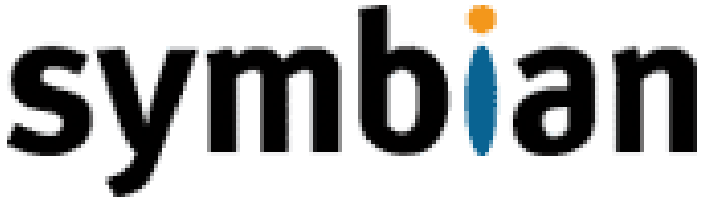
ICE503 Operating Systems



Symbian OS Introduction

Symbian Conventions

Symbian OS Classes

- Classes should have a clear role
- One class to one header file (the ideal)
- Layout of header files:
  - #include files; friend classes; public, protected, private methods; private, protected, public data.
- **T**-, **C**-, **M**-, **R**- convention to denote different class types

Fundamental Types

- *Typedefs* of built-in types for integer, floating-point, character, and pointer types (e32def.h), compiler-independent
  - **TInt, Tuint**:
    signed and unsigned 32-bit integers
  - **Tbool**:
    values ETrue (=1) and EFalse (=0)
  - **Treal**:
    Double precision (64-bit) floating point number
  - **Ttext, Tany*** :
    Pointer to anything (use instead of void*)

T Classes

- Behave like C++ built-in types
- No destructor => can be created on the stack and will be cleaned up correctly when the scope of function exits
- Contain member data which is
  - Built-in types
  - Pointers and references with “uses a” relationship
- Contain all data internally
- Example

```
struct TInitParams
 { TInt iN1;
   TInt iN2; }
```

C Classes

- **CBase** (defined in e32base.h) is the base class for all C classes.
- All C objects are allocated on the heap.
- When it is first allocated on the heap all member data will be zero-filled
- When no longer needed heap-based objects must be destroyed
- It has a virtual destructor (destroyed properly by deletion through a CBase pointer)

```
class CExample : public CBase
   { ... }
CExample* myExample = new(ELeave) CExample;
   ...
delete myExample;
```

R Classes

- **R classes** are proxies for objects usually owned elsewhere.
- There are two main motivations for this,
  - the real object is owned by a server in a different thread or address space
  - the real object's implementation must be hidden from the client
- There is no common base class for all R classes. The initialisation function has a variety of names, like *Open(), Create(), Allocate(),* etc
- The termination function has a variety of names, like *Close(), Destroy(), Free(),* etc

R Classes

```
RTimer myTimer;
RFs myFileServerSession;
RWsSession myWindServSession;

// e.g.

myTimer.CreateLocal();
   .
   .
   .
myTimer.Close();
```

M Classes

- The only form of multiple inheritance allowed by Symbian OS is where the extra classes are **Mixins**.
- Mixins (**M classes**), define an interface but do not provide an implementation of it
- They do not not have any data members.
- Only pure virtual functions
- A concrete class derived from a Mixin must implement its interface.

M Classes

```
class MRadio
{
public:
   virtual void TuneL() =0;
};

class MClock
{
public:
   virtual void CurrentTimeL(TTime& aTime) =0; };

class CClockRadio : public CBase, public MRadio, public MClock
{
public:
   void TuneL();
   void CurrentTimeL(TTime& aTime); };
```

Class members

- Data members: use prefix **i** -, should be private
- Arguments: prefix **a**-
- variables start with lower case, Functions Start With Capitals
- Setter functions: **SetThing(aThing);**
- Getter functions: **myThing = Thing();** (return it)

  GetThing(myThing); (pass it by ref.)
- Variables including arguments: **&** for uses-a and * for has-a
- Trailing **L**,**C** for functions that cause exceptions (more in Resource Management)
- Trailing **D** for functions that delete object

Symbian OS Application Framework

- The basic application consists of 4 objects of following classes:
  - **Application** (e.g. class CExampleApp)
  - **Document** (e.g. class CExampleDocument)
  - **AppUi** (e.g. class CExampleAppUi)
  - **View** (e.g. class CExampleAppView)

- In ’basic’ C/C++ console programming, there is no application framework. Instead, a **main** method or function is the ’starting’ and ending point of the program. We can consider Application object as a ’main function’ of the application.

Application

- It contains 2 basic methods.
  - **CreateDocumentL()** which creates the document object for the application.
  - **AppDllUid()** which returns the UID of the application. An UID (unique identifier) is an identifier for this application. No other application should use the same ID number.
- The OS does the following procedures, when the user selects the application from the application menu of the device:
  - Starts the application by creating an instance object of this class
  - Calls the **`CreateDocumentL`** –method

Document

- The main task of the document is to write and read data from the file system.
- Document class also instantiates engine(s)
- If files are not used by the application, document contains only one method
  - **`CreateAppUiL()`** which creates the **AppUi** object.
- The basic task of document object is to create **AppUi** object

AppUi – user interface object

- The main task of AppUi is to handle events generated by user actions. These events are mainly:
  - Menu actions. The user selects a command from an application menu. Menu events are defined in the application resource file (.rss file)
  - Key events. The user presses a key.
- The methods handling those events are as follows:
  - **HandleKeyEvent()** for key event handling
  - **HandleCommand()** for menu generated events
- The operating system takes care of calling these methods. You only have to implement the functionality in the code.

