File Server and Streams

File Server and Streams

This Lecture Examines

- The file system server
- Streams and stores

File Server and Streams

The File system server

- Known simply as the file server
- Handles all aspects of managing files and directories
- Provides a consistent interface across the ROM, RAM, Flash memory and removable-media devices
- The file server runs as a process **`EFILE.EXE`**
- The client-side implementation classes supplied by **`EFSRV.DLL`**

File Server and Streams

As the file server contains the loader

- That loads executable files (DLLs and EXEs) from the data-caged **`\sys\bin`** directory
- The file server is part of the trusted computer base (TCB)
- More on Platform Security in a later lecture

The Symbian OS File System

- Understand the role of the file server in the system
- Know the basic functionality offered by class **`RFs`**
- Recognize code which correctly opens a fileserver session (**`RFs`**) and a file subsession (**`RFile`**) and reads from and writes to the file
- Know the characteristics of the four **`RFile`** API methods which open a file
- Understand how **`TParse`** can be used to manipulate and query file names

File Server Session Class

The file server provides

- The basic services that allow calling code to manipulate drives, directories and files

In order to use the file server

- A caller must first create a file server session
- Represented by an instance of the **`RFs`** class

The general pattern

- For connecting to the file server is using the **`RFS`** session
- To create and use an **`RFile`** subsession
- Then releasing both the session and subsession
- As demonstrated in the following code example

File Server Session Class

Connect the session
Closes **fs** if a leave occurs
Create a file
A subsession which represents a file, as below
Closes file if a leave occurs
Submit a read request using the subsession
Clean up the **RFile** subsession and **RFs** session
This calls **RFile::Close()** on file
and **RFs::Close** on **fs**

```
RFs fs;
User::LeaveIfError(fs.Connect());
CleanupClosePushL(fs);
...
_LIT(KASDExampleIni, "c:\\ASDExample.ini");
RFile file;
User::LeaveIfError(file.Create(fs, KASDExampleIni,
          EFileRead|EFileWrite|EFileShareExclusive));
CleanupClosePushL(file);
TBuf8<32> buf;
...
User::LeaveIfError(file.Read(buf));
...

CleanupStack::PopAndDestroy(2, &fs);
```

File Server Session Class

The code example uses the cleanup stack

- To ensure that the resources associated with the open file server session and file subsession are leave-safe

Note: If the sessions objects are members of a class

- It is not necessary to use the cleanup stack to protect them as the class destructor will ensure the session and subsession are closed

If a file is not closed explicitly

- By using **`RFile::Close()`**
- It will be closed when the server session associated with it is closed
- But it is good practice to clean up any file handle when it is no longer required

File Server Session Class

A connected **RFs** session

- Can be used to open any number of files or directories (as subsessions)
- Or to perform any other file-related operations

A file server session

- Can be kept open for the lifetime of an application

The **RFs** class

- Provides many useful file system-related operations

Including the following ...

The **RFs** class

`Delete()` and **`Rename()`**

- Used to delete or rename the file specified

`Replace()`

- Used to move a file to a different location

`MkDir()`, **`MkDirAll()`**, **`RmDir()`** and **`Rename()`**

- Used to create, remove and rename the directories specified

`Att()`, **`SetAtt()`**, **`Modified()`** and **`SetModified()`**

- Used to read and modify directory and file attributes
- Such as hidden, system or read-only flags

symbian Academy

The **`RFs`** class

`NotifyChange()`

- An asynchronous request for notification of changes to files, directories or directory entries

`NotifyChangeCancel()`

- Used to cancel the outstanding request

`Drive()`, `SetDriveName()`, `Volume()` and **`SetVolumeLabel()`**

- Used to manipulate drive and volume names

`ReadFileSection()`

- Used to “peek” at file data without opening the file

The **`RFs`** class

`AddFileSystem()`, **`MountFileSystem()`**, **`DismountFileSystem()`** and **`RemoveFileSystem()`**

- Used to dynamically add and remove file system plug-ins
- That extend the file server types Symbian OS can support

Examples of potential file system plug-ins include

- Support for a remote file system over a network
- Encryption of file data before it is stored

The plug-in file system modules

- Are implemented as polymorphic DLLs of **`targettype fsy`**

File Handle Class

The **`RFile`** class

- Is a subsession of an RFs client session to the file server

An **`RFile`** object

- Represents access to a named, individual file

Providing functions to

- Open, create or replace the file
- Or to open a temporary file
- To read from and write to the file

The **RFile** Class

`RFile::Open()`

- Used to open an existing file; an error is returned if it does not already exist

`RFile::Create()`

- Used to create and open a new file
- An error - **`KErrAlreadyExists`** - is returned if the file already exists

`RFile::Replace()`

- Creates a file if it does not yet exist
- Deletes an existing version of the file and replaces it with an empty one if it does exist

