An Introduction to C Classes

This article provides a comprehensive introduction to Symbian C classes; heap based classes that require cleanup support.

Introduction to C Classes

Symbian OS standards define a set of class types which have different characteristics. These types are used to group classes with similar properties and behavior. They include:

- M classes – these are similar to Java interfaces. They can only contain pure virtual functions, that is, functions which must be implemented in derived classes, as M classes will only provide the declaration. M classes are always prefixed with the letter 'M', for example, `MNotify`.
- T classes – these are 'flat' classes. In other words they cannot own external data and as such usually do not need a destructor. They may own a pointer to external data but must not be responsible for maintaining it, for example, deleting it. T classes represent simple types and are always prefixed with the letter 'T', for example, `TRect` and `TPoint`.
- L classes – these are a new Core Idiom comprising a self-managing string handling class and template classes providing automated cleanup.
- R classes – these classes access external resources such as files. R classes are always prefixed with the letter 'R', for example, `RFile` and `RSocket`.

The most common type of class is the C class. C classes may own external data and are always a subclass of `CBase` or a `CBase`-derived class. Any class derived from `CBase` or a `CBase`-derived class has the prefix C, which stands for ‘class’ . See *Stichbury, Jo (2004), Symbian OS Explained* page 4.

How do you choose which type of class to use? One useful rule is if you need a destructor, you need a C class. Also, if you intend to do anything non-trivial, or particularly need a heap-based class, you need to use a C class.

Anatomy of a `CBase`-Derived Class

`CBase`-derived classes have the following properties:

- All member data is initialized to binary zeroes; this is performed by the overloaded operator new, as defined in `CBase`.
- They are allocated on the heap. One of the reasons for this is to ensure uniform initialization among C classes; as stack-based objects are not created using operator new, their member data would not be initialized to zero. If it were permitted to have C classes on the stack, we would end up in a situation where some of our newly initialized C classes had zeroed variables and others did not.
- They must be cleaned up if a leave occurs due to their being allocated on the heap; this imposes a requirement for cleanup consciousness when dealing with all C classes (leaves are discussed in Section 3.3).
- They are passed by pointer or reference and therefore do not need an explicit copy constructor or assignment operator unless there is clear intention that a particular class supports copying.
- They usually have non-trivial construction and, because of the possibility that a leave might occur during the construction process, a two-phase construction protocol is used. In this, the C++ constructor is only used for aspects of construction which cannot leave, and a `ConstructL()` function is used for aspects of construction which might leave. We will explore some of these points in later sections.

Generally, C classes cannot be used as inline data members in other C classes (although there are exceptions to this rule which will be discussed later). The reason for this is that if a C class appears as a member variable, a default constructor will be used for its creation. Therefore, you will not be able to make use of Symbian C++ two-phase construction and will lose the benefits that this provides. All C classes should have a private constructor to avoid this eventuality.

What is `CBase`?

CBase is the ultimate base class of all C classes. It is defined in `e32base.h` and has three main characteristics:

- It has a virtual destructor. This is used for standard cleanup processing, making cleanup of `CBase`-derived objects using the cleanup stack easy (see next sub-section). Having the virtual destructor in `CBase` means that we don’t have to declare our derived classes’ destructors as virtual, as it is already done for us.
- It overloads operator new to zero-initialize C class objects. The benefits of this are discussed later.
- It has a private copy constructor and assignment operator. This prevents a client from making shallow copies of a class unintentionally: a copy constructor and assignment operator needs to be defined if a class is to support copying.

The cleanup stack

The cleanup stack is the Symbian C++ method of managing memory when errors occur. It stores pointers to objects and uses these to safely destroy objects which would otherwise be orphaned and thus cause memory leaks.

Class `CleanupStack` is a collection of functions that are used to add resources to and remove resources from the cleanup stack. CBase provides seamless support for the cleanup stack (see section 4 for more details on this). Defined in `CleanupStack` are the following functions:

- `Check()` – checks whether the expected item is at the top of the cleanup stack.
- `Pop()` – pops an item from the cleanup stack. This can take a TInt specifying the number of items to pop, a pointer to the expected item, a pointer to the last item to pop along with `TInt aCount` (`aCount` -1 items are popped before checking whether the next item is the expected one; if so, it is popped) or no parameters, in which case the item at the top of the stack will be popped.
- `PopAndDestroy()` – pops and destroys items from the cleanup stack. This can take the same parameters as `Pop()`.
- `PushL()` – pushes an item onto the cleanup stack.

