An Introduction to R Classes

Original author: Sorin Basca

This article provides a comprehensive introduction to Symbian R (resource) classes.

Introduction

Symbian C++ standards define a number of types of classes which are easily identifiable by the naming conventions used.1 'R classes,' or 'resource classes,' are one of these types of classes defined in the standards. They are so named because they hold a handle to a resource owned elsewhere, for example, in another local object, a server or the kernel. All the Symbian OS resource classes have their name preceded by an R so that they are easily recognized. Internal resource classes are often created to provide a clean, usable interface to a more complicated object, abstracting implementation detail which is not useful to the client. For example, the user of an array needs to know how to insert and delete items but not how the array allocates and grows the underlying memory block: the array R classes hide this unnecessary detail. The handle that the R class itself owns is to the start of an area of memory where the array data is stored. Many resource classes are used for accessing kernel services and have a handle to a matching object in the kernel. These classes provide functionality including library loading, using publish and subscribe, accessing timers and communicating with other threads and processes. R classes with handles to kernel objects are used for accessing services provided by other components, like the file server, the location acquisition framework and the central repository (the client-server framework uses the kernel for inter-process communications).

Anatomy of an R Class

By convention, all resource classes have the R prefix. The rest of the name usually refers to the type of object this is a handle to: for example, a timer (`RTimer`), an array (`RArray`) and a thread (`RThread`). The handle itself could be a pointer to a thread local block of memory or a `TInt` which specifies the number of the handle on the kernel side, if the object is external to the thread.

Usually the member variables of an R class are stack-based variables; the only exception to this is when it is the handle to a local resource, which is usually a pointer (for example, to a block of memory on the heap). In addition, R classes usually have few member variables and hence are small enough to create on the stack.

After the object is instantiated, it needs to be initialized. Most of the R classes have a method for opening the handle which can be called `Open()`, `OpenL()`, `Connect()` or a similar name which would indicate some sort of initialization. If this method is not called then the class cannot be used properly.

Because they are small and usually contain no other member data besides the resource handle, R classes rarely have a destructor. Instead, the classes provide the means of releasing any handle they own, which will result in cleaning up the associated resource. This is done through a `Close()` method or equivalent (for example, `Free()` or `Reset()`). An R class should be designed and implemented such that it is always safe to call `Close()`, even if the object hasn't been opened. Also, calling `Close()` repeatedly should not cause any double deletion. These conventions are in place so that the user of the R class is always able to call `Close()` on the class when it is not needed any more, without worrying about what happened previously.

Usually the interfaces of the resource classes are quite simple, abstracting unnecessary implementation detail. For example, an array resource class would provide methods that allow the appending, insertion, deletion and accessing of the elements – nothing that would require the user of the class to worry about the allocation of memory.

When designing an R class, you need to be careful about copying. The default '=' operator creates a shallow bitwise copy, so both objects point to the same resource, without actually having duplicated it. This can sometimes cause double deletion or inconsistencies when using the copies. If your class is not freely bitwise 'copyable,' you should prevent this by making the '=' operator and the copy constructor private. If you want to make a deep copy, implement a method called `Duplicate()` (or something similar). This ensures that the resource is being duplicated correctly.

These features can be illustrated through a simple example. Below is the definition of an R class which is used to store an array of integers using a handle to a local resource:

```
class RIntArray
    {
public:
    RIntArray();
    void OpenL(TInt aGranularity);
```


(C) Copyright Nokia 2014. All rights reserved.

```
    void Close();
    void AppendL(TInt aElement);
    void InsertL(TInt aElement);
    void Remove(TInt aIndex);
    TInt Remove(TInt aElement);
    TInt operator[](TInt aIndex) const;
    TInt Duplicate(const RIntArray& aIntArray);
private:
    void ReAllocL(TInt aNewSize);
    RIntArray(const RIntArray&);
    RIntArray& operator=(const RIntArray&);
private:
    TInt iSize;
    TInt* iArray;
    };
```

The class is small, having just two stack-based member variables, and can hence be created on the stack.

The interface is simple, providing just the methods needed to append, insert, remove, navigate through the elements and to do a copy of the array. The user does not need to understand the actual implementation: the header above is sufficient to be able to use the class. However, we will discuss some implementation details in the following sections.

