symbian Academy

# Two-Phase Construction and Object Destruction

# Introduction

## Symbian OS is designed to perform well on devices with limited memory

- It uses memory-management models such as the cleanup stack to ensure that memory is not leaked, even under error conditions or in exceptional circumstances, such as when there is insufficient free memory to complete an allocation
- Two-phase construction is an idiom used extensively in Symbian OS code to provide a means by which heap-based objects may be constructed fully initialized, even when that initialization code may leave


symbian Academy

# Two-Phase Construction and Object Destruction

## Two-Phase Construction

- Know why code should not leave inside a constructor
- Recognize that two-phase construction is used to avoid the accidental creation of objects with undefined state
- 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
- 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
- Know the Symbian OS types (C classes) which typically use two-phase construction

# No code within a C++ constructor should ever leave

Consider this example:

```
class CFoo : public CBase
   {...                          //Details omitted for clarity
public:
    CFoo(TInt aVal)
        {
        iVal=aVal;
        iBar = new(ELeave) CBar(aVal);
        }
private:
    CBar* iBar;
   };
...
CFoo* foo = new(ELeave) CFoo(42);
```

- Memory is allocated for **foo** and its constructor is called
- But what if **iBar = new(ELeave) CBar(aVal);** in the **CFoo** constructor leaves?
- Memory has already been allocated for the **CFoo** object
- Because of the leave there is no valid pointer for the partially-constructed **CFoo** object
- Without a valid pointer there is no way to clean up the **CFoo** object

# Two-Phase Construction

Two-phase construction breaks object construction into two parts

1. A basic C++ constructor which cannot leave
   - It is this constructor which is called by the new operator.
   - It implicitly calls base-class constructors
   - It may also invoke functions that cannot leave and/or initialize member variables with default values or those supplied as arguments to the constructor

# Two-Phase Construction

Two-phase construction breaks object construction into two parts

2. A class method (typically called `ConstructL()`)
   - This method may be called once the allocated and constructed object has been pushed onto the cleanup stack
   - It completes construction of the object and may safely perform operations that may leave.
   - If a leave does occur, the cleanup stack calls the destructor to free any resources that had already been successfully allocated, and destroys the memory allocated for the object itself

# Two-Phase Construction

Splitting the construction allows a more atomic approach to handling leaves.

- The memory is allocated for foo and safely placed on the cleanup stack
- The leaving function **`ConstructL`** is called separately

```
class CFoo : public CBase
   {...                          //Details omitted for clarity
public:
    CFoo(TInt aVal){iVal=aVal;}

   void ConstructL()
      {
      iBar = new (ELeave) CBar(iVal);
      }
private:
  CBar* iBar;
   };
...
CFoo* foo = new (ELeave)CFoo(42);
CleanupStack::PushL(foo);
foo->ConstructL();
CleanupStack::Pop(foo);
```

# Introducing `NewL` and `NewLC`

## The previous example can introduce programming errors

- Some callers may forget to call `ConstructL()` after instantiating the object
  - after all, it is not a standard C++ requirement for construction

## The idiom

- Is best implemented if the class itself makes the call to the second-phase construction function, rather than the caller
- Obviously the code cannot do this from within the simple constructor, since this takes it back to the original problem of calling a method which may leave

# Introducing **`NewL`** and **`NewLC`**

## The Symbian OS idiom

- Is to provide a static function which wraps both phases of construction
- Providing a simple and easily identifiable means to instantiate objects of a class on the heap.
- The function is typically called **`NewL()`**
- A **`NewLC()`** function is often provided too and leaves the constructed object on the cleanup stack for convenience (as discussed previously in the cleanup stack section)

# Introducing `NewL` and `NewLC`

The `NewL()` and `NewLC` functions are static

- Thus they can be called without first having an existing instance of the class

The non-leaving constructors and second-phase `ConstructL()` functions have been made `private`

- A caller cannot instantiate objects of the class except through `NewL()`/`NewLC`
- Prevents duplicate calls to `ConstructL()` after the object has been fully constructed