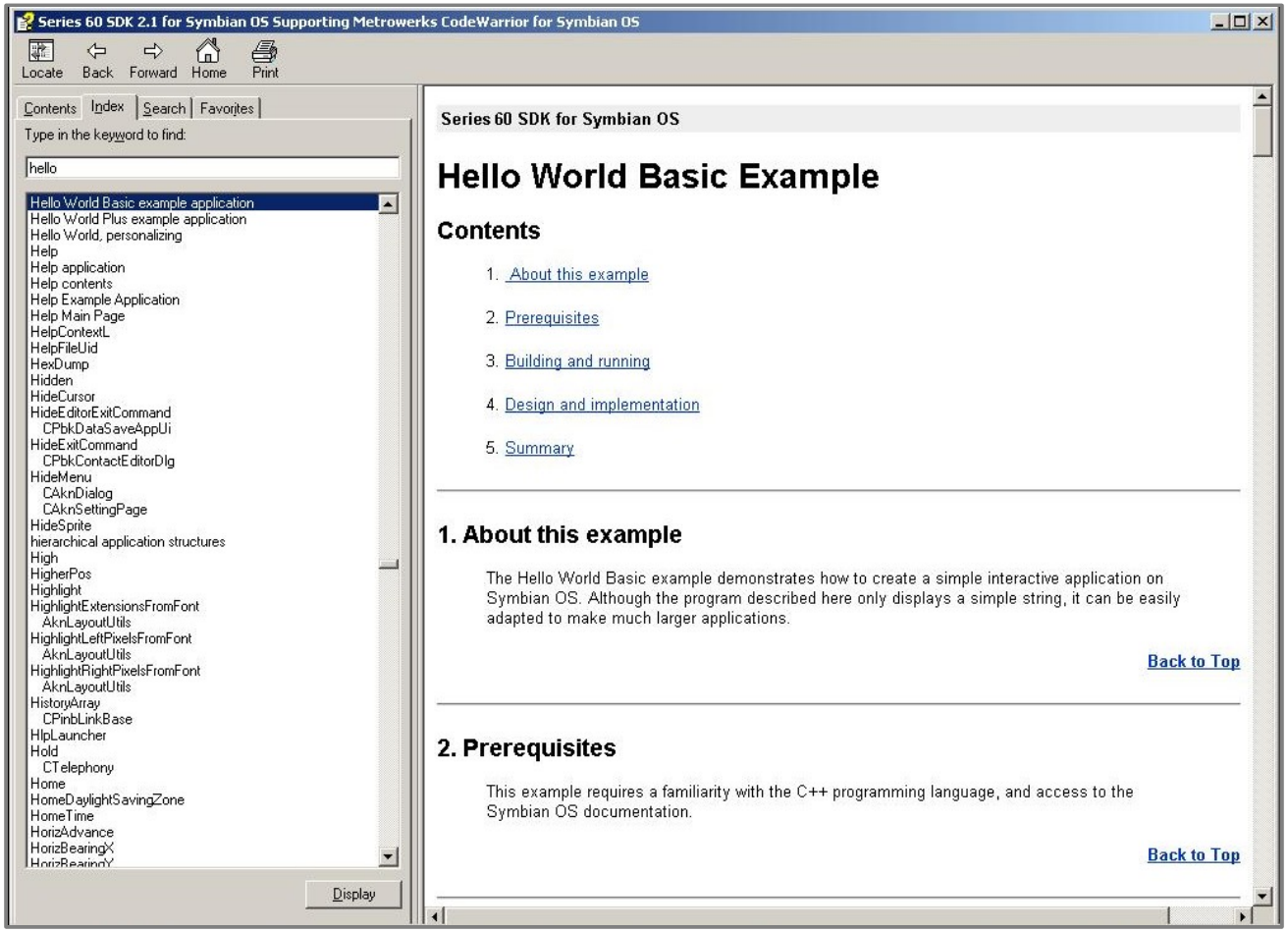
AppUi – user interface object

- AppUi class has also other methods that can you can implement, e.g.
  - **HandleForegroundEvent()** which is called when the application is switched to background (e.g. in case of incoming call)
  - **HandleSwitchOnEvent()** which is called when the device is started up.
  - etc…
- AppUi also generates one or more **Control objects** which handles Drawing and contains the user interface items.

View

- View is a conceptual term: *”the representation of the model’s data on the screen”*
- View is rendered by one or more UI controls (inherited from **CCoeControl** class)
- Parent control is sometimes called *Container*.
- The most important method is **Draw()**, where you can implement drawing by using pen, brush, shapes and colors.
- Draw –method is called by system when the drawing area needs an update or when **DrawNow()** is called from the code.
  - **NOTE:** Never call Draw() in code, if you need to update the screen, use DrawNow()

Hello World Example



About drawing

- Draw method is called by system whenever some part of the display needs to be updated (e.g. in case a 'Battery Low' notification has appeared on the display).
- Draw method gets a rectangle area to be redrawn. Parameter TRect aRect defines the area.
- Drawing area starts from the up-left corner coordinates (0,0) and ends at bottom-right coordinates (maxX, maxY) which you can get in Draw method by;

```
TInt maxY = Rect().Height();
TInt maxX = Rect().Width();
```

About Drawing

- Drawing can be made through CWindowGC object (GC = Graphics Context).
- CWindowGC offers methods for
  - drawing points, lines, ellipses, bitmaps etc...
  - change colors and styles of pen or brush
  - etc...
- Some example methods:
  - `void DrawLine(const TPoint& aPoint1,const TPoint& aPoint2);`
  - `void DrawRect(const TRect& aRect);`
  - `void SetBrushColor(const TRgb &aColor);`
  - `void SetPenColor(const TRgb &aColor);`
  - `void SetPenSize(const TSize& aSize);`
  - `void SetPenStyle(TPenStyle aPenStyle);`

About Drawing

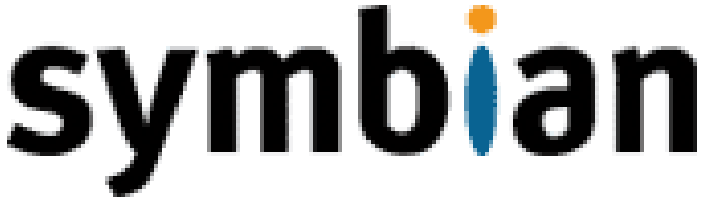
Common structs and variables needed when drawing:

- Structs
  - **TPoint** – point (e32std.h)
  - **TRect** – rectangle
  - **TRgb** – color (gdi.h)
  - **TSize** – size
- Some enumerations (gdi.h)
  - **TBrushStyle**
  - **TPenStyle**

About Drawing

```
void CHelloWorldAppView::Draw(const TRect& aRect) const
    {
   CWindowGc& gc = SystemGc();
   gc.SetBrushColor(KRgbBlue);
   gc.SetBrushStyle(CGraphicsContext::ESolidBrush);
   gc.DrawRect(aRect);
   gc.SetBrushColor( KRgbRed );
   TRect area( 20,20,50,50 );
      gc.DrawRect( area );
      _LIT (KSomeText, "Welcome to ICE503 course!");
      TPoint tLoc (40,80);
   gc.UseFont(iCoeEnv->NormalFont());
   gc.SetPenColor(KRgbRed);
   gc.DrawText(KSomeText, tLoc);
}
```

ICE503 Operating Systems



Symbian OS Introduction

Resource Management

Exceptions

- **Exception** is a runtime error that is not the programmer's fault
  - Lack of memory
  - Inability to open a socket because of dropped internet connection
  - Failure of a non-existing file to open
- **Panic** is a programmatic error, which is the programmer's fault
  - Out-of-bounds array
  - Divide by zero

Standard C++ exception handling

```
void PrintSequence(int StopNum)
   {
   int Num;
   Num = 1;
   while (true) {
      if (Num >= StopNum)
         throw Num;
      cout << Num << endl;
      Num++;
      }  }
int main(void)
   {
   try {
        PrintSequence(20);
      }
   catch(int ExNum) {
        cout << "Caught an exception with value: " << ExNum << endl;
      }
   return 0;
   }
```

Print successive integers 1, 2, 3...
Throws an integer exception when StopNum is reached.

Symbian OS exception mechanism

- The C++ standardization was in progress while Symbian OS was developed.
- **Symbian OS does not support standard C++ throw-catch mechanism** (so it does not matter if you never coded C++ exceptions)
- Symbian OS exception mechanism differs from standard C++ - **Leave process:**

•

- "Throw" exception up to the call stack to some centralized exception handler – in Symbian OS this is referred as a "leave"
- Exception handler is called **trap** or **trap harness**

Leaving

- Use of explicit leave (similar to throw in C++)
  - User::Leave()
  - User::LeaveIfError()
  - User::LeaveNoMemory()
  - User::LeaveIfNull()
- Calling a leaving function (function performing an operation that is not **guaranteed** to succeed)
  - If a function may leave, its name must be suffixed with ”L”
- Use of overloaded new(ELeave) operator

Explicit Leave

- Throwing an exception:
  - User::Leave() – simply leaves at that point (leave code passed to trap harness)
  - User::LeaveIfError() – leaves if the value passed is negative (KErrXXX constants defined in e32std.h)

```
…
RFs fileSession;
User::LeaveIfError(fileSession.Connect());
…
```

Out of memory in C++

- In 'normal' C++ you allocate an object as follows:

```
if ((myObject = new CSomeObject() == NULL) {
    //Out of memory. Handle it somehow, e.g. give error message
}
```

