Symbian OS

(Dynamic) Arrays

Disclaimer

- **These slides are provided free of charge at http://www.symbianresources.com and are used during Symbian OS courses at the University of Applied Sciences in Hagenberg, Austria (http://www.fh-hagenberg.at/)**
- **Respecting the copyright laws, you are allowed to use them:**
  - **for your own, personal, non-commercial use**
  - **in the academic environment**
- **In all other cases (e.g. for commercial training), please contact andreas.jakl@fh-hagenberg.at**
- The correctness of the contents of these materials cannot be guaranteed. Andreas Jakl is not liable for incorrect information or damage that may arise from using the materials.
- Parts of these materials are based on information from Symbian Press-books published by John Wiley & Sons, Ltd. This document contains copyright materials which are proprietary to Symbian, UIQ, Nokia and SonyEricsson. “S60™” is a trademark of Nokia. “UIQ™” is a trademark of UIQ Technology. Pictures of mobile phones or applications are copyright their respective manufacturers / developers. “Symbian ™”, “Symbian OS ™” and all other Symbian-based marks and logos are trademarks of Symbian Software Limited and are used under license. © Symbian Software Limited 2006.

Contents

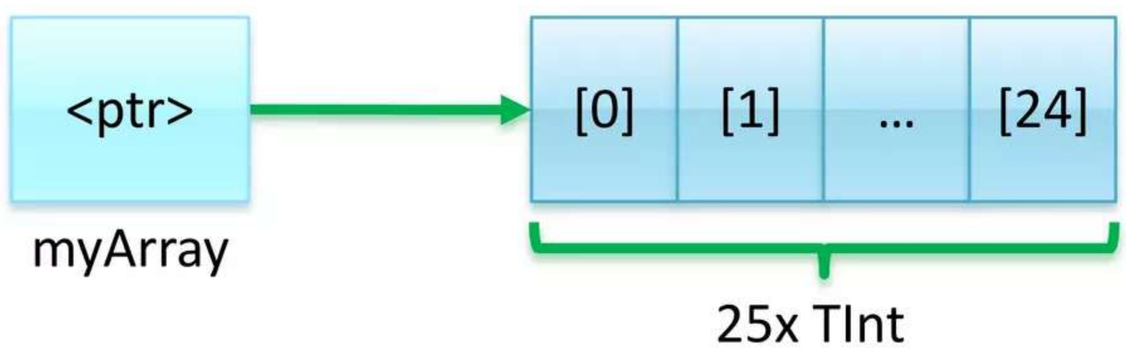
- Fixed Arrays
- Dynamic Arrays
  - RArray, RPointerArray
  - CArray*X*
- When to use R(Pointer)Array or CArray*X*

Starting simple

Fixed Arrays

Fixed Arrays

- **Standard C++ Array:**
  - TInt myArray[25];
- The same using **Symbian OS wrapper class:**
  - **TFixedArray<TInt, 25> myArray;**



TFixedArray - Advantages

- **Range checking** (Panic if out of range)
  - At(): Checks range in debug- and release-builds
  - []: Checks range in debug-builds only
- **Comfort functions**, e.g.:
  - DeleteAll(): invokes delete on each element
  - Reset(): fills each element with zeros
  - Count(): number of elements in the array
  - Begin(), End(): to navigate the array

Overview

Dynamic Arrays

Dynamic Arrays – Overview

- **Available dynamic array types:**
  - Flat / Segmented arrays
  - Pointer / direct arrays

  } **RArray*X* / CArray*X***

  - Descriptor Arrays

  } CDesCArray*X* / CPtrCArray

  - (Doubly) Linked Lists

  } TSglQue / TDblQue

  - Circular Arrays

  } CCirBuf

  - Balanced Trees

  } TBTree / TBTreeFix

- STL collections not supported (footprint too large for mobile devices)

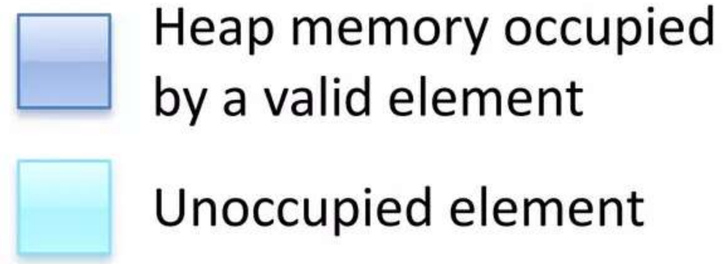
Usually the best way to store your data:

