

# 1

## Class Name Conventions on Symbian OS

**Llanfairpwllgwyngyllgogerychwyrndrobwlllantysiliogogoch**

*The longest place name in the British Isles, said to be invented to attract tourists*

Symbian OS defines several different class types, each of which has different characteristics. The categories are used to describe the main properties and behavior of objects of each class, such as where they may be created (on the heap, on the stack or on either) and, particularly, how they should be cleaned up. Each of the class types has a well-defined set of rules which makes the creation and destruction of instances of that class quite straightforward.

To enable the types to be easily distinguished, Symbian OS uses a simple naming convention which prefixes the class name with a letter (usually T, C, R or M). Naming conventions aren't always popular, but this one isn't difficult to follow and is clearly of value, since it allows the behavior of a class to be easily identified, particularly with respect to cleanup. As a class designer, the class types simplify matters. You consider the required behavior of your class, and match it to the definitions of the basic Symbian OS types. Having chosen a type, you can then concentrate on the role of the class. By the same token, as a client of an unfamiliar class, the naming convention indicates how you are expected to instantiate an object, use it and then destroy it in a leave-safe way.

### 1.1 Fundamental Types

I'll discuss the main features of each class type in this chapter. However, before doing so let's go back to basics and consider the fundamental types. Symbian OS provides a set of `typedefs` of the built-in types,

which are guaranteed to be compiler-independent; these should always be used instead of the native types.

- `TIntX` and `TUIntX` (for  $X = 8, 16$  and  $32$ ) for 8-, 16- and 32-bit signed and unsigned integers respectively. Unless you have a good reason to do so, such as for size optimization or compatibility, you should use the non-specific `TInt` or `TUInt` types, which correspond to signed and unsigned 32-bit integers, respectively.
- `TInt64`. Releases of Symbian OS prior to v8.0 had no built-in ARM support for 64-bit arithmetic, so the `TInt64` class implemented a 64-bit integer as two 32-bit values. On Symbian OS v8.0, `TInt64` and `TUInt64` are `typedef'd` to `long long` and use the available native 64-bit support.
- `TReal32` and `TReal64` (and `TReal`, which equates to `TReal64`) for single- and double-precision floating point numbers, equivalent to `float` and `double` respectively.<sup>1</sup> Operations on these are likely to be slower than upon integers so you should try to avoid using them unless necessary.
- `TTextX` (for  $X = 8$  or  $16$ ), for narrow and wide characters, correspond to 8-bit and 16-bit unsigned integers, respectively.
- `TAny*` should be used in preference to `void*`, effectively replacing it with a `typedef'd` "pointer to anything". `TAny` is thus equivalent to `void` but, in the context where `void` means "nothing", it is not necessary to replace the native `void` type. Thus, a function taking a `void*` pointer (to anything) and returning `void` (nothing) will typically have a signature as follows on Symbian OS:

```
void TypicalFunction(TAny* aPointerParameter);
```

This is the one exception to the rule of replacing a native type with a Symbian OS `typedef`; it occurs because `void` is effectively compiler-independent when referring to "nothing".

- `TBool` should be used for boolean types. For historical reasons `TBool` is equivalent to `int` and you should use the Symbian OS `typedef'd` values of `ETrue` ( $=1$ ) and `EFalse` ( $=0$ ). Bear in mind that C++ will interpret any nonzero value as true. For this reason, you should refrain from making direct comparisons with `ETrue`.
- Each `TBool` represents 32 bits, which may be quite wasteful of memory in classes with a number of flags representing state or settings.

---

<sup>1</sup> Note that these are `typedefs` and should not be confused with the Symbian OS `TRealX` class, which describes an extended 64-bit precision real value.

You may wish to use a bitfield rather than a number of booleans, given that the 32 bits of a single `TBOOL` could hold 32 boolean values in a bitfield. Of course, you should consider the potential code complexity, and trade that off against the benefits of a smaller object.



**The `typedef`'d set of Symbian OS built-in types are guaranteed to be compiler-independent and should be used instead of the native types except when returning `void` which is equivalent to "nothing".**

## 1.2 T Classes

T classes behave much like the C++ built-in types, hence they are prefixed with the same letter as the `typedefs` described above (the "T" is for "Type"). Like built-in types they have no destructor and, consequently, T classes should not contain any member data which itself has a destructor. Thus, a T class will contain member data which is either:

- "plain ol' data" (built-in types) and objects of other T classes
- pointers and references with a "uses a" relationship rather than a "has a" relationship, which implies ownership. A good example of this is the `TPtrC` descriptor class, described in Chapter 5.

