WSD and the Singleton Pattern

Original Author: Jo Stichbury

This article examines the limitations on using modifiable global data (also known as writable static data, or WSD) in Symbian platform DLLs. There are some circumstances where it's advisable not to use modifiable global data on Symbian platform, but the limitation occurs only in DLLs – if you're writing an EXE you are not affected.

The article also explains the use of alternatives to WSD: TLS and CCoeStatic.

What is Modifiable Global Data?

What we mean by "modifiable global data" in this discussion, is any non-constant globally-scoped variable or any non-constant function-scoped static variable. For example:

```
TBufC<20> fileName; // modifiable global data
void SetFileName()
  {
  static TInt iCount; // static variable
  ...
  }
```

As shorthand, Symbian typically refer to this kind of variable as WSD, which stands for writable static data.

The Bad News

WSD was not allowed in DLLs built to run on phone hardware containing the original kernel architecture, EKA1. That means no WSD in DLLs for **Symbian OS v8.1a or earlier** (see the table below for the corresponding versions of S60 and UIQ).

With the advent of the EKA2 kernel architecture (Symbian^1, Symbian OS v9.x and Symbian OS v8.1b), the situation changed for the better, and the use of WSD in DLLs became possible. However, because it can be expensive in terms of memory usage, it is not recommended when writing a shared DLL that will be loaded into a number of processes. This is because WSD in a DLL typically consumes 4KB of memory for *each* process into which the DLL is loaded.

When a process loads its first DLL containing WSD, it creates a single chunk to store the WSD. The minimum size of a chunk is 4KB, and that amount of memory is consumed irrespective of how much static data is actually required. Any memory not used for WSD will be wasted (however, if subsequent DLLs are loaded into the process and also use WSD, the same chunk can be used, rather than separate 4KB chunks for every DLL that uses WSD).

Since the memory is per-process, the potential memory wastage is:

```
(4KB – WSD Bytes) × number of client processes
```

If, for example, a DLL is used by 4 processes, that’s potentially an “invisible” cost of 16KB (minus the memory occupied by the WSD itself).

Furthermore, DLLs that use WSD are not fully supported by the Symbian OS emulator that runs on a Windows PC. The emulator can only load a DLL with WSD into a single running process (because of the way the Symbian emulator is implemented on top of Windows). If a second process attempts to load the same DLL on the emulator, it will fail with `KErrNotSupported`. Symbian^1 introduces a workaround to this. Further information is available in Symbian Platform Support for Writeable Static Data in DLLs.

There are a number of occasions where the benefits of using WSD in a DLL outweigh the disadvantages (for example, when porting code that uses WSD significantly, and for DLLs that will only ever be loaded into one process). As a third party developer, you may find the memory costs of WSD acceptable, for example, if you create a DLL that is intended only to be loaded into a single application. Note, however, that the current supported version of the GCC-E compiler has a defect such that DLLs with static data may cause a panic during loading.

However, if you are working in device creation (for example, to create a shared DLL that ships within Symbian OS, one of the UI platforms, or on a phone handset) the tradeoffs are different. Your DLL may be used by a number of processes, and the memory costs and constraints are such that using WSD is less likely to be justifiable.

The Good News

If you're working on Symbian^1,Symbian OS v9, you are less likely to be affected by the limitation on WSD than if you are working on earlier versions of Symbian OS. Prior to Symbian OS v9, applications were built as DLLs, and developers porting applications that used WSD had to find a workaround. A change in the application framework architecture in Symbian OS v9 means that all applications are now EXEs rather than DLLs. Writable static data has always been allowed in EXEs, so an application can now use WSD if it is required.

