



**The Fundamentals of Symbian OS Curriculum
V2.1 (2008-2009)**

**As assessed by the
Accredited Symbian Developer exam**

The following document constitutes the Fundamentals of Symbian OS Curriculum V2.1 (2008-2009). Major changes to this curriculum from the previous published version V2.0 (for 2006-2007) are as follows:

- Modifications to accommodate the implementation of leaves in terms of C++ exceptions (changes to curriculum sections 4 and 5)
- Removal of the requirement to know about Recognizers in curriculum section 10.

The questions contained within the Accredited Symbian Developer examination are derived from the [Fundamentals of Symbian OS](#) curriculum.

Any university or training course adhering to the curriculum thus prepares a student to become an Accredited Symbian Developer. For self-study, the [Accredited Symbian Developer Primer](#) also follows the curriculum, and provides further information for each objective described herein.

Note the following version numbering convention has been adopted:

- V1 Refers to the 2005-2006 curriculum, based on Symbian OS V7 & Symbian OS V8
- V2 Refers to the 2006-2007 curriculum, based on Symbian OS v9 and first covered in detail in the [Accredited Symbian Developer Primer](#).

1. C++ LANGUAGE FUNDAMENTALS

Evaluates basic understanding of the structure, syntax and semantics of the C++ language.

Summary

The following objectives cover:

- built in types
- statements and expressions
- function use and syntax
- dynamic memory
- basic tools (compiler and linker)

Objectives

Types

- 1.1. Understand that C++ has a number of different types in various categories e.g. integral, arithmetic
- 1.2. Recognize typedef as defining a synonym for an existing type but not a new type in itself
- 1.3. Recognize enumeration types as user-defined value sets
- 1.4. Specify the similarities, differences and advantages of `const` over `#define`
- 1.5. Understand the use and properties of the C++ reference type
- 1.6. Specify the difference between pointers and references
- 1.7. Understand the semantics of pointer arithmetic
- 1.8. Recognize pointer operations and the purpose of the `NULL` pointer value
- 1.9. Differentiate `const` pointers and pointers to `const`

Statements

- 1.10. Know the use and properties of the declaration and definition statements
- 1.11. Recognize the use of the `extern` keyword
- 1.12. Cite and understand initialization of variables and their scope
- 1.13. Understand the purpose, syntax and behavior of C++ loop statements (`while`, `for` and `do`)
- 1.14. Specify the behavior and effect of the `continue` and `break` keywords in a loop
- 1.15. Specify the syntax and behavior of C++ conditional statements (`if` and `switch`)

Expressions and Operators

- 1.16. Specify the syntax and meaning of unary and binary operator expressions
- 1.17. Understand the difference between precedence and associativity
- 1.18. Recognize common operator categories including logical, prefix and postfix
- 1.19. Understand the operator precedence and associativity rules

Functions

- 1.20. Understand the syntax of a function prototype
- 1.21. Cite the purpose of the `inline` keyword
- 1.22. Understand the rules for passing default arguments and an unspecified number of arguments to a function
- 1.23. Recognize value, reference, array and pointer parameter passing

- 1.24. Understand the scope of function blocks and return by reference and value
- 1.25. Specify the syntax for pointers to function assignments
- 1.26. Recognize pointer to functions as callback parameter arguments

Dynamic Memory Allocation

- 1.27. Understand C++ free store allocation scope using the `new` and `delete` operators
- 1.28. Recognize the syntax and purpose of placement `new`

Tool Chain Basics

- 1.29. Understand the function of the tools, the C++ tool chain (e.g. compiler and linker)
- 1.30. Recognize there are lexical and syntax-parsing stages of compilation
- 1.31. Be able to describe the purpose of the C++ preprocessor, specifying common directives
- 1.32. Understand the role `inline` functions play in C++
- 1.33. Know how to use the `extern` keyword

2. CLASSES AND OBJECTS

Measures the understanding of C++ object-oriented programming support and class semantics

Summary

The following objectives cover:

- scope rules, data abstraction and structures
- construction and destruction
- access control

Objectives

Scope and C++ Object-Oriented Programming (OOP) Support

- 2.1. Understand the scope and lifetime properties of blocks and namespaces
- 2.2. Understand C++ support for data abstraction
- 2.3. Specify the attributes of a C++ object with respect to object-oriented programming
- 2.4. Know the syntax of a class declaration
- 2.5. Cite the differences between a class and an object
- 2.6. Differentiate between basic data structures (`struct` keyword) and classes