- Out of memory situations are very uncommon in desktop Windows environments.
- If memory can not be allocated, new operator returns NULL

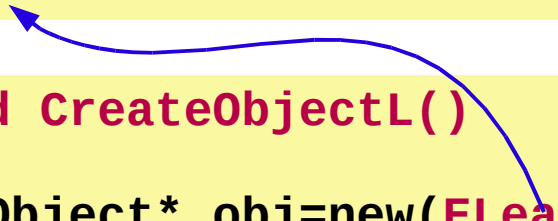
new (ELeave)

- Symbian OS has its own overloaded new operator for creating objects: **new (ELeave)**
- **new (ELeave)** is similar to new but it will call User::Leave() if insufficient memory to allocate object
- Can be called from any class
  - Implemented globally
- Next code demonstrates new (ELeave) in Symbian OS compared to standard C++:

```
//Creating object with 'standard' new
   CSomeObject* myObject = new CSomeObject;
   if (myObject == null) User::Leave(KErrNoMemory);

//Creating object with Symbian OS new (ELeave).
   CSomeObject* myObject = new (ELeave) CSomeObject;
```

Leaving on Exception

```
TInt err;
 TRAP(err,CreateObjectL());
```



```
void CreateObjectL()
  {
  CObject* obj=new(ELeave) CObject;
```

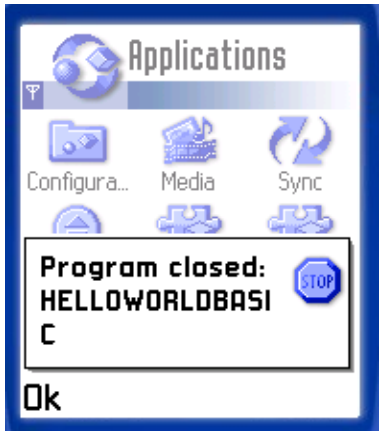
- Write code assuming success
  - if exception occurs, then Leave - call **`User::Leave()`**
- Run it within a harness that traps exceptions

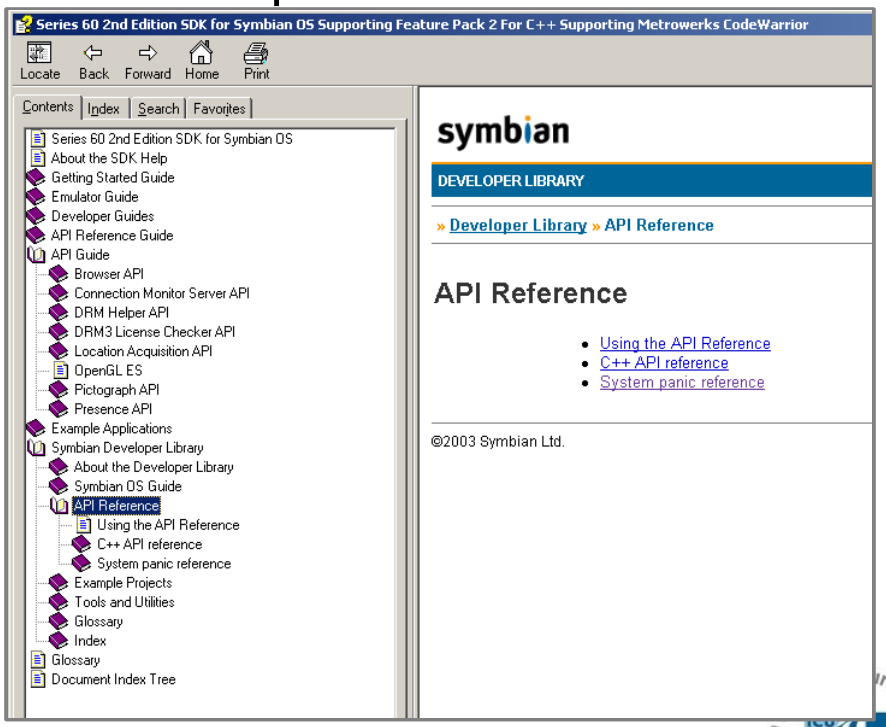
TRAP & Leave Tips

- TRAPs are expensive, never have several in sequence, end the function name in L and let caller deal with leave appropriately
- Make sure to use **new(ELeave)** not just **new**
- Every program must have at least one TRAP to catch any leaves that are not trapped elsewhere.

Panics

- Often the system will invoke a panic to indicate that something has gone seriously wrong and immediately terminate the offending application with brief message
- Developer shouldn't use panics except as a means of eliminating programming errors
- Symbian OS has a series of well-documented panic categories

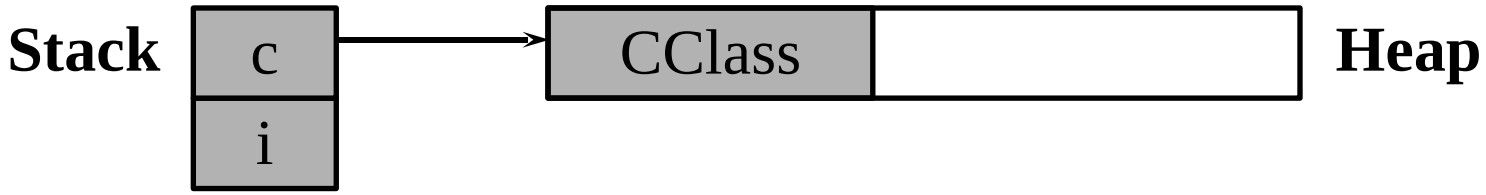




Stack & Heap Objects

- **Stack:** object deletion automatic
- **Heap:** object deletion explicit by programmer
- Pointers on stack point to heap

```
{
TInt i;
CClass* c = new(ELeave) CClass;
...
```

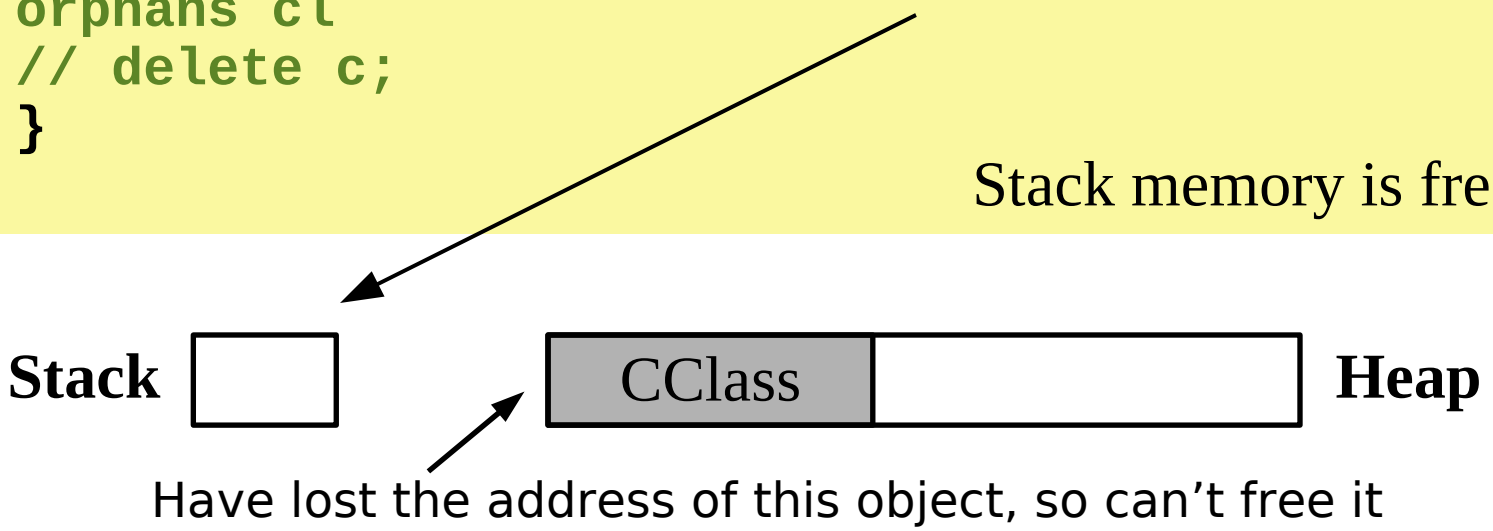


Memory Leak

Can only free heap memory if have a pointer to it

```
{
TInt i;
CClass* cl = new(ELeave) CClass;
cl->DoSomethingL(); // potential leaving function
orphans cl
// delete c;
}
```