`RFile::Temp()`

- Opens a new temporary file and assigns a unique name to it

File Handle Class

A common pattern

- Is to call **Open()** to attempt to open an existing file
- Then call **Create()** if it does not yet exist

For example

- When using a log file, an existing log file should not be replaced
- But simply have data appended to it:

```
RFile logFile;
TInt err=logFile.Open(fsSession,fileName,shareMode);

if (err==KErrNotFound)
    err=logFile.Create(fsSession,fileName,shareMode);
```


symbian Academy

File Handle Class

When opening a file

- A bitmask of **`TFileMode`** values is passed
- Indicating the mode in which the file is to be used
- Such as for reading or writing

The share mode

- Indicates whether other **`RFile`** objects can access the open file
- And whether this access is read-only
- i.e. files may be opened exclusively or shared

For shared files

- A region may be locked using **`RFile::Lock()`** to claim temporary exclusive access to a region of the file
- Unlocked using **`RFile::Unlock()`**

File Handle Class

When a file is already open for sharing

- It can only be opened by another program using the same share mode as the one in which it was originally opened

For example

- To open a file as writable and shared with other clients:

```
RFile file;
_LIT(KFileName,"ASDExample.ini");
file.Open(fsSession,KFileName,EFileWrite|EFileShareAny);
```

- If another **`RFile`** object tries to open **`ASDExample.ini`** in **`EFileShareExclusive`** or **`EFileShareReadersOnly`** mode access is denied
- It can only be accessed in **`EFileShareAny`** mode
- Or through use of the **`RFs::ReadFileSection()`** method ...

File Handle Class

`RFs::ReadFileSection()` method

- Can read from a file without opening it
- Thus the contents of a file can never be truly locked
- Either through use of **`RFile::Open()`** methods with **`EFileShareExclusive`** flags
- Or by calling **`RFile::Lock()`**

`RFs::ReadFileSection()`

- Is used by **`Apparc`** and the recognizer framework to determine the type of a file by rapid inspection of its contents.

File Handle Class

The **`RFile::Write()`** methods

- Write data from a non-modifiable 8-bit descriptor object - **`const TDesC8&`**

The **`RFile::Read()`** methods

- Read data into an 8-bit descriptor - **`TDes8&`**

Both **`Read()`** and **`Write()`** methods

- Are available in synchronous and asynchronous forms

Although

- Neither the asynchronous **`Read()`** nor the asynchronous **`Write()`** method can be cancelled

File Handle Class

Open ASDExample.ini

```
_LIT(KASDExample,"c:\\ASDExample.ini");
RFile file;
User::LeaveIfError(file.Open(fs, KASDExample,
              EFileShareExclusive|EFileWrite));
```

Write to the file

```
_LIT8(KWriteData,"Hello ASD");
file.Write(KWriteData);
```

Read from the file

readBuf contains "**Hello**"

```
TBuf8<5> readBuf;
file.Read(readBuf);
file.Close();
```

File Handle Class

There are several variants of

- `RFile::Read()` and `RFile::Write()`

There are overloads which allow

- The receiving descriptor length to be overridden
- The seek position of the first byte to be specified
- Asynchronous completion
- Or combinations of these

In all cases

- 8-bit descriptors are used ...

Use of 8-bit descriptors

As a consequence

- Of using 8-bit descriptors

`RFile` is not particularly well suited

- To reading or writing the rich variety of data types that may be found in a Symbian OS application

This is not an accident!

- But a deliberate design decision to encourage the use of streams
- Which provide the necessary functionality and additional optimizations

File Name Manipulation

Files on Symbian OS

- Are identified by a file name specification which may be up to 256 characters in length

A file specification consists of:

- A device, or drive, such as c:
- A path such as `\Document\Unfiled\` where the directory names are separated by backslashes (\)
- A file name
- An optional file name extension, separated from the file name by a period (.)

File Name Manipulation

Symbian OS applications

- Do not normally rely on the extension to determine the file type
- They use one or more UIDs stored within the file
- To ensure that the file type matches the application

Subject to the overall limitation of 256 characters

- A directory name, file name or extension may be of any length

The `RFs::IsValidName()` method

- Returns a boolean value to indicate whether a path name is valid

File Name Manipulation

The Symbian OS file system

- Supports up to 26 drives, from a: to z:

On Symbian OS phones

- The z: drive is always reserved for the system ROM
- The c: drive is always an internal read–write drive
- On some phones may have limited capacity

Drives from d: onwards

- May be internal or may contain removable media

It may not be possible to write to all such drives

- Many phones have one or more read-only drives in addition to z:
- That are used only by the system

File Name Manipulation

The file system

- Preserves the case of file and directory names
- All operations on those names are case-independent
- This means that there cannot be two or more files in the same directory
- With names which differ only in the case of some of their letters.

