Dynamic Arrays

Dynamic Arrays

Introduction

- Dynamic container classes are very useful for manipulating collections of data without needing to know in advance how much memory to allocate for their storage
- They expand as elements are added to them and do not need to be created with a fixed size

This lecture deals

- With the two families of dynamic array classes provided by Symbian OS

The fixed-length array class

- Which wraps the standard C++ [] array is also covered

Dynamic Arrays in Symbian OS

- Demonstrate an understanding of the basics of Symbian OS dynamic arrays (**`CArrayX`** and **`RArray`** families)
- Understand the different types of Symbian OS dynamic arrays with respect to memory arrangement (flat or segmented), object storage (within array or elsewhere), object length (fixed or variable) and object ownership.
- Recognize the appropriate circumstances for using a segmented-buffer array class rather than a flat array class

Dynamic Arrays in Symbian OS

The logical layout of an array

- Is like a vector

The implementation of a dynamic array either:

- Uses a single heap cell as a "flat" buffer to hold the array elements
- Allocates the array buffer in a number of segments using a doubly-linked list to manage the segmented heap memory

Dynamic Arrays in Symbian OS

Segmented buffers are preferable

- For large arrays which are expected to resize frequently
- Where elements are frequently inserted into or deleted from the array

  Repeated reallocations of a single flat buffer may result in heap thrashing and copying

Contiguous flat buffers are typically used when

- High-speed pointer lookup is an important consideration
- Array resizing is expected to be infrequent

Insertion and deletion are typically more efficient with a segmented buffer than with a flat buffer

- Since it does not require that all the elements after the modification point be shuffled into a new position.

Dynamic Arrays in Symbian OS

Symbian OS provides

- Two distinct class families for creating and accessing dynamic arrays
- The original array classes from the early days of Symbian OS are C classes

There are a number of different types of array class

- All of which have names prefixed by "**`CArray`**"
- e.g. **`CArrayFixFlat`**, **`CArrayFixSeg`** and **`CArrayVarSeg`**
- They are referred to generically as the "**`CArrayX`**" classes

The **`RArray`** and **`RPointerArray`** classes

- Were introduced at a later stage to improve efficiency

`CArrayX` Classes

The number of **`CArrayX`** classes

- Makes this array family very flexible
- But they can have a significant performance overhead (more later)
- Use is no longer encouraged

`CArrayX` Classes

For each class the naming scheme **`CArray`** prefix is followed by:

`Fix` for elements which have the same length

- And are copied so they may be contained in the array buffer

`Var` where the elements are of different lengths

- Each element is contained within its own heap cell and the array buffer contains pointers to the elements

`Pak` for a packed array where the elements are of variable length

- Elements are copied into the array buffer, each element preceded by its length information

`Ptr` for an array of pointers to **`CBase`**-derived objects

`CArrayX` Classes

Following the **`Fix`**, **`Var`**, **`Pak`** and **`Ptr`** the array class name ends with:

`Flat`

- e.g. **`CArrayFixFlat`** for classes which use an underlying flat buffer for the dynamic memory of the array

`Seg`

- e.g. **`CArrayPtrSeg`** for those that use a segmented buffer

`CArrayX` Classes

The inheritance hierarchy of the **`CArrayX`** classes is fairly straightforward

- All of the classes are C classes and thus ultimately derive from **`CBase`**

Each class is a thin template specialization

- Of one of the array base classes:

  `CArrayVarBase`

  `CArrayPakBase`

  `CArrayFixBase`

`CArrayX` Classes

`CArrayVarSeg<class T>` and `CArrayVarFlat<class T>`

- Derive from `CArrayVar<class T>`
- Which is a template specialization of `CArrayVarBase`