RArray / RPointerArray

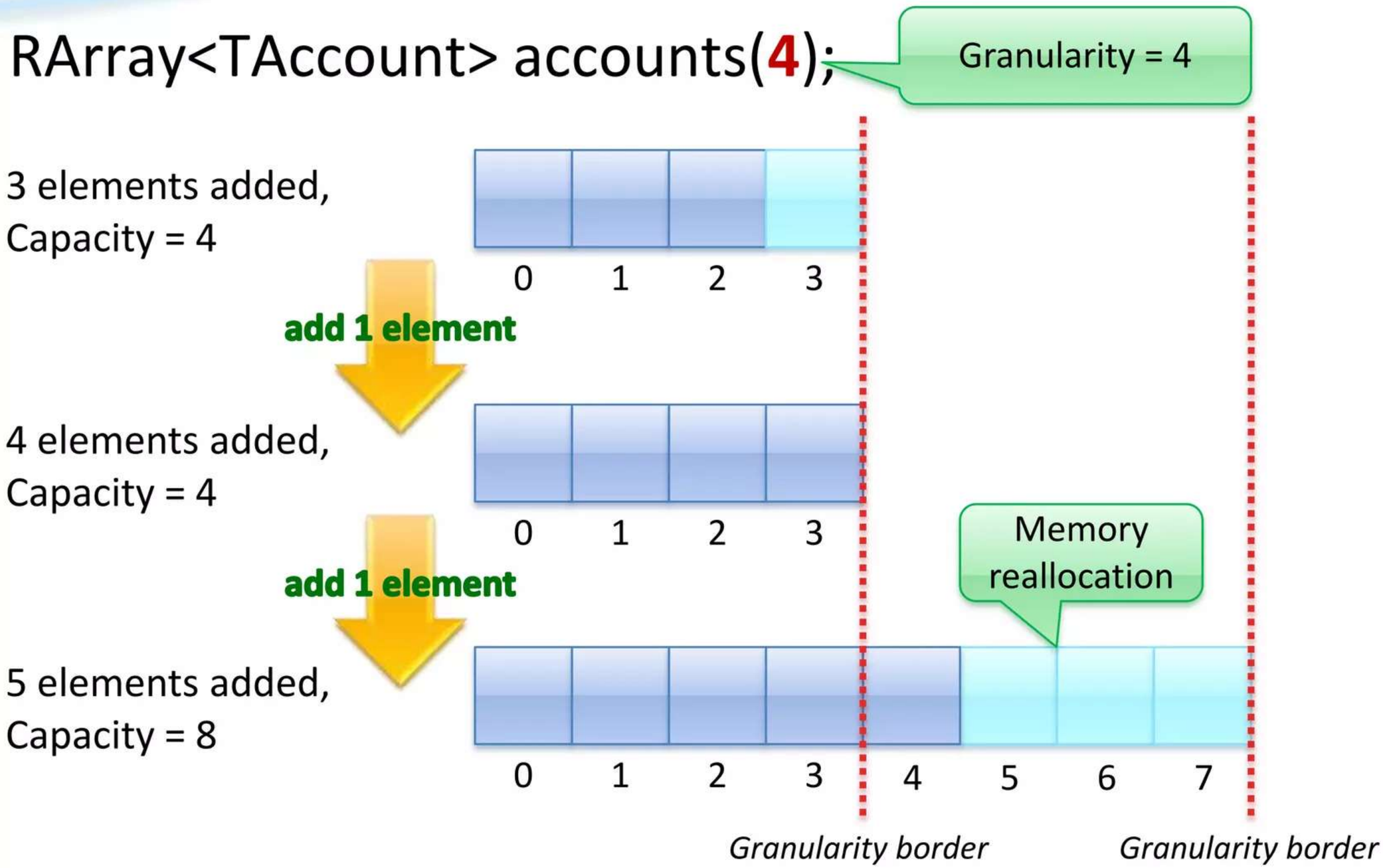
RArray

- **RArray<MyObj> myArray**
  - **Dynamic Array for fixed-length objects**
    (T- and R-type (no C-type!), max. size: 640 bytes)
  - **Features:** insertion, sorting, finding elements, …
  - **Free memory with:**
    - myArray.Close(); // Free memory and close array
    - ***or*** myArray.Reset(); // Free memory, prepare for reuse
  - Specify **granularity:**
    RArray<TAccount> accounts(4);

Granularity



- RArray<TAccount> accounts(**4**);