File names

- Are constructed and manipulated using the **`TParse`** class and its member functions
- For example, to set an instance of **`TParse`** to contain the file specification **`c:\Documents\Oandx\Oandx.dat`**:

```
_LIT(KFileSpec, "c:\\Documents\\Oandx\\Oandx.dat");
TParse fileSpec;
fileSpec.Set(KFileSpec,NULL,NULL);
```

The **`TParse`** Class

Following this code

- The **`TParse`** getter functions can be used to determine the various components of the file specification

For example:

```
filespec.Drive(); // returns the string "c:"
fileSpec.Path();  // returns the string "\Documents\Oandx\"
```

The `TParse` Class

`TParse::Set()`

- Takes three parameters

The first parameter

- Is the file specification to be parsed
- The second and third parameters are pointers to two other `TDesC` descriptors, and either or both may be `NULL`

The second parameter

- Is used to supply any missing components in the first file specification

The `TParse` Class

The third parameter

- Should point to a default file specification
- From which any components not supplied by the first and second parameters will be taken

Any path, file name or extension

- May contain the wildcard characters `?` or `*`
- Representing any single character or any character sequence

The **`TParse`** Class

A **`TParse`** object owns an instance of **`TFileName`**

- Which is a **`TBuf16<256>`**
- Each character is 2 bytes in size
- The data buffer occupies 512 bytes

This is a large object!

- Its use on the stack should be avoided where possible

Common Errors and Inefficiencies

A common compile-time error

- Experienced by novice Symbian OS file system users occurs
- When attempting to use a 16-bit descriptor to read from or write
- To a file using an **`RFile`** handle.

The **`RFile::Read()`** and **`RFile::Write()`** methods

- Take only 8-bit descriptors
- Meaning wide strings must first be converted

Another common error

- Is the failure to make stack-based **`RFs`** or **`RFile`** objects leave-safe
- Through use of the cleanup stack

Common Errors and Inefficiencies

Connections to the file server

- Can take a significant amount of time to set up
- `RFs` sessions should be passed between functions where possible
- Or stored and reused

It is also possible to share `RFile` handles

- Within a single process
- Or between two processes

Allowing an open file handle

- To be passed from one process to another
- Is a necessary feature in secure versions of Symbian OS

Common Errors and Inefficiencies

File system access code

- Can also be made more efficient
- By remembering the implications of client–server interaction
- Efficiency can be improved by minimizing the number of client–server calls
- Transfer more data and thus make fewer file server requests

For example

- It is more efficient to read once from a file into one large buffer
- Then access and manipulate this client-side
- Rather than make multiple read requests for smaller sections of a file

Common Errors and Inefficiencies

Most file servers data-transfer clients

- Use the stream store or a relational database
- Which perform buffering automatically
- These components have optimized their use of the file server
- Callers that use these APIs rather than access the file server directly
- Gain efficiency automatically

Streams and Stores

- Know the reasons why use of the stream APIs may be preferred over use of **`RFile`**
- Understand how to use the stream and store classes to manage large documents most efficiently
- Be able to recognize the Symbian OS store and stream classes and know the basic characteristics of each (for example base class, memory storage, persistence, modification, etc.)
- Understand how to use **`ExternalizeL()`** and operator << with **`RWriteStream`** to write an object to a stream, and **`InternalizeL()`** and operator >> with **`RReadStream`** to read it back
- Recognize that operators >> and << can leave

Streams

A Symbian OS stream

- Is the external representation of one or more objects

Externalization

- Is the process of writing an object's data to a stream

Internalization

- The reverse process - reading an object's data from a stream

The stream

- May reside in a variety of media
- Including stores, files or memory
- Streams provide an abstraction layer over the final persistent storage media.

Streams

The external representation

- Of an object's data needs to be agnostic of the object's internal storage
- Such as byte order and data alignment
- It is meaningless to externalize a pointer
- It must be replaced in the external representation by the data to which it points

Streams

The representation

- Of each item of data must also have an unambiguously defined length
- Special care is needed when externalizing data types such as **`TInt`**
- Whose internal representation may vary in size between different processors and/or C++ compilers

Streams

Storing multiple data items

- That may come from more than one object
- In a single stream implies that they are placed in a specific order

Internalization code

- Which restores the objects by reading from the stream
- Must therefore follow exactly the same order used to externalize them

Streams

The concept of a stream

- Is implemented in two base classes

`RReadStream` and `RWriteStream`

- With concrete classes derived from them to support streams that reside in specific media

For example:

`RFileWriteStream` and `RFileReadStream`

- Implement a stream that resides in a file

`RDesWriteStream` and `RDesReadStream`

- Implement a memory-resident stream whose memory is identified by a descriptor

Streams

The `RReadStream` and `RWriteStream` base classes

- Provide a variety of `WriteXxxL()` and `ReadXxxL()` functions
- That handle specific data types
- Ranging from 8-bit integers e.g. `WriteInt8L()`
- To 64-bit real numbers e.g. `WriteReal64L()`