Constructors and Destructors

- 2.7. Understand the order of construction and destruction for the class and its member variables
- 2.8. Recognize implicit constructor invocation (overloading and pattern matching) and the role the `explicit` keyword plays in constructor declaration
- 2.9. Specify what the compiler automatically generates for user-defined classes
- 2.10. Understand the purpose and use of copy constructors (including parameter passing)
- 2.11. Understand the difference between assignment and initialization
- 2.12. Specify the required function members needed in a class to support safe member pointer data ownership

Class Members

- 2.13. Describe `private`, `protected` and `public` access control for class members
- 2.14. Declare and specify the syntax for pointers to class members
- 2.15. Identify nested classes and their scope and life-span
- 2.16. Cite the scope access rules and lifetime of a nested class
- 2.17. Understand the scope rules and syntax of the `friend` keyword
- 2.18. Understand the semantics of member function and data addressing
- 2.19. Understand the purpose of the scope-resolution operator `::`
- 2.20. Understand the role of the `this` pointer
- 2.21. Specify the properties of `static` class members

3. CLASS DESIGN AND INHERITANCE

Tests the understanding of more advanced C++ properties including design considerations when using inheritance and both dynamic and static polymorphism.

Summary

The following objectives cover:

- class relationships
- the fundamental properties of the base-derived inheritance hierarchy
- virtual methods
- templates

Objectives

Class Relationships

- 3.1. Understand the key benefits and purpose of inheritance
- 3.2. Specify the differences between composition, aggregation and inheritance
- 3.3. Cite the object-oriented relationship for inheritance

Inheritance

- 3.4. Be able to define public, private and protected inheritance
- 3.5. Understand the scope resolution operator syntax for accessing the base-derived class hierarchy
- 3.6. Given a base-derived hierarchy, specify the access rules (including those of friend classes)
- 3.7. Describe the scope access rules and purpose of public, protected and private inheritance
- 3.8. Specify the implicit invocation order of constructors and destructors in a base-derived hierarchy

Dynamic Polymorphism – Virtual Methods

- 3.9. Specify the mechanisms of OO reuse available in C++
- 3.10. Be able to state C++ support for polymorphism
- 3.11. Understand the importance of virtual destructors
- 3.12. Understand the purpose of and difference between overriding and overloading
- 3.13. Understand the use of overriding to modify behavior in base-derived class inheritance
- 3.14. Understand the rules and pattern-matching criteria for correct overloaded-function invocation
- 3.15. Identify the typical uses and behavior for operator overloading
- 3.16. Describe the purpose of the virtual table, citing constraints and overheads
- 3.17. Specify the use of virtual functions and their implementation tradeoffs
- 3.18. Understand how an abstract base class is implemented in C++
- 3.19. State the differences between interface and implementation inheritance
- 3.20. Understand and recognize the problems associated with multiple inheritance
- 3.21. Cite the implementation requirements to support the `static_cast` operator in user-defined classes

Static Polymorphism and Templates

- 3.22. Specify the syntax for a simple function template specialization
- 3.23. Be able to cite the advantages of function templates (e.g. over macros)
- 3.24. Understand the inheritance rules and syntax supported by class templates

- 3.25. Understand the syntax and semantics of a template type/class declaration
- 3.26. Recognize the prototype declaration and pattern-matching properties of a template declaration and its use
- 3.27. Understand the purpose and implementation differences of the Symbian OS thin template and mainstream C++ templates

4. SYMBIAN OS TYPES & DECLARATIONS

Ensures that the candidate understands the fundamental Symbian OS types, naming conventions and coding style, and the usage paradigms of each basic type of Symbian OS class.

Summary

The following objectives cover the Symbian OS class naming conventions, usage and behaviour.

Objectives

The Fundamental Symbian OS Types

- 4.1. Know how the fundamental Symbian OS types relate to native built-in C++ types
- 4.2. Understand that the fundamental types should always be used in preference to the native built-in C++ types (`bool`, `int`, `float` etc) because they are compiler-independent, and know the exception case – when to use `void` instead of `TAny`.