Stack memory is freed



Solution: using CleanupStack

- CleanupStack is a stack where you can store pointers of allocated objects e.g. **cl** pointer in the scenario in previous slide.
- If a Leave occurs, the TRAP harness empties its Cleanup Stack frame by calling atomic **PopAndDestroy()** on pushed objects
  - This pops each item and calls object's destructor
- You can store pointers by **pushing** them to CleanupStack as follows:
  - **CleanupStack::PushL(cl),** pushes object **cl** to the CleanupStack.
- You can remove pointers by **popping** them from the CleanupStack as follows:
  - **CleanupStack::Pop()**
- You can pop and destroy objects in cleanup stack by calling: **CleanupStack::PopAndDestroy()**

Cleanup Stack

```
TRAPD(err,CreateObjectsL());
...
void CreateObjectsL()
{
  CObject* obj1=new(ELeave) CObject;
  CleanupStack::PushL(obj1);
  CObject* obj2=new(ELeave) CObject;
```

If second allocate fails, we have a pointer to obj1

Checking what's Popped

- Series of Pop() calls must occur in reverse order of PushL() - first in last out

```
CleanupStack::PushL(obj1);
CleanupStack::PushL(obj2);
CleanupStack::PushL(obj3);

CleanupStack::PopAndDestroy(3);
      or
CleanupStack::PopAndDestroy(3, obj1);
      or
CleanupStack::PopAndDestroy(obj3);
CleanupStack::PopAndDestroy(obj2);
CleanupStack::PopAndDestroy(obj1);
```

Two-phase construction

- **`CExample* foo = new CExample();`**
  - Call new operator, which allocates CExample object on the heap (if enough memory available) and then calls the constructor of class CExample to initialise the object
  - If CExample contructor itself leaves => memory orphaned

=> no code within a C++ constructor should ever leave

=> two-phase construction:

- **A basic constructor which cannot leave**, e.g CExample() – new operator calls this base-class constructor
- **A class method**, e.g. ConstructL – called separately once object (allocated and constructed) has been pushed onto cleanup stack

Two-phase construction

```
CClass* CClass::NewL(TInt aInt,CBase& aObj)
  {
  CClass* self=new(ELeave) CClass(aInt);
  CleanupStack::PushL(self);
  self->ConstructL(aObj);
  CleanupStack::Pop(self);
  return self;
  }
```

Two-phase construction

- Construct compound classes in two phases
  1. the normal constructor **`CClass::CClass()`** for all safe construction
  2. the second phase constructor **`ConstructL`** to safely construct things that can leave

- Factory function **`NewL`** ties two phases together
- Self contained classes only require single phase construction

Two-phase construction

```
class CClass : public CBase
  {
public:
  static CClass* NewL(TInt aInt,CBase& aObj);
  ~CClass();
private: // or protected
  CClass(TInt aInt);
  void ConstructL(CBase& aObj);
private:
  TInt          iInt;
  CSimple*      iSimple;
  CCompound*    iCompound;
  };
```

Two phase construction

1st phase

```
CClass* CClass::NewL(TInt aInt,CBase& aObj)
  {
  CClass* self=new(ELeave) CClass(aInt);
```

```
CClass::CClass(TInt aInt)
:iInt(aInt)     // do safe construction
  { ... }
```

2nd phase

```
void CClass::ConstructL(CBase& aObj)
  {
  iSimple=new(ELeave) CSimple;
  iCompound=CCompound::NewL(KVal,aObj);  }
```

```
CClass::~CClass()
  {
  delete iSimple; // delete child objects
  delete iCompound;  }
```

NewLC factory function

- **NewL** - for when pointer to object is on heap
- **NewLC** - for when pointer to object is on cleanup stack

```
iHeapPtr=CClass::NewL(val, p);
iHeapPtr->DoSomethingThatWillL();
// Parent destructor will delete CClass

CClass* stackPtr=CClass::NewLC(val, p);
stackPtr->DoSomethingL();
CleanupStack::PopAndDestroy(stackPtr);
```

Summary of two phased construction

- Create instances by calling NewL if the object to be created has one.
- NewL method calls first the constructor and so the object is created except its dynamically allocated members.
- NewL puts the created object to the CleanupStack and calls its ConstructL.
- ConstructL method creates the dynamically allocated members. (If there is no enough memory, it leaves causing NewL to leave and CleanupStack to be emptied.)
- After this the execution goes back to NewL, which pops the object from the cleanup stack and returns the pointer to the caller.
- **In the end**: Two phased construction guarantees that memory leaks will not occur at any stage of object memory allocation

Theory in a nutshell

"What if an exception occurred right here?"

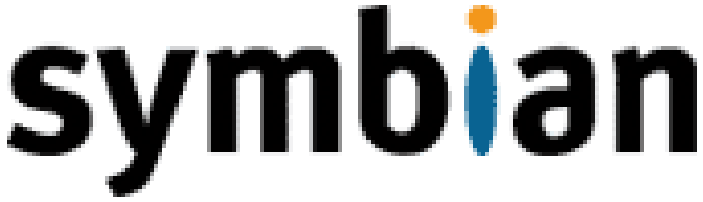
- Handle resource exceptions & Maintain pointers to heap memory, by using 3 strategies / disciplines:
  - TRAP harness & Leave: to encapsulate Exceptions that Leave
  - Cleanup Stack: to save local heap-pointers
  - Two phase construction: safe construction; exception-raising

Practice In A Nutshell

All three strategies are present in this function

```
CClass* CClass::NewL(TInt aInt,CBase& aObj)
  {
  CClass* self=new(ELeave) CClass(aInt);
  CleanupStack::PushL(self);
  self->ConstructL(aObj);
  CleanupStack::Pop(self);
  return self;
  }
```

ICE503 Operating Systems



Symbian OS Introduction

Descriptors

Introduction

- Symbian OS string is known as a “descriptor”, because it is self-describing
  - Length of the string
  - “Type” of the string – based on underlying memory layout
- Descriptors are fundamental classes
- They handle strings & binary data (unlike ‘C’ strings which must be null terminated)
- Programmer should take care of memory allocation and cleanup
- T-Descriptors behave like built-in types e.g TInt, can be safely created and orphaned on stack