The following table summarizes the support for WSD in DLLs and applications on various versions of the Symbian UI platforms:

UI Platform	Symbian OS version	Symbian OS kernel	WSD allowed for DLLs running on the device?	Applications allowed WSD?
UIQ 2.x S60 1st and 2nd Editions	v7.0 (UIQ) v7.0s, v8.0a, v8.1a (S60)	EKA1	Not supported	No
UIQ 3 S60 3rd Edition	v9	EKA2	Supported, but not recommended without understanding the limitations on the Windows emulator and memory consumption required.	Yes – applications are EXEs

Usage

By now, I should have answered your question about the limitations of modifiable global data in Symbian OS DLLs but, for more information, please see Symbian Platform Support for Writeable Static Data in DLLs. Let's now move on to what happens when you do use WSD. The following discussion will assume that you're working on Symbian OS v9 only.

First of all, there are occasions where you may find you use WSD inadvertently. Some classes have a non-trivial constructor, which means that the objects must be constructed at run time. You may not think you are using WSD but, because an object isn't initialized until the constructor for the class has executed, it counts as modifiable rather than constant. Here are a few examples:

```
static const TPoint KGlobalStartingPoint(50, 50);
static const TChar KExclamation('!');
static const TRgb KDefaultColour(0, 0, 0);
```

On most versions of Symbian OS, if you attempt to build DLL code that uses WSD, deliberately or inadvertently, you'll get a error when you build the DLL for phone hardware. The error will look something as follows:

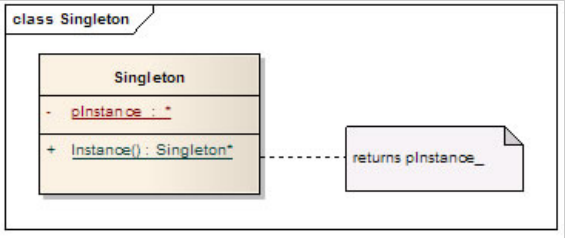
```
ERROR: Dll 'TASKMANAGER[1000C001].DLL' has uninitialised data
```

You'll always find that a DLL that uses WSD builds for the Windows emulator. It is only when code is compiled for phone hardware that the use of WSD is flagged as an error. Reference [1] explains how to track down the WSD if you're not using it deliberately. If you are sure you want to use WSD, you can explicitly enable it by adding the keyword `EPOCALLOWDLLDATA` to the DLL's MMP file.

Note: **Note:** there are some builds of Symbian OS (for example, Symbian OS v9.3, found in S60 3rd Edition FP2) that don't actually flag WSD in a DLL as an error, irrespective of whether `EPOCALLOWDLLDATA` is present in the MMP file. This occurs because a particular build flag is turned on by default.

Implementation of Singleton in the Absence of WSD

One of the common snags for developers porting code from other platforms, is that the restriction on WSD in Symbian OS DLLs affects the implementation of the classic Singleton design pattern. Singleton is arguably the most popular pattern found in the classic book Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison Wesley, 1995. It is one of the simplest design patterns, and involves only one class that provides a global point of access to a single instance, which instantiates itself.



UML Diagram for a typical Singleton

The classic implementation of the Singleton pattern in C++ uses WSD, as shown below:

```
class Singleton
  {
public:
  static Singleton* Instance();
  ... // Operations supplied by Singleton
private: // For clarity, these methods aren't implemented below
  Singleton();

  ~Singleton();
private: // Static Singleton member
  static Singleton* pInstance_;
  };

/*static*/ Singleton* Singleton::pInstance_ = NULL;

/*static*/ Singleton* Singleton::Instance()
  {
  if (!pInstance_)
    pInstance_ = new Singleton();
  return (pInstance_);
  }
```

As I described earlier, it's possible to implement Singleton in a DLL on Symbian OS v9, simply by explicitly enabling WSD in the MMP file. You can then define a Singleton class such as the one shown above, using Symbian C++ naming conventions and the standard idioms used to cope with the potential for leaves during instantiation.

However, if you want to avoid the potential of extra memory consumption and emulator testing limitations, there is an alternative mechanism available: thread local storage (TLS). TLS can be used in the implementation of Singleton in DLLs on all versions of Symbian OS (it can also be used in EXEs if desired). TLS is a single per-thread storage area, one machine word in size (32 bits on Symbian OS v9). A pointer to a local, heap-based, Singleton object is saved to the TLS area, and whenever access to the Singleton is required, the pointer in TLS is used to access it.