When C classes that were pushed onto the cleanup stack are popped and destroyed, the object's destructor is invoked.

Deriving from `CBase`

Remember your roots! What if you forget to derive from `CBase`? When the object is pushed onto the cleanup stack, the cleanup stack will try to push a `TAny*` rather than a `CBase*`. If the cleanup stack later tries to destroy the object, it will invoke `User::Free()` – no destructors are called and therefore a memory leak can occur.

`CBase` comes first

It is good practice that any class that derives from other non 'C' classes lists its base classes in the correct order. `CBase` or a `CBase`-derived class should be the first base class in the list:

```
class COurClass : public CBase, public MClass1
```

or, for a class further down the inheritance tree:

```
class CAnotherClass : public COurClass, public MClass2
```

This will emphasize the primary inheritance tree. If any other class is the first base class, you may have problems when using the cleanup stack.. This is because when a `CBase`-derived object is pushed onto the cleanup stack, an implicit conversion takes place: one of the overloaded `CleanupStack::PushL()` methods takes a `CBase*` and, as `COurClass` derives from `CBase`, it is this overloaded method that is called. The pointer passed to the cleanup stack actually points to the memory address of the `CBase` 'sub-object' rather than `COurClass`.

When the object is popped from the cleanup stack, the `CleanupStack::Pop(TAny* aExpectedItem)` method is called. This method compares the address of aExpectedItem with the address of the memory pointed to by the top-most pointer on the cleanup stack. If they match, the object is popped.

What happens when a `CBase` class is not first in the list? Let's look at it step by step:

- We have a class as follows:

```
class CMyCClass : public MMyInterface, public CBase
```

- As mentioned earlier, when a pointer to an instance of `CMyCClass` is pushed onto the cleanup stack, `PushL(CBase*)` is called. In order that the pointer points to the `CBase sub-object`, rather than the M class sub-object, the pointer is automatically incremented by 4 bytes for us (see Figure 1).
- Unfortunately, there is not a corresponding `Pop(CBase*)` function. What is called when your object is popped is `Pop(TAny*)`. As our M class is a `TAny, Pop(TAny*)` happily accepts the parameter without incrementing the pointer to point to the `CBase` sub-object. However, behind the scenes `CleanupStack::Pop(TAny*)` calls `Check()`, which in debug builds checks whether the pointer passed to `Pop()` matches the address at the top of the cleanup stack; of course, they differ by 4 bytes as the cleanup

stack's top-most address is the incremented `CBase` address, and an `E32USER-CBase 90` panic is raised. (A 'panic' is an error which cannot be dealt with. Unlike a leave, a panic will cause the application to simply exit.)

File:C Classes.jpg

Figure 1 : The layout of CTestClass (taken from Mixin Inheritance and the Cleanup Stack - detailed discussion as to why CBase must appear first in the base class list)

`CBase` compared to… It is interesting to compare `CBase` and its place in the class hierarchy with frameworks of other languages. For example, all Java classes are implicitly descended from the Object class, that is, there is no need to state this in your Java classes. In contrast, ISO C++ has no implicit base class, therefore developers have a 'blank canvas' to work with. However, the 'Microsoft Foundation Class Library' of C++ classes includes `CObject`, which is the base for all user-defined classes and provides the following basic services See Microsoft MSDN Visual C++ Developer Center - Microsoft MSDN Visual C++ Developer Center – reference page for CObject:

- serialization support
- run-time class information
- object diagnostic output
- compatibility with collection classes.

Thus we can see a CBase type scheme here, in that it provides a number of 'presets' which relieves developers of some tasks (`CBase`'s 'presets' and their benefits are discussed in more detail later.)

The virtual destructor The virtual destructor, inherited from `CBase`, means we can have a C class, `COurClass`, and a different object with a `CBase` pointer to an instance of `COurClass`. When the object is deleted, the correct destructor will be called:

```
  class COurClass : public CBase
    {
    // implementation of our class
    }
 void SomeOtherObject::SomeFunction()
    {
    CBase* anObject = new (ELeave) COurClass();
    delete anObject;
    }
```

The example above shows a `CBase`-derived class pointed to by a `CBase` pointer. When delete is called, the destructor in `COurClass` will be called.