Descriptor Variants

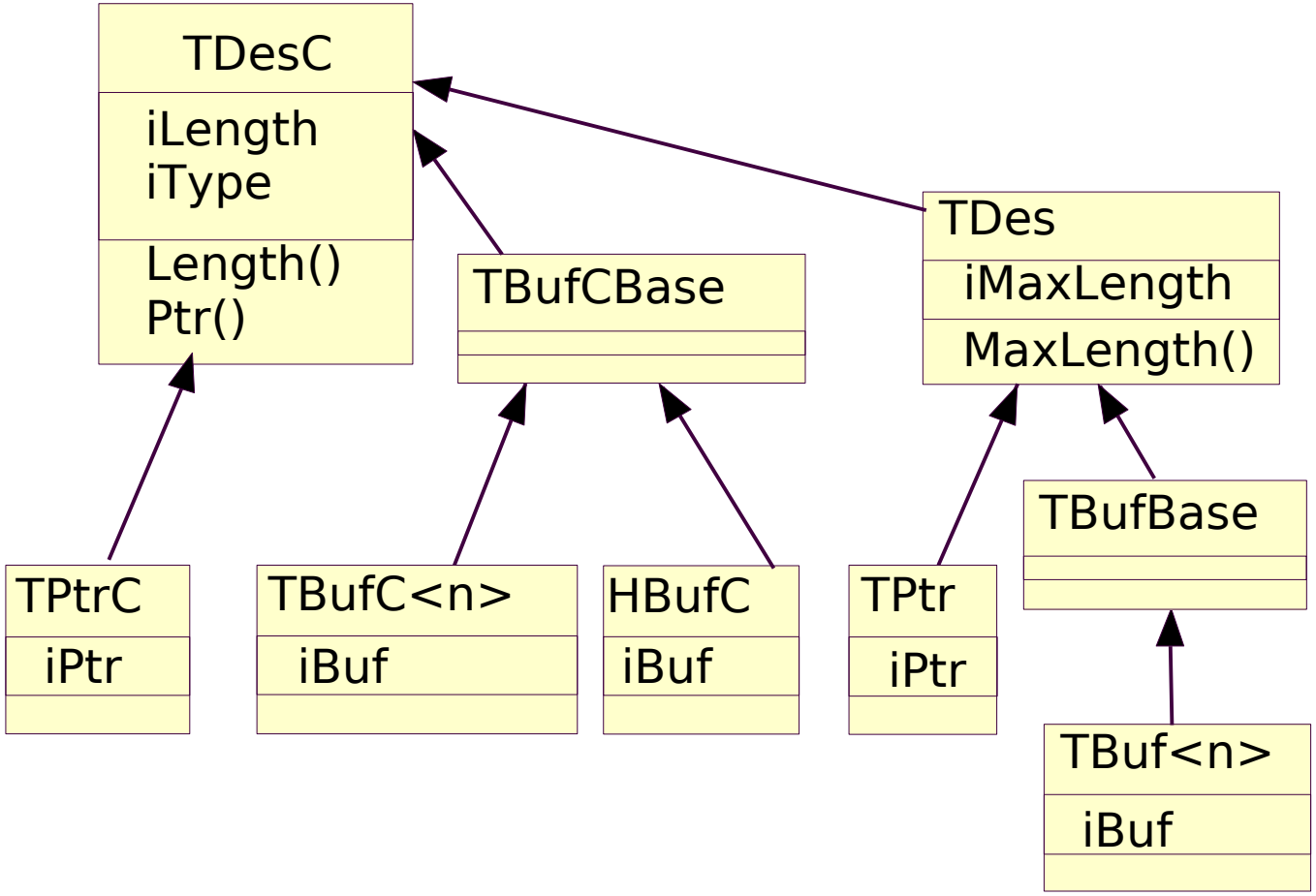
- Descriptors come in 8-bit and 16-bit variants: TPtr8, TPtr16 etc.
- Neutral classes (e.g. TPtr) are typedef'd to either narrow or wide classes depending on the build target
- A good coding convention is to use neutral classes except when working with binary data (use 8-bit descriptor classes)

The Instantiable Classes

- **TBufC** Buffer descriptor (non-modifiable)
- **TBuf** Buffer descriptor (modifiable)
- **TPtrC** Pointer descriptor (non-modifiable)
- **TPtr** Pointer descriptor (modifiable)
- **HBufC** Heap descriptor (non-modifiable, can't declare on stack)
- **_LIT** Literal Descriptors
  - Used to introduce constant string literals into program binaries

```
_LIT(KHelloDesc,"Hello");
TBufC<5> helloStack(KHelloDesc);
```

Descriptor Classes



Buffer length, size and maximum length

- **Length**: number of data items stored in buffer
- **Size**: number of bytes occupied by valid data in the buffer
- **Maximum length**: maximum number of possible data items stored in descriptor's buffer

```
_LIT(KHello, "Hello");
TBuf<12> buf(KHello);  // store "Hello"  in buffer
```

In above example:

length = 5; ***size*** = 5 (ASCII build) / 10 (unicode build) ;

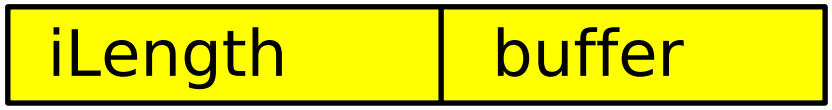
maximum length = 12 (given by templated TInt value at construction)

The out-of-bounds problem

- All descriptors carry length and maximum length information so:
  - Impossible to construct a descriptor with a length which exceeds the buffer's capacity (panic)
  - Impossible to assign data into a descriptor with a length which exceeds the buffer's capacity (panic)
  - Modifiable descriptors: TBuf, TPtr
    - Impossible to alter the length of an existing descriptor beyond iMaxLength (panic)

TBuf & TBufC

TBufC<S>



- Buffer is part of the descriptor object
- Data can be assigned at construction, or by assignment operator

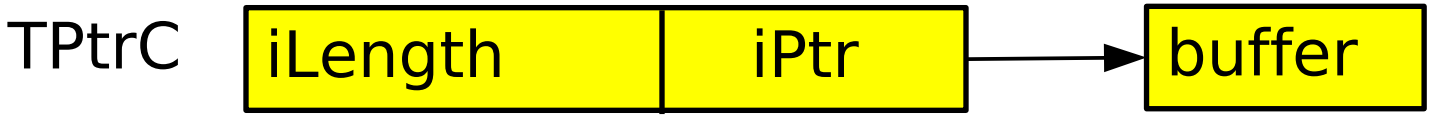
TBuf<S>

- Data can be altered and extended up to maximum length
- Set of member functions to manipulate data

Examples

```
TText cstr[12] =  {'H', 'e' ,'l' ,'l' ,'o', ' ','W',
   'o','r', 'l', 'd', '!'};
TBufC<16> bufc1(&cstr[0]);

TBufC<18> bufc4(bufc1);

_LIT(KTxtHelloWorld,"Hello World!");
TBufC<16> bufc2(KTxtHelloWorld);

_LIT(KTxtRepText,"Replacement text");
bufc2 = KTxtRepText; //can't modify data, but can
   replace it
```

TPtrC and TPtr



- Buffer separate from the descriptor
- Data represented is strictly read-only

- Data can be altered or extended up to maximum length

Examples

```
TUint8 data[7] = {0x09,0x02,0x20,0x00,0x02,0x01,0x0};
TInt bSize = sizeof(data);
TPtrC8 tP8(&data[0],bSize);

TBufC8<10>  bDesc(tP8);

TPtrC8 ptr8(bDesc);

_LIT(KTxtHW,"Hello World!");
TPtrC ptr(KTxtHW);
```

