Symbian OS C++ coding standards

Symbian DevNet
Jan 2003

These are the coding standards used by Symbian's own system developers. Some of the recommendations, such as those relating to the inclusion of copyright notices in source and header files, are only relevant to Symbian OS system developers, but most are relevant to all Symbian OS developers.

1. Naming conventions

1.1. General

- always use meaningful and carefully considered names
  in general, first letter of words capitalized (exceptions noted explicitly)
- all words adjoined
- avoid the use of '_' <underscore> except in macros and resource IDs.

```
void TObject::PrepareForCommit();
class CGlobalText;  // see note on class names and types
TInt elementOffset;  // automatic variables begin in lowercase
```

(for further details and references see Ambler, 2000)

- use full English descriptors that accurately describe the variable/field/class/. For example, use names like firstName, grandTotal, or CorporateCustomer. Although names like x1, y1, or fn are easy to type because they're short, they do not provide any indication of what they represent and result in code that is difficult to understand, maintain, and enhance (Nagler, 1995; Ambler, 1998a).
- use terminology applicable to the domain. If your users refer to their clients as customers, then use the term Customer for the class, not Client. Many developers will make the mistake of creating generic terms for concepts when perfectly good terms already exist in the industry/domain
- use mixed case to make names readable. You should use lower case letters in general, but capitalize the first letter of class names and interface names, as well as the first letter of any non-initial word (Kanerva, 1997)
- use abbreviations sparingly, but if you do so then use them intelligently. This means you should maintain a list of standard short forms (abbreviations), you should choose them wisely, and you should use them consistently. For example, if you want to use a short form for the word number, then choose one of nbr, no, or num, document the one you choose (it doesn't really matter which one), and use only that one
- avoid long names (less than 15 characters is a good idea). Although the class name PhysicalOrVirtualProductOrService might seem to be a good class name at the time (an extreme example) this name is simply too long and you should consider renaming it to something shorter, perhaps something like Offering (NPS, 1996)
- avoid names that are similar or differ only in case. For example, the variable names persistentObject and persistentObjects should not be used together, nor should anSqlDatabase and anSQLDatabase (NPS, 1996).
- capitalize the first letter of standard acronyms. Names will often contain standard abbreviations, such as SQL for Standard Query Language. Names such as sqlDatabase for an attribute, or SqlDatabase for a class, are easier to read than sQLDatabase and SQLDatabase.

1.2. Automatic variables

- first letter lowercase
- do not declare automatics until required; (avoid the K&R C-style of declaring all the automatics used in a routine at the top of that routine)
- always try to initialize variable when they are declared rather than assigning values to them
- never initialize or instantiate multiple items on the same line

```
TInt first=0;
TInt second=0;
```

Explanation: having the first letter of automatic variable lowercase is simply a convention to help reviewers and maintainers to follow your code

- the main reason for not declaring automatics until required is because they may not be required; declaring them at the start of a routine is therefore inefficient in terms of execution speed and use of stack space
- for lengthy routines, declaring automatics as close as possible to the point of use can also help people to understand code fragments without having to refer back to the top
- initializing variables when they are declared is more efficient (smaller and faster code) and avoids the risk of using the variable in an uninitialized state
- declaring multiple items on the same line can lead to errors such as:

```
   TText* data1, data2;
TInt int1, int2 = 0;
```

when the programmer meant:

```
TText* data1;
TText* data2;
TInt int1=0;
TInt int2=0;
```

1.3. Global variables

- use of global variables is discouraged
- start names with a capital letter. In cases where confusion might be caused (for instance, if the capital letter in question is an H or a T or a C, and there's no obvious alternative name) you may use a small g prefix, e.g. `gWhatever`
- non-const global data is not supported in DLLs; better to use thread local storage (TLS).

1.4. Macros

- all capitalized
- an underscore separates words

```
IMPORT_C
__TEST_INVARIANT
__ASSERT_ALWAYS
```

1.5. Pointer and reference types

- specifier placed next to type; not next to name

```
TText* data;
void TDemo::Append(const TDesC& aData);
```

```
TEntry* TDemo::Entry() const;
```

1.6. Class names

- 'C', 'R', 'T', 'M' class types only (the first letter of the class name). The class naming distinction helps reinforce valuable programming idioms
- structs are 'T'; *structs with attitude*
- static classes have no prefix letter
- exceptionally, driver classes known by the kernel are 'D' classes. See Symbian Developer Library for more information.

```
class CBase;
class TTypefaceInfo;
class RFont;
class MLaydoc;
class User;  // static class
class EikUtils;  // static class
class Time;  // static class
```

