



This is intended to be a simple list of things you should and should not do when writing Symbian C++ programs. A short explanation is provided in each case. They are not in any particular order, just how they came to mind.

## Contents

- [1 Never leave from a destructor](#)
- [2 Think about the need for a virtual destructor](#)
- [3 Initialize member data in T classes and structs](#)
- [4 Think about the error handling](#)
- [5 Don't use writable static data \(WSD\) in a dll](#)
- [6 Be careful with casting](#)
- [7 Use the checked functions in CleanupStack](#)
- [8 Think about the access specifiers \(public, protected, private\)](#)
- [9 Never ignore function return values](#)
- [10 Never ignore compiler warnings](#)
- [11 Don't use Copy\(\) to convert descriptors](#)
  - ◆ [11.1 External links](#)
- [12 Don't confuse CIdle with CPeriodic](#)
  - ◆ [12.1 External links](#)
- [13 Always make CBase the first base of a C class](#)
- [14 Be careful when comparing TBool variables](#)
- [15 Use bit shifts to define bit flag values](#)

## Never leave from a destructor

Do not allow a destructor to leave! The destructor may have been called as a part of a routine to handle an exception (leave) and raising another exception will cause the program to be terminated. All calls to leaving functions must therefore be trapped.

## Think about the need for a virtual destructor

Virtual destructor should always be defined for base classes. There are however a few exceptions to the rule:

**C classes** have a virtual destructor automatically because it is defined in **CBase**. Leaving the virtual keyword out from a C class makes no difference.

If an **M class** has a destructor, it has to be virtual. If the M class is used to define an **observer** it should not have a destructor at all. If the object is meant to be deleted through the M class pointer, the destructor has to be virtual for it to work.

**T class** *should* not have a destructor at all because they are mostly allocated on stack. This is an old Symbian convention that is not really necessary in S60 3rd Edition or later versions. Leaves and traps were previously implemented with setjmp and longjmp which meant that when the stack was being unwound the destructors of stack-based objects were not called. This would cause a very common memory leak problem so the

convention was introduced to avoid it. Now that the leaves are implemented in terms of exceptions the destructors are called and the memory leak problem doesn't exist anymore.

**R class** can have a destructor and it should be virtual if the class is to be derived from. Usually however the cleanup is done with a Close() function.

It is worth thinking about whether or not the destructor of a class should be virtual or not. If the class is not meant to be derived from and it doesn't itself derive from another class where it inherits virtual functions the compiler will not need to generate a vtable for it. If the destructor is declared virtual "just in case" in a simple class like this, the generated vtable will be a waste of memory.

## Initialize member data in T classes and structs

Always initialize member variables in T class and struct constructors (Note, a struct is a class so it can have a constructor). Only classes derived from CBase (C classes) get their member variables zero initialized automatically. If the variables of T classes or structs are not explicitly initialized, they will contain garbage. The best place to do the initialization is the constructor initializer list like so:

```
TClass::TClass()  
    : iMemberTInt( 0 )  
    , iMemberTBool( EFalse )  
    , iMemberTBuf()  
    {  
    }  
}
```

Destructors should not be defined for T classes or structs because they can not own heap based objects. In earlier Symbian and S60 releases the destructors of automatics (objects in stack) were not even called as part of leave handling. The stack was simply unwound. In S60 3rd Edition and later releases the leaves are implemented as C++ exceptions and C++ guarantees that the destructors are called. Even so, T class destructors are just a curiosity in Symbian. They should not be defined except for perhaps logging purposes.

## Think about the error handling

Keep error handling as simple as possible. A leaving should not return an error code and a function that returns an error code should not leave. Do not use leaves as part of normal program logic. They should only be used for error situations.

If you define your own error codes, they should be negative values like the system codes because **User::LeaveIfError()** for example will not do anything if it is given a positive value. Start the codes from a low enough value that they don't overlap with the system error codes.

Never **Panic** the UI thread except from server when the client is behaving badly. **Panic** is a very abrupt and ugly resolution to a problem. The user is not shown any meaningful error message about what happened, only that the application has been closed. Always aim to handle the error situation gracefully and let the user know what has happened.

Always keep the error messages to a level that the user can understand. The user doesn't know or care about the internal logic of an application so the error messages should not include any debug information except in debug builds.

Think about the need for a virtual destructor

## Don't use writable static data (WSD) in a dll

No not use it unless you really have to. It is very expensive memory-wise. Symbian build tools strip off the data segment of a dll to save memory. It makes the dll binary smaller and the loader doesn't need to allocate a memory chunk for the writable data.

In Symbian v9.1 onwards (S60 3rd Edition) the writable static data can be enabled by putting EPOCALLOWDLLDATA in the mmp file, but this should not be used lightly. It is a huge waste of memory and in the emulator the dll can only be loaded to one single process. It should only be considered when porting code that will not work without it.