These functions are called

- When the << and >> operators are used on the built-in types

To handle raw data

- The stream base classes also provide
- A range of `WriteL()` and `ReadL()` functions
- Including overloads to read and write 16-bit Unicode characters
- Rather than bytes

Streams

The raw data functions

- Should be used with caution:

1. The raw data is written to the stream exactly as it appears in memory

- It must be in an implementation-agnostic format before calling **`WriteL()`**

2. A call to **`ReadL()`**

- Must read exactly the same amount of data as was written by the corresponding **`WriteL()`** call
- This can be ensured by writing the length immediately before the data
- Or terminating the data with a uniquely recognizable delimiter

3. There must be a way to acquire the maximum expected length of the data

4. The 16-bit **`WriteL()`** and **`ReadL()`** functions

- Do not provide standard Unicode compression and decompression

Streams - Externalize Example

The following example

- Externalizes a **TInt16** to a file named **aFileName**
- Assumed not to exist before **WriteToStreamFileL()** is called

```
void WriteToStreamFileL(RFs& aFs, TDesC& aFileName, TInt16* aInt)
    {
    RFileWriteStream writer;
    writer.PushL(); // put writer on cleanup stack
    User::LeaveIfError(writer.Create(aFs, aFileName, EFileWrite));
    writer << *aInt;
    writer.CommitL();
    writer.Pop();
    writer.Release();
    }
```

Streams - Externalize Example

The only reference

- To the stream is on the stack and code following it can leave
- It is necessary to push the stream to the cleanup stack
- Using the stream's (not the cleanup stack's) **`PushL()`** function

Streams - Externalize Example

Once the file has been created

- The data is externalized using operator <<

The the write stream's **`CommitL()`** function is then called

- To ensure that any buffered data is written to the stream

The stream removed from the cleanup stack

- Using the stream's **`Pop()`** function

Finally the stream is closed by calling **`Release()`**

- Which frees the resources it has been using

Streams

A common pattern:

- Operator << is used to externalize the data
- Operator >> to internalize it
- Can be used for all built-in types
- Except those like **`TInt`** whose size is unspecified and compiler-dependent

On Symbian OS

- A **`TInt`** is only specified to be at least 32 bits
- It may be longer!
- Thus externalizing it with operator << would produce an external representation of indefinite size

So ...

- The maximum length of the value
- Is used to select an appropriate internalization and externalization method

Streams

For example

- If the value stored in a **TInt** can never exceed 16 bits
- **RWriteStream::WriteInt16L()** can be used to externalize it
- **RReadStream::ReadInt16L()** to internalize it:

```
TInt i = 1234;
writer.WriteInt16L(i);
...
TInt j = reader.ReadInt16L();
... // Cleanup etc
```

Streams

Operators << and >>

- Can be used for any class that provides an implementation
- Of **ExternalizeL()** and **InternalizeL()**

Which are prototyped as:

```
class TAsdExample
    {
public:
    ...
    void ExternalizeL(RWriteStream& aStream) const;
    void InternalizeL(RReadStream& aStream);
    ...
    }
```

Streams

For such a class

- Externalization can use either:

```
TAsdExample asd;
...
writer << asd; // writer is RFileWriteStream
               // initialized and leave-safe
```

- Or

```
TAsdExample asd;
...
asd.ExternalizeL(writer); // writer is RFileWriteStream
                          // initialized and leave-safe
```

- Which are functionally equivalent

Likewise for internalization

- using operator >> or **`InternalizeL()`**

Streams

Note: Operators << and >> can leave

- As the resulting operations allocate resources
- Thus can fail if insufficient memory is available
- The operators must be used within a TRAP harness
- If they are called within a non-leaving function

Stores

Stores

A Symbian OS store

- Is a collection of streams
- Generally used to implement the persistence of objects

The abstract base class is **`CStreamStore`**

- Its API defines all the functionality needed to create and modify streams
- Used for all stores
- Classes derived from **`CStreamStore`** selectively implement the API according to their needs

Stores

Stores can use a variety of different media including:

- Memory **`CBufStore`**
- A stream **`CEmbeddedStore`**

And other stores

- e.g. **`CSecureStore`** - which allows an entire store to be encrypted and decrypted
- The most commonly used medium is a file