T Classes

- 4.3. Know the purpose of a T class, what types of member data it may and may not own, and that it typically does not have a destructor
- 4.4. Know what types of function a T class may have
- 4.5. Understand that a T class may be created on the heap or stack
- 4.6. Understand that a T class may be used as an alternative to the traditional C/C++ `struct`
- 4.7. Know that the T prefix is also used to define an `enum`.

C Classes

- 4.8. Recognize that a C class always derives from `CBase`
- 4.9. Know the purpose of a C class, and what types of data it may own
- 4.10. Understand that a C class must always be instantiated on the heap
- 4.11. Know that a C class uses two-phase construction and has its member data zero-filled when it is allocated on the heap
- 4.12. Understand the destruction of C classes via the virtual destructor defined in `CBase`

R Classes

- 4.13. Know the purpose of an R class, to own a resource
- 4.14. Understand that an R class can be instantiated on the heap or stack
- 4.15. Understand the separate construction and initialization of R classes
- 4.16. Understand the separate cleanup and destruction of R classes, and the consequences of forgetting to call the `Close()` or `Reset()` method before destruction

M Classes

- 4.17. Know the purpose of an M class, to define an interface
- 4.18. Understand the use of M classes for multiple inheritance, and the order in which to derive an implementation class from C and M classes
- 4.19. Know that an M class should never contain member data and does not have constructors
- 4.20. Know what types of function an M class may include, and the unusual circumstances where it is appropriate to define their implementation
- 4.21. Understand that an M class cannot be instantiated

Static Classes

- 4.22. Know that static classes do not have a prefix letter
- 4.23. Understand that static classes cannot be instantiated because they contain only static functions

Factors to Consider when Creating a Symbian OS Class

- 4.24. Know the important factors to consider when creating a new class, and how this determines the choice of Symbian OS class type

Why Is the Symbian OS Naming Convention Important?

- 4.25. Understand that the use of a class prefix makes it clear to anyone wishing to use a class how it should be instantiated, used and destroyed with leave-safety.
- 4.26. Recognize that the naming convention forces a class designer to think about the class design and, having decided on the fundamental behavior, can concentrate on the role of the class, knowing that leave-safe construction, destruction and ownership are already handled.

5. CLEANUP STACK

Demonstrates practical knowledge of the Cleanup Stack and the difference between standard C++ and Symbian OS in handling leaks and exceptions.

Summary

These objectives cover the following idioms:

- Symbian OS leaves vs standard C++ exception support
- Cleanup stack
- TRAP handlers

in terms of their purpose, the relationship between them, their usage and common pitfalls and practical programming techniques (e.g. debug macros to detect memory leaks).

Objectives

Leaves: Lightweight Exceptions for Symbian OS

- 5.1. Know that, pre-Symbian OS v9, standard C++ exceptions (`try/catch/throw`) were not supported and that Symbian OS used a lightweight alternative: `TRAP` and `leave`. In addition, understand that, from Symbian OS v9.1, leaves are implemented in terms of standard C++ exceptions.
- 5.2. Know that leaves are a fundamental part of Symbian error handling and used throughout the system
- 5.3. Understand that `User::Leave()` is implemented in terms of C++ exceptions from Symbian OS v9.1. When `User::Leave()` is called, an exception is thrown, which is always of the type `XLeaveException` (this is a simple type that wraps a single `TInt`).
- 5.4. Understand that Symbian OS does not support nested exceptions on target builds, because this requires dynamic heap allocation. For code executing on hardware, if an exception is thrown while another is being handled, `abort()` is called on the thread.
- 5.5. Recognize the typical system functions which may cause a leave, including the `User::LeaveXXX()` functions and `new(ELeave)`
- 5.6. Be able to list typical circumstances which cause a leave (for example, insufficient memory for a heap allocation)
- 5.7. Understand that `new(ELeave)` guarantees that the pointer return value will always be valid if a leave has not occurred

How to Work with Leaves

- 5.8. Know that leaves are indicated by use of a trailing L or LC suffix on functions containing code which may leave (for example, `InitializeL()`)
- 5.9. Be able to spot functions which are not leave-safe and those which are
- 5.10. Understand that leaves are used for error handling; code should very rarely both return an error and be able to leave
- 5.11. Understand the reason why a leave should not occur in a constructor or destructor and be able to recognise the following cases:
 - leaves should never occur in the destructors of stack-based objects (because nested exceptions are not allowed, and destructors are executed as part of exception handling)
 - leaves should never occur in constructors of heap-based classes because of the potential for memory leaks and double-deletion panics

Comparing Leaves and Panics

- 5.12. Understand the difference between a leave and a panic
- 5.13. Recognize that panics come about through assertion failures, which should be used to flag programming errors during development
- 5.14. Recognize that a leave should not be used to direct normal code logic

What Is a TRAP?