The operations for accessing TLS are found in class Dll, in e32std.h:

```
static TInt SetTls(TAny* aPtr); // Set up the TLS data
static TAny* Tls(); // Retrieves the pointer stored to TLS
static void FreeTls(); // Clears the TLS data
```

We'll look at a typical implementation of Singleton using TLS shortly. But first, what's the catch? In short, it's a reduction in runtime performance. Data is retrieved from TLS about 30 times slower than direct access, because the lookup involves a switch to the kernel executive through an executive call (see Symbian OS Internals for more information).

In addition, there is only one slot for TLS per thread. If TLS is used for anything else in the thread, all the TLS data must be put into a single class and accessed as appropriate through the TLS pointer. This can be difficult to maintain (although, for application developers, there is a solution, as the later section, Singleton For Application Developers, will describe).

On Symbian OS, the implementation of Singleton must take into account the possibility of instantiation failure (for example, a leave occurring if there is insufficient memory to allocate the Singleton instance). One option is to provide two separate methods:

- A factory method, `NewL()` that instantiates the Singleton instance and stores it in the TLS slot. This may fail, so the caller must handle any possible leaves that occur.
- A separate non-leaving method called `Instance()` that can be used to access the Singleton instance once it has been instantiated, by retrieving its location from TLS.

This approach gives the caller more flexibility. Only one call to a method that can fail is necessary in order to instantiate the Singleton instance. Upon instantiation, the Singleton is guaranteed to be returned as a reference, so removing the requirement for pointer verification or leave-safe code.

```
class CSingleton : public CBase
  {
public:
  // To create the Singleton instance
  IMPORT_C static void NewL();
  // To access the Singleton instance
  IMPORT_C static CSingleton& Instance();

private: // For clarity, these methods aren't implemented below
  CSingleton();
  ~CSingleton();
  void ConstructL();
  };

EXPORT_C /*static*/ void CSingleton::NewL()
  {
  if (!Dll::Tls()) // No Singleton instance exists yet. Create one.
    {
    CSingleton* singleton = new(ELeave) CSingleton();
    CleanupStack::PushL(singleton);
    singleton->ConstructL();
    User::LeaveIfError( Dll::SetTls(static_cast<TAny*>(singleton)) );
    CleanupStack::Pop(singleton);
    }
  }

EXPORT_C /*static*/ CSingleton& CSingleton::Instance()
  {
  CSingleton* singleton = static_cast<CSingleton*>(Dll::Tls());
  ASSERT(singleton); // Panics in debug builds
  return (*singleton);
  }
```

To align more closely with the classic pattern, some implementations of Singleton may prefer to provide a single access method, `InstanceL()`, that can leave.

```
class CSingleton : public CBase
  {
public:
  // To access/create the Singleton instance
  IMPORT_C static CSingleton& InstanceL();

private: // For clarity, these methods aren't implemented below
  CSingleton();
  ~CSingleton();
```

```
  void ConstructL();
  };

EXPORT_C /*static*/ CSingleton& CSingleton::InstanceL()
  {
  CSingleton* singleton = static_cast<CSingleton*>(Dll::Tls());
  if (!singleton) // No Singleton instance exists yet. Create one.
    {
    singleton = new(ELeave) CSingleton();
    CleanupStack::PushL(singleton);
    singleton->ConstructL();
    User::LeaveIfError( Dll::SetTls(static_cast<TAny*>(singleton)) );
    CleanupStack::Pop(singleton);
    }
  return (*singleton);
  }
```

The advantage of this approach is that the implementation can be customized, for example, to perform reference counting. However, every call to `InstanceL()` must take the potential for leaves into account, which places more of a burden on the caller, and raises the potential for inefficiencies through the proliferation of `TRAP`s, or for more complex code, to make use of the cleanup stack.