Stores

An important distinction

- Between the different store types is whether or not they are persistent

A persistent store

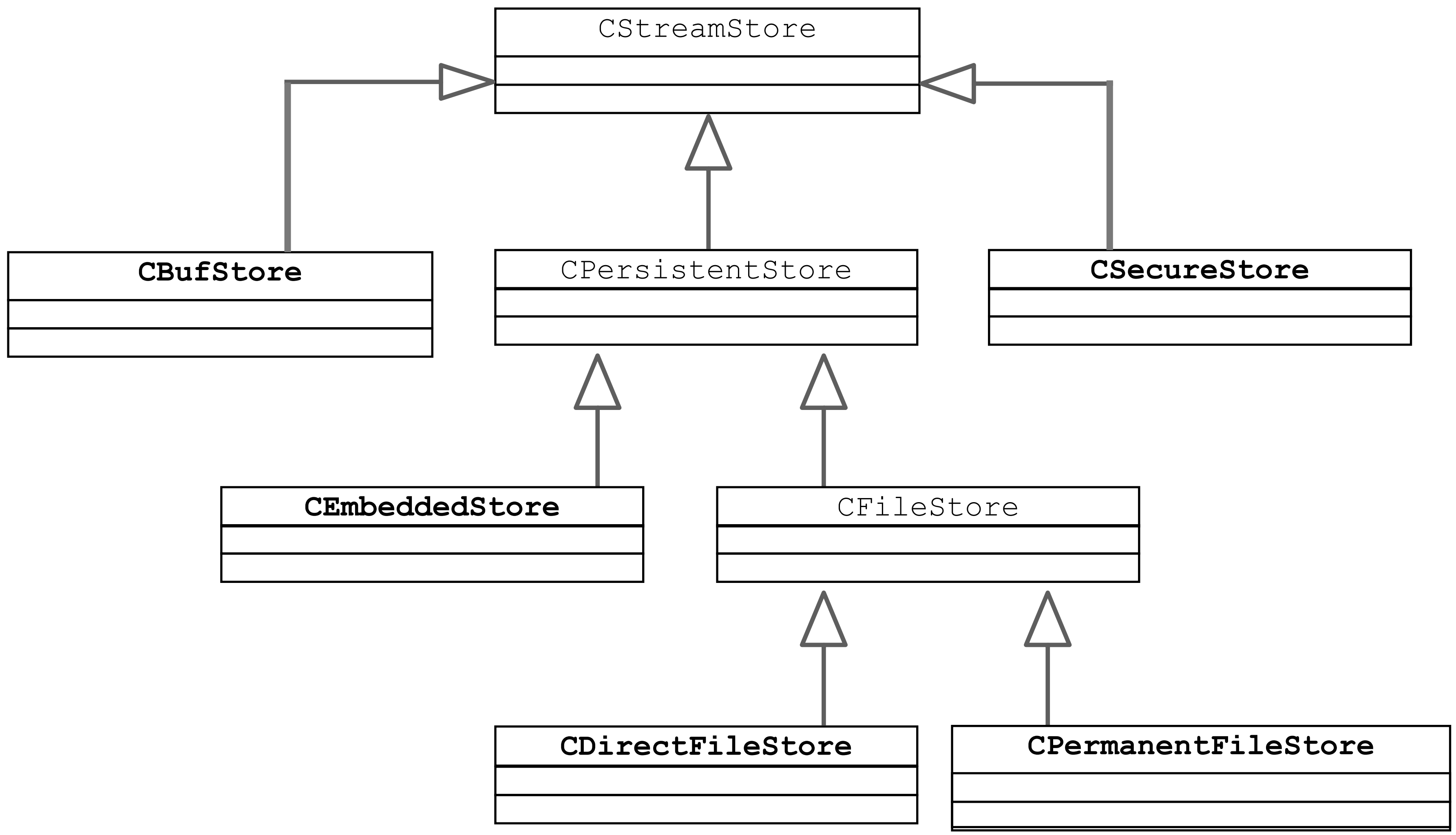
- Can be closed and re-opened and its content accessed
- The data in such a store continues after a program has closed it
- Even after the program itself has terminated
- A file-based store is persistent

`CBufStore` is not persistent

- Since the store consists of in-memory data which will be lost when it is closed