{{Note| that Symbian overrides C++ new with new (`ELeave}). This overridden operator will leave if there is not enough memory available for the new object.`

Basic Usage Patterns

Cleaning up

The destructor of C classes should be coded to release all the resources owned by the object. Cleanup may occur on partially constructed objects, but this is still safe as `CBase`-derived classes are zeroed when operator new is used. It is always safe to delete NULL objects.

Resources will usually be indicated by pointers or handles. It is important that such pointers or handles are NULL when there is no resource allocated. This behavior is facilitated in `CBase`-derived classes because their memory is guaranteed to be initially set to binary zeroes.. Note that this automatic initialization only occurs when the object is first created: programmers must take care with pointers to owned objects which are repeatedly allocated and de-allocated throughout an object's lifetime. When such owned objects are de-allocated, the pointer should always be set to NULL, for example:

```
void CMyClass::SomeFunctionL()
 {
 delete iMemberData; // 1
 iMemberData = NULL; // 2
 iMemberData = CMyOtherClass::NewL(); // 3
 }
```

After the delete statement, the iMemberData variable points to the same chunk of memory but the object it pointed to has disappeared. If we didn't assign NULL to our member variable (comment 2), further attempts to delete iMemberData will be a case of double deletion and a NULL pointer exception panic will ensue. If `CMyOtherClass::NewL()` leaves at comment 3, the destructor of `CMyClass` will be called, which will delete the already-deleted iMemberData and will cause double deletion. So, whenever you delete a member pointer variable, always assign it to NULL afterwards.

It is not necessary to assign NULL to a deleted variable in the destructor, as the owning object will very shortly cease to exist and therefore the member variable cannot be dereferenced, for example:

```
CMyClass::~CMyClass()
 {
 delete iMyDataMember;
 // don't need to assign NULL!
 }
```

(Remember to be careful when deleting objects which interact with each other; it is important to get the order of deletion correct otherwise we may end up with an object relying on another object which no longer exists!)

Construction

Symbian C classes are usually constructed in two phases.

Create the class using new `(ELeave) CMyTestClass()`. This will in turn call the standard C++ `CMyTestClass()` constructor.

Initialize the object using `ConstructL()`.

Finally, when finished with the object, we delete it using delete.

It is asking a lot to expect developers to call new (`ELeave`) followed by `ConstructL()` every time they want to create an object; if the object will be created in lots of places, this would soon become a chore (although it may be more efficient as sometimes it is possible to avoid pushing the object onto the cleanup stack before calling `ConstructL()`). To automate this, it is usual Symbian OS practice to provide factory functions which do the work for us. For example:

Declaration of class:

```
class CMyTestClass : public CBase
   {
 public:
   static CMyTestClass* NewL(); // factory function
   static CMyTestClass* NewLC(); // factory function
   ~CTest();
 private:
   CMyTestClass();
   void ConstructL();
   TInt iSomeInt;
   RFile iMyMemberFile;
   CAClass* iMyMemberCClass;
   };
```

Notice that the first phase constructor, `CMyTestClass()`, and the second phase constructor, `ConstructL()`, are declared private; users are forced to safely create instances using two-phase construction: `NewL()` and `NewLC()` (and less commonly, `New()`):

Implementation of class:

```
CMyTestClass* CMyTestClass::NewL()
   {
   CMyTestClass* self = NewLC();
   CleanupStack::Pop(self);
   return self;
   }
CMyTestClass* CMyTestClass::NewLC()
  {
   CMyTestClass* self = new (ELeave) CMyTestClass();
   CleanupStack::PushL(self);
   self->ConstructL();
   return self;
   }
```