T classes contain all their data internally and have no pointers, references or handles to data they own (although references to data owned by other objects is allowed). The reason for not allowing ownership of external data is because the T class must not have a destructor.

Without a destructor, an object of a T class can be created on the stack and will be cleaned up correctly when the scope of that function exits, either through a normal return or a leave ("leaving" is discussed in detail in Chapter 2). If a T class were to have a destructor, Symbian OS would not call it in the event of a leave because leaves do not emulate the standard C++ `throw` semantics. If a call to the destructor were necessary for the object to be safely cleaned up, the object could only be created on the stack in the scope of code which can be guaranteed not to leave – which is clearly somewhat restrictive.

An object of a T class can also be created on the heap. Such an object should be pushed onto the cleanup stack prior to calling code with the potential to leave. In the event of a leave, the memory for the T object is deallocated by the cleanup stack (which is discussed in detail in Chapter 3) but no destructor call is made.

T classes are also often defined without default constructors; indeed, if a T class consists only of built-in types, a constructor would prevent member initialization as follows:

```
TMyPODClass local = {2000, 2001, 2003};
```

However, in the rare case that the T class has exported virtual functions, a default constructor must be exported because it is required for any client code to link against. The reasons for this are discussed in Chapter 20, which explains the `EXPORT_C` syntax.

As a rule, the members of a T class will be simple enough to be bitwise copied, so copy constructors and assignment operators are trivial and the implicit compiler-generated versions are likely to be more efficient. So you generally don't need to write a copy constructor or assignment operator unless, of course, you want to prevent cloning, in which case you should declare them both private to your class and leave them unimplemented.

Some T classes have fairly complex APIs, such as `TLex`, for lexical analysis, and the descriptor base classes `TDesC` and `TDes`, described in Chapter 5. In other cases, a T class is simply a C-style `struct` consisting only of public data (traditionally, a `struct` was prefixed with S instead of T, but more recent Symbian OS code tends to make these T classes too).

You'll find the T prefix used for enumerations too, since these are simple types. For example:

```
enum TMonthsOfYear {EJanuary = 1, EFebruary = 2, ..., EDecember = 12};
```



**A T class must not have a destructor.**

## 1.3 C Classes

Classes with the C prefix<sup>2</sup> ultimately derive from class `CBase` (defined in `e32base.h`). This class has two characteristics which are inherited by its subtypes and thus guaranteed for every C class.

Firstly, `CBase` has a virtual destructor so a `CBase`-derived object may be destroyed properly by deletion through a `CBase` pointer. This

<sup>2</sup> In case you are wondering, the "C" stands for "Class", which perhaps makes a "C class" something of a tautology, though it is an accurate reflection of the fact that it is more than the simple "Type" described by a T class.

is common when using the cleanup stack, since the function invoked when pushing a `CBase`-derived object onto the cleanup stack is the `CCleanupStack::PushL(CBase* aPtr)` overload.

If `CCleanupStack::PopAndDestroy()` is called for that object (or if a leave occurs), the object is deleted through the `CBase` pointer. The virtual destructor in `CBase` ensures that C++ calls the destructors of the derived class(es) in the correct order (starting from the most derived class and calling up the inheritance hierarchy). So, as you'll have gathered, unlike T classes, it's fine for a C class to have a destructor, and they usually do.

It is worth noting at this point that, if the class does not derive from `CBase` and is pushed onto the cleanup stack, the `CCleanupStack::PushL(TAny* aPtr)` overload will be used instead. As described above, when `PopAndDestroy()` is called, or a leave occurs, the memory for the object will be deallocated but no destructor calls made. So if you do not inherit from `CBase`, directly or indirectly, even if your base class destructor has a virtual destructor, objects of your class will not be cleaned up as you expect.

The second characteristic of `CBase`, and its derived classes, is that it overloads `operator new` to zero-initialize an object when it is first allocated on the heap. This means that all member data in a `CBase`-derived object will be zero-filled when it is first created. You don't have to do this explicitly yourself in the constructor. Zero initialization will not occur for stack objects because allocation on the stack does not use `operator new`. This could potentially cause unexpected or different behavior between zero-filled heap-based and non-zeroed stack-based objects. For this reason, among others such as managing cleanup in the event of a leave, objects of a C class must *always* be allocated on the heap.