TLS (Thread Local Storage) should be used instead. TLS allocates space for one pointer in every thread created. The pointer can be used to store any object.

## Be careful with casting

If the code contains lots of casting, there is probably something wrong with the design.

Do not use old C-style casts. They can be dangerous because they allow casting from anything to anything. C++ casts provide additional error checking by separating them into different categories depending on their purpose. The compiler can complain if the code attempts to do a cast that doesn't make sense.

C++ casts are intentionally "ugly" to encourage the programmer to make sure casting is necessary and also to make them easier to spot in the code. Because the old C-style casts are so hard to spot from the code Symbian has defined macros that correspond to the C++ casts: `STATIC_CAST`, `CONST_CAST`, `REINTERPRET_CAST` and `MUTABLE_CAST(const_cast in C++)`. Nowadays they are defined in terms of the C++ casts so the macros should not be used anymore.

Casts can also be grouped based on their "safety" and the safer casts should be preferred to the less safe ones. List of casts from the safest to the least safe is as follows:

1. **static\_cast<T>** is the simplest and safest one to convert one type to another type. It is safe because it relies on compile-time type information. If it compiles, it works.
2. **const\_cast<T>** is used to cast away the const-ness or volatile-ness of a type for example from a function parameter. If the value passed in the function is declared as `const`, then casting away the const-ness of the parameter will not give write access to the value. Should not be used unless really necessary. Casting away const-ness of **this** pointer in a `const` function is an indication of bad design. **mutable** keyword should be used to declare the variable that must be modifiable from a `const` function.
3. **dynamic\_cast<T>** is not usable in Symbian C++ because it is implemented in terms of Runtime Type Identification (RTTI).
4. **reinterpret\_cast<T>** can be used to convert between two unrelated types. Use only when absolutely necessary.

## Use the checked functions in CleanupStack

Always use the type-checking versions of Pop() and PopAndDestroy(). There are very few cases when they won't work and the other overloads must be used. This additional type-checking makes sure any small mistakes are caught as early as possible. Type-checking versions can be used to pop stack objects, but the & operator must be used because the Pop() and PopAndDestroy() functions expect a pointer parameter.

```
// Not recommended
CleanupStack::Pop();
CleanupStack::Pop( 3 );
CleanupStack::PopAndDestroy();
CleanupStack::PopAndDestroy( 3 );

// Recommended
CleanupStack::Pop( object );
CleanupStack::PopAndDestroy( &stackObject );
CleanupStack::PopAndDestroy( 2, object );
```

You Should also be careful about where to use CleanupStack::Pop(\*PObject) and CleanupStack::PopAndDestroy(\*PObject).

Keep in Mind that- **"It is never possible for an object on heap to be cleaned-up more than once."** If You attempt to delete an object which is already been released from the heap will definitely cause a system panic. e.g. If a pointer to an object which is pushed onto the CleanupStack is also kept elsewhere and it is accessible even after the leave, then we are supposed to use CleanupStack::Pop(\*PObject).Dont use CleanupStack::PopAndDestroy(\*PObject).

## Think about the access specifiers (public, protected, private)

The access specifiers define API of the class so you should pay attention to what you declare **public**, **protected** or **private**.

Public data is almost always a bad idea because it breaks encapsulation.

As a general rule you should always try to declare everything private except the functions and data that are meant to be accessible from outside of the class. Protected keyword should only be used in base classes to give the deriving classes more access. When deriving a class, the inherited overwritten functions should be declared private except if they specifically need to be public or protected. In the observer pattern for example the public pure virtual functions should be private in the concrete observer implementation.

When deriving a class, the access specifier works like when declaring a function. It defines whether or not the base class will be visible from outside of the class. Public specifier is used almost always, private very rarely and protected never.

## Never ignore function return values

Don't ignore function return values. Return values should be checked and appropriate action should be taken if

an error occurs. There are situations where no action can be taken when an error occurs so the return value can be ignored, but the fact that it is ignored should be clearly commented. The simplest way in an L function is to wrap the call to the function inside **User::LeaveIfError()**.

## Never ignore compiler warnings

Never ignore compiler warnings. The times when the compiler complains "for nothing" are rare. Even though warnings of unused function parameters seem annoying, they should be fixed either removing the parameters or commenting them out. If you get into the habit of ignoring minor warnings you'll soon ignore major ones as well.