Granularity – Usage Pattern

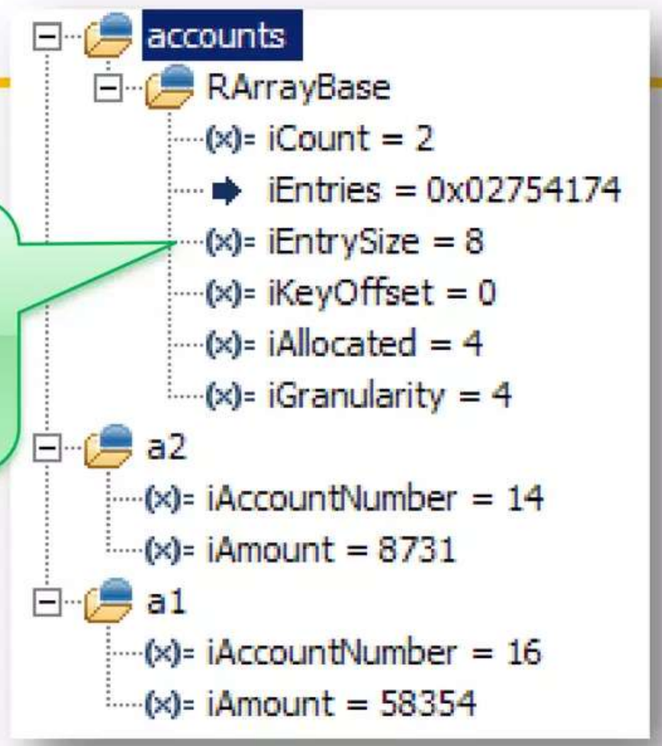
- **Choose granularity according to expected contents**
  - **Too small:** overhead through many reallocations
  - **Too large:** wastes storage space
- **Examples:**
  - Array typically holds 8 to 10 objects
    - Granularity 10 good, 100 wastes space
  - Usually 11 objects
    - Granularity 10 wastes memory for 9 objects
  - Granularity 1
    - Too many reallocations

Example: Adding Data

```
...
// Create an array which can store TAccount-objs.
RArray<TAccount> accounts(4);
// Make sure cleanup of the array's heap memory
// is done in case of a leave!
CleanupClosePushL(accounts);
// Create some test-accounts
TAccount a1(16, 58354);
TAccount a2(14, 8731);
// Add
accounts.AppendL(a1);
accounts.AppendL(a2);
// Do some stuff with the array
// ...
// Finally, do cleanup
CleanupStack::PopAndDestroy(1);
...
```

```
class TAccount {
public:
   TAccount(TInt aNumber, TInt aAmount)
      { iAccountNumber = aNumber;
      iAmount = aAmount; }
public:
   TInt iAccountNumber;
   TInt iAmount;
};
```

2 TInt-variables
(iAccountNumber, iAccount)
= 2x4 bytes
= 8 byte entry size



Inserting

- **Insert:**
  - TInt success = accounts.Insert(const T& aEntry, TInt aPos);
  - **aPos:** Insert at this position, shuffle everything up. e.g. position 0 = insert at beginning, move everything back
  - **Return value:** *KErrNone* (Success) or system wide error code

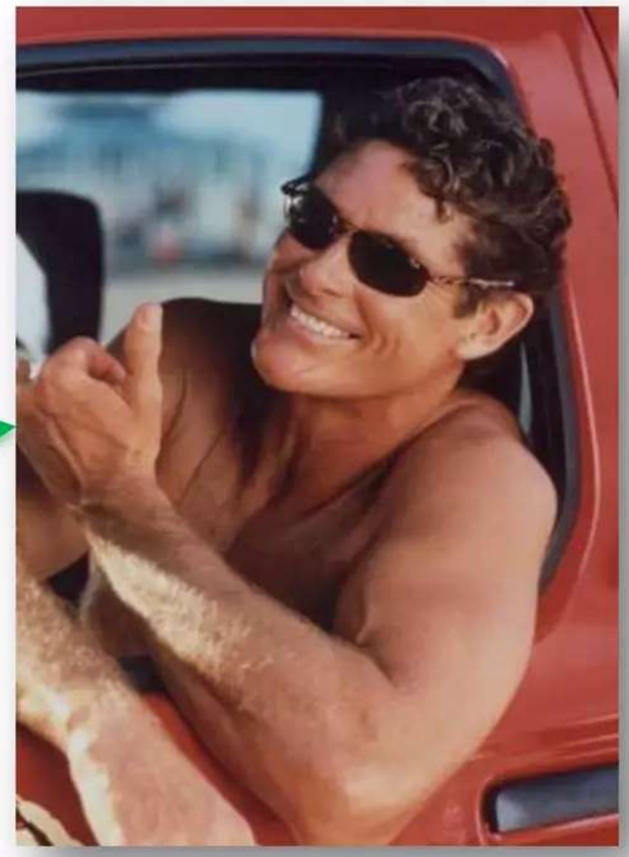
Defining an Order

- **Simple version:**
  - Define **order based on integer key**
  - Key = Data **member of array element**
  - Used for searching, inserting and sorting
  - **Define which variable to use in RArray-constructor:**
    RArray(TInt aGranularity, **TInt aKeyOffset**)

