CleanupResetAndDestroyPushL()` may be included or copied from the implementation in `%EPOC32%\include\mmf\common\mmfcontrol\erplugiresolver.h`.

In the second code block we have a pointer array, but it does not own the pointers and so it is safe in this case to push the array onto the cleanup stack using `CleanupClosePushL()`.

When the array is destroyed the memory is released for the second array, but no attempt is made to delete the items contained in the array as it does not own the memory that the pointers refer to.

## 5.5 Dynamically creating RArray objects

Another common mistake is to allocate the `RArray` object on the heap. Actually this is a common problem that is not just restricted to the `RArray` class, but rather applies to all 'R' classes.

There are a number of reasons why this is a very dangerous thing to do, but the main reason is that it leaks resources in many mysterious ways.

```
RArray<TInt>* array = new (ELeave) RArray;
CleanupStack::PushL(array);
User::LeaveIfError(array->Append(3));
CleanupStack::PopAndDestroy(array);
```

The above code leaks when the array is removed from the cleanup stack. The answer to the problem is not obvious, however: because the array that gets pushed onto the cleanup stack is anonymous (it's a TAny pointer) and is not derived from CBase, its destructor will never be invoked but its underlying memory will be deleted. Furthermore, remembering the rules for 'R' classes, it is a requirement that the Close() be called explicitly and not in the destructor. Close()/Reset() is not called in the above example, and so the underlying blob containing the data for the array is never deleted, leading to a memory leak.

This problem is actually much worse with RPointerArray as the objects with the RPointerArray will also not be deleted, because ResetAndDestroy is not being called.

## 5.6 Copy constructors and assignment of RArray objects

It is also essential to remember that the copy constructor for 'R' classes is a bitwise copy, so it is up to the developer to perform the correct resource management.

```
RArray<TInt> array1;
... add some items to array 1
RArray<TInt> array2 = array1
... do some work

... remember array1 owns the memory, array2 is just a copy
array1.Reset();
```

In the example above, because there are effectively two objects containing the same handle or pointer to a block of memory, the developer needs to consider which of these owns the resources and which is a copy. Often it seems that when one array is assigned to another a deep copy should occur, rather than a shallow copy. This does have unfortunate side effects, particularly when one array is being thrashed heavily. This can cause the underlying blob containing the data to be reallocated to a different address so that the second array will now point to an invalid memory block and strange crashes can occur.

For the most part, it is recommended that copy operations for 'R' classes be avoided; instead, use a reference which provides the same functionality without the risk, as the reference points to the actual object and not to a copy.

## 6 Tips for get the most out of using the RArray classes

Here are some useful tips that will squeeze the most out of the array classes when using it in your own classes.

### 6.1 Adding items efficiently

If the intention is to add many items into an empty array, then in Symbian OS v9.x it is much more beneficial to call Reserve(), which will pre-allocate sufficient space for the number items specified

in the parameter. This has the effect that the underlying memory block is already allocated and so reallocation and copy is not required. This will improve runtime performance quite considerably when many items are added to the array in sequence.

## 6.2 Recovering memory

If the number of items in the array was large and then many items were removed, it will be beneficial to call `Compress()` or `GranularCompress()` to recover the memory that is no longer used. These functions will not affect any items currently in the array but will shrink the underlying memory block to either the number of items or to the next highest granularity for the current number of items.

## 6.3 Duplicating an RArray efficiently

Instead of copying an RArray item by item, it is often faster to copy the array in a memory block operation, assuming that the array to be copied contains items.

```
RArray<TMyClass> source;
CleanupClousePushL(source);

... Load source with data from somewhere ...

// now duplicate the array quickly by allocating a block of memory
TMyClass* data = User::AllocL(source.Count() * sizeof(TMyClass));
// duplicate the source array in the destination array
Mem::Copy(data, &(source[0]), source.Count() * sizeof(TMyClass));
// now create a clone of the array and transfer ownership of the array
RArray<TMyClass> destination(sizeof(TMyClass), data, source.Count());
CleanupClousePushL(destination);

// source and destination are now identical but separate copies of the same data.
```

## 6.4 Passing RArrays across processes with IPC

The standard approach to get items in an RArray from a server process to the client is to heuristically guess the number of items in the server, pre-allocate the memory and then handle the error condition if there is insufficient space to transfer all the items back. The actual number of items in the array will be returned, and so the caller knows how many items to expect.