- 5.15. Recognize the characteristics of a TRAP handler
- 5.16. Understand that, for efficiency, use of TRAPS should be kept to a minimum

The Cleanup Stack

- 5.17. Know how to use the cleanup stack to make code leave-safe, so memory is not leaked in the event of a leave
- 5.18. Understand that `CleanupStack::PushL()` will not leak memory even if it leaves
- 5.19. Know the order in which to remove items from the cleanup stack, and how to use `CleanupStack::PopAndDestroy()` and `CleanupStack::Pop()`
- 5.20. Recognize correct and incorrect use of the cleanup stack
- 5.21. Understand the consequences of putting a C class on the cleanup stack if it does not derive from `CBase`
- 5.22. Know how to use `CleanupStack::PushL()` and `CleanupXXXPushL()` for objects of C, R, M and T classes and `CleanupArrayDeletePushL()` for C++ arrays
- 5.23. Understand the meaning of the Symbian OS function suffixes C and D

Detecting Memory Leaks

- 5.24. Recognize the use of the `__UHEAP_MARK` and `__UHEAP_MARKEND` macros to detect memory leaks

6. OBJECT CONSTRUCTION

Evaluates knowledge of 2-phase construction in Symbian OS, safely creating objects and avoiding memory leaks.

Summary

The following objectives test an understanding of the importance, implementation and reasons for 2 phase construction, and how to ensure object destruction is efficient and safe.

Objectives

Two-Phase Construction

- 6.1. Know the reasons why code should not leave inside a constructor (see also objective 5.11)
- 6.2. Recognize that two-phase construction is used to avoid the accidental creation of objects with undefined state
- 6.3. Understand that constructors and second-phase `ConstructL()` methods are given private or protected access specifiers in classes which use two-phase construction, to prevent their inadvertent use.
- 6.4. Recognise that public static factory methods, called `NewL()` or `NewLC()` are provided to instantiate an object,
- 6.5. Understand how to implement two-phase construction, and how to construct an object which derives from a base class which also uses a two-phase method of initialization
- 6.6. Know the Symbian OS types (C classes) which typically use two-phase construction

Object Destruction

- 6.7. Know that it is not efficient or necessary to set a pointer to `NULL` after deleting it in destructor code
- 6.8. Understand that a destructor must check before dereferencing a pointer in case it is `NULL`, but need not check if simply calling `delete` on that pointer

7. DESCRIPTORS

Test the understanding of the motivation for using descriptors, how to use buffer and pointer descriptors and when to use package descriptor classes.

Summary

The following objectives cover the characteristics of the Symbian OS descriptor classes including memory management, inheritance hierarchy, use as function parameters, conversion methods and key aspects of the API.

Objectives

Features of Symbian OS Descriptors

- 7.1. Understand that Symbian OS descriptors may contain text or binary data
- 7.2. Know that descriptors may be narrow (8-bit), wide (16-bit) or neutral (which is 16-bit since Symbian OS is built for Unicode)
- 7.3. Understand that descriptors do not dynamically extend the data they reference, so will panic if too small to store data resulting from a method call

The Symbian OS Descriptor Classes

- 7.4. Know the characteristics of the `TDesC`, `TDes`, `TBufC`, `TBuf`, `TPtrC`, `TPtr`, `RBuf` and `HBufC` descriptor classes
- 7.5. Understand that the descriptor base classes `TDesC` and `TDes` implement all generic descriptor manipulation code, while the derived descriptor classes merely add construction and assignment code specific to their type
- 7.6. Identify the correct and incorrect use of modifier methods in the `TDesC` and `TDes` classes
- 7.7. Recognize that there is no `HBuf` class, but that `RBuf` can be used instead as a modifiable dynamically allocated descriptor

The Inheritance Hierarchy of the Descriptor Classes

- 7.8. Know the inheritance hierarchy of the descriptor classes
- 7.9. Understand the memory efficiency of the descriptor class inheritance model and its implications

Using the Descriptor APIs

- 7.10. Understand that the descriptor base classes `TDesC` and `TDes` cannot be instantiated
- 7.11. Understand the difference between `Size()`, `Length()` and `MaxLength()` descriptor methods
- 7.12. Understand the difference between `Copy()` and `Set()` descriptor methods and how to use assignment correctly

Descriptors as Function Parameters

- 7.13. Understand that the correct way to specify a descriptor as a function parameter is to use a reference, for both constant data and data that may be modified by the function in question.