Prior to Symbian OS v9, applications were DLLs, and could not use WSD. The TLS-based implementation of Singleton was found to be a reasonably straightforward alternative to the use of WSD in the classic pattern. To make it simpler still, Symbian OS provides an additional mechanism for application developers, which is discussed in the Singleton For Application Developers section below.

From Symbian OS v9, applications are EXEs instead of DLLs, and hence WSD can be used in application code without restriction.

Multi-threaded Code

Note also that the TLS implementation shown above will work only when a single thread needs to access the Singleton. As the name "thread local storage" suggests, the storage word used by TLS is local to the thread; each thread in a process has its own storage location. Where TLS is used to access a Singleton in a multi-threaded environment, the pointer that references the location of the Singleton must be passed to each thread and stored in the TLS slot. This can be done when each new thread is created, using the appropriate parameter of `RThread::Create()`. If this is not done, when the new thread calls `Dll::Tls()` to retrieve the Singleton's location, it will be returned a `NULL` pointer.

Thus the creation of the Singleton must be managed by the parent thread. It creates the Singleton before the other threads exist, and passes its location to other threads as they too are created. They must use `Dll:SetTls()` to store the location of the Singleton.

Let's look at some code to clarify how this works. Firstly, the CSingleton class exports two additional methods that are used by calling code to retrieve the Singleton instance pointer from the main thread (`SingletonPtr()`) and set it in the created thread (`InitSingleton()`).

```
class CSingleton : public CBase
  {
public:
  IMPORT_C static void NewL();
  IMPORT_C static CSingleton& Instance();
  // To pass the location of the Singleton to a new thread
  IMPORT_C static TAny* SingletonPtr();
  // To initialize access to the Singleton in a new thread
  IMPORT_C static TInt InitSingleton(TAny* aLocation);

private:
  CSingleton();
```

```
  ~CSingleton();
  void ConstructL();
  };

EXPORT_C TAny* CSingleton::SingletonPtr()
  {
  return (Dll::Tls());
  }

EXPORT_C TInt CSingleton::InitSingleton(TAny* aLocation)
  {
return (Dll::SetTls(aLocation));
  }

// Other methods are omitted for clarity
// See above for NewL() and Instance()
```

To illustrate, here's some basic code for a main thread of a process that creates a Singleton instance and then creates a secondary thread, passing in the Singleton's location

```
// Main (parent) thread creates the Singleton
CSingleton::NewL();

// Creates a secondary thread
RThread childThread;
User::LeaveIfError(childThread.Create(_L("childThread"),
  ChildThreadEntryPoint, KDefaultStackSize, KMinHeapSize, KMaxHeapSize,
CSingleton::SingletonPtr()));

CleanupClosePushL(childThread);

// Resume thread1, etc...
```

Note that the thread creation function now takes the return value of `CSingleton::SingletonPtr()` as a parameter value. The parameter value must then be passed to `CSingleton::InitSingleton()` in `childThread`'s entry point method:

```
TInt ChildThreadEntryPoint(TAny* aParam)
  {// Store Singleton's location in TLS
  if ( CSingleton::InitSingleton(aParam)==KErrNone )
    {// Succeeded, now run as normal
    ...
    }
  return (0);
  }
```

Singleton For Application Developers

Symbian OS provides class `CCoeStatic` to help application developers porting application code from other platforms where WSD was available. It was useful in the earlier days of Symbian OS (pre Symbian OS v9) when applications were DLLs and could not use WSD. Nowadays, on Symbian OS v9, it's not necessary to use it, since applications are EXEs and WSD can be used instead. However, if you do decide to use TLS, it is a simple way to do so, and allows more than one DLL per thread to use the single TLS slot.