Member functions available to all descriptors

- **Locate()** locate character (returns offset)
- **Compare()** compare for =, < or >
- **Match()** pattern matching
- **Find()** search for sub-string
- **Left()** extract leftmost part
- **Right()** extract rightmost part
- **Mid()** extract portion from within (the extractors return TPtrC)

Member functions only available to TPtr and TBuf

- **Append()** Append a char
- **Lowercase()** Convert to lower
- **Uppercase()** Convert to upper
- **Insert()** Insert descriptor @ pos
- **Delete()** Delete data @ pos
- **Fold()** Fold chars
- **Collate()** Collate chars

Heap Descriptor (HBufC)

- Heap-based descriptors can be used for string data that is too big to be placed on the stack or the size of the buffer is unknown at compile time
- Heap-based descriptors can be used where they have a longer lifetime than their creator (parameter for asynchronous function)
- HBufC-class represents these descriptors
- Always referred to by pointer, HBufC*

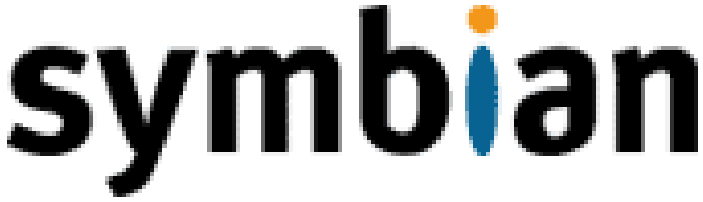
Examples

```
_LIT(KTxtHW,"Hello World!");
TBufC<14> buf(KTxtHW);

HBufC* hBuf = HBufC::NewLC(20);
TPtr ptr  (hBuf->Des()); \\Creates a Tptr to represent hbuf
ptr = buf;

HBufC* hBuf2 = buf.AllocLC();\\TDesC Alloc method to spawn an
                             \\HBufC copy of any descriptor
_LIT(KTxtHW,"Hello World!");
*hBuf2 = KTxtHW;
CleanupStack::PopAndDestroy(2, hBuf);
```

Summary

- Descriptors provide:
  - Safe way to define strings and data buffers: detecting out-of-bounds error early in development process by carrying length information
  - Space efficient way of implementing above
  - ASCII and UNICODE implementations of above
- TBufC, TBuf, TPtrC, TPtr - fundamental types: go on stack
- HBufC for large amounts of data: goes on heap
- TDesC functions available to all descriptors
- TDes functions available to TBuf and TPtr only
- Above functions behave identically for strings and binary data

ICE503 Operating Systems



Symbian OS Introduction

Dynamic Arrays

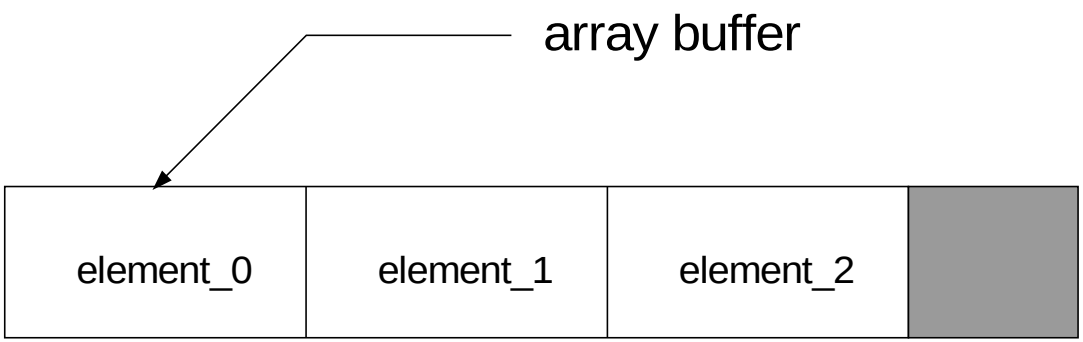
Array buffers

- Flat vs. segmented buffers
  - flat: allocate and manage storage in a single allocation cell
  - segmented: allocate and manage storage in several allocation cells
- Issues concerning choice of buffer:
  - frequency of re-allocation
  - frequency of insertion/deletion of elements
  - speed of access to buffer elements
- Fixed length vs. variable length elements
  - fixed: elements contained within array buffer
  - variable: each element contained within own heap cell; buffer contains pointers to the elements.

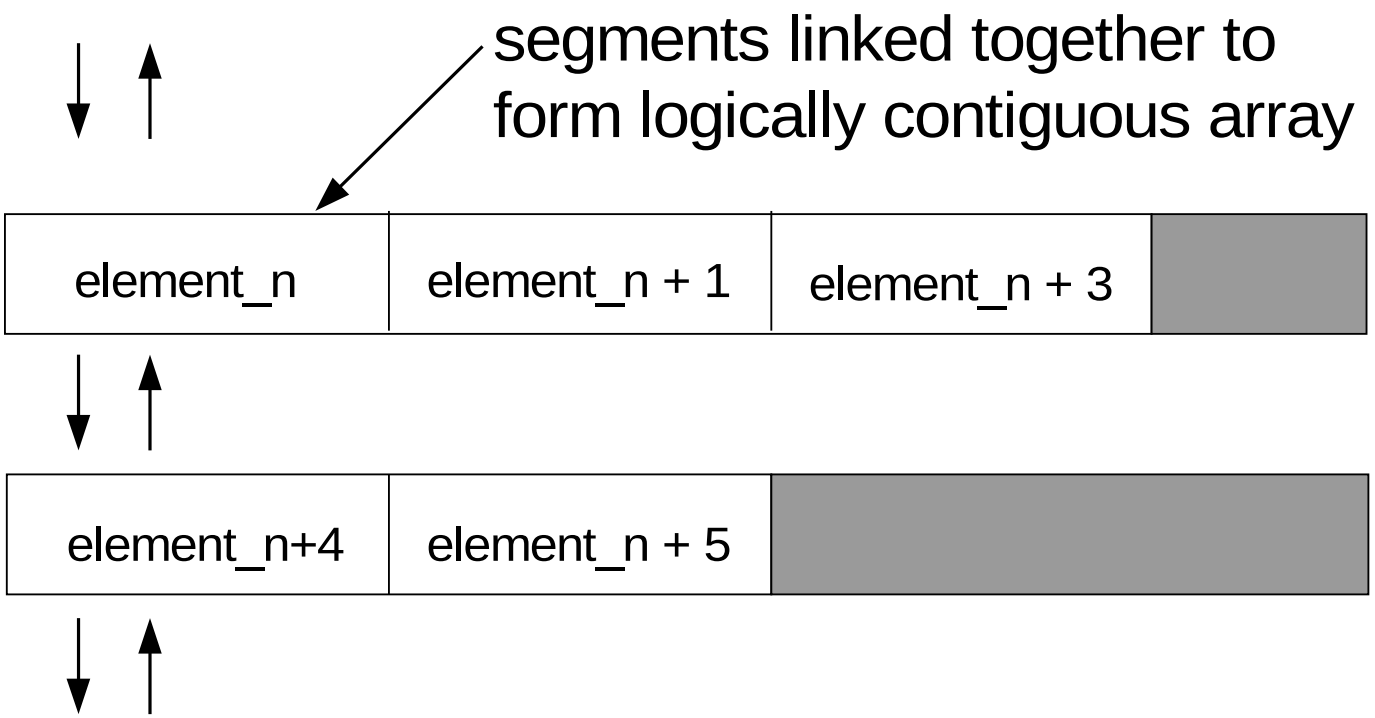
Instantiable array classes

- Elements contained in buffer:
  - **CArrayFixFlat<class T>** fixed-length, flat buffer.
  - **CArrayFixSeg<class T>** fixed-length, segmented buffer.
  - **CArrayPakFlat<class T>** packed (i.e. variable length), flat buffer.
- Elements contained in heap cells with pointers in buffer:
  - **CArrayVarFlat<class T>** variable-length, flat buffer.
  - **CArrayVarSeg<class T>** variable-length, segmented buffer.