Key Offset

- **Example:**

```
class TAccount {
public:
   TAccount(TInt aNumber, TInt aAmount);
public:
   TInt iAccountNumber;
   TInt iAmount;
};
```



The HOFF!*

- **Create Array using:**

```
RArray<TAccount> accounts(4, _FOFF(TAccount, iAccountNumber));
```

... well, in reality it's the Field OFFset macro.

* Image copyright:
www.davidhasselhoff.com

Sorting

- Sort objects within the array (ascending)
- **Based on key-variable:**
  - Sort**Signed**(): for **TInt** member variable
  - Sort**Unsigned**(): for **TUint** member variable

Ordered Inserting

- **Based on key-variable:**
  - InsertIn**Signed**KeyOrder(): for **TInt** member variable
  - InsertIn**Unsigned**KeyOrder(): for **TUint** member variable
- Inserts in **ascending order** based on key
- **If key already exists:**
  - Return code: *KErrAlreadyExists*
  - Or use the […]**AllowRepeats**()-version of Insert (e.g. InsertInSignedKeyOrderAllowRepeats())

Example

- Demonstrates sorting and ordered inserting:

```
RArray<TAccount> accounts(4, _FOFF(TAccount, iAccountNumber));
CleanupClosePushL(accounts);
accounts.AppendL(TAccount(16, 58354));
accounts.AppendL(TAccount(14, 8731));
// Initial array
accounts.SortSigned();
// Array is now sorted
accounts.InsertInSignedKeyOrderL(TAccount(15, 16923));
// After ordered insert
CleanupStack::PopAndDestroy(1);
```

```
Element 0: Account: 16, Amount: 58354
Element 1: Account: 14, Amount: 8731
```

```
Element 0: Account: 14, Amount: 8731
Element 1: Account: 16, Amount: 58354
```

```
Element 0: Account: 14, Amount: 8731
Element 1: Account: 15, Amount: 16923
Element 2: Account: 16, Amount: 58354
```

Finding

- **Linear search**
  - Find(const &T aObject)
- **Binary search**
  - FindInSignedKeyOrder(const &T aObject)
  - FindInUnsignedKeyOrder(const &T aObject)
  - ... assumes that the list is currently ordered by key value

```
TInt idx = accounts.FindInSignedKeyOrder(TAccount(16, 0));
// idx == 2
```

Comparison is done based on account number.
Here: Search for account# 16

More control over the order

Advanced Comparison

Define your own order

- For **more control** over comparison process, e.g.:
  - Sort based on more than one variable
  - Sort based on non-Integer key
- **→ Own call-back function to compare two elements**
- **Returns:**
  - **Negative value:** $1^{st} < 2^{nd}$
  - **0:** $1^{st} == 2^{nd}$
  - **Positive value:** $1^{st} > 2^{nd}$

TLinearOrder<>

- Create a **TLinearOrder-object based on the element-class**
- **Function pointer** specifies which comparison function to use
- Can be used for sorting, ordered insertion and ordered finding

TLinearOrder-object is based on this class

Function pointer to the comparison function (see full example on the following slides)

```
TLinearOrder<TAccount> order(TAccount::CompareName);
// Now sort the array using your custom comparison function
accounts.Sort(order);
accounts.InsertInOrderL(newEntry, order);
```

Complete Example

Extended TAccount-class

```
class TAccount {
public:
   TAccount(const TDesC& aOwnerName, TInt aNumber, TInt aAmount);
   static TInt CompareName(const TAccount& aElement1, const TAccount& aElement2);
public:
   TInt iAccountNumber;
   TInt iAmount;
   TBuf<40> iOwnerName;
};

// Constructor
TAccount::TAccount(const TDesC& aOwnerName, TInt aNumber, TInt aAmount)
: iAccountNumber(aNumber), iAmount(aAmount) {
   iOwnerName.Copy(aOwnerName);
}

// Comparison function
TInt TAccount::CompareName(const TAccount& aElement1, const TAccount& aElement2) {
   // Use built-in comparison function of descriptor base class!
   return aElement1.iOwnerName.CompareF(aElement2.iOwnerName);
}
```