Clearly, when it is no longer needed, a heap-based object must be destroyed. Objects of C classes typically exist either as pointer members of another class or are accessed by local pointer variables. If owned, the `CBase`-derived object should be destroyed by a call to `delete`, for example in the destructor of the owning class. If the C class object is not owned, instead being accessed through a local pointer, that pointer must be placed on the cleanup stack prior to calling any code with the potential to leave – otherwise it will be orphaned on the heap in the event of a leave. I'll discuss this further in Chapter 3.

If you look at `e32base.h`, you'll notice that `CBase` declares a private copy constructor and assignment operator. This is a common strategy used to prevent a client from making accidental shallow copies of, or assignments to, objects of a class. If such an operation is valid for your class you must explicitly declare and define a public copy constructor and assignment operator, because their private declaration in the base class means they cannot be called implicitly. However, given the nature

of C classes, a deep copy operation may well have the potential to leave, and you should never allow a constructor (or destructor) to leave (I describe the reasons for this in more detail in Chapter 4). If you need to allow C class copying, rather than defining and implementing a public copy constructor, you should add a leaving function, e.g. `CloneL()` or `CopyL()`, to your class to perform the same role.

Since most C classes tend not to be straightforward enough to be bitwise copied, the implicit copy is best avoided, and this is yet another benefit of deriving from `CBase`. The private declaration of a copy constructor and assignment operator in the `CBase` class means you don't have to privately declare them in every C class you write in order to prevent clients from making potentially unsafe "shallow" copies.



**Objects of a C class must always be allocated on the heap.**

## 1.4 R Classes

The "R" which prefixes an R class indicates a resource, usually a handle to an external resource such as a session with the file server. There is no equivalent `RBase` class, so a typical R class will have a constructor to set its resource handle to zero, indicating that no resource is currently associated with the newly constructed object. You should not attempt to initialize the resource handle in the constructor, since it may fail; you should never leave in a constructor, as I'll explain in Chapter 4.

To associate the R class object with a resource, the R class will typically have a function such as `Open()`, `Create()` or `Initialize()` which will allocate the resource and set the handle member variable as appropriate or fail, returning an error code or leaving. The class will also have a corresponding `Close()` or `Reset()` method, which releases the resource and resets the handle value to indicate that no resource is associated with the object. It should thus be safe to call such a function more than once on the same object. Although, in theory, the cleanup function can be named anything, by convention it is almost always called `Close()`.

A common mistake when using R classes is to forget to call `Close()` or to assume that there is a destructor which cleans up the owned resource. This can lead to serious memory leaks.

R classes are often small, and usually contain no other member data besides the resource handle. It is rare for an R class to have a destructor – it generally does not need one because cleanup is performed in the `Close()` method.

R classes may exist as class members or as automatic variables on the stack, or occasionally on the heap. You must ensure the resource

will be released in the event of a leave, typically by using the cleanup stack, as described in Chapter 3. Remember, if using an R class object as a heap-based automatic variable, you must make sure that the resource is released as well as freeing the memory associated with the object itself, typically by using two push calls: `CleanupClosePushL()`, or a similar function, to ensure that the resource is cleaned up, and a standard `CleanupStack::PushL(TAny*)` which simply calls `User::Free()` on the heap cell.

The member data of an R class is typically straightforward enough to be bitwise copied, so you would not expect to see an explicit copy constructor or assignment operator unless a shallow copy would result in undefined behavior. An example of this might be where the duplication of a handle value by bitwise copy would make it ambiguous as to which object owns the resource. In these circumstances undefined behavior might result if both copies attempt to release it. Calling the `Close()` function on the same object may be safe because the handle value member is reset, but the same cannot always be said for calling `Close()` twice on the same underlying resource via two separate handle objects. If one object frees the resource, the handle held by the other is invalid.

If your class contains a handle to a resource which cannot be shared safely by bitwise copy, you should declare a cloning function in the event that it must be copied. If you want to prevent any kind of copying for objects of your R class, you should declare, but not define, a private copy constructor and assignment operator.

The rules for R classes are more open to interpretation than C or T classes, so you'll find more different "species" of R class. Within Symbian OS the types of resources owned by R classes vary from ownership of a file server session (class `RFs`) to ownership of memory allocated on the heap (class `RArray`).



**`Close()` must be called on an R class object to cleanup its resource.**

## 1.5 M Classes

Computing folklore relates that "mix-ins" originated from Symbolic's Flavors, an early object-oriented programming system. The designers were apparently inspired by Steve's Ice Cream Parlor, a favorite ice cream shop of MIT students, where customers selected a flavor of ice cream (vanilla, strawberry, chocolate, etc) and added any combination of mix-ins (nuts, fudge, chocolate chips and so on).