Stores

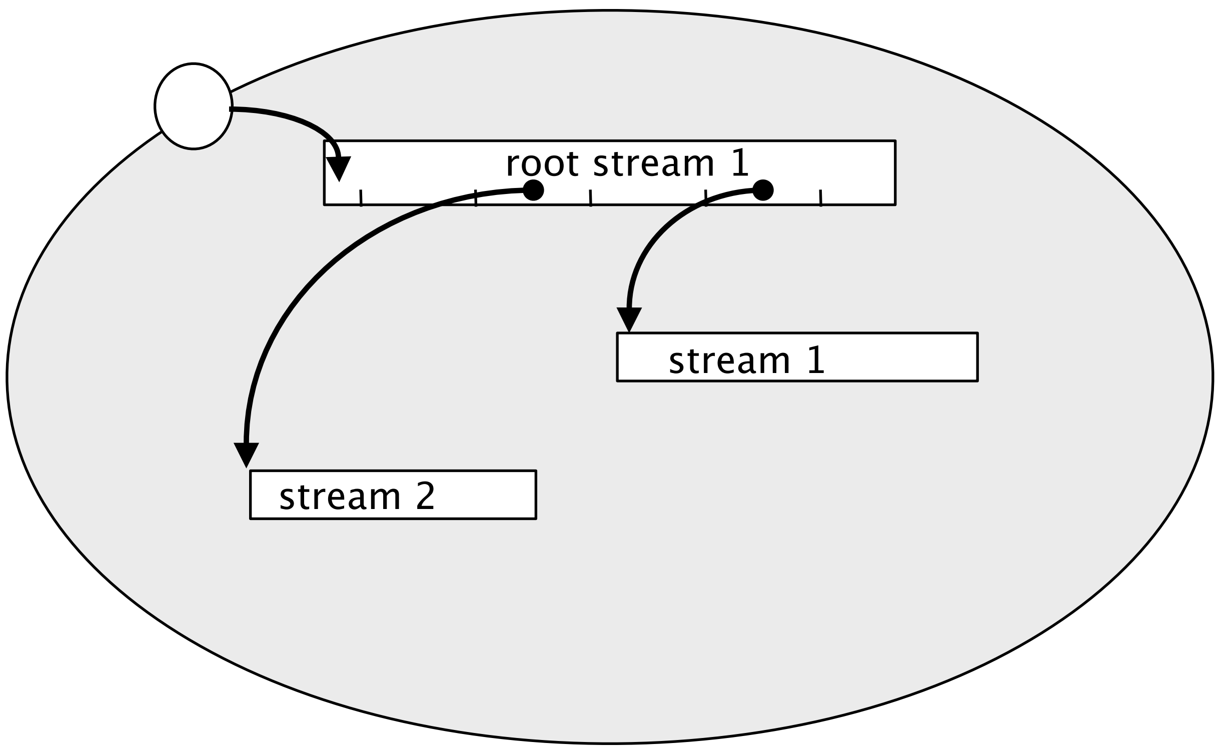
Store class hierarchy



Concrete classes are highlighted in bold text

Stores

Logical view of a persistent store



- The persistence of a store is implemented in the **`CPersistentStore`** abstract class
- It defines a root stream which is always accessible on opening the store
- The root stream (1) contains a stream dictionary of pointers to the remaining streams
- Access to the rest of the data in the store is maintained

Stores

The two file-based stores

- **`CDirectFileStore`** and **`CPermanentFileStore`**
- **`CPermanentFileStore`** allows the modification of streams after they have been written to the store
- **`CDirectFileStore`** does not allow modification after they have been written

This difference results

- In the two stores being used to store persistent data for two different types of application
- Depending on whether the store or the application itself
- Is considered to contain the primary copy of the application's data ...

Stores

For an application such as a database

- The primary copy of the data is the database file itself
- The application holds in memory only a small number of records from the file
- Any modified data is written back to the file - replacing the original version

Thus would use

- An instance of **`CPermanentFileStore`**
- With each record being stored in a separate stream

Stores

Other applications

- Such as games which store level data hold all their data in memory
- load or save the data in its entirety

Such applications can use a `CDirectFileStore`

- Since they never modify the store content
- but replace the whole store with an updated version

Creating a Persistent Store

The following code

- Illustrates how to create a persistent store
- The example creates a direct file store
- But creating a permanent file store follows a similar pattern

Creating a Persistent Store

```
void CreateDirectFileStoreL(RFs& aFs, TDesC& aFileName, TUid aAppUid)
    {
    CFileStore* store = CDirectFileStore::ReplaceLC(aFs, aFileName,
                                                       EFileWrite);
    store->SetTypeL(TUidType(KDirectFileStoreLayoutUid,
                             KUidAppDllDoc, aAppUid));
    CStreamDictionary* dictionary = CStreamDictionary::NewLC();
    RStoreWriteStream stream;
    TStreamId id = stream.CreateLC(*store);
    TInt16 i = 0x1234;
    stream << i;
    stream.CommitL();
    CleanupStack::PopAndDestroy(); // stream
    dictionary->AssignL(aAppUid,id);
    RStoreWriteStream rootStream;
    TStreamId rootId = rootStream.CreateLC(*store);
    rootStream << *dictionary;
    rootStream.CommitL();
    CleanupStack::PopAndDestroy(2); // rootStream, dictionary
    store->SetRootL(rootId);
    store->CommitL();
    CleanupStack::PopAndDestroy(); // store
    }
```

We will walk through line by line ...

Creating a Persistent Store

The call to **ReplaceLC()**

- Will create the file if it does not exist
- Otherwise it will replace any existing file
- The name of the `ReplaceLC()` method indicates that a reference to the store is left on the cleanup stack - to make it leave-safe
- In a real application - it may be more convenient to store the pointer in an object's member data

Once created it is essential to set the store's type:

```
store->SetTypeL( TUidType(KDirectFileStoreLayoutUid,
                          KUidAppDllDoc,
                          aAppUid));
```

Creating a Persistent Store

```
store->SetTypeL( TUidType(KDirectFileStoreLayoutUid,
                          KUidAppDllDoc,
                          aAppUid));
```