Complete Example (cont'd)

```
[...]
// Fill array with data
_LIT(KName1, "Mopius");
_LIT(KName2, "Anna");
_LIT(KName3, "Tony");
accounts.AppendL(TAccount(KName1, 16, 58354));
accounts.AppendL(TAccount(KName2, 14, 8731));
accounts.AppendL(TAccount(KName3, 15, 16923));
// Array is now sorted by order of insertion
// Create a TLinearOrder-object with a
// function pointer to our comparison function
TLinearOrder<TAccount> order(TAccount::CompareName);
// Sort the array using the comparison function
accounts.Sort(order);
// Array is now sorted alphabetically by the owner name
```

```
Element 0:
Owner: Mopius, Account: 16, Amount: 58354
Element 1:
Owner: Anna, Account: 14, Amount: 8731
Element 2:
Owner: Tony, Account: 15, Amount: 16923
```

```
Element 0:
Owner: Anna, Account: 14, Amount: 8731
Element 1:
Owner: Mopius, Account: 16, Amount: 58354
Element 2:
Owner: Tony, Account: 15, Amount: 16923
```

Compare as you like

- Similar to defining own order
- Define an **own matching function**. Returns:
  - **ETrue:** Objects are identical
  - **EFalse:** Objects are non-identical

… at least according to your own rules!

- **Find an element using:**

TIdentityRelation-object is based on this class

Function pointer to the matching function

```
TIdentityRelation<TAccount> matcher(TAccount::MatchApproxAmount);
TInt idx = accounts.Find(TAccount(KNullDesC, 0, 58330), matcher);
```

The value to search for

Find a matching element using your own matching function

Example

```
// Match the amount of both elements approximately (difference of ± 100 is allowed!)
TBool TAccount::MatchApproxAmount(const TAccount& aElement1, const TAccount& aElement2) {
   if ((aElement1.iAmount >= aElement2.iAmount - 100) &&
       (aElement1.iAmount <= aElement2.iAmount + 100)) {
           return ETrue;                    // Elements are identical based on our approximate rule
   }
   return EFalse;                           // Elements are not identical
}

// Create a matcher object that uses our approximate matching function
TIdentityRelation<TAccount> matcher(TAccount::MatchApproxAmount);
const TInt findAmount = 58330;              // Account to find should have approximately this amount of money
TInt idx = accounts.Find(TAccount(KNullDesC, 0, findAmount), matcher);

_LIT(KFindThis, "Search account with approx. amount: %d\n");
console->Printf(KFindThis, findAmount);
if (idx == KErrNotFound) {
   console->Printf(_L("No account was found\n"));
} else {
   _LIT(KFindSuccess, "Found account owned by: %S\n");
   console->Printf(KFindSuccess, &accounts[idx].iOwnerName);
}
```

Use KNullDesC instead of 0 if you don't want to specify a descriptor.

```
Element 0:
Owner: Anna, Account: 14, Amount: 8731
Element 1:
Owner: Mopius, Account: 16, Amount: 58354
Element 2:
Owner: Tony, Account: 15, Amount: 16923
Search account with approx. amount: 58330
Found account owned by: Mopius
```

RPointerArray

- **Similar to RArray**, stores pointers to objects instead of directly storing copies of them
- Can be used for **any type objects** (also C-type!)
- **(Nearly) the same functionality as RArray**
  - But does not support a key index due to complex structure of C-type objects.
  - **Always use TLinearOrder** instead! (sorting, inserting, ...)
- **Free memory with:**
  - Close() or Reset() // If array goes out of scope
  - ResetAndDestroy() // If objects are owned by the array