Correct Use of the Dynamic Descriptor Classes

- 7.14. Identify the correct techniques and methods to instantiate an `HBufC` heap buffer object
- 7.15. Recognize and demonstrate knowledge of how to use the new descriptor class `RBuf`

Common Inefficiencies in Descriptor Usage

- 7.16. Know that `TFileName` objects should not be used indiscriminately, because of the stack space each consumes

- 7.17. Understand when to dereference an `HBufC` object directly, and when to call `Des()` to obtain a modifiable descriptor (`TDes&`)

Literal Descriptors

- 7.18. Know how to manipulate literal descriptors and know that those specified using `_L` are deprecated
- 7.19. Specify the difference between literal descriptors using `_L` and those using `_LIT` and the disadvantages of using the former

Descriptor Conversion

- 7.20. Know how to convert 8-bit descriptors into 16-bit descriptors and vice versa using the descriptor `Copy()` method or the `CnvUtfConverter` class
- 7.21. Recognize how to read data from file into an 8-bit descriptor and then 'translate' the data to 16-bit without padding, and vice versa
- 7.22. Know how to use the `TLex` class to convert a descriptor to a number, and `TDes::Num()` to convert a number to a descriptor

8. DYNAMIC ARRAYS

Measures proficiency in the use of Symbian OS dynamic arrays in preference to standard C++ arrays, and the choice of dynamic array class depending on desired usage and characteristics of array elements.

Summary

The following objectives cover an understanding of the Symbian OS dynamic array classes including their practical application, memory layout and expansion strategy (granularity).

Objectives

Dynamic Arrays in Symbian OS

- 8.1. Demonstrate an understanding of the basics of Symbian OS dynamic arrays (`CArrayX` and `RArray` families)
- 8.2. 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.
- 8.3. Recognize the appropriate circumstances for using a segmented-buffer array class rather than a flat array class

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

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

Array Granularities

- 8.5. Understanding the meaning of array granularity
- 8.6. Know how to choose the granularity of an array as appropriate to its perceived use

Array Sorting and Searching

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

`TFixedArray`

- 8.9. Recognize that the `TFixedArray` class should be preferred over a C++ array, since it gives the benefit of bounds checking (debug-only or debug and release).

9. ACTIVE OBJECTS

Tests understanding of why Active Objects are preferred over Threads and under what conditions, the use and implementation of simple Active Objects and their relation to the Active Scheduler.

Summary

The following objectives cover the use of active objects for non-preemptive event handling on Symbian OS including: how to derive an active object class (priority assignment and virtual overrides, and their characteristics), “conditions of use”, common errors (bad practice) and the role of the active scheduler.

Objectives

Event-Driven Multitasking on Symbian OS

- 9.1. Demonstrate an understanding of the difference between synchronous and asynchronous requests, and differentiate between typical examples of each
- 9.2. Recognize the typical use of active objects to allow asynchronous tasks to be requested without blocking a thread
- 9.3. Understand the difference between multi-tasking using multiple threads and multiple active objects, and why the latter is preferred in Symbian OS code

Class `CActive`

- 9.4. Understand the significance of an active object’s priority level
- 9.5. Recognize that the active object event handler method (`RunL()`) is non-preemptive
- 9.6. Know the inheritance characteristics of active objects, and the functions they are required to implement and override
- 9.7. Know how to correctly construct, use and destroy an active object

The Active Scheduler

- 9.8. Understand the role and characteristics of the active scheduler
- 9.9. Know that `CActiveScheduler::Start()` should only be called after at least one active object has an outstanding request
- 9.10. Recognize that a typical reason for a thread to fail to handle events may be that the active scheduler has not been started or has been stopped prematurely
- 9.11. Understand that `CActiveScheduler` may be sub-classed, and the reasons for creating a derived active scheduler class