The R class should have a constructor that initializes all the member variables to zero. Since the constructor cannot leave, there can be no allocation of memory in the constructor because this can fail. Space is allocated usually in the `Open()` type methods.

There is no need to define a destructor since the cleaning up gets done in the `Close()` method, which needs to be called by the user of the class when the object is no longer needed.

You can see that some methods are leaving methods: for example, `OpenL()`, `AppendL()` and `InsertL()`.

These methods leave in case of an error being raised, for example, if there is not enough memory. Other methods, like Remove(`TInt aElement`), do not leave but return an integer which is used for an error code, or `KErrNone`. So, if some error is raised in these methods they will not leave but instead return the error to the user who can decide what to do next. In the example of Remove(`TInt aElement`), this function can look for the given element in the array and if it finds it then removes it, otherwise it returns `KErrNotFound`.

Basic Usage Pattern

Initializing R classes

R classes are usually not fully initialized on construction. To access the resource that they contain, you need to call some method that connects to the resource or opens it. Typically, this method is called `Open()`, or `Connect()`, or something similar.

For example, taking the previously defined RIntArray class, the local handle iArray is a pointer to an area of memory. The class constructor will not allocate any space but will only initialize the size to 0 and the pointer to NULL, as follows:

```
RIntArray:: RIntArray()
      : iSize(0), iArray(NULL)
      {}
```

The constructor of the `RIntArray` class will not open the resource; to do this, the resource needs to be initialized by calling the `OpenL()` method (or some other equivalent). In the `RIntArray` example, `OpenL()` calls `ReAllocL()` to allocate an initial chunk of memory to the array before it is used, according to the granularity (how much space to pre-allocate when expanding the array) specified in the parameter. If there is not enough memory to allocate, `OpenL()` leaves. If {{Icode|OpenL()}] is not called, the class cannot be used properly.

When appending or inserting elements, the array can fill up, in which case the `ReAllocL()` method is called to allocate a new chunk of memory with a bigger size, the array is copied from the old memory location to the new one and, finally, the old memory is freed. If there is not enough memory, `ReAllocL()` leaves with `KErrNoMemory`, which causes `OpenL()`, `AppendL()` or `InsertL()` to leave.

(C) Copyright Nokia 2014. All rights reserved.

Closing R classes

The class has no destructor; instead, the `Close()` method must be called to release allocated memory before the R class goes out of scope. If `Close()` is not called, a memory leak can occur because the destructor of the R class does not actually clean up the resources.

In case a leave occurs after the object is initialized, the object should be added to the cleanup stack with the `CleanupClosePushL()` method (or an equivalent, like `CleanupReleasePushL()`, `CleanupDeletePushL()`, or `CleanupArrayDeletePushL()`) or by creating a `TCleanupItem` and adding it to the cleanup stack. The `CleanupClosePushL()` method adds a pointer to the object to the cleanup stack and ensures that `Close()` is called for it instead of delete in case of a leave or when `CleanupStack::PopAndDestroy()` gets called. The `TCleanupItem` is used when a different method needs to be called for cleaning up instead of `Close()`: for example, `Reset()`. Here is an example of using an R class object when it is a local variable:

```
void SomeMethodL()
     {
     RIntArray array;
     CleanupClosePushL(array);
     array.OpenL();
     AddThreeElementsL(array);
     CleanupStack::PopAndDestroy(&array);
     }
```

Note: Note:The R classes are not like C classes, which are derived from CBase. Therefore, since R classes are not derived from CBase, they are not automatically filled with zero on construction and hence all the member variables should be initialized.

You can see that the object is defined on the stack and is added to the cleanup stack as soon as it is declared. This is because `Close()` will do nothing if the `OpenL()` hasn't been called, otherwise it will release the memory. Also, at the end of `SomeMethod()`, `Close()` is not called; instead, `CleanupStack::PopAndDestroy()` is called with the address of the object as argument. This call will ensure that `Close()` gets called. The call is equivalent to:

```
CleanupStack::Pop(&array);
array.Close();
```

Notice that the argument to the `CleanupStack::PopAndDestroy()` is the address of the R class that is being destroyed. This is because the cleanup stack just holds pointers to the objects that need to be cleaned up.