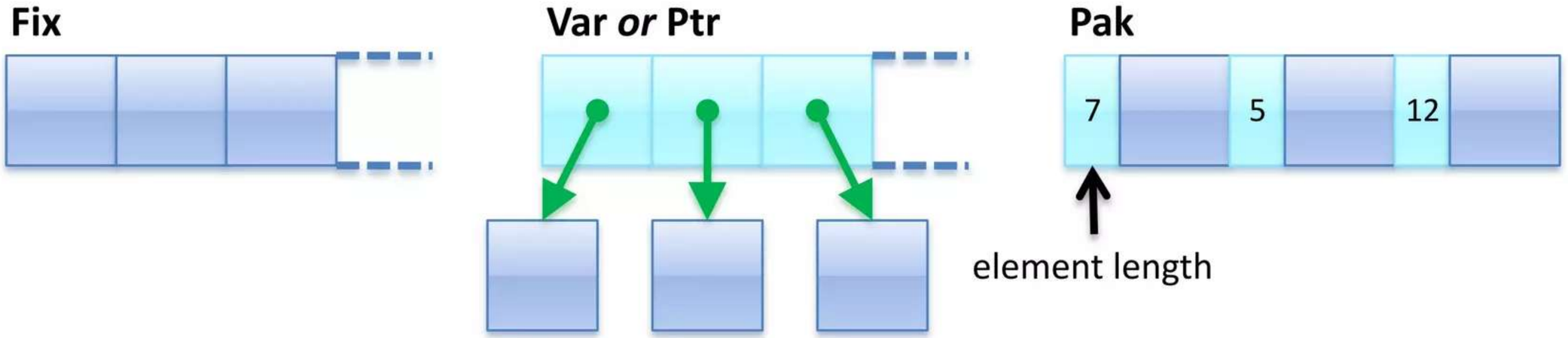
More flexible, but they have their disadvantages:

CArrays

Storing data

Array type:
CArray*XX*

- Objects can be stored in different ways:

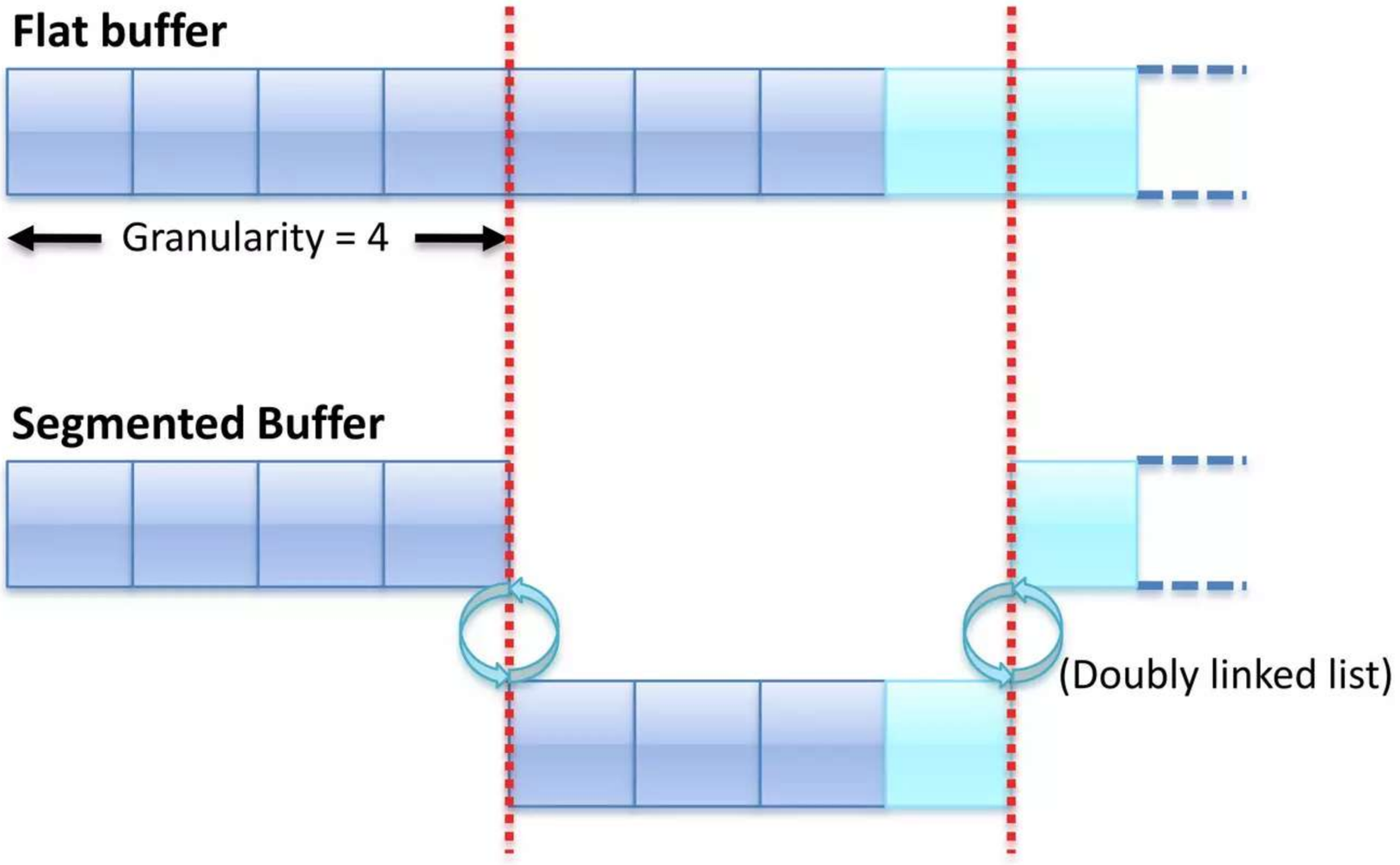


Type	Description
Fix	Objects have the same length and are copied to the array buffer.
Var	Objects are of different length and have their own heap cells. The array contains pointers.
Ptr	For an array of pointers to CBase-derived objects.
Pak	Packed array, elements are of variable length. Elements copied to buffer, preceded by length information.

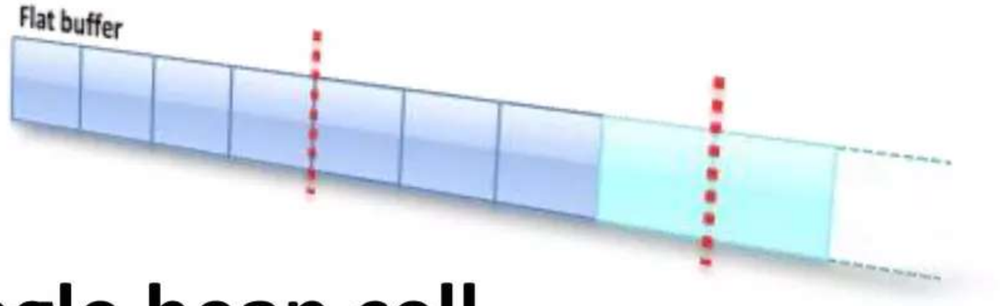
Memory Layout

Array type:
CArray*XX*

- Various memory layouts are supported:

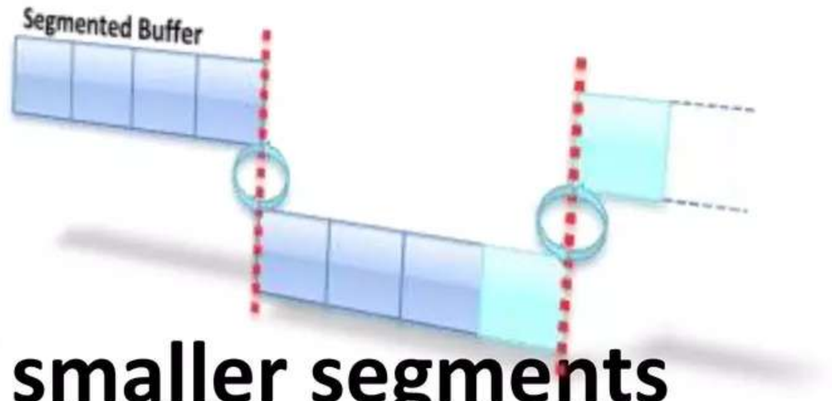


Flat / Segmented Buffers



- **Flat Buffer:**
  - Stores entire data within a **single heap cell**
  - **When it's full:**
    - During next append, data has to be transferred to a new allocated heap cell
  - **Use when:**
    - **High speed pointer lookup** is required
    - Array **resizing** is expected to be **infrequent**

Flat / Segmented Buffers



- **Segmented Buffer:**
  - Stores data in **doubly-linked list of smaller segments**
  - Each segment = separate heap cell with fixed size
  - **When it's full:**
    - New segment is allocated, old data remains in place
- **Use for/when:**
  - Large arrays which **resize frequently**
  - Elements **frequently inserted** into or **deleted** from the array

Overview

- Several combinations are possible:

Array Class	Use	Expected Reallocation	Cleanup*
CArrayFixFlat	Fixed-sized T- or R-type objects.	Infrequent	1
CArrayFixSeg	Fixed-sized T- or R-type objects.	Frequent	1
CArrayVarFlat	Variable-sized T- or R-type objects.	Infrequent	1
CArrayVarSeg	Variable-sized T- or R-type objects.	Frequent	1
CArrayPtrFlat	Pointers to objects.	Infrequent	2
CArrayPtrSeg	Pointers to objects.	Frequent	2
CArrayPakFlat	Variable-sized T- or R-type objects in one heap cell.	Infrequent	1

*** Cleanup:**

1.... Elements are owned and destroyed by the array (Reset(), Close())

2.... Elements must be destroyed separately (ResetAndDestroy())

CArray*X* vs. RArray

- **In short:**
  - **Use RArray** except when you need segmented memory!