Canceling an Outstanding Request

- 9.12. Understand the different paths in code that the active object uses when an asynchronous request completes normally, and as the result of a call to `Cancel()`

Background Tasks

- 9.13. Understand how to use an active object to carry out a long-running (or background) task
- 9.14. Demonstrate an understanding of how self-completion is implemented

Common Problems

- 9.15. Know some of the possible causes of a stray signal panic
- 9.16. Recognize that a call to `User::After()` blocks a thread until the time specified as a parameter has elapsed
- 9.17. Recognize that a typical reason for a blocked thread may be the inappropriate use of `User::WaitForRequest()`

10. SYSTEM STRUCTURE

Tests knowledge of the underlying structure of the Symbian OS, including threads, processes, DLLs and memory management. Also assesses the candidate's understanding of common system components.

Summary

The following objectives cover

- writable static data
- UIDs
- threads and processes
- memory management idioms
- system architecture including non-functional requirements
- IPC (preferred mechanisms on Symbian OS)
- recognizers
- synchronization primitives
- defensive programming
- debugging techniques
- publish and subscribe

Objectives

DLLs in Symbian OS

- 10.1. Know and understand the characteristics of polymorphic interface and shared library (static) DLLs
- 10.2. Know that UID2 values are used to distinguish between static and polymorphic DLLs, and between plug-in types
- 10.3. For a shared library, understand which functions must be exported if other binary components are to be able to access them

Writable Static Data

- 10.4. Recognize that writable static data is not allowed in DLLs on EKA1 and discouraged on EKA2
- 10.5. Know the basic porting strategies for removing writable static data from DLLs

Executables in ROM and RAM

- 10.6. Recognize the correctness of basic statements about Symbian OS execution of DLLs and EXEs in ROM and RAM

Threads and Processes

- 10.7. Recognize the correctness of basic statements about threads and processes on Symbian OS
- 10.8. Recognize the role and the characteristics of the synchronization primitives `RMutex`, `RCriticalSection` and `RSemaphore`

Inter-Process Communication (IPC)

- 10.9. Recognize the preferred mechanisms for IPC on Symbian OS (client-server, publish and subscribe and message queues), and demonstrate awareness of which mechanism is most appropriate for given scenarios

- 10.10. Understand the use of publish and subscribe to retrieve and subscribe to changes in system-wide properties, including the role of platform security in protecting properties against malicious manipulation

Panics and Assertions

- 10.11. Know the type of parameters to pass to `User::Panic()` and understand how to make them meaningful
- 10.12. Understand the use of `__ASSERT_DEBUG` statements to detect programming errors in debug code by breaking the flow of code execution using a panic
- 10.13. Recognize that `__ASSERT_ALWAYS` should be used more sparingly because it will test statements in released code too and cause code to panic if the assertion fails

11. CLIENT SERVER

Recognizes the use cases of the Client / Server model within a handset, system components utilizing the model and applies knowledge to simple Server implementations.

Summary

The following objectives cover the following:

- the role of client-server on Symbian OS
- reasons for client-server
- characteristics of the communication protocol
- runtime impact and memory overheads
- threading model
- classes used by the client-server framework
- server-side objects which must be created on start up (cleanup stack, active scheduler)
- client-server data transfer mechanism, marshalling and data lifetime requirements
- client subsessions
- system and transient servers

Objectives

The Client–Server Pattern

- 11.1. Know the structure and benefits of the client–server framework
- 11.2. Understand the different roles of system and transient servers, and match the appropriate server type to examples of server applications

Fundamentals of the Symbian OS Client–Server Framework

- 11.3. Know the fundamentals of the Symbian OS client–server implementation

Symbian OS Client–Server Classes

- 11.4. Know the following classes used by the Symbian OS client–server framework, and basic information about the role of each:
- 11.5. Recognize the objects that a server must instantiate when it starts up
- 11.6. Understand the mechanism used to prevent the spoofing of servers in Symbian OS

Client–Server Data Transfer

- 11.7. Know the basics of how clients and servers transfer data for synchronous and asynchronous requests
- 11.8. Recognize the correct code to transfer data from a client derived from `RSessionBase` to a Symbian OS server
- 11.9. Know how to submit both synchronous and asynchronous client–server requests
- 11.10. Know how to convert basic and custom data types into the appropriate payload which can be passed to the server, as both read-only and read/write request arguments

Impact of the Client–Server Framework

- 11.11. Understand the potential impact on run-time speed from using a client–server session and differentiate between circumstances where it is useful or necessary and where it is inefficient
- 11.12. Recognize scenarios where an implementation which uses client subsessions with the server would be recommended
- 11.13. Understand the impact of the context switch required when making a client–server request, and the best way to manage communication between a client and server to maximize run-time efficiency

12. FILE SERVER, STORE & STREAMS

Identifies an understanding of the use of files, stores and streams for storing persistent and temporary data. Also measures whether the candidate has knowledge of the conditions and intended usage which determine the different classes to use for storing data.