The three UIDs in the **`TUidType`** indicate

- The file contains a direct file store
- The store is a document associated with a Unicode application
- It is associated with the particular application whose UID is **`aAppUid`**

For the file

- To be recognized as containing a direct file store
- It is strictly necessary only to specify the first UID - **`KDirectFileStoreLayoutUid`**
- Leaving the other two as **`KNullUid`**
- Including the other two allows an application to be certain that it is opening the correct file

Creating a Persistent Store

For comparison

- The following code creates a permanent file store:

```
CFileStore* store = CPermanentFileStore::CreateLC(aFs, aFileName,
                                                   EFileWrite);

store->SetTypeL(TUidType(KPermanentFileStoreLayoutUid,
                         KUidAppDllDoc, aAppUid));
```

- Note that the **CreateLC()** function is typically used
- Rather than **ReplaceLC()** since it is less usual to need to replace a permanent file store

symbian Academy

Creating a Persistent Store

Creating, writing and closing a stream

- Follow a similar pattern to that discussed above:

```
RStoreWriteStream stream;
TStreamId id = stream.CreateLC(*store);
TInt16 i = 0x1234;
stream << i;
stream.CommitL();
CleanupStack::PopAndDestroy();
```

- The important difference is that an instance of **RStoreWriteStream**
- Rather than **RFileWriteStream**
- Must be used to write a stream to a store
- The **CreateL()** and **CreateLC()** functions return a **TStreamId**

symbian Academy

Creating a Persistent Store

Once writing the stream is complete

- The stream dictionary created earlier in the example
- Can be used to make an association between the stream ID and an externally known UID:

```
dictionary->AssignL(aAppUid,id);
```

Once all the data streams

- Have been written and added to the stream dictionary
- The stream dictionary itself must be stored ...

Creating a Persistent Store

This is done by creating a stream to contain it

- Then marking it in the store as the root stream:

```
RStoreWriteStream rootStream;
TStreamId rootId = rootStream.CreateLC(*store);
rootStream << *dictionary;
rootStream.CommitL();
CleanupStack::PopAndDestroy(); // rootStream
...
store->SetRootL(rootId);
```

Creating a Persistent Store

All that remains

- Commit all the changes made to the store
- Then to free its resources by the calling the cleanup stack's **PopAndDestroy**()

```
store->CommitL();
CleanupStack::PopAndDestroy(); // store
```

- The store's destructor takes care of closing the file and freeing any other resources

Creating a Persistent Store

If a permanent file store is created

- It can later be re-opened and new streams added
- Or existing streams replaced or deleted

To ensure that the modifications

- Are made efficiently, replaced or deleted streams are not physically removed from the store
- Thus the store will increase in size with each such change

To counteract this

- The stream store API includes functions to compact the store
- By removing replaced or deleted streams

Note

It is important

- Not to lose a reference to any stream within the store
- This is analogous to a memory leak within an application
- Resulting in the presence of a stream that can never be accessed or removed

Arguably

- Losing access to a stream is more serious than a memory leak
- As a persistent file store outlives the application that created it

The stream store API contains a tool

- Whose central class is **`CStoreMap`** to assist with stream cleanup
- Not covered here please see SDK

Reading a Persistent Store

The following code

- Opens and reads the direct file store created in the previous example:

```
void ReadDirectFileStoreL(RFs& aFs, TDesC& aFileName, TUid aAppUid)
    {
    CFileStore* store = CDirectFileStore::OpenLC(aFs, aFileName,
                                                  EFileRead);
    CStreamDictionary* dictionary = CStreamDictionary::NewLC();
    RStoreReadStream rootStream;
    rootStream.OpenLC(*store, store->Root());
    rootStream >> *dictionary;
    CleanupStack::PopAndDestroy(); // rootStream
    TStreamId id = dictionary->At(aAppUid);
    CleanupStack::PopAndDestroy(); // dictionary
    RStoreReadStream stream;
    stream.OpenLC(*store, id);
    TInt16 j;
    stream >> j;
    CleanupStack::PopAndDestroy(2); // stream, store
    }
```

Reading a Persistent Store

After opening the file store

- For reading, and creating a stream dictionary
- The code opens the root stream by calling `RStoreReadStream::OpenLC()`
- Passing in the `TStreamId` associated with root stream
- Which can be acquired from the store using `store->Root()`

Reading a Persistent Store

Once the root stream is opened

- Its content can be internalized to the stream dictionary
- Using the dictionary's **`At()`** function
- The dictionary is then used to extract the IDs of the other streams in the store
- Each stream can then be opened individually and internalized
- As appropriate for the application concerned

Embedded Stores

A store

- May contain an arbitrarily complex network of streams
- Any stream may contain another stream — by including its ID
- A stream may itself contain an embedded store