Notice the difference between `NewL()` and `NewLC()`: `NewLC()` pushes a pointer to the new object on the cleanup stack and leaves it there (in fact, the 'C' suffix in a function's name is a sign stating 'I have left an object on the cleanup stack, don't push it again!'), whilst `NewL()` pushes and then pops the pointer off the cleanup stack. Use `NewL()` when initializing member data, otherwise the object will be deleted twice; if `NewLC()` is used with a data member, both the cleanup stack and the created object will hold pointers to the data. Both the cleanup stack and the object's destructor will attempt to delete the object: one of these will effectively be trying to free memory which has already been freed and a panic will occur.

```
CMyTestClass::CMyTestClass() : iSomeInt(100)
  {
  // only non-leaving code here
  }
CMyTestClass::~CMyTestClass()
  {
  /* do object cleanup such as deleting C-classes and releasing R-classes */
  delete iMyMemberCClass;
  iMyMemberRFile.Close();
  }
void CMyTestClass::ConstructL()
  {
  // leaving code can go here
  }
Use of static New() functions
CMyTestClass* myObject = CMyTestClass::NewL();
or
CMyTestClass* myObject = CMyTestClass::NewLC();
```

Note that `NewL()` and `NewLC()` were declared as static functions; this allows them to be called before the object is created.

The reason we do it this way is so we can be sure that the created object is fully constructed and ready for use. If we have any leaving code in the constructor then a leave may occur part way through initialization of the object. This will mean the destructor of the partially constructed object will not be called, thus causing memory leak. So it is only safe to construct the objects in two phases.

Leaves

Historically, Symbian OS did not use C++ exceptions to deal with unforeseen error states. This is because C++ exceptions were not standardized when Symbian OS was designed. (When C++ exceptions were standardized they were thought to cost too much in terms of RAM overheads and compiled code size and so were not adopted [4].) Since v9, Symbian OS has used C++ exceptions behind the scenes to deal with error states and developers can use them directly if they wish to. However, the usual way to handle errors is through leaves.

Leaves are generated when an exception event occurs, such as there not being enough memory. Any function which attempts some task but may not succeed is called a 'leaving' function, and has the suffix 'L'. Any function which calls a leaving function is itself a leaving function and therefore must also have the 'L' suffix, unless it traps the calls. This naming convention is vital: if clients are aware that the function may leave, they will be able to defend against memory leaks. Leaves must eventually be trapped and dealt with. This is the job of the TRAPD and TRAP macros:

```
ThisFunctionDoesNotLeave()
  {
  TRAPD(result, ThisFunctionMayLeaveL());
  // handle any error from ThisFunctionMayLeave()
  }
```

TRAPD declares a variable to store the leave error code. When using TRAP, you need to declare this variable yourself:

```
ThisFunctionDoesNotLeave()
  {
  TInt result;
  TRAP(result, ThisFunctionMayLeaveL());
  // handle any error from ThisFunctionMayLeave()
  }
```

All programs must use at least one TRAP macro in order that any leave may eventually be caught and dealt with. This may mean providing a single TRAP at the uppermost level which deals with all leaves, or having TRAP macros nested in the code where they can deal more specifically with particular errors. Before C++ exceptions were used to implement leaves, TRAPs were expensive to use in terms of CPU cycles and memory consumption, so developers were well advised to keep their use to a minimum. Fortunately, the use of exceptions means TRAPs are now far more efficient.

One last point regarding TRAPs: don't add anything to the cleanup stack within a TRAP macro, for example, by calling an LC function (a function which may leave and will push an item onto the cleanup stack).

```
TRAPD(err, myVar = CSomething::NewLC()); // causes a panic
```

The reason for this is that TRAP (TRAPD in this example) will mark the cleanup stack before calling the leaving function. In the event of a leave, when execution returns to the TRAP macro, it expects all objects pushed onto the cleanup within the TRAP's scope to have been popped off. However, there will be one extra item on the cleanup stack (the caller of `CSomething::NewLC()` function will not pop the item) therefore an item will remain on the cleanup stack which cannot be destroyed and an E32User Panic 71 will be raised.

Coding Without C Classes

C classes, via their ultimate base, `CBase`, afford us many luxuries. To fully appreciate them, let's see what is involved when we avoid the use of `CBase`. For this investigation, we're going to create a heap-based object and then delete it. This sounds simple, but when broken down it becomes:

1. create the object on the heap
2. check whether the object is NULL
3. zero initialize the object – we have to do this ourselves as there is no CBase to do it for us
4. write a cleanup item for the object's destruction so that any leave will be correctly handled
5. push the cleanup item onto the cleanup stack
6. use the object, as required
7. when finished, pop and destroy the cleanup item.