# Example Implementation

```
/*static*/ CFoo* CFoo::NewLC(TInt aVal)
    {
    CFoo*self = new(ELeave) CFoo(aVal); // First-phase construction
    CleanupStack::PushL(self);
    self->ConstructL(aVal);             // Second-phase construction
    return self;
    }

/*static*/ CFoo* CFoo::NewL(TInt aVal)
    {
    CFoo* self = CFoo::NewLC(aVal);
    CleanupStack::Pop(this);
    return self;
    }
```

- The **`NewL()`** function is implemented in terms of the **`NewLC()`** function rather than the other way around (which would be slightly less efficient since this would require an extra **`PushL()`** call on the cleanup stack)
- Note the use of the Symbian OS overload of operator **`new(ELeave)`** this implementation will leave if it fails to allocate the required amount of memory.
- This means that there is no need to check for a **`NULL`** pointer after a call to **`new(ELeave)`**

# Deriving from a Class which uses Two-Phase Construction

## If any class is to be sub-classed

- The default constructor should be made **`protected`** rather than **`private`** so the compiler can construct the derived classes
- C++ will ensure that the first-phase constructor of a base class is called prior to calling the derived-class constructor

## Two-phase construction is not part of standard C++ construction

- The second-phase constructor of a base class will not be called automatically when constructing a derived class
- The second-phase **`ConstructL()`** method should be made private if it is not be called by a derived class, or protected if it does need to be called - *document this in your code*

# Deriving from a Class which uses Two-Phase Construction

## A class intended

- For extension through inheritance which uses the two-phase construction pattern typically supplies a protected method to do this
- Called **`BaseConstructL()`** rather than **`ConstructL()`**

## The derived class

- Calls this method in its own **`ConstructL()`** method to ensure that the base-class object is fully initialized

# Discussion Point: Using `PushL` in a Constructor

## Two-stage construction for a class could be avoided

- By including a `CleanupStack::PushL(this)` at the start of the class's C++ constructor
- This would achieve the same effect as using `ConstructL()` (since `PushL()` will always store the this pointer safely for later cleanup)

## If the class is to be used as a base class

- the constructor of any classes derived from it will incur the overhead of one `push` and `pop` in the constructor called at each level in the inheritance hierarchy

## This is less efficient

- Than one pop and push in its own `NewLC()`
- Better for C++ to automatically initialize the base classes, while the developer manages the resource allocation of base class objects in the most derived class's `NewLC()`

# Two-phase construction and Symbian OS types

## Two-phase construction is typically used only for C classes

- T classes do not usually require complex construction code as they do not contain heap-based member data
- R classes are usually created uninitialized requiring their callers to call `Connect()` or `Open()` to associate the R-class object with a particular resource


symbian Academy

# Two-Phase Construction and Object Destruction

## Object Destruction

- Know that it is neither efficient nor necessary to set a pointer to NULL after deleting it in destructor code
- 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

# Object Destruction

When implementing

- The standard Symbian OS two-phase construction idiom it is important to consider the destructor code carefully
- A destructor must be coded to release all the resources that an object owns
- The destructor may be called to clean up partially constructed objects if a leave occurs in the second-phase `ConstructL()` function

The memory for a `CBase`-derived object

- Is guaranteed to be set to binary zeroes on first construction
- It is safe for a destructor to call delete on a NULL pointer

But ...

# Object Destruction

## The destructor code

- Cannot assume that the object is fully initialized
- Should beware of calling functions on pointers which may not yet be set to point to valid objects
- Should beware of attempting to free other resources - by dereferencing them - without checking whether the handle or pointer which refers to them is valid:

```
CExample::~CExample()
    {
    if (iMyAllocatedMember)   // iMyAllocatedMember may be NULL
      {
      iMyAllocatedMember->DoSomeCleanupPreDestruction();
      delete iMyAllocatedMember;   // No need to set it to NULL
      }
    }
```


symbian Academy

# Two-Phase Construction and Object Destruction

- ✓ Two-Phase Construction
- ✓ Object Destruction