It may be useful

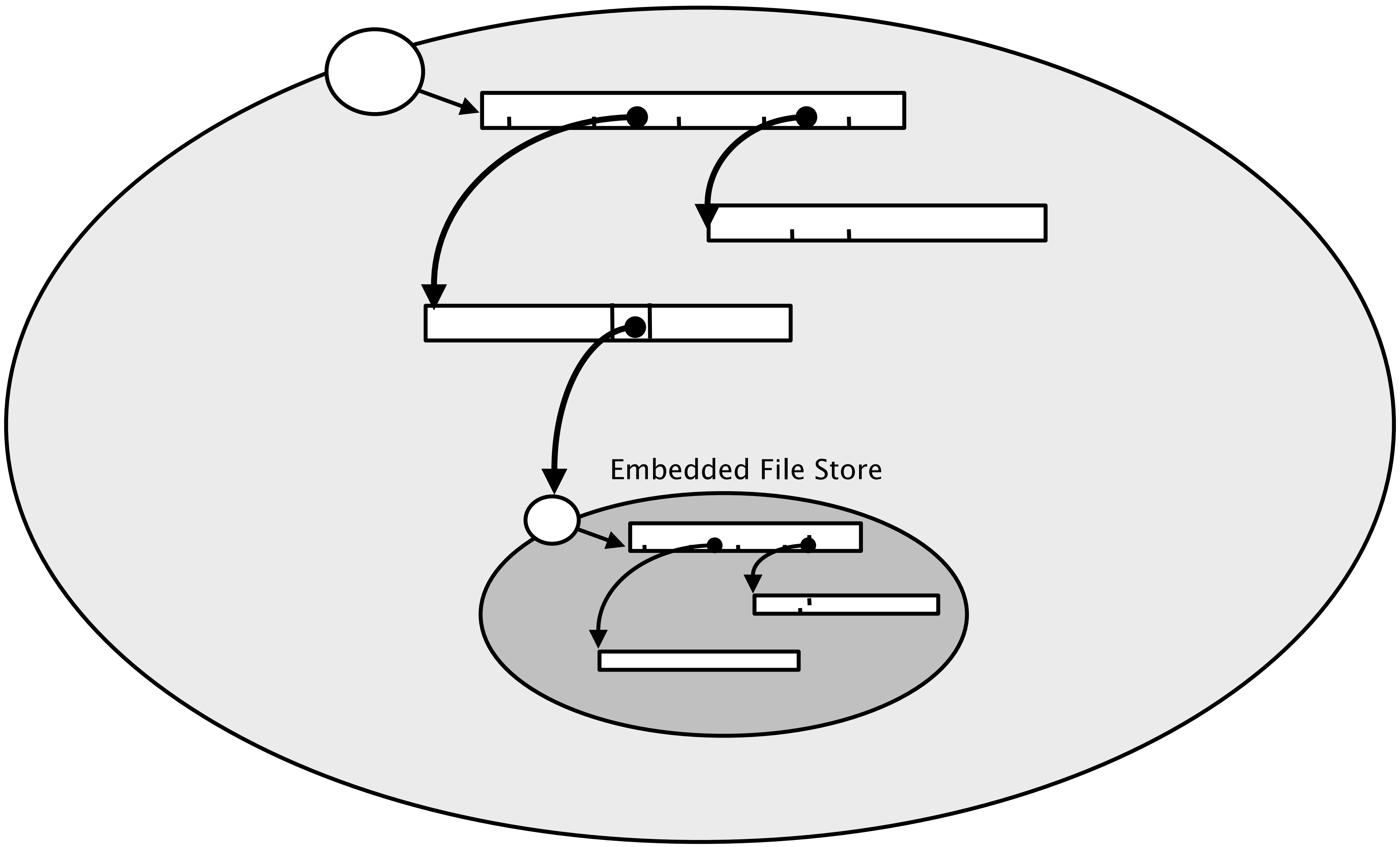
- To store a collection of streams in an embedded store
- From the outside - the embedded store appears as a single stream
- Can be copied or deleted as a whole without the need to consider its internal complexities

Embedded Stores

An embedded store cannot be modified

- Thus behaves like a direct file store
- Which means that a permanent file store cannot be embedded

Embedded Stores



Swizzles

Stores can be used to manage complex data relationships

- Such as that in a large document which may embed other documents within itself

An efficient way to manage memory in cases like this

- Is to use a class which maintains a dual representation of the data
- Defer loading it into memory from a store until required to do so

Swizzles

The templated swizzle classes

- `CSwizzleC<class T>` and `CSwizzle<class T>`
- Can be used to represent an object

Either by:

- Stream ID the stream contains the external representation of that object
- Pointer - if the object is in memory

Swizzles

Externalizing a swizzle is a two-stage process which involves:

- Externalizing the in-memory object which the swizzle represents - to its own stream
- Externalizing the resulting stream ID

A typical container-type object

- Does not hold a pointer directly to a contained object
- But owns a swizzle object which can represent the contained object
- Either as a pointer or as a stream ID

File Server and Streams

- ✓ The Symbian OS File System
- ✓ Streams and Stores