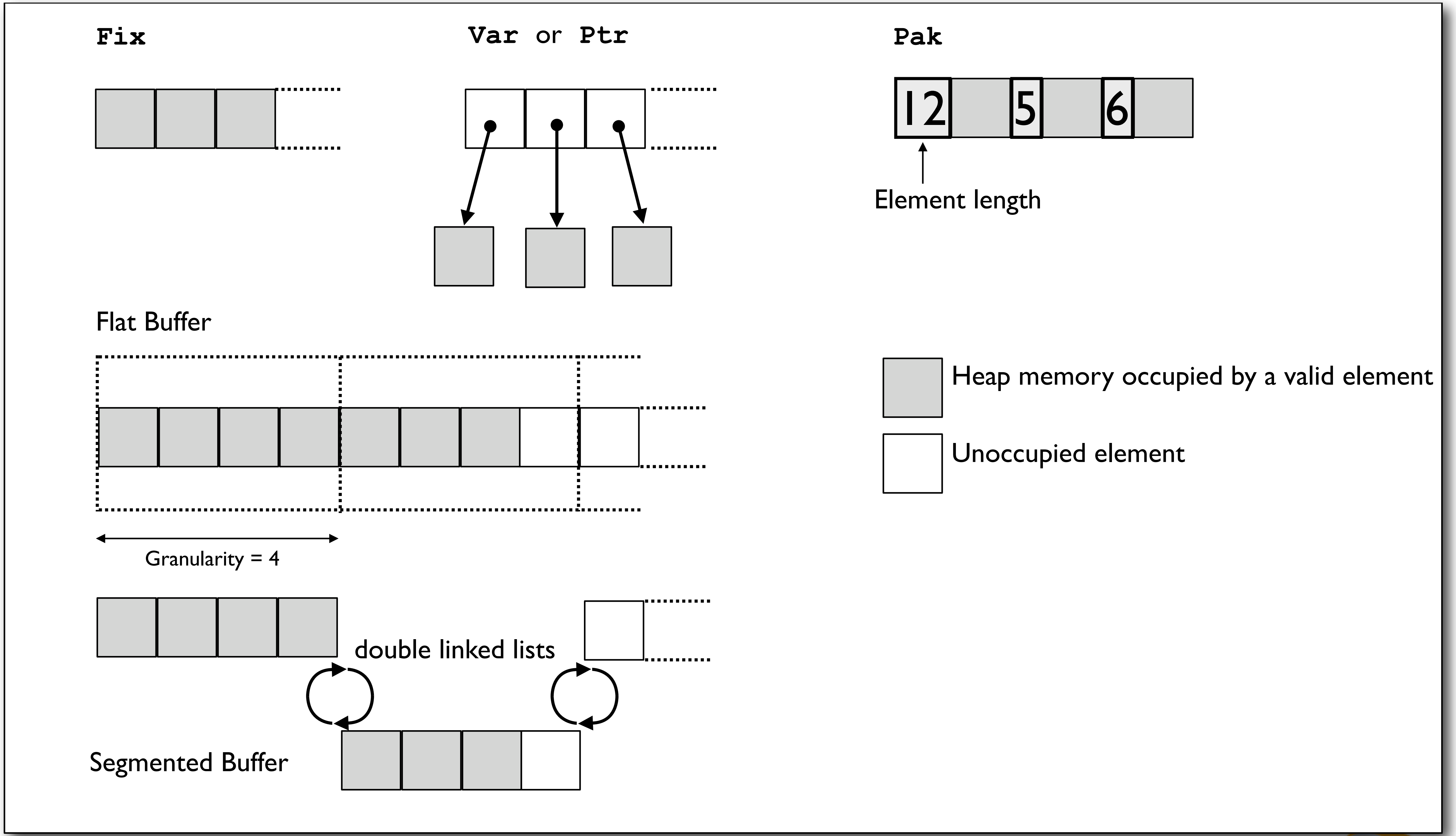
`CArrayVarBase` owns an object

- Which derives from `CBufBase` the dynamic buffer base class
- Which is used to store the elements of the array

The object is a concrete instance of

- `CBufFlat` a flat dynamic storage buffer
- `CBufSeg` a segmented dynamic buffer

Memory Layout of Dynamic Arrays



`CArrayX` Classes Available

Array Class	Description	Cleanup behavior
`CArrayFixFlat`	Elements are of fixed size and are contained in the array itself. The array occupies a single area in memory.	Elements are owned and destroyed by the array.
`CArrayFixSeg`	Elements are of fixed size and are contained in the array itself. The array occupies multiple areas (segments) of memory.	Elements are owned and destroyed by the array.
`CArrayVarFlat`	Elements are of variable size. Each element exists separately on the heap, and the array consists of pointers to those items. The array occupies a single area in memory.	Elements are owned and destroyed by the array.