- **Next two slides:**
  - Detailed information, important for ASD-Exam

CArray*X* vs. RArray?

- **CArray**
  - **Disadvantages:**
    - Older API, sorting etc. is **more difficult**
    - Constructs **TPtr8-object** for every array access
      → **performance** overhead!
    - **Two (!) assertion checks** for each array access
    - All **manipulation-functions** (AppendL(), InsertL(), ...) are only available with **leave** – can have disadvantages, requires TRAP for every access in functions that should not leave!
  - **Advantage:**
    - **Segmented-memory** versions available (CArrayFixSeg, CArrayPtrSeg)

CArray*X* vs. **RArray?**

- **RArray**
  - **Disadvantages:**
    - **Element-size max. 640 bytes**
    - **No segmented memory** version available
    - May have problems with non-word-aligned element size on hardware that enforces strict alignment
  - **Advantages:**
    - **Better performance** than counterparts (RArray ↔ CArrayFixFlat, RPointerArray ↔ CArrayPtrFlat)
    - R-Classes **lower overhead** than C-classes (No zero-fill on allocation, no virtual function table pointer)

Did you understand everything?

Test your Knowledge

ASD-like Question – Easy

- **Which of the following statements correctly describe the 'granularity' of Symbian OS dynamic arrays?**
  - **A.** Granularity represents the current number of elements in the array.
  - **B.** Granularity represents the maximum size of the array.
  - **C.** Granularity represents the amount by which the capacity of the array increases during a reallocation.
  - **D.** Granularity represents the maximum number of elements the array can hold without reallocation when the array is first created.
  - **E.** Granularity applies to arrays contained in either flat or segmented memory buffers.

Copyright Meme Education, 2006
http://www.meme-education.com/

Solution

- **A.** Incorrect. Number of elements is equal to or less than the granularity, as arrays are not resized for every added / removed element (except when the granularity would be 1)
- **B.** Incorrect. Dynamic arrays are automatically reallocated, the granularity defines when this happens.
- **C.** Correct.
- **D.** Correct.
- **E.** Correct.

ASD-like Question – Medium

- **Which of the following statements about Symbian OS dynamic arrays are incorrect?**
  - **A.** When searching for an element in the array the search is always started at the low index and the search will always return with the first matching element.
  - **B.** Symbian OS dynamic arrays must always be constructed on the heap.
  - **C.** The maximum size of an element stored in RArray is bounded to an upper limit of 640 bytes.
  - **D.** Only flat and not pointer type dynamic arrays can be sorted.
  - **E.** Only the CArrayX classes can be used in user-side code. RArray and RPointerArray are intended for kernel-side arrays only.

Copyright Meme Education, 2006
http://www.meme-education.com/

Solution

- **A.** Correct.
- **B.** Incorrect. RArrays can be constructed on the stack, even though they store their data on the heap.
- **C.** Correct.
- **D.** Incorrect. All dynamic array types provide methods for sorting.
- **E.** Incorrect. RArrays and RPointerArrays can be used on the Kernel-side as well. Only a few functions are not available to code running on the kernel side (see SDK-doc).

ASD-like Question

- **Which of the following are valid reasons for using one of the CArrayX classes rather than RArray or RPointerArray?**
  - **A.** When using the CArrayX classes, two assertion checks occur for every array access. As a result, CArrayX element access is less error-prone.
  - **B.** To access an element of RArray requires a TPtr8 to be constructed around it. This makes access slower than for CArrayX arrays.
  - **C.** Unlike CArrayX flat arrays, RArray stores objects in the array as word (4 byte) aligned quantities and it is possible to get an unhandled exception on hardware that enforces strict alignment.
  - **D.** Some CArrayX classes provide support for segmented memory storage.
  - **E.** The size of elements in RArray is limited to a maximum of 640 bytes. This is not the case for elements of the CArrayX array classes.

Copyright Meme Education, 2006
http://www.meme-education.com/

Solution

- **A.** Not valid. All Symbian OS arrays include range-checks. Two of them are not twice as secure...
- **B.** Not valid. It's the other way round, CArrayX-arrays require a TPtr8-object. RArrays are more efficient.
- **C.** Valid.
- **D.** Valid.
- **E.** Valid.

Try it for your own

… let’s move to the Challenges!