Summary

The following objectives cover:

- RFile API methods
- RFs API methods
- Stream operators
- Stream and store classes and their use to manipulate large documents
- Difference between use of RFile and stream classes and reasons for preferring each

Objectives

The Symbian OS File System

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

Streams and Stores

- 12.6. Know the reasons why use of the stream APIs may be preferred over use of `RFile`
- 12.7. Understand how to use the stream and store classes to manage large documents most efficiently
- 12.8. Be able to recognize the Symbian OS store and stream classes and know the basic characteristics of each (e.g. base class, memory storage, persistence, modification, etc.)
- 12.9. 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
- 12.10. Recognize that operators `>>` and `<<` can leave

13. SOCKETS

Evaluates the ability to add communication services to an application and handling asynchronous communication events with the socket server architecture.

Summary

The following objectives cover practical usage of the `RSocket` API and `RHostResolver` class.

Objectives

Introducing Sockets

- 13.1. Recognize correct high-level statements which define and describe a network socket
- 13.2. Recognize correct statements about transport independence
- 13.3. Know the difference between connected and connectionless sockets
- 13.4. Differentiate between streamed and datagram communication and their relationship with connected/connectionless sockets

The Symbian OS Sockets Architecture

- 13.5. Demonstrate a basic understanding of the use of sockets on Symbian OS
- 13.6. Recognize the characteristics of the `RSocketServ`, `RSocket` and `RHostResolver` classes
- 13.7. Understand the role and purpose of PRT protocol modules on Symbian OS

Using Symbian OS Sockets

- 13.8. Recognize correct patterns for opening and configuring connected and connectionless sockets
- 13.9. Know which `RSocket` API methods should be used for connected and unconnected sockets to send and receive data
- 13.10. Know the characteristics of the synchronous and asynchronous methods for closing a `RSocket` sub-session

14. TOOL CHAIN

Measures that the candidate has a practical working knowledge of the Symbian OS tool chain and development environment.

Summary

The following objectives cover an understanding of the mmp file syntax, use of bldmake, the role of resource files, emulator settings, the difference between running code on emulator and hardware, the ABI binary standard and tool specific errors (not programming errors).

Objectives

Build Tools

- 14.1. Understand the basic use of bldmake, bld.inf and abld.bat
- 14.2. Understand the purpose and typical syntax of project definition (MMP) files
- 14.3. Understand the role of Symbian OS resource and text localization files

Hardware Builds

- 14.4. Understand that the ARM C++ EABI is an industry standard optimized for embedded application development
- 14.5. Recognize basic information about the RVCT and GCCE compilers, which can be used for target hardware builds
- 14.6. Understand that ARMv5 supports both 32-bit ARM and 16-bit THUMB instructions, and appreciate the difference with respect to speed and size

Installing an Application to Phone Hardware

- 14.7. Recognize the package file format used for creation of SIS installation files

The Symbian OS Emulator

- 14.8. Understand the purpose of the Symbian OS emulator for Windows
- 14.9. Recognize differences between running code on the emulator and on target hardware

15. PLATFORM SECURITY

Assesses the understanding of the three core concepts of platform security: The trust model, capabilities and data caging. Evaluates the candidate's practical knowledge of designing, developing and distributing software on Symbian OS v9.