`CArrayX` Classes Available

Array Class	Description	Cleanup behavior
`CArrayVarSeg`	Elements are of variable size. Each element exists separately on the heap, and the array consists of pointers to those items. The array occupies multiple areas (segments) of memory.	Elements are owned and destroyed by the array.
`CArrayPtrFlat`	Elements are pointers to `CBase`-derived objects. The array occupies a single area in memory.	Elements must be destroyed separately before array deletion by calling `ResetAndDestroy()`
`CArrayPtrSeg`	Elements are pointers to `CBase`-derived objects. The array occupies multiple areas (segments) of memory	Elements must be destroyed separately before array deletion by calling `ResetAndDestroy()`

symbian Academy

`RArray` and **`RPointerArray`**

`RArray` and **`RPointerArray`** are R classes

- Indicating that they own a resource which in this case is the heap memory allocated to hold the array

`RArray<class T>`

- Is a thin template specialization of class **`RArrayBase`**
- It comprises a simple array of elements of the same size
- The array class uses a flat vector-like block of heap memory which is resized when necessary

`RArray`

`RArray` objects

- May be either stack or heap-based
- The **`Close()`** or **`Reset()`** functions must be called to clean up properly i.e. to free the memory allocated for the array

`RArray::Close()`

- Frees the memory used to store the array and closes it

`RArray::Reset()`

- Frees the memory associated with the array
- Resets its internal state allowing the array to be reused.

It is acceptable to call **`Reset()`**

- Before allowing the array object to go out of scope
- Since all the heap memory associated with the object will have been cleaned up

RPointerArray

`RPointerArray<class T>`

- Is a thin template class deriving from **`RPointerArrayBase`**
- It comprises a simple array of pointer elements and uses flat linear memory
- Each of the pointer elements addresses objects stored on the heap
- The ownership of these objects must be considered when the array is destroyed

If the objects are owned by other components

- It is sufficient to call **`Close()`** or **`Reset()`** on the array object to clean up the memory associated with it

If the objects are owned by the array

- They are not destroyed automatically when the array is cleaned up
- Thus as part of cleanup, **`ResetAndDestroy()`** must be called to delete the heap object associated with each pointer element in the array

`RArray`, `RPointerArray` or `CArrayX`?

- Know the reasons for preferring **`RArrayX`** to **`CArrayX`**, and the exceptional cases where **`CArrayX`** classes are a better choice

`RArray`, **`RPointerArray`** or **`CArrayX`**?

There are performance implications of **`CArrayX`** dynamic arrays

- The original **`CArrayX`** classes use the **`CBufBase`** base class to access the memory allocated to the array
- **`CBufBase`** works with byte buffers and requires a **`TPtr8`** object to be constructed for every array access
- This results in a performance overhead even for a simple flat array containing fixed-length elements

RArray, **RPointerArray** or **CArrayX**?

Furthermore

- For every method which accesses the array there are a minimum of two assertion checks on the incoming parameters - even in release builds.

For example to access a position in a **CArrayFixX** array ...

a. **operator[]** calls **CArrayFixBase::At()**

b. **CArrayFixBase::At()** uses an **__ASSERT_ALWAYS** statement to range-check the index

c. **CArrayFixBase::At()** calls **CBufFlat::Ptr()**

d. **CBufFlat::Ptr()** also asserts that the position specified lies within the array buffer

`RArray`, **`RPointerArray`** or **`CArrayX`**?

A second issue

- Is that a number of the array-manipulation functions of **`CArrayX`**, such as **`AppendL()`**, can leave when there is insufficient memory to resize the array buffer

In cases where the kernel uses the dynamic arrays

- Or where the array must be called within a function which cannot leave

The leaving functions must be called in a TRAP macro to catch any leaves

- As described previously the TRAP macro has an associated performance overhead and it's undesirable to need to use one

`RArray`, `RPointerArray` or `CArrayX`?