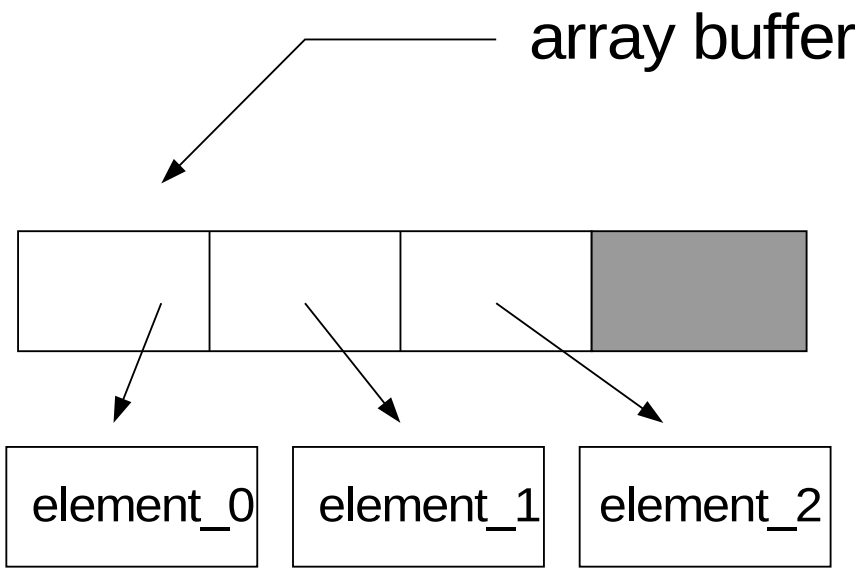
CArrayFixFlat<class T>



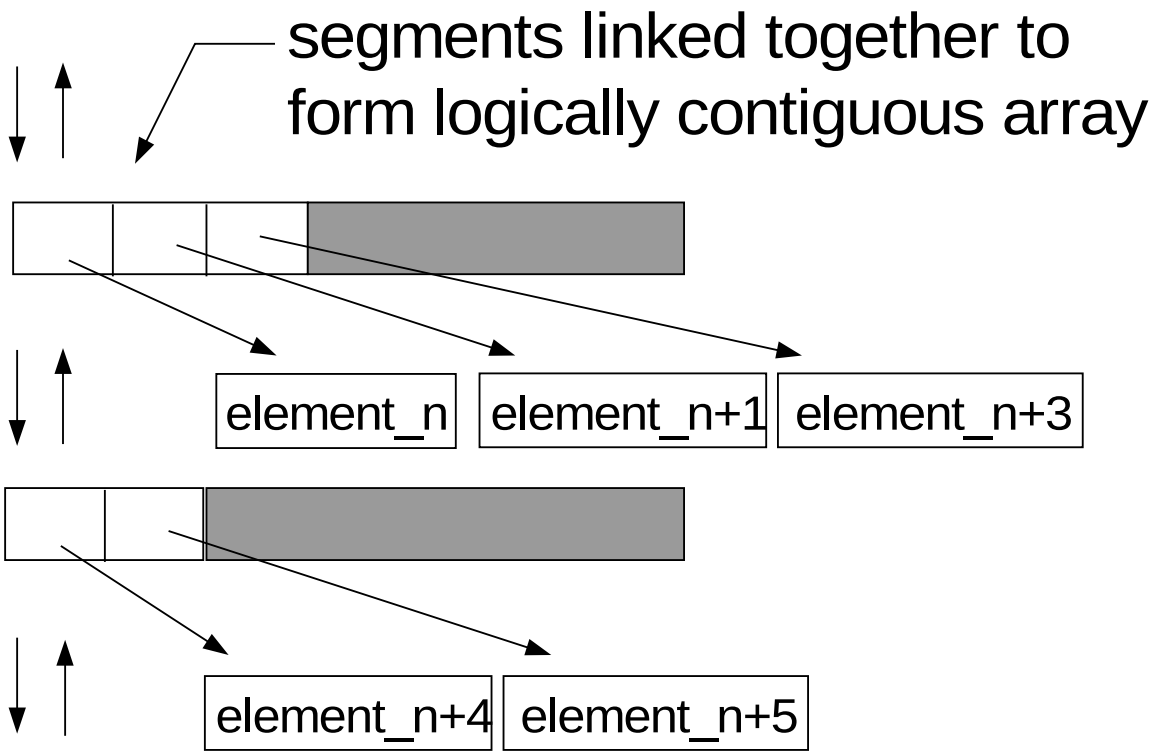
CArrayFixSeg<class T>



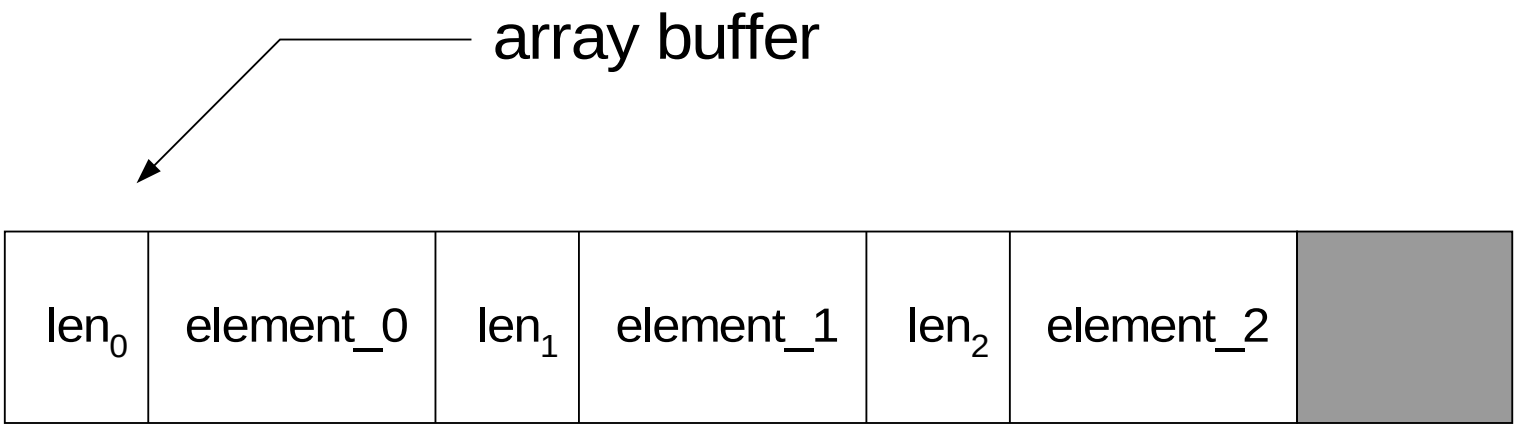
CArrayVarFlat<class T>



CArrayVarSeg<class T>



CArrayPakFlat<class T>



Array Keys

- Used to sort and find elements and locate insertion point for new elements
- 3 instantiable classes: all derived from TKey
  - **TKeyArrayFix:** used by Sort(), InsertIsqL(), Find(), FindIsq() of CArrayFix-derived classes
  - **TKeyArrayVar**: used by Sort(), InsertIsqL(), Find(), FindIsq() of CArrayVar-derived classes and CArrayPakFlat:: SortL()
  - **TKeyArrayPak:** used by InsertIsqL(), Find(), FindIsq() of CArrayVar-derived classes