When referring to multiple inheritance, it implies inheriting from a main "flavor" base class with a combination of additional "mix-in" classes which extend its functionality. I believe the designers of Symbian OS adopted the term "mixin", and hence the M prefix in turn, although the use of multiple inheritance and mixins on Symbian OS should be more controlled than a trip to an ice cream parlor.

In general terms, an M class is an abstract interface class. A concrete class deriving from such a class typically inherits a "flavor" base class (such as a CBase, or a CBase-derived class) as its first base class, and one or more M class "mixin" interfaces, implementing the interface functions. On Symbian OS, M classes are often used to define callback interfaces or observer classes.

An M class may be inherited by other M classes. I've shown two examples below. The first illustrates a concrete class which derives from CBase and a single mixin, MDomesticAnimal, implementing the functions of that interface. The MDomesticAnimal class itself derives from, but does not implement, MAnimal; it is, in effect, a specialization of that interface.

```
class MAnimal
{
public:
    virtual void EatL() =0;
};

class MDomesticAnimal : public MAnimal
{
public:
    virtual void NameL() =0;
};

class CCat : public CBase, public MDomesticAnimal
{
public:
    virtual void EatL(); // From MAnimal, via MDomesticAnimal
    virtual void NameL(); // Inherited from MDomesticAnimal
    ... // Other functions omitted for clarity
};
```

The second example illustrates a class which inherits from CBase and two mixin interfaces, MRadio and MClock. In this case, MClock is not a specialization of MRadio and the concrete class instead inherits from them separately and implements both interfaces. For M class inheritance, various combinations of this sort are possible.

```
class MRadio
{
public:
    virtual void TuneL() =0;
};
```

```

class MClock
{
public:
    virtual void CurrentTimeL(TTime& aTime) =0;
};

class CClockRadio : public CBase, public MRadio, public MClock
{
public:
    virtual void TuneL();
    virtual void CurrentTimeL(TTime& aTime);
    ... // Other functions omitted for clarity
};

```

The use of multiple *interface* inheritance, as shown in the previous examples, is the *only* form of multiple inheritance encouraged on Symbian OS. Other forms of multiple inheritance can introduce significant levels of complexity, and the standard base classes were not designed with it in mind. For example, multiple inheritance from two CBase-derived classes will cause problems of ambiguity at compile time, which can only be solved by using virtual inheritance:

```

class CClass1 : public CBase
{...};

class CClass2 : public CBase
{...};

class CDerived : public CClass1, public CClass2
{...};

void TestMultipleInheritance()
{
    // Does not compile, CDerived::new is ambiguous
    // Should it call CBase::new from CClass1 or CClass2?
    CDerived* derived = new (ELeave) CDerived();
}

```

Let's consider the characteristics of M classes, which can be thought of as equivalent to Java interfaces. Like Java interface, an M class should have no member data; since an object of an M class is never instantiated and has no member data, there is no need for an M class to have a constructor.

In general, you should also consider carefully whether an M class should have a destructor (virtual or otherwise). A destructor places a restriction on how the mixin is mixed in, forcing it to be implemented only by a CBase-derived class. This is because a destructor means that `delete` will be called, which in turn demands that the object cannot reside on the stack, and must always be heap-based. This implies that an implementing class must derive from CBase, since T classes never possess destructors and R classes do so only rarely.

In cases where the ownership of an object that implements an M class is through a pointer to that interface class, it is necessary to provide some means of destroying the object. If you know you *can* restrict implementers of your interface class to derivation from `CBase`, then you can provide a virtual destructor in the M class. This allows the owner of the M class pointer to call `delete` on it.

If you do not define a virtual destructor, deletion through an M class pointer results in a `USER 42` panic, the reasons for which are as follows: a concrete class that derives from an M class also inherits from another class, for example `CBase`, or a derived class thereof. The mixin pointer should be the second class in the inheritance declaration order<sup>3</sup>, and the M class pointer will thus have been cast to point at the M class subobject which is some way into the allocated heap cell for the object, rather than at the start of a valid heap cell.

The memory management code attempting to deallocate the heap space, `User::Free()`, will panic if it cannot locate the appropriate structure it needs to cleanup the cell. This is resolved by using a virtual destructor.