In code this becomes:

```
 void UseNonCBaseClass()
   {
```

```
    SomeObject* myObject = new SomeObject(); // step 1
    if (myObject) // step 2
      {
      Mem::FillZ(myObject, sizeof(myObject)); // step 3
      TCleanupItem item(FreeAnObject, myObject); // step 4
      CleanupStack::PushL(item); // step 5
      /* use the object */ // step 6
      CleanupStack::PopAndDestroy(myObject); // step 7
      }
  else
      {
      /* handle class not initialized */
      }
    }
  void FreeSomeObject(TAny* aObject)
    {
    SomeObject* object = static_cast< SomeObject*>(aObject);
    delete object;
    }
```

As you can see, not making use of `CBase` makes the developer's life considerably more difficult. If we had used `CBase` as a base class, some of the above steps would have been automated. For example:

- step 3 would be handled by `CBase`'s overloaded operator new
- step 4 would be taken care of due to `CBase`'s cleanup stack support (in step 4 we construct a `TCleanupItem` with a pointer to a cleanup operation and a pointer to the object to be cleaned up. When `myObject` is popped from the cleanup stack, `FreeAnObject()` will be called).

In fact, if we derived from `CBase` and make use of two-phase construction, we don't need to worry about steps 2-5. (Of course, we could have used new (ELeave) in step 2 to cater for out of memory issues but this is another thing for users of the class to remember. The point is that correctly coded two-phase construction will eliminate this and other responsibilities.)

Exceptions to the Common Usage Pattern

Even though the rules about how C classes are defined and used apply in general, there are sometimes exceptions. For example, some C classes can be used as inline data members in other `CBase`-derived classes. One such class is `CDesCArrayFlat`.

The main reason that `CDesCArrayFlat` need not be a pointer member is that it has no second phase constructor. Initialization is done when you try to add an element. To make this work on the stack, the user must initialize the object to zero after it has been declared. This is required because when you try to insert any data at that point it checks if the memory is allocated for `CArrayFixBase::iBase`. If the object is not zeroed and it tries to use the member, a KERN-EXEC 3 panic will result. However, if it is zeroed then it will create the object properly.

`CActiveSchedulerWait`, like `CDesCArrayFlat`, can be used within a C class without needing a pointer. Again, this is because there is no second phase constructor, and as it is inside a C class, it is initialized to zero. If you create it on the stack you need to zero it with `Mem::FillZ()`.

Some Common C Classes

The `CArray` family

The `CArray` family includes classes for fixed and variable length arrays as well as pointer arrays and arrays of objects contained in flat and segmented buffers. Elements can either be deleted individually:

```
myVariableCArray.Delete(aIndex);
```

or all together:

```
myVariableCArray.Reset();
```

As well as deleting all elements, `Reset()` will free the memory allocated to the array.

In this family, there are classes that store objects by value and other classes that store pointers to objects. The latter type are all derived from `CArrayPtr`. When storing objects by pointer, you need to be careful with ownership: does the array own the items or is it just a useful way to access them. If the array owns the objects, then a useful function is:

```
myVariableCArray.ResetAndDestroy();
```

which will call delete on all the entries in the array before calling `Reset()`.

Graphics classes: devices, contexts and bitmaps

The base class for devices is `CGraphicsDevice`. This class inherits from two classes:

`class CGraphicsDevice : public CBase`, public `MGraphicsDeviceMap` deriving from `CBase` first. It has three main subclasses: `CFbsBitmapDevice`, `CFbsScreenDevice` and `CWsScreenDevice`. The first of these is for use with bitmap and the other two are for using the screen either directly, in the case of `CFbsScreenDevice`, or via the window server component.

The bitmap class is `CFbsBitmap`. This class represents a block of memory, which can be shared between processes, in which pixel data can be stored. The class has a size (width and height in pixels) and a color depth which specifies the format of the data for each pixel.

When drawing to a bitmap or the screen, a context is used. There are various base classes for contexts, including `CBitmapContext` and `CGraphicsContext`. There are two main context classes: `CFbsBitGc and CWindowGc`; the former can be used for drawing to a bitmap or directly to the screen and the latter for drawing to windows. These classes provide a device independent abstract interface, and also provide settings necessary for drawing to a device. Settings include:

- pen/brush color, style and size
- text alignment
- drawing modes, such as inverting the screen color.