Here is a similar example using `TCleanupItem`:

```
void RSomeClass::ResetCall(RSomeClass* aObject)
     {
     aObject->Reset();
     }
...
void SomeOtherMethodL()
     {
     RSomeClass rObject;
     TCleanupItem rObjectCleanup(RSomeClass::ResetCall(), &rObject)
     CleanupStack::PushL(rObjectCleanup);
     rObject.DoSomethingL();
     CleanupStack::PopAndDestroy(&rObject); // this will call
                                                                //
RSomeClass::ResetCall(&rObject)
                                                               // which in turn calls
rObject.Reset()
     }
```


(C) Copyright Nokia 2014. All rights reserved.

In this example, `RSomeClass` needs to have `Reset()` called for cleanup. A `TCleanupItem` object is created that has a pointer to the `RSomeClass` object and to a static method that receives as a parameter a pointer to the `RSomeClass` object. This method, which is usually written by the class implementer, knows how to cleanup the object, by calling the `Reset()` method. In the example above you can see how the static method `RSomeClass:ResetCall()` is implemented. So when `PopAndDestroy()` gets called, the right method will be used to do the cleanup.

Note: Note that you should not add a pointer to an R class directly to the cleanup stack because this means that when the object has to be cleaned up, the destructor will be invoked, which does not actually clean the resources. So, it is a mistake to add a stack-based R class object directly to the cleanup stack.

.

Although it is not recommended, if you declare an R class on the heap, you need to take care of its cleanup and of freeing the memory allocated on the heap. An example of how you would do this in a non-leaving method is:

```
TInt SomeNonLeavingMethod()
    {
    RSomeClass* rObject = new RSomeClass; // construction can’t leave
    if(!rObject) // but will return a NULL pointer if there isn’t enough memory
        {
        return KErrNoMemory;
        }
    rObject->Open();
    rObject->DoSomething();
    rObject->Close(); // need to close first
    delete rObject; // and to delete after
    return KErrNone;
    }
```

The example above shows the ‘two-stage’ cleanup but none of the methods can produce any errors. If they did, then you would have to check for an error and return it in case it is different from `KErrNone`. It is easier to allow leaving but in this case you need to add the object on the cleanup stack, and this has to happen twice since the object is on the heap. Here is an example of how this is done:

```
void SomeLeavingMethodL()
    {
    RSomeClass* rObject = new(ELeave) RSomeClass;
    CleanupStack::PushL(rObject); // first add the pointer on the cleanup stack
    CleanupClosePushL(*rObject); // then add the object for closing
    User::LeaveIfError(rObject->Open()); // leave if there is an error in Open
    rObject->DoSomethingL();
    CleanupStack::PopAndDestroy(rObject); // this will close
    CleanupStack::PopAndDestroy(rObject); // this will free the memory
    }
```

Having an R class object on the heap complicates its cleanup; because of this and because R classes are small in size, it is not recommended to instantiate them on the heap.

To automatically call `Close()` on R classes (that support cleanup by calling `Close()`) when they go out of scope, you can use the `TAutoClose` helper class to implement behavior similar to Resource Acquisition Is Initialization (more popularly known as `RAII`). This is especially helpful for R classes that are member variables:

```
class CFoo : public CBase
    {
protected:
```


(C) Copyright Nokia 2014. All rights reserved.

```
    void ConstructL();
private:
    TAutoClose<RSocketServ> iSockServ;
    HBufC* iBuf;
    };

void CFoo::ConstructL()
    {
    User::LeaveIfError(iSockServ.iObj.Connect());
    iBuf = HBufC::NewL(10);
    }

CFoo::~CFoo()
    {
    // Note no need to call iSockServ.Close()
    delete iBuf;
    }
```

`TAutoClose` also adds value when used with automatic variables, producing cleaner code that would otherwise require multiple explicit calls to {{lcode|Close()}}, as the example below illustrates:

```
TInt ReadFile(const TDesC& aFile)
    {
    TAutoClose<RFile> file;
    TInt r = file.iObj.Open(aFile, KFileStreamText|EFileExclusive);
    if (r != KErrNone)
    {
    return(r);
    }
TBuf<100> buf;
if ((r = file.iObj.Read(buf)) != KErrNone)
    {
    return(r);
    }

...
}
```

A note about `Close()`: an R class is usually designed and implemented such that it is always safe to call `Close()`, even if the object hasn't been opened. However, this behavior is not guaranteed.