1.7. Function names

- general rules apply
- *Setters* are typically `SetThing()`.
- *Getters* are typically `Thing()` if the function returns the item.

```
void TObject::PrepareForCommit()
 {DoPrepareForCommit();}
void SetOffset(TInt aOffset);
TInt Offset() const;
TInt offset=Offset()
```

- 'Get' used for functions that set values into reference arguments.

```
TCharFormat format;
GetCharFormat(format);
```

- Trailing 'L' means function may *leave*; lack of trailing ‘L’ means it does not *leave*
- Trailing 'C' means function places an item on the cleanup stack
- Trailing 'D' means the object in question will be destroyed

```
CStoreMap* map=CStoreMap::NewLC();  // map returned on cleanup stack
```

1.8. Member data

- preceding 'i' lowercase; for *instance* data.

```
class TObject
 {
 TType iType;
 TInt iElementOffset;
 TPtrC iComponentValue;
 };
```

1.9. Function arguments

- preceding 'a' lowercase; for *argument* data
- do not use 'an' before a vowel

```
void TObject::TObject(TType aType,TInt aElementOffset)
 {
 iType=aType;
 iElementOffset=aElementOffset;
 }
```

1.10. Constants

- preceding 'K', uppercase.

const TInt KMaxNameLength=0x20;

```
const TUid KEditableTextUid={268435548}
```

1.11. Enumerations

- should be scoped within the relevant class
- do not pollute the global name space
- must have a meaningful, unambiguous name
- enumerations are types; therefore 'T' classes
- uppercase 'E' for enum members
- class-specific constants can be implemented as enums and in that case are ‘K’

```
class TDemo
 {
public:
 enum TShape {EShapeRound, EShapeSquare};
 enum TFruit
  {// Fruit definitions
  EFruitOrange,
  EFruitBanana,
  EFruitApple
  }
 }
TDemo::TShape shape=TDemo::EShapeSquare;
```

2. Header files

2.1. General

- never leave commented out code in production code, especially in published headers
- do not include more headers than are needed; use forward references
- in the project source…
  public headers kept in the `..inc` directory
  private headers kept in the relevant source directory.

- use standard anti-nesting mechanism to avoid multiple inclusion of headers
- use standard headers, giving standard information; be consistent; for example:

```
// EXAMPLE.H
// Copyright (c) 2003 Symbian Ltd. All rights reserved.
//
// Example module header
//
```

Copyright messages
The example above shows the type of copyright message that should appear at the top of source files. Where a file has been part of the source for some time, the year in the header can be given as a range in the form:

```
// Copyright (c) 1997-2003 Symbian Ltd.  All rights reserved.
```

The start year should be the year that the component was started. However, this is not vital - the important bit is asserting copyright.

Components that include source files which are partially copyrighted by third parties should ensure that a Symbian copyright message is added to any files modified. Typically, such files have a large block of comments at their head detailing the copyrights. The addition of the following line to that block is sufficient to assert copyright to the code added by Symbian:

```
// Portions Copyright (c) 1997-2003 Symbian Ltd.  All rights reserved.
```

2.2. *Class design*

2.2.1. Declaring functions: parameters and return values

- use references in preference to pointers - if none of the criteria for using pointers apply; (see below)
- pay attention to const; use const for parameters and return values if the data is not to be modified
- encourage generality; use the least derived class possible

```
void Foo(const CArrayFix<X>& aX);
```
rather than
```
void Foo(const CArrayFixFlat<X>& aX);
```

2.2.2. When to use which form of parameter/return

- by **const value** `const X`

- don't do this; it only restricts the code in the implementation, but has no impact on clients.

- by **value** `X`

- for return values or arguments that are concrete data types or are small (< 8 bytes)
- for return values where the object has to be built (i.e., cannot return a reference to one we have already); Note for large classes it is preferable to use a reference argument to get a value, as this reduces stack usage.

- by **const reference** `const X&`

- do not do this for concrete (built-in) types, especially enumerations, as the GCC compiler will fault (current as of July 1997).
- use for larger arguments (> 8 bytes)
- if in doubt pass all class arguments using this form.

- by **reference** `X&`

- to allow a function to return values through these arguments, or modify the contents of them.

- for return values, where the object exists and can be modified by the caller; Note, the lifetime of the object being returned must extend beyond the scope of the function.

- By **const pointer** `const X*`

- Const for items which can be used (but not modified) by the recipient; e.g. a buffer
- if a NULL (non-existent) object is meaningful (so const X& cannot be used); i.e. the argument/return value is optional; if this is the case consider using a function overload instead - one taking a reference and one taking no arguments - if it makes the API clearer.