Summary

The objectives cover the following:

- Trusted Computer Base and Trusted Computer Environment
- Capabilities and rules for dll loading and static linking
- Data caging for processes and dlls
- The capabilities required to access the protected filesystem areas
- Secure IDs and UIDs
- A practical understanding of writing a secure application
- The role of the software installer and supporting tools for sis file generation
- The signing process
- Platform security and Symbian OS servers

Objectives

The Trust Model

- 15.1. Understand what is meant by the axiom "a process is a unit of trust" and how Symbian OS enforces this
- 15.2. Understand the purpose of the Trusted Computing Base and why it is important
- 15.3. Recognize that a number of Symbian OS APIs do not require security checks before they can be used
- 15.4. Know that self-signed software that does not use sensitive system services is "untrusted" and can be installed and run on the phone, although it is effectively "sandboxed"

Capability Model

- 15.5. Understand the relationship between capabilities and the Trusted Computing Base (TCB)
- 15.6. Understand the relationship between the TCB/TCE, capability assignment, software install as the "gatekeeper" and the role of application signing
- 15.7. Recognize the different groups of capabilities, demonstrating a broad understanding of the privileges granted
- 15.8. Recognize how to specify platform security capabilities within an MMP file
- 15.9. Demonstrate an understanding of the capability rules

Data Caging

- 15.10. Understand how data caging works to protect all types of files via the three special directories (`\sys`, `\resource` and `\private`); in particular, that data caging is used to partition all executables in the file system so, once trusted, they are protected from modification
- 15.11. Understand the implications of data caging for naming executable code
- 15.12. Recognize that data caging can be used to provide a secure area for an application's data
- 15.13. Recognize the capabilities needed to read from and write to specific directories and subdirectories

- 15.14. Know that DLLs do not have a private data-caged area and use that of the process in which they are loaded, and that this directory can be acquired by the DLL using the file system methods
`RFs::PrivatePath()`

Secure Identifier, Vendor Identifier and Unique Identifier

- 15.15. Explain what a Secure Identifier (SID) is, where it is defined and what is it used for
15.16. Understand the similarities and differences between Secure Identifier (SID), Vendor Identifier (VID) and a binary's Unique Identifiers (UID)
15.17. Know the rules by which an application is identified, according to the specification of SID, VID and UID
15.18. Understand that SID and VID may be assigned, but are not relevant, to DLLs
15.19. Recognize how to specify VID and SID within an MMP file
15.20. Understand that UIDs are now split into 2 groups (protected and unprotected ranges) with different implications for test and commercial code

Application Design for a Secure Platform

- 15.21. Demonstrate an understanding of the key considerations when writing a secure application, including the parties interested in application security, typical attacks, countermeasures and secure application design, and the costs of various countermeasures

Releasing a Secure Application on Symbian OS v9

- 15.22. Understand the basic process of testing and releasing a signed Symbian OS v9 application

The Native Software Installer

- 15.23. Recognize the key functions of the Symbian OS v9 Native Software Installer, including the compatibility break in SIS file format between Symbian OS v9 and previous versions of Symbian OS

16. BINARY COMPATABILITY

Assesses understanding of binary compatibility and the programming and design techniques which are required to maintain binary compatibility in code modules and APIs.

Summary

The following objectives cover:

- the definition of compatibility at different levels (binary, link, source, forwards and backwards)
- what can't be changed without breaking compatibility
- what can be changed without breaking compatibility
- best practice - designing to ensure compatibility is retained

Objectives

Levels of Compatibility

- 16.1. Demonstrate an understanding of source, binary, library, semantic and forward/backward compatibility

Preventing Compatibility Breaks – What Cannot Be Changed?

- 16.2. Recognize which attributes of a class are necessary for a change in the size of the class data not to break compatibility
- 16.3. Understand which class-level changes will break source compatibility
- 16.4. Understand which class-level changes will break binary compatibility
- 16.5. Understand which library-level changes will break binary compatibility
- 16.6. Understand which function-level changes will break binary and source compatibility
- 16.7. Differentiate between derivable and non-derivable C++ classes in terms of what cannot be changed without breaking binary compatibility

What Can Be Changed Without Breaking Compatibility?

- 16.8. Understand which class-level changes will not break source compatibility
- 16.9. Understand which class-level changes will not break binary compatibility
- 16.10. Understand which library-level changes will not break binary compatibility
- 16.11. Understand which function-level changes will not break binary and source compatibility
- 16.12. Differentiate between derivable and non-derivable C++ classes in terms of what can be changed without breaking binary compatibility

Best Practice – Designing to Ensure Future Compatibility

- 16.13. Recognize best practice for maintaining source and binary compatibility
- 16.14. Recognize the coupling arising from the use of inline functions and differentiate between cases where it will make maintaining binary compatibility more difficult and where it will be less significant