Copying R classes

If you use the '=' operator to copy a class it will usually result in a shallow copy. This can be dangerous; for example, for the `RIntArray` class we defined above, if we hadn't made the '=' operator private, the following code will cause a double deletion of the memory where the array is stored:

```
RIntArray arrayA;
arrayA.OpenL(8);
arrayA.AppendL(5);
RIntArray arrayB;
arrayB = arrayA;
arrayA.Close();
arrayB.Close();
```


(C) Copyright Nokia 2014. All rights reserved.

So, shallow copying is not recommended, but if you need to use it then take care with sharing and closing the handle. Taking the example above, having made the copy, if `arrayA` is expanded and memory needs to be reallocated, any attempt to access arrayB will actually access a deallocated part of memory. This can be avoided by keeping arrayB up-to-date when `arrayA` is updated. To make it safer it is best to disallow shallow copying.

In the `RIntArray`, the copy constructor and '=' operator are made private so that shallow copy is avoided. The `Duplicate()` method provided allocates an area of memory and copies the information from the original array into it. This ensures safety when copying. You could override the '=' operator, but then you do not have the possibility of returning an error, or of leaving, in case of some exception (for example, where there is not enough memory).

In general, because the copy is usually a shallow copy, avoid passing an R class into a function by value; always pass by reference instead.

Exceptions to the Common Usage Pattern

Although the rules about how R classes are defined and used apply in general, there are some exceptions; these are presented in some of the examples of R classes below.

- R classes that do not have an `Open()` function, or equivalent:

These include `RArray` and `RPointerArray`. These classes don't need an `Open()` function because the memory is allocated locally and the allocation happens when elements are appended or inserted.

- R classes where calling `Close()` alone will not free the memory:

Classes in this category include `REComSession`, `RProperty` and `RPointerArray`. For these classes, additional steps are required to ensure that no leaks occur. See the examples of common R classes in section 5 for more details on the cleanup of these classes.

- R classes with no `Open()` or `Close()`:

Classes such as `RDebug` have all their methods defined as static, so there is no need to open or close any handles.

- R classes which initialize with a default handle:

Classes in this category include `RThread` and `RProcess`. These classes are created with a default initialization to the current thread or process. Therefore you do not need to call `Open()` unless you want to open a new thread or process. Also, you only need to call `Close()` on an object if you called `Open()` on it.

Some Common R Classes

This section provides information about some commonly used Symbian C++ R classes. The examples are divided into R classes that have a handle to a local resource and those that have a handle to a resource in the kernel. The latter are further divided into R classes that access resources in the kernel and/or just use some kernel service, and those classes that communicate with a different process through the client-server architecture.

Resource classes that have handles to local resources

`RArray` is a class for holding fixed length objects in an array. It provides a handle to a chunk of memory where an array is stored.

This class is a templated class and the type of the elements in the array is taken from the template type. This is used for arrays of simple structures that are stack-based: more precisely, T classes. But the objects added to the array have to be word aligned, so the size of the T class should be a multiple of four. `RArray` does not provide any `Open()` equivalent because as soon as it is instantiated it can be used. The allocation of the memory happens when necessary, such as for the first append or when the size of the array reaches the limit of memory allocated. When the array is no longer used, `Close()` has to be called to free the space taken by the array. The class provides functionality for working with arrays, for example, finding, appending, inserting, sorting or accessing individual elements. Copying is shallow and should be avoided.

`RPointerArray` is similar to `RArray`, but instead of holding the elements directly, it holds the pointers to the objects. `RPointerArray` is used for arrays of more complex, bigger classes such as the C classes. Sometimes the `RPointerArray` objects take ownership of the elements added to them, other times they do not. You need to check the documentation of the code that is using the `RPointerArray` object to see if ownership should be passed to the array or not.