- By **pointer** `X*`

- for a modifiable (non-const) object that is optional; again, consider an overload if it makes the interface more intuitive.
- to imply transfer of ownership, rather than just available for use (owns vs. uses); the recipient of the object is now responsible for deleting the object.

2.2.3. What to export

- when writing a Symbian OS library, attention must be given to which functions are defined as EXPORT_C
- do **not** export (const) data; rather, export a function returning a reference/pointer to the data.
- only export private functions if…

- they are accessed via public inline members.
- They are virtual and the class is intended for derivation outside the library.

- For public or protected members…

- do **not** export functions that are not designed for use outside the library
- do **not** export pure virtual functions
- do **not** export inline functions.

2.3. Class Layout

- always write access control specifiers; removes all ambiguity
- list the public methods first, with special member functions at the top
- list public member data before private member data (otherwise changing the private data would break binary compatability)
- use explicit access specifiers at the start of each class section…
  - friend classes
  - public methods
  - protected methods
  - private methods
  - public data
  - protected data
  - private data

- heed binary compatibility issues with respect to promoting functions' access control specifiers; (if you change the access for something after an API freeze, you may have to leave it in exactly the same place as before).
- use white space to group related functions and improve readability
- function arguments are named in the function declaration (as well as the definition). Improves readability
- obey the indentation conventions.

2.3.1. Virtual functions

- virtual functions that replace inherited behaviour should be grouped together (inside the groupings defined above) in the header, with a comment documenting which class the behaviour is inherited from

- it is not necessary to specify the `virtual` keyword for these functions
- do not make virtual functions inline; it is difficult to know exactly how a compiler treats these, and can lead to code bloat. Exception; a virtual inline destructor is okay.

2.3.2. Inline functions

- specify `inline` explicitly for inline functions.
- do not leave it to the compiler.
- do not provide the inline implementation in the class definition; this confuses and pollutes the header, making it less readable.
- provide inline implementations at the bottom of the header or in a separate .inl file.

2.3.3. Header layout

Note that this header is an illustration only; it will not compile as there is no definition of `MEmptyable` available

```
// DEMHEADR.H
// Copyright (C) Symbian LTD, 1998
//
// Example header layout
//
#ifndef __DEMHEADR_H__
#define __DEMHEADR_H__
#include <e32std.h>
#include <e32base.h>
//
// Illustrates coding conventions for header layout
class CContainer;  // forward reference avoids header inclusion
class CShape : public CBase, public MEmptyable
// CShape class representation of a shape to draw on the screen
// also implements MEmptyable so that CWhangDoodle can control it.
 {
public:
 enum TShape
  {
  EShapeCircle,
  EShapeSquare,
  EShapeTriangle
  }
public:
 IMPORT_C static CShape* NewL(TShape aValue);
 IMPORT_C static CShape* NewLC(TShape aValue);
 IMPORT_C virtual ~CExample();
 inline CContainer* Container() const;
 inline void SetContainer(CContainer* aContainer);
 //
 // MEmptyable implementation
 IMPORT_C void Empty();  // virtual specifier unnecessary
private:
 CShape(TInt aShape);
 void ConstructL();
 CContainer* iContainer;
private:
 TInt iValue;
 };

#include <demheadr.inl>
```

```
#endif  // __DEMHEADR_H__
```

Explanation: Symbian's convention for generating unique names for the #define guards in header file is simply to replace the dot in the filename with an underscore and use double underscores as prefixes and suffixes on the resulting variable name.

Note: The `#ifndef XXX` form of the guard against multiple inclusion is significantly quicker than the apparently equivalent `#if !defined(XXX)` form. The latter is optimized by the compiler in such a way that the inclusion of the other header files `e32std.h` and `e32base.h` don't have to be guarded themselves.

3. Source modules

3.1. Writing good code

- build code that gives 0 compiler warnings and 0 link errors; it is dangerous to ignore compiler warnings, and these are highly likely to generate errors on other compilers. Complacency regarding compiler warnings leads to important new compiler warnings not being noticed among a pile of familiar warnings
- be consistent in the use of programming conventions
- strive to write tight, efficient code
- minimize on stack usage
- minimize on the number of calls to the allocator
- always ask... *what would happen if this code were to leave?*
- avoid hard-coding magic numbers; use constants or enumerations.