The memory block can then be transferred to the RArray using the tip in the previous section.

### 6.4.1 Client side Code

```
TInt32 elementCount = KDefaultElementCount;
TMyClass * data = NULL;

FOREVER
{
// first check to see if there are actually elements on the server
// side as the first time through this will be set to the default
if (elementCount == 0)
break;

// Allocate the memory and a variable to hold the number of items
// Note that if there is insufficient space then the function will leave
data = STATIC_CAST(TMyClass *, User::AllocL(elementCount * sizeof(TMyClass)));
```

```

// now make these visible to the server so the server can fill them in
TPckg<TInt32> pkg(elementCount);
TPtr8 blob((TUint8*)data, elementCount * sizeof(TMyClass));

// do the IPC call
TIPCArgs args(&blob, &pkg);

// You put your own opcode here and device from RSessionBase
const TInt err = SendReceive(EOpCode, args);

// on return elementCount contains the actual number of items
if (err == KErrNone)
{
CleanupStack::Pop(data);
break;
}

// There could have been for example a server died error
// note the memory is already on the cleanup stack so
// no deallocation is necessary
if (err != KErrNoMemory)
User::Leave(err);

// There was insufficient space reserved in the client
// so free up the block, elementCount contains the actual number
// of items
CleanupStack::PopAndDestroy(data);
data = NULL;
}

// you need to handle the case where elementCount is zero!
if (elementCount > 0)
{
RArray<TExampleStruct> myArray(sizeof(TMyClass), data, elementCount);
CleanupClosePushL(myArray);
// ...
}

```

### 6.4.2 Server side Code

This will normally be called from the switch statement in the `ServiceL()` method. For example:

```

void ServerSidePCHandler(RMessage2& aMessage, const RArray< TMyClass >&
aArrayToCopy)

TInt32 clientCount = KErrNotFound;
TPckg<TInt32> pkg(clientCount);
aMessage.ReadL(1, pkg); // remember index 0 has the data, 1 has the number of items
__ASSERT_ALWAYS(clientCount > 0, User::Panic(_L("ServerSidePCHandler"), 1));

// first check we have sufficient space
if (clientCount > aArrayToCopy.Count())
{
// Make sure there are items to copy
if (aArrayToCopy.Count() > 0)

```

```

{
const TPtrC8 ptr((const TUint8*)&(aArrayToCopy[0]), sizeof(TMyClass) *
aArrayToCopy.Count());
aMessage.Wri teL(0, ptr);
}

// Now update the actual number of items copied
clientCount = aArrayToCopy.Count();
aMessage.Wri te(1, pkg);
aMessage.Compl ete(KErrNone);
}
else
{
// The client did not allocate sufficient space so
// tell the client how many items we have on the server
clientCount = aArrayToCopy.Count();
aMessage.Wri te(1, pkg);
aMessage.Compl ete(KErrNoMemory);
}
}

```

## 6.5 Avoid calling Count() in loops

Each call to Count() results in a call to the RArrayBase::Count() function, so if the array elements count does not change while executing the loop, consider either shifting the Count() function out of the loop into a new const variable, or counting down instead of up in the loop. Of course if performance is key, consider not using a loop at all – instead, use one of the techniques mentioned to keep the array sorted, then finding items is a trivial task.

## 6.6 Know your limits

The maximum structure an RArray can hold is 640 bytes, the minimum is four bytes. Where the minimum is less than four bytes it will be rounded up to four bytes.