If the array of pointers does not take ownership of the objects then, since someone else is responsible for deleting the objects, `Close()` is sufficient for releasing the memory. However, if the array does take ownership, `Close()` will not delete the objects themselves and `ResetAndDestroy()` needs to be called instead; this will go through the array, delete each object and then close the array.

(C) Copyright Nokia 2014. All rights reserved.

Note: Note that ResetAndDestroy() calls Close(), so there is no need to do this separately.

`RBuf` provides a buffer that contains, accesses and manipulates data. It is a descriptor which provides the means of handling data in the same way as a `TPtr` but it holds its own resources, rather than pointing to another descriptor. `RBuf` inherits from {{icode|TDes}}, so it can be passed around as a descriptor. Underneath, it handles memory allocated on the heap, either through an HBufC or through a TPtr. The equivalent method to `Open()` is `Create()`. The `Assign()` method copies another descriptor and can be used instead of `Create()` or it can be used at a later stage; however, take care when using `Assign()` as it doesn't free what is already contained in the `RBuf`.

`RDesReadStream` and `RDesWriteStream` provide support for reading and writing a stream from an 8-bit descriptor.

`Open()` only needs to be used if the default constructor is used, otherwise the other constructor will call `Open()`. The handle is to the descriptor provided either in the constructor or to `Open()`.

Reading or writing is done in the functions defined in the base classes `RReadStream` and `RWriteStream`. Since the descriptor handled by these classes should be cleaned up by whoever owns it, these classes don't leak any memory if not closed. However, it is always good practice to ensure `Close()` is called.

Note: Note that calling Close() will detach from the descriptor and will guarantee that no writing is done to it afterwards. This avoids the conflict of the descriptor being cleaned up while the stream still points to it and tries to write or read from it.

Resource classes that have handles to resources in the kernel for using its services

`RProperty` is the class used for publish and subscribe. It holds a handle to a property defined on the kernel side. The property can be defined, set, read and deleted statically from any thread that has the right capabilities, without having an instance of the class. If you want to be notified of changes to the property, you need an instance of the class. Closing the class is necessary to release the handle, but the property exists as long as it is defined. So, to release the memory for the property, `Delete()` has to be called explicitly.

`RHandleBase` is the base class for holding a handle to an object on the kernel side. This object is reference counted. The class is not meant for instantiation, but for inheritance; this is why the `Open()` method for the handle is protected. The handle is actually an integer which is uniquely identified with an object on the kernel-side. This class provides the means of closing this handle. If `Close()` is not called, the resources leaked will be on the kernel-side until the thread dies (the kernel will free all the thread-owned resources on thread termination). Copying is shallow but `Duplicate()` is provided for correctly creating an additional handle to the object. `RHandleBase` is generally used as the base class for a client-side interface to a server. If `Close()` is not called specifically, it is possible that the server will never know it is not needed any more and so it will not shut down, which means that part of the memory will be occupied by an idle server.

`RTimer` provides asynchronous timer services. The class holds a handle to a timer resource on the kernel side. To open a handle to the resource you need to call `CreateLocal()`. When the timer is no longer needed, `Close()` needs to be called.

Other examples of classes which use kernel services are `RThread`, `RProcess`, `RChunk`, `RCriticalSection`, `RSemaphore`, etc. All these classes inherit from `RHandleBase`, which is the base class for all classes using kernel-side objects.

Resource classes that have handles to resources in the kernel for client-server communication

`RSessionBase` is the client side handle to a session with a server. Communications from the client are made through this class. An actual client implementation inherits from this class. `CreateSession()` is used to open a session and connect to the server. If the server is not present because it is not started, then the client has to try and start it. Messages to the server are sent through `Send()` or `SendReceive()`, which can be synchronous or asynchronous. This class inherits from `RHandleBase` and so holds the handle to a reference-counted object on the kernel-side, which enables all of the communication with the server.

`RSubSessionBase` is used for sub-sessions with a server. A sub-session is more lightweight than a session; it does not need so many resources allocated for the communication, as it uses what the session provides already. A sub-session can only be started within a session.

`RMessage` is used to read the information in the IPC call from the server-side and is used by the server to respond to the client.

It provides a handle to an object pointing to memory in the client-side and so it can retrieve information sent to the client and write to that memory. There is no `Close()` method but if `SendReceive()` was used from the client-side, the server should call `Complete()` with an error code (or `KErrNone`) to notify the client of the outcome of its request.