Contexts are not created in the way that most C classes are. They are created using a factory function on the associated device:

```
TInt CreateContext(CGraphicsContext*& aGc);
```

However, the context returned by this function is then owned by the calling code, so it will need to be deleted.

CRichText

`CRichText` lets you create text with rich formatting. In rich text, each paragraph can have a different paragraph format and each character can have a different character format. Formatting is based on global format layers, as well as specific formatting applied to particular parts of the text with `ApplyCharFormat()` and `ApplyParaFormat()`. In the case of conflict, specific formatting overrides global formatting. Rich text also supports object embedding: objects are represented by `CPicture`-derived objects.

This class shows good object oriented design, with the functionality being built up through the base classes. It derives from `CGlobalText`, which provides the global format layers, in turn deriving from `CPlainText` which provides the text storage functionality.

CServer2

This is an abstract base class for servers. It is an active object (see Section 6.7) and so needs to be passed a priority upon creation. It accepts requests from client threads and forwards them to the relevant server-side client session. It also handles the creation of server-side client sessions as a result of requests from client threads. A basic outline of the handling of requests is as follows:

- A `CServer2` object is created by the server.
- A request for a connection by a client results in the creation of a new session. This is facilitated by a call to `CServer2::NewSessionL()`.
- This results in the creation of a `CSession2`-derived class.

CSession2

This represents a session for a client thread on the server-side. A session acts as a channel of communication between the client and the server. A client thread can have multiple concurrent sessions with a server, but this is unusual. More often, many different

client threads have sessions at the same time. This class includes the methods:

- `ServiceL()` – used to handle all messages except connect/disconnect requests (which are handled by `CreateL()` and `Disconnect()` respectively; the default implementation of these functions in `CSession2` is often all that is required). This method takes an `RMessage2` as parameter and must be implemented by the server writer.
- `CountResources()` – gets the number of resources currently in use.
- `ServiceError()` – handles the error situation if `ServiceL()` leaves.

CDictionaryStore

The abstract class `CDictionaryStore` provides the interface for a store which is accessed by a UID rather than directly by a stream ID. Its interface includes:

- `IsPresentL()` – this takes a UID as a parameter and tests whether the dictionary store holds a stream with the UID.
- `RevertL()` – this rolls back the dictionary to its state at the last commit point and leaves if it is unsuccessful. (Leaves were discussed in Section 3.3).
- `Commit()` and `CommitL()` – these methods commit changes, with `CommitL()` leaving and `Commit()` returning an error if it is unsuccessful.

`{{{1}}}== CActive` is the abstract base class which encapsulates the issuing of a request to an asynchronous service provider and the handling of completed requests. An application can have one or more active objects whose processing is controlled by an active scheduler. To implement an active object, a class needs to derive from `CActive` and takes a priority as a parameter to its default constructor:

```
CMyActiveObject::CMyActiveObject() : CActive(CActive::EPriorityStandard)
  {
  }
```

(If the priority needs to be changed during the life of the object, `SetPriority() can be called`.)

The active object can be registered with the active scheduler in `ConstructL()` (the second phase constructor, as discussed in Section 3.2):

```
void CMyActiveObject::ConstructL()
  {
  CActiveScheduler::Add(this);
  }
```

or even in the first phase construction, as the `CActiveScheduler::Add()` function cannot fail. To implement an active object, two functions need to be provided: `RunL()` is called by the active scheduler when the asynchronous service has occurred, and `DoCancel()` is called by `CActive::Cancel()` when the owner of the object is no longer interested in the asynchronous service.

CEikAppUi

Part of the Symbian application framework, `CEikAppUi`-derived classes handle events generated by the user, such as key and menu events, toolbar pop-up menus, opening and closing of files and exiting the application gracefully. Commands are handled in the `HandleCommandL()` method – this is overridden in subclasses to handle their particular requirements. In addition, `CEikAppUi` inherits a number of event-handling methods from `CCoeAppUi`, including:

- `HandleKeyEventL()` and `HandleSystemEventL()` – for key and system events.
- `HandleSwitchOnEventL()` – to handle the event of the device being switched on.
- `HandleForegroundEventL()` – to handle the event of the application being switched to the foreground.