Consider using a descriptor (such as TPtr8 or HBufC8) when working with raw memory as this will avoid alignment fault issues as well as consuming far less memory.

## 6.7 Heuristically guess the growth of your array

The granularity of the array is important: too big and there is the risk that too much memory will be unused and wasted; too small and frequent additions or insertions require the entire memory block to be reallocated resulting in substantial performance degradation. When in doubt, attempt to over- rather than underestimate the granularity that is required.

This can be somewhat mitigated by using the overloaded RArray constructor, where both granularity and the factor may be specified so that the array can grow faster if it is being used extensively. If you know your array is likely to grow quickly, then using the factor version may help reduce memory thrashing.

## 6.8 Check error codes

The RArray classes do not generally leave. From Symbian OS 9.x there are now overloads to provide a leave if the function fails – which makes the functions much safer to use.

It is often the case that values returned from array methods are not checked for failure conditions. This is especially true of the Append() function which can fail and, in doing so, can orphan

memory. It is also important to note that the `Append()` function returns `KErrNone` when the function succeeds or an error code. Many non-Symbian OS array implementations return the index of the newly added item.

The following functions return error codes:

- `Append()`
- `FindXxx()` functions
- `InsertXxx()` Functions
- `Reserve()`

## 6.9 Managing memory safely

When adding items to `RPointerArray` it is important to get the memory management correct.

The item should be pushed onto the cleanup stack, then added, and then popped off the cleanup stack:

```
CMyClass* object = new (ELeave) CMyClass;
CleanupStack::PushL(object);
User::LeaveIfError(myArray.Append(object));
CleanupStack::Pop(object);
```

Whilst this is a lot of work, in most cases C classes have a `NewLC()` function, so this can make it easier to code correctly:

```
myArray.AppendL(CMyObject::NewLC());
CleanupStack::Pop(); // remove the NewLC'd object
```

The code below illustrates two common incorrect scenarios:

- Failing to check the error code from the `Append()` function, so that ownership of the object is not transferred to the array class, resulting in a leak.

```
CMyClass* object = new (ELeave) CMyClass;
CleanupStack::PushL(object);
myArray.Append(object);
CleanupStack::Pop(object);
```

- Failing to put the object onto the cleanup stack before a leave occurs, so that the ownership of the object is not transferred to the array and a leak results.

```
CMyClass* object = new (ELeave) CMyClass;
User::LeaveIfError(myArray.Append(object));
```

## 7 Code Examples

The supplied code illustrates how to use RArray methods.

Function name	What it does
NonAlloca tingArrayL()	Create an array that does not allocate memory.
TakeOwnersh i pL()	Transfers ownership from a blob to an RArray.
ReserveAndAppendL()	Uses Reserve() to reserve space for the items then appends them to the array.
Inserti onUsi ngKeyOffsetsL()	Inserts into a sorted array using the offset functions.
Inserti onUsi ngFuncti onL()	Inserts into the array using the TLi nearOrder functions.
Fi ndUsi ngLi nearSearchL()	Finds an item using the TLi nearOrder functions.
Fi ndUsi ngAKeyL()	Finds a item using the key functionality.
SortExi sti ngArrayL()	Sorts the array using the inbuilt sort function and a TLi nearOrder.
SortExi sti ngArrayUsi ngKeyOffsetsL()	Sorts the array using the offset.
Thrashi ngExampl esL()	Shows the impact of using the incorrect granularity on an array.
All ocRArrayObj ectsL()	Leaky example showing what happens if an RArray is allocated on the stack.
Unal i gnedDataL()	Unaligned data example.

## 8 Further Information

- [Alignment issues on ARM processors](#)
- [RArray documentation](#)
- [RPointerArray documentation](#)
- [RArray and Resource Management](#)
- [Thin template idioms in Symbian](#)
- [Essential Symbian OS Booklets - Coding Standards, Coding Tips and Performance Tips](#)