## Don't use Copy() to convert descriptors

Never use a simple Copy() to convert 16-bit descriptor to an 8-bit one because it doesn't do any real conversion, it simply strips off the high bits. Copy() is safe in the other direction because it fills the high bits with zeros and the meaning of the data doesn't change. It is recommended to always use **CnvUtfConverter** to do the job because it will perform an actual conversion.

```
// Bad
_LIT( K16BitTestData, "Unicode characters" );
TBuf8<30> buf8;
buf8.Copy( K16BitTestData );

// Acceptable
_LIT( K8BitTestData, "Non-unicode characters" );
TBuf16<30> buf16;
buf16.Copy( K8BitTestData ); // Fills high bits with zeros

// Good
_LIT( K16BitTestData, "Unicode characters" );
TBuf8<30> buf8;
CnvUtfConverter::ConvertFromUnicodeToUtf8( buf8, K16BitTestData );
```

## External links

[CnvUtfConverter](#)

## Don't confuse CIdle with CPeriodic

**CIdle** and **CPeriodic** are basically both timers. They both call a static **TCallback** function but the similarities end there. **CIdle** is triggered immediately and **CPeriodic** is triggered after a specified delay. The important difference between them is how they are stopped and that is the thing that should not be confused. **CIdle** can be stopped by returning **EFalse** from the callback function, returning **ETrue** will cause the callback to be called again as soon as possible. **CPeriodic** will keep on ticking regardless of the callback return value, it doesn't even check it. It has to be canceled explicitly by calling **Cancel()**.

```
TInt CSomeClass::CIdleCallback( TAny* aSelf )
{
```

```

// Do something
...
return EFalse; // Not called again
}

TInt CSomeClass::CPeriodicCallback( TAny* aSelf )
{
// Do something
...
static_cast<CSomeClass*>( aSelf )->iPeriodic->Cancel();
return 0; // Value ignored
}

```

## External links

[TCallback](#)

[CIdle](#)

[CPeriodic](#)

## Always make CBase the first base of a C class

If you are writing a C class that also derives from M classes, make sure **CBase** or some other C class is the first base class. If an M class is the first base, you will not be able use the [CleanupStack](#) properly. The type-checking versions of **Pop()** and **PopAndDestroy()** will panic with **E32USER-CBase 90** when they don't find **CBase** as the first base.

## Be careful when comparing TBool variables

Comparing two TBool variables is not as simple as it seems. From a logical perspective it would seem obvious that you should simply use the **==** operator directly to compare them. In some environments this is the case, but not in Symbian. Be careful.

If you take a look at the **TBool** definition you can see that it is simply a typedef to an **int**. It does not use the built-in **bool** type. This is because the **bool** type didn't exist in the C++ standard when Symbian was first implemented and for compatibility reasons it hasn't been changed since.

Because integer values can be evaluated logically as true or false the **TBool** works almost like **bool**, but not quite. The **bool** type can have only two values, namely true or false whereas **TBool** can have **KMaxTInt** values which is a lot. 0 is evaluated as logically false and all other values are evaluated as logically true. So if you compare two **TBools** that are both logically true your code might not work because the actual values of the **TBools** are different.

```

TBool first = ETrue;    // Actual value is 1
TBool second = 16;     // Still logically true

// WRONG:
if ( first == second ) // Will not match because values are different
...

```

Don't confuse CIdle with CPeriodic

## DOs\_and\_DON'Ts\_of\_Symbian\_C++

```
// RIGHT:
if ( !first == !second ) // Will match because negating the values makes them 0
...

```

So the code may look a bit strange if you're not used to seeing that, but you have to learn to live with it. Negating both of the values makes them 0 and they match.

You might think that this is just a minor peculiarity that is not important because you use the **ETrue** and **EFalse** macros and you would be wrong. **TBitFlags** is just one example class whose member functions return **TBool** values that are not limited to 1 and 0. It will return 0 for false values and the bit position as the true value. If you use such components in your code but you want to limit the **TBools** your interface returns you can simply negate the values twice. 16 => 0 => 1.

## Use bit shifts to define bit flag values

If you need bit flags in your code you can minimize the chance of accidentally defining wrong by using bit shifts. For some developers that are used to implementing lots of low level code defining bit values is second nature, but others may need to think about it a little more to get it right. So if the sequence 0x1, 0x2, 0x4, 0x8, 0x10, 0x20, 0x40 feels too error prone, consider using bit shifts.

```
enum TBitValues
{
    EFlagFirst           = 1 << 0 // Binary: 000001 Hex: 0x1
    ,EFlagSecond         = 1 << 1 // Binary: 000010 Hex: 0x2
    ,EFlagThird          = 1 << 2 // Binary: 000100 Hex: 0x4
    ,EFlagFourth         = 1 << 3 // Binary: 001000 Hex: 0x8
    ,EFlagFifth          = 1 << 4 // Binary: 010000 Hex: 0x10
    ,EFlagSixth          = 1 << 5 // Binary: 100000 Hex: 0x20
    ....
};

```