3.2. Basic types

`e32def.h` defines concrete types; use them.

```
TInt32 integer;
```

... and not...

```
long int integer;
```

3.3. String and buffer classes

Use the Symbian OS descriptor classes, such as `TPtr`, `TDesC` and TBuf<n>; instead of 'native' text string classes or hand-crafted buffer classes. This results in code that is much more easily converted between wide-character and narrow-character builds, and also has gains in efficiency and robustness over other string classes.

3.4. Date classes

Use the Symbian OS date and time classes, such as `TDateTime`, instead of hand-crafted alternatives.

3.5. Indentation

- indentation is performed using 4 space tabs, not spaces
- opening braces are indented to align with the code that follows; they are not aligned with the preceding code; i.e. braces form part of the indented code.

```
void CItem::Function()
```

```
 {
 Foo();
 }
```

...and not...

```
void CItem::Function()
{
 Foo();
}
```

3.6. Class types

- it is strongly recommended that you use direct initialization for class types
- the use of copy initialization is deprecated.

Explanation: If you declare a class type using the copy constructor = like this:

```
TParsePtrC parser=TPtrC(FileName);
```

…then the initialization of `TParsePtrC` effectively requires a call to `TParsePtrC::TParsePtrC(const TPtrC&)` followed by a call to `TParsePtrC`'s copy constructor `TParsePtrC::TParsePtrC(const TParsePtrC&)`.

This is significantly more expensive than directly initalizing the class type, which is done like this:
`TParsePtrC parser((TPtrC(FileName)));`

and eliminates the redundant call of the copy constructor.

Note that you need the extra set of brackets round `(TPtrC(FileName))` in order to allow the compiler to recognize that you are instantiating an object of type `TParsePtrC` and not declaring a function that returns something of type `TParsePtrC`. The brackets serve to disambiguate the code.

4. Recurring code patterns

4.1. Two-phase construction

- complex objects (typically those deriving from CBase) require two-phase construction:
  - trivial constructor
  - non-trivial construction - allocation of resources
- object only fully initialized when both phases have completed
- static factory functions can wrap both phases into a single call.

4.1.1. Trivial constructor

- default C++ constructor or custom constructor - cannot leave
- use the C++ member initializer list (applies to simple 'T' classes also)

```
CComplexObject::CComplexObject(TInt aValue,TInt aLength))
// c'tor
//
:iValue(aValue),iLength(aLength)
 {}
```

4.1.2. Non-trivial construction

- implemented by `ConstructL()` members (and overloads) - may leave

```
void CComplexObject::ConstructL()
// This will complete initialization of the object
//
 {
 iResource=new(ELeave) CResource;
 }
```

4.1.3. Static factory construction

CComplexObject* CComplexObject::NewL(TInt aLength,TInt aValue)

```
//
// return a handle to a new fully initialised instance of this class
//
 {
 CComplexObject* self=new(ELeave) CComplexObject(aLength,aValue);  // calls c'tor
 CleanupStack::PushL(self);
 self->ConstructL();
 CleanupStack::Pop(self);
 return self;
 }
```

4.2. Protect your objects

Catch programming and run-time errors early by using pre- and post-conditions in functions i.e., assert that those conditions required for correct execution hold true. Two mechanisms support this programming style:

__ASSERT_ALWAYS / __ASSERT_DEBUG class invariants

Both these mechanisms must be used. They catch programming errors early and aid in communicating the design and purpose of the class.

4.2.1. Assertions

__ASSERT_ALWAYS to catch run-time invalid input
__ASSERT_DEBUG to catch programming errors

4.2.2. Class invariants

Define class invariants for non-trivial classes using…

- __DECLARE_TEST
  Specifies the allowed *stable states* for that class.

Call the invariant at the start of all public methods (where feasible) using…

- __TEST_INVARIANT
  Ensures the object is in a stable state prior to executing the function.
  Calls are compiled out in *release* software builds.

For non-const methods call the invariant at the end of the method. This ensures the object has been left in a stable state after executing the method:

```
void CComplexTextObject::Delete(TInt aPos,TInt aLength)
//
```

```
// Removes text content, commencing at position aPos, over aLength number of
characters
//
 {
 __TEST_INVARIANT;

 __ASSERT_ALWAYS(aPos>0,Panic(EPosOutsideTextObject));
 __ASSERT_ALWAYS(aLength>=0,Panic(EDeleteNegativeLength);

 iTextBuffer->Delete(aPos,aLength);

 __TEST_INVARIANT;
 }
```

4.3. Cleanup stack

Use the cleanup stack properly…

- where appropriate use it in preference to TRAP harnesses as it is quicker and more efficient
- use the checking methods of CleanupStack::Pop to check that the PushL and Pop are balanced if possible
- use CleanupStack::PopAndDestroy if you are finished using the object

Standard usage
For `CBase`'d heap-created objects, push and pop the pointer to the object, e.g:

```
CObject* object = CObject::NewL();
CleanupStack::PushL(object);
object->LeavingFunctionL();
CleanupStack::PopAndDestroy(object);
```

RClasses
For RClass objects that define a Close method use CleanupClosePushL where applicable

```
RObject object;
object.Open();
CleanupClosePushL(object);
object.LeavingFunctionL();
CleanupStack::PopAndDestroy(&object);
```

This may not be suitable for all RClass objects. For instance it may not be suitable for RPointerArray if you have already added objects to the array. Calling Close in that instance could orphan the added objects unless they are also on the CleanupStack.