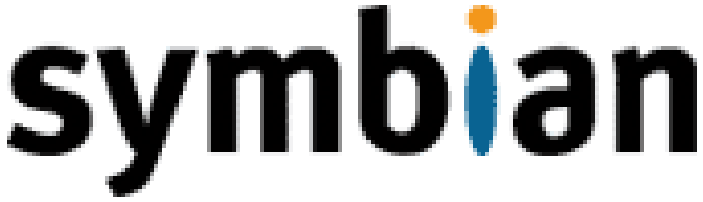
Array Keys

- Key type
  - Descriptor, text or numeric
- Offset
  - The offset of the key from the start of an array element
  - Macro _FOFF(c,f) will calculate offset (c = Class, f = field)
- Comparison:
  - Folded (no case sensitive), collated (includes special characters) or normal for text,signed or unsigned for int
- Key length
  - Implicit in descriptor, numeric types
  - Explicitly-defined for text

Sort Example

```
class TElement
   {
public :
   TElement();
public :
   TBuf<4> iData;
   };
…
CArrayFixFlat<TElement>* fixflat;
fixflat = new (ELeave) CArrayFixFlat<TElement>(3);
…
TKeyArrayFix cKey(_FOFF(TElement, iData), ECmpNormal);
TInt sRes;
sRes = fixflat->Sort(cKey); //success, if sRes = KErrNone
```

ICE503 Operating Systems



Symbian OS Introduction

Active Objects

Need for Active Objects

- Typically, Symbian OS applications are event based:
  - The application is constructed and initialised.
  - The application starts waiting for events from the user or from the OS services
  - When an event occurs, the application handles the event and starts waiting for a new event.
- Events can be generated by e.g:
  - The user
    - Key events
    - Events generated by the UI (e.g. menu)
  - Service providers
    - Timers
    - File server
    - Network servers
- The UI events are handled by AppUi but we need event handlers for other services.

Synchronous vs. asynchronous services

- In Symbian OS, most services are provided through servers, which can be accessed through the functions of R-classes (those are for example RFile or REtel).
- R-classes provide ***synchronous*** and ***asynchronous*** service functions:
  - If your application calls ***synchronous*** function, the code (the application's thread) stops until the service is completed and the function returns.
  - If your application calls ***asynchronous*** function, the function returns immediately, while the request itself is processed in the background.
- Synchronous services should be quick operations e.g. requesting a system state information which can be responded immediately.

Asynchronous services

- Many services require a lot of time to complete which makes synchronous functions unusable (thread blocked). Examples:
  - An application opens a large document.
  - An application waits for an picture to be processed.
- Instead, we use **asynchronous** service functions, where a service function returns immediately and the service is processed in the background.
- Since the application thread is not blocked, application can process other tasks like responding to user input or updating the display.
- When the request is complete, the program receives a notification, which will then be handled by e.g:
  - **Threads** (in many systems)
  - **Active objects** (in typical Symbian OS application)

Asynchronous services and multi-threaded applications

- In many systems, applications are implemented as multi-threaded processes, where asynchronous services are handled as follows:
  - For each new asynchronous task a new execution thread is spawned to handle it.
  - A scheduler makes decisions on which thread is executed.
  - A thread polls the service provider to see if the request is completed.
  - Once a service is completed the corresponding thread makes the required actions.
- Disadvantages of multi-threaded practice:
  - Multiple threads lead to increasing number of context switches increasing system overhead.
  - Programmer need to take care of synchronization, deadlock and other process management issues making programming more complex.

Asynchronous services and Active Objects

- A typical Symbian OS application is implemented as a single thread process which can handle multiple asynchronous services.
- The technique is **cooperative multitasking** where there is a **wait loop** going through the outstanding task requests.
- Once the wait loop finds a completed task, it calls the event handler code of the corresponding **handler object**.
- This is done by using **active object** framework where each asynchronous service request has an active object waiting the request to be completed.
- In Symbian OS:
  - The wait loop is implemented as an **Active Scheduler.**
  - Handler objects are implemented as **Active Objects.**

Active Object Classes

- **Active Object:** derived from **CActive** class. Encapsulates:
  - Asynchronous request (e.g. to server)
  - Application's handler for the completed request
  - Plus functionality for cancellation of the request
- **Active Scheduler:** derived from **CActiveScheduler** class.
  - Schedules Active Objects non-pre-emptively within an application (AOs have priorities but cannot pre-empt each other)
  - Encapsulates waitloop processing of events
  - One per thread

The Active Scheduler

- The Active Scheduler is implemented by **CActiveScheduler.**
- The Active Scheduler maintains a list ordered by priority, of all Active Objects of the application.
- The Active Scheduler implements the wait loop for an application thread. The wait loop goes through the iStatus boolean flags of Active Objects of **iActive** flag set on.
- If the iStatus is other than **KRequestPending** (meaning that the request is completed), the Active Scheduler calls the **RunL** method of the Active Object.

Active object scheduling

- Active object mechanism makes possible to create multi-asynchronous event based application with one thread.
- Active object mechanism uses pre-emptive scheduling inside the thread. **RunL** pre-empties all other code in the thread until it is completed.
  - Too long RunLs might lead to problems so try to keep it short.
- On OS –level there are, of course, processes that might execute during your RunL and so delaying the execution of your code.

Active object priorities

- If more than one active object gets the service completed in the same time, the Active Scheduler chooses the one with the highest priority. The priorities are:

EPriorityIdle	A low priority, useful for active objects representing background processing
EPriorityLow	A priority higher than EPriorityIdle but lower than EPriorityStandard
EPriorityStandard	Most active objects will have this priority.
EPriorityUserInput	A priority higher than EPriorityStandard; useful for active objects handling user input
EPriorityHigh	A priority higher than EPriorityUserInput

- You can set the priority in constructor.

Active Objects

- Active Objects are classes which request asynchronous services and once completed, handle the requests. They contain:
  - A data member representing the status of the request (iStatus)
  - A handle on the asynchronous service provider (R-class object).
  - Connection to the service provider (established during the construction through the R-class object).
  - The user-defined function to issue the asynchronous request.
  - The handler function to be called by the Active Scheduler when the request completes (RunL).
  - The function to cancel an outstanding request (Cancel).
- An application can contain 0-n Active Objects.
- Active Objects are implemented by deriving CActive class.

Summary

- Active Objects used instead of multi-threading: non pre-emptive scheduling within a single-threaded process
- Active Objects avoid the use of mutexes for data sharing
- One **CActiveScheduler** per thread
- CActive-derived concrete classes must implement **RunL()** and (and also another one called DoCancel())
- Note: real-time/time critical tasks should be implemented using threads and processes