However, as an alternative to defining the interface as an M class, you should consider simply defining it as an abstract `CBase`-derived class instead, which can be inherited as usual by a C class. `CBase` already provides a virtual destructor, and nothing else, so is ideal for defining an interface class where its implementation classes can be limited to C classes. Of course, in these cases, the implementation classes will be limited to single inheritance, because, as I described above, multiple inheritance from `CBase` results in ambiguity and the dreaded "diamond-shape" inheritance hierarchy.

In general, a mixin interface class should not be concerned with the implementation details of ownership. If it is likely that callers will own an object through a pointer to the M class interface, as described above, you must certainly provide a means for the owner to relinquish it when it is no longer needed. However, this need not be limited to cleaning up through a destructor. You may instead provide a pure virtual `Release()` method so the owner can say "I'm done" – it's up to the implementing code to decide what this means (for a C class, the function can just call "delete this"). This is a more flexible interface – the implementation class can be stack- or heap-based, perform reference

---

<sup>3</sup> The correct class definition is

```
class CCat : public CBase, public MDomesticAnimal{...};
and not
class CCat : public MDomesticAnimal, public CBase{...};
```

The "flavor" C class must always be the first specified class of the base class list, to emphasize the primary inheritance tree. It also enables C class objects of derived classes such as these to be placed on the cleanup stack using the correct `CleanupStack::PushL()` overload (see Chapter 3 for more details).

counting, special cleanup or whatever. By the way, it isn't essential to call your cleanup method `Release()` or `Close()`, but it can help your callers if you do. First of all, it's recognizable and easy enough to guess what it does. More importantly, it enables the client to make use of the `CleanupReleasePushL()` function described in Chapter 3.

Like a Java interface, an M class should usually have only pure virtual functions. However, there may be cases where non-pure virtual functions may be appropriate. A good example of this occurs when all the implementation classes of that interface have common default behavior. Adding a shared implementation to the interface reduces code duplication and its related bloat and maintenance headaches. Of course, there's a restriction on what this default implementation can do, because the mixin class must have no member data. Typically, all virtual functions are implemented in terms of calls to the pure virtual functions of the interface.



**An M class has similar characteristics to a Java interface. It has no member data and no constructor. The use of multiple interface inheritance using M classes is the only form of multiple inheritance encouraged on Symbian OS.**

## 1.6 Static Classes

We've reached the end of the naming convention prefixes, though not quite the end of the types of class commonly found on Symbian OS. There are a number of classes, with no prefix letter, that provide utility code through a set of static member functions, for example, `User` and `Mem`. The classes themselves cannot be instantiated; their functions must instead be called using the scope resolution operator:

```
User::After(1000); // Suspends the current thread for 1000 microseconds
Mem::FillZ(&targetData, 12); // Zero-fills the 12-byte block starting
                             // from &targetData
```



**Classes which contain only static functions do not need to have a name prefix.**

## 1.7 Buyer Beware

It's a good rule of thumb that for all rules there are exceptions. The class name conventions on Symbian OS are no exception to that rule and there

are classes in Symbian OS code itself which do not fit the ideals I've put to you above. There are a few classes in Symbian OS which don't even conform to the naming conventions. Two well-documented exceptions are the kernel-side driver classes and the heap descriptor (`HBufC`), which is discussed further in Chapter 5.

This doesn't mean that the code is wrong – in many cases there are good reasons why they do not fit the theory. In the lower-level code, in particular, you'll find cases which may have been written before the name conventions were fully established or which, for efficiency reasons, have different characteristics and behavior. Whenever you come across a new class, it's worth comparing it to the rules above to see if it fits and, if not, considering why it doesn't. You'll be able to add a set of good exceptions to the rules to your list of cunning Symbian OS tricks, while discarding any code that unnecessarily contravenes the conventions – thus learning from others' mistakes.

## 1.8 Summary

This chapter reviewed the major class types used when coding for Symbian OS, and described their main features such as any special requirements when constructing or destroying objects of the class, whether they can be stack- or heap-based and typical member data contained by the class (if any). In particular, the chapter discussed the use of the class name conventions to indicate the cleanup characteristics of objects of the class in the event of a leave.

The guidelines within this chapter should be useful when writing a class – they can help you save time when coding and testing it, if only by cutting down on the rewrite time. If you can stick as closely as possible to Symbian OS conventions, your clients will know how you mean your class to be constructed, used and destroyed. This naturally benefits them, but can also help you reduce documentation and support-time further down the track.