The **`RArray`** and **`RPointerArray`** classes

- Were added to Symbian OS to provide more efficient simple flat-memory arrays

Comparing

- **`RArray`** with **`CArrayFixFlat`**
- **`RPointerArray`** with **`CArrayPtrFlat`**

The **`RArray`** and **`RPointerArray`** classes

- Have significantly better performance than **`CArrayX`** classes

`RArray` and **`RPointerArray`**

- So do not need a **`TRAP`** harness to ensure leave-safe operations when inserting or appending to the array
- Thus can be used efficiently both kernel- and user-side (more on this later)

`RArray`, `RPointerArray` or `CArrayX`?

R classes have a lower overhead than C classes because they do not need the characteristic features of a C class:

- Zero-fill on allocation
- A virtual function table pointer
- Mandatory creation on the heap

The searching and ordering functions

- Of the **`RArray`** classes were also optimized over those of the original classes

RArray, **RPointerArray** or **CArrayX**?

The **RArray** or **RPointerArray** classes should be used in preference to the **CArrayFixFlat** and **CArrayPtrFlat** classes whenever the array has the following characteristics:

- The size of an array element is bounded

  The current implementation for **RArray** imposes an upper limit of 640 bytes

- Insertions into the array are relatively infrequent

  There is no segmented-memory implementation for **RArray** or **RPointerArray** - both classes use a fixed rather than segmented memory layout

`RArray`, `RPointerArray` or `CArrayX`?

When a segmented-memory implementation is required for reasons of efficiency (e.g. large arrays which are expected to resize frequently)

- it may be more appropriate to use the **`CArrayX`** family of dynamic arrays

For this purpose

- **`CArrayFixSeg`** and **`CArrayPtrSeg`** are useful alternatives to **`RArray`** and **`RPointerArray`**

Implementation Note

For performance reasons

- **`RArray`** stores objects in the array with word (4-byte) alignment.

This means

- That some member functions do not work when **`RArray`** is used for classes which are not word-aligned (i.e. not aligned along the 4-byte boundary)
- An unhandled exception may occur on hardware that enforces strict alignment

Implementation Note

The functions affected are:

- The constructor **`RArray(TInt, T*, TInt)`**
- **`Append(const T&)`**
- **`Insert(const T&, TInt)`**
- **`Operator []`**, if the returned pointer is used to iterate through the array as for a C array

Array Granularities

- Understand the meaning of array granularity and capacity
- Know how to choose the granularity of an array as appropriate to its intended use

Array Granularities

The capacity of a dynamic array

- Is the number of elements the array can hold within the space currently allocated to its buffer

When the capacity is filled

- The array dynamically resizes itself by reallocating heap memory when the next element is added
- The <u>number of additional elements</u> allocated to the buffer is <u>determined by the granularity</u> which is specified at construction time.

All dynamic container classes

- Regardless of whether they have a segmented or flat memory layout have a granularity for reallocation

Array Granularities

It is important to choose an array granularity consistent with the expected usage pattern of the array

- Too small an overhead will be incurred for multiple extra allocations when a large number of elements is added to the array
- Too large a granularity is chosen the array will waste storage space

For example, if an array typically holds 8 to 10 objects

- A granularity of 10 would be sensible
- A granularity of 100 would be unnecessary.

If there are usually 11 objects

- A granularity of 10 wastes memory for 9 objects unnecessarily
- A granularity of 1 would also be foolish since it would incur multiple reallocations

Array Sorting and Searching

- Demonstrate an understanding of how to sort and seek in dynamic arrays
- Recognize that **`RArray`**, **`RPointerArray`** and the **`CArrayX`** family can all be sorted, although the **`CArrayX`** classes are not as efficient

Array Sorting and Searching

For the **`CArrayX`** classes

- An array key can be used to define a property of an array element by which the entire array can be sorted and searched

For example

- For an array of task elements that have an <u>integer priority</u> value and a <u>string name</u>