The approach is straightforward - just derive your Singleton class from `CCoeStatic`. For example:

```
class CAppSingleton : public CCoeStatic
  {
public:
  static CAppSingleton& InstanceL();
  static CAppSingleton& InstanceL(CCoeEnv* aCoeEnv);

private:
  CAppSingleton();
  ~CAppSingleton();
  };
```

The implementation of the class must associate itself with a UID to allow the Singleton instance to be "registered" by the application framework (class `CCoeEnv`). When the Singleton is instantiated, the `CCoeStatic` base class constructor adds the object to the list of Singletons stored by `CCoeEnv`. Internally, `CCoeEnv` uses TLS to store a pointer to each registering Singleton (using a doubly linked list of pointers to `CCoeStatic`-derived objects). Thus, for class `CAppSingleton`:

```
const TUid KUidMySingleton = {0x10204232};

// "Register" the singleton
CAppSingleton::CAppSingleton() : CCoeStatic(KUidMySingleton, CCoeStatic::EThread)
  {}

// Uses CCoeEnv::Static() to access the Singleton
CAppSingleton& CAppSingleton::InstanceL()
  {
  CAppSingleton* singleton =
static_cast<CAppSingleton*>(CCoeEnv::Static(KUidMySingleton));

  if (!singleton)
    {// Two-phase construction omitted
    singleton = new(ELeave) CAppSingleton();
    }
  return (*singleton);
  }

// Uses CCoeStatic::FindStatic() to access the Singleton
CAppSingleton& CAppSingleton::InstanceL(CCoeEnv* aCoeEnv)
  {
  CAppSingleton* singleton = static_cast<CAppSingleton*>
(aCoeEnv->FindStatic(KUidMySingleton));

  if (!singleton)
    {// Two-phase construction omitted
    singleton = new(ELeave) CAppSingleton();
    }
  return (*singleton);
  }
```

Once the `CAppSingleton` class has been instantiated, it can be accessed through its leaving `InstanceL()` methods, but can also be accessed directly by calling `CCoeEnv::Static()` or `CCoeEnv::FindStatic()`. Note that the former is a static method, so can be used in an application class where no `CCoeEnv` pointer is available.

```
// From coemain.h
```

```
static CCoeStatic* Static(TUid aUid);
CCoeStatic* FindStatic(TUid aUid);
```

These methods iterate through the doubly linked list, attempting to match each CCoeStatic-derived object with the appropriate UID. The use of `CCoeStatic` can only be used by code running inside the application framework i.e. within the control environment (CONE).

Singleton Cleanup

All the implementations discussed in this article have specified the destructor for the Singleton class as private, and returned the Singleton by reference rather than pointer. This is because, if the destructor was public, and a pointer to the Singleton instance returned, it could be inadvertently deleted by any caller, leaving the `Instance()` method handing out "dangling references" to the deleted instance. The implementations shown prevent this, and give the Singleton class responsibility of creation, ownership and, ultimately, cleanup of the Singleton instance.

A common approach to cleanup is to use the `atexit` function provided by the standard C library, to register a cleanup function to be called explicitly when the process terminates. The cleanup function can be a member of the Singleton class, and simply delete the Singleton instance. Please see [2] for further details.

However, you may want to destroy the Singleton before the process terminates (for example, to free up memory if the object is no longer needed). In this case, you will have to consider reference counting, as discussed in [3] to avoid dangling references arising from premature deletion.

Further Information

This discussion has been taken in part from the Singleton pattern description in the book Common Design Patterns for Symbian OS.

References

1. ↑ Symbian Foundation Knowledgebase Q&As#How can I find the "uninitialised data" in my DLL?
2. ↑ Modern C++ Design: Generic Programming and Design Patterns Applied, Andrei Alexandrescu, Addison-Wesley Professional 2001.
3. ↑ Modern C++ Design: Generic Programming and Design Patterns Applied, Andrei Alexandrescu, Addison-Wesley Professional 2001.

Contributors

I'd like to thank Adrian Issott, Mark Jacobs, Antti Juustila, hamishwillee and Tim Williams, who all provided feedback on aspects of this article.