(C) Copyright Nokia 2014. All rights reserved.

`RComSession` is a singleton session class that communicates with the ECOM server. It uses the server to retrieve plug-ins dynamically. It is a special R class because the session is a singleton, so there is only one session to the ECOM server per thread. `OpenL()` is a static function that will return the single session instance and will increase the reference count for clients that use this session. Each client should call `Close()` on the session; this will decrease the reference count but this will not close the session. The peculiarity of the class is that it has a two-step closing process and so to close the actual session to the server, `FinalClose()` must be called. This method does nothing if there are still clients to the session but if all of them are closed, it will complete the session to the server. Not calling `FinalClose()` after all the work with the ECOM framework has been done will cause a memory leak panic on the client-side.

Other examples of R classes used for client-server communication are the client-side classes `RDbs`, `RFs`, `RDir`, `RFile`, `RPositioner`, etc.

Best Practices

With R classes it is important to be careful about cleanup. Make sure that `Close()`, or the equivalent method for releasing the resources, is always called before the class goes out of scope or in the event of a leave. If the R class object is a member variable, then call `Close()` in the destructor of the class that owns it. Calling `Close()` is safe even if the object is not open! There are some rare exceptions to this but these should be in the documentation, so make sure you are using the class as documented.

If the R class object is instantiated locally it is good to add it to the cleanup stack and `PopAndDestroy()` it at the end. Make sure you add the object to the cleanup stack in the appropriate way, either by calling `CleanupClosePushL()` (or other similar functions) or by using a `TCleanupItem`. Never add a stack-based R class object to the cleanup stack directly by just adding a pointer to it through `CleanupStack::PushL()`. Alternatively, you can use TAutoClose to ensure `Close()` will be called on return.

Here is an example of using the R class as a member variable:

```
class CMyClass : public CBase
    {
public:
    ~CMyClass();
private:
    void DoSomethingL();
private:
    RHandleClass iHandle;
    };

void CMyClass::DoSomethingL()
    {
    User::LeaveIfError(iHandle.Connect());

    ...
    }
CMyClass::~CMyClass()
    {
iHandle.Close();
    }
```

In the example, if `DoSomethingL()` is never called, `iHandle` does not get connected. Even in this case, calling `Close()` will be safe. So, we ensure `Close()` will always be called by having this call made in the destructor of `CMyClass`.

Avoid copying an R class directly, unless it has a documented operator= or a Duplicate() method. Otherwise, try to copy everything manually; for example, for an array, copy each element. If you want to share a resource and there is no `Duplicate()` or equivalent method, be very clear about who actually owns the resource and who is responsible for calling `Close()`. Also ensure that all copies of the handles to the resource are kept up-to-date as the resource changes.

Due to the issues with copy, always have R classes parameters as a reference and use const if they are not going to be modified. When you use an R class, make sure you know all of its peculiarities, especially when it comes to cleanup. For example, be aware that you need to call `FinalClose()` for `RComSession`, or that you need to call `ResetAndDestroy()` for `RPointerArray` if you

(C) Copyright Nokia 2014. All rights reserved.

own the objects in the array.

If you write an R class, remember that it will be used on the stack and so the size has to be small. If you need to hold a lot of information, consider holding it in another class and make the R class an interface to this larger class. It is very unusual to put an R class object on the heap, as it adds complications and is usually considered an error.

Conclusion

R classes abstract away from the actual implementation and provide a cleaner interface than, for example, C classes would. If you need to access, or share, one resource then use an R class; if you need something more complex, like accessing multiple resources, then another class type might be more appropriate.

R classes are usually used as interface classes for services and for interfacing to some data types. They are stack-based and easy to use. The only complexity comes in the cleanup, more precisely the necessity to call `Close()`, and in the sharing of the resources.

© 2010 Symbian Foundation Limited. This document is licensed under the Creative Commons Attribution-Share Alike 2.0 license. See http://creativecommons.org/licenses/by-sa/2.0/legalcode for the full terms of the license.

Note that this content was originally hosted on the Symbian Foundation developer wiki.

(C) Copyright Nokia 2014. All rights reserved.