A key based on the <u>priority</u>

- May be used to sort the array into priority order

A key based on the <u>name</u>

- May be used to search for a task of a particular name

Array Sorting and Searching

The abstract base class

- For the array key is **`TKey`**

The following **`TKey`**-derived classes implement keys for different types of array:

- **`TKeyArrayFix`** for arrays of fixed-length elements
- **`TKeyArrayVar`** for arrays of variable-length elements
- **`TKeyArrayPak`** for arrays of packed (variable-length) elements

Array Sorting and Searching

Accessing an element by key requires the appropriate

- `TKeyArrayFix`
- `TKeyArrayVar`
- `TKeyArrayPak`

Object to be passed to:

- `Sort()`
- `InsertIsqL()`
- `Find()` or `FindIsq()` array-class member function

Array Sorting and Searching

A search can be made for elements

- For example, based on the value of a key in one of two ways:

Sequentially

- Through the array, starting with the first element performed using the **`Find()`** member function

Using a binary-search (binary-chop) technique

- Performed using the **`FindIsq()`** member function
- This technique assumes that the array elements are sorted in key sequence

Both functions

- Indicate the success or failure of the search and, if successful, supply the position of the element within the array

Array Sorting and Searching

`RArray` classes provide searching and ordering methods

- That are more efficient and easier to use than that of their **`CArrayX`** counterparts

The objects

- Contained in **`RArray`** and **`RPointerArray`** may be ordered
- Using a comparator function provided by the element class

The class typically

- Supplies a method which is used to order the objects

Which is passed

- To the **`InsertInOrder()`** or **`Sort()`** method by wrapping it in a **`TLinearOrder<class T>`** package

See the example in the ASD Primer for how to implement sort code

Array Sorting and Searching

It is also possible

- To perform lookup operations on the **`RArray`** and **`RPointerArray`** classes in a similar manner

The **`RArray`** classes

- Have several **`Find()`** methods
- One of which is overloaded to take an object of type **`TIdentityRelation<class T>`**

This object packages a function,

- Usually provided by the element class
- Which determines whether two objects of type T match

See the example in the ASD Primer for how to implement search code

`TFixedArray`

- Recognize that, when a dynamic array is not required, the **`TFixedArray`** class should be preferred over a C++ array, since it gives the benefit of bounds checking (debug-only or debug and release)

`TFixedArray`

`TFixedArray` is an alternative to dynamic arrays

- This is useful when the number of elements that will occupy an array is known at compile time
- The **`TFixedArray`** class wraps the standard fixed-length C++ array
- Adding range checking to prevent out-of-bounds access

The **`TFixedArray`** class

- Can be used as a member of a **`CBase`** class on the heap
- Or on the stack as it is a T class

`TFixedArray`

Access is automatically bounds-checked

- On both release and debug builds if the `At()` function is called

Where run-time efficiency is required in production code

- `operator[]` can be invoked instead of `At()` so access is bounds-checked in debug builds only
- A panic occurs if an attempt is made to use an out-of-range array index

`TFixedArray`

Once a **`TFixedArray`** has been allocated

- It cannot be resized dynamically, which is a disadvantage of the class

Because the allocation has been made

- Insertion within the bounds of the array is guaranteed to succeed at run-time

Which means

- There is no need to check for out-of-memory errors or leaves on array insertion
- Access to the array is fast in release mode

TFixedArray

The main drawbacks to the use of fixed-length arrays are that:

- Any additions to an array must occur at the end
- **`TFixedArray`** does not support ordering and matching

`TFixedArray`

The **`TFixedArray`** class has some useful additional functions which extend the generic C++ [] array:

- **`Begin()`** and **`End()`** for navigating the array
- **`Count()`** returns the number of elements in the array
- **`Length()`** returns the size of an array element in bytes
- **`DeleteAll()`** invokes delete on each element of the array
- **`Reset()`** clears the array by filling each element with zeroes

Dynamic Arrays

- ✓ Dynamic Arrays in Symbian OS
- ✓ **`RArray`**, **`RPointerArray`** or **`CArrayX`**?
- ✓ Array Granularities
- ✓ Array Sorting and Searching
- ✓ **`TFixedArray`**