CCoeEnv

`CCoeEnv` provides an active environment for creating controls, such as a list box. It implements active objects and an active scheduler. It also provides utility functions that are useful to many applications, such as:

- `ReadResource()` – this reads a resource into a descriptor.
- `SystemGc()` – returns the system graphics context.
- `WsSession()` – returns the windows server session owned by the application.
- `FsSession()` – returns the file server session owned by the CCoeEnv.
- `Format256()` – this reads a 256 byte resource into a formatted string.

For more details on all of the above classes, please see the latest SDK.

Best Practices

As you are probably aware by now, there are some recommendations regarding the use of C classes. To summarize:

- Make use of the two-phase construction protocol – this ensures objects are constructed safely.
- The destructor must be coded to release any resources owned by the object to avoid memory leaks.
- Generally, they cannot be used as inline data members in other C classes, as this would mean a default constructor would be used for their creation rather than the preferred two-phase construction.
- They must derive from `CBase` and, in order to avoid possible problems with the cleanup stack, `CBase` should be the first class in the derivation list.

Conclusion

This article has given a brief overview of C classes, some rules for their usage and also some exceptions to the rules. C classes contribute strongly to the robustness of Symbian OS applications; along with R, M, L and T classes, they allow the functionality required of a program to be separated into sensible ‘chunks’ – simple types, access to resources, interfaces and everything in between can be handled by one of these types of classes. They give the Symbian developer a helping hand in the design of code and, along with suffixes such as ‘L’ and ‘LC’, greatly improve the task of navigating third-party code. C classes are the most common class types you’ll use in Symbian development, so good luck, keep the guidelines in mind and you won’t go far wrong!

Further Reading

- The Implications of Leaving in a Constructor - why 2 phase construction is recommended
- Nokia Developer – discussion on two-phase construction: Nokia Developer
- NewLC – six essentials of a C class: six essentials of a C class
- NewLC – an overview of the different types of Symbian classes: www.newlc.com
- Symbian Application Reference – description of C, T, M and R classes
- Nokia Developer wiki – exception handling in Symbian OS: Exception handling in native Symbian C++
- *Symbian OS Explained*, Jo Stichbury
- An Introduction to R Classes
- An Introduction to M Classes
- An Introduction to T Classes
- Archived:An Introduction to L Classes
- A Comparison of Leaves and Exceptions
- Fundamentals of Symbian C++/Leaves & The Cleanup Stack

Company Profile

Penrillian makes mobile software work. We are experts in developing and porting applications across all the major mobile software platforms, turning great ideas in to real products, quickly and efficiently. Our diverse customer base includes global service providers and software companies such as Handmark, Sybase, T Mobile and Vodafone. They choose us because we are agile enough to respond to changing demands and always deliver, on time, and on budget.

Our core competencies include: Symbian OS, S60, UIQ, Java, Windows Mobile, secure applications, connectivity, RFID, and user interface design. We maintain close links with the major influencers and knowledge centres in the mobile software industry and we’re pleased to be a Symbian Platinum Partner, UIQ Alliance Partner and a member of Nokia Developer PRO.

To find out more about our consultancy, software porting, and application development services go to penrillian.com.

Author Biographies

Barry Drinkwater

Barry Drinkwater is a software developer at Penrillian. Barry graduated with first class honors in Computing. He started his career

in the software industry as a Software Test Engineer before progressing to his current position as Software Developer. He has been involved in numerous Symbian projects, such as a Bluetooth locator application, a live video streaming application and a network monitoring service.

Seeta Rama Murthy Velamakanni

Seeta Rama Murthy Velamakanni, BSc, MCA, ASD is a postgraduate from Osmania University, India. He is a Senior Symbian OS Developer at Penrillian and has 10 years of commercial programming experience, including eight years developing for Symbian OS. Projects that Murthy has worked on include: porting a Java environment to support a new Symbian OS look-and-feel, consultancy for an email synchronization development, and development of native SDK extensions for a well known Java environment. Murthy was lead developer for a port of a well known email synchronization client to UIQ3. He originally worked on Windows application development before becoming involved in Symbian OS development projects.