If you wanted to call non-leaving functions after LeavingFunctionL, then it is preferable to leave the object on the CleanupStack rather than calling CleanupStack::Pop and then the non-leaving function as it results in less code.

TCleanupItems
Some classes or methods will require more complicated cleanup. Use a TCleanupItem and provide a function to call that does the required cleanup. Declare these functions as either local functions or static class functions:

```
void RObject::ReleaseOnCleanup(TAny* aObject)
// Allows correct cleanup of specified objects by
// using the cleanup stack instead of a trap harness.
 {
 reinterpret_cast(aObject)->Release();
 }
```

```
RObject* object=GetObject();// may increase share on a reference counted object
CleanupStack::PushL(TCleanupItem(ReleaseOnCleanup,object));
iContainer->AddL(object);
CleanupStack::Pop(object); // Pop the TCleanupItem::iPtr
```

Heap allocated arrays
Use CleanupArrayDeletePushL() when pushing an array of objects onto the cleanup stack. This ensures that PopAndDestroy() will call delete[] to delete the array correctly

```
TObject objectArray = new(ELeave) TObject[numberOfElements];
CleanupArrayDeletePushL(objectArray);
...
CleanupStack::Pop(objectArray);
```

4.4. Private inheritance

Avoid private inheritance: use composition instead. Inheritance should be restricted to cases when the relationship really *is a kind of*.

4.5. Multiple inheritance

In general…

- M.I. is fine; only 'M' classes (mixins) are inherited
- Mixins specify interface only, not implementation
- inherit from CBase-derived class first; achieve correct layout of the v-table
- inherit from only one CBase-derived class; other super-classes must be mixins
- refer to the Symbian Developer Library for further information

```
class CGlobalText : public CPlainText, public MLayDoc, public MFormatText
 {
 ...
 };
```

5. Casting

- casts may indicate questionable code, always use with caution
- use the correct casting operator as this improves readability and removes ambiguity
- always use the C++ casts
- avoid using C-style casts.

Note: be aware that, for historical reasons, Symbian OS provides the following macros to encapsulate the C++ cast operators.

```
REINTERPRET_CAST
STATIC_CAST
CONST_CAST
MUTABLE_CAST
```

These are historical and should not be used in any new code. These used to be needed because some older compilers did not fully support all the C++ casts. Always use the standard C++ casts in any new code e.g. static_cast<>()

6. Test code

An essential part of all software development…

- must be kept up to date.
- as a general rule, if you have no test code to test a specific code area, that code WILL be bugged.
- testing is not 100% proof of bug-free code.

7. Code reviews

Every software author is responsible for having their own code reviewed.

8. Code examples

Complex conditionals

```
if (conditionA && conditionB)
 {
 DoSomethingA();  //  even a single line requires braces
 }
else
 {
 DoSomethingB();
 DoSomethingC();
 }
```

1) brackets round single lines are essential as anyone who's tracked down a missing bracket bug knows. It's even more necessary if you want to do automatic transforms on your source code (say adding some logging prints). *Much* easier if you don't have to explicitly deal with single-line ifs. Remember, it's not your source, it's the next reader's source. And they will want to add ‘just one more line’.

2) source differencing and merging is easier - the source differences will just contain the added lines of code (the essential complexity) and not any added braces (accidental complexity).

Case statements

```
switch(type)
 {
case(KValueA):
case(KValueB):
 SomeActionA();
 SomeActionB();
 break;
case(KValueC):
 SomeActionC();
 break;
default:
 SomeActionD();
 }
```

Conditionals

Testing for NULL handles

```
// aHandle!=NULL is not required
```

```
if (aHandle)
 ActionA();
else
 ActionB();
```

Testing for zero values

```
// if (aLength) --> this is unconventional
if (aLength>0)
 ActionA();
else
 ActionB();
```

Nested ifs

```
if (aScore>80)
 degree=1;
else if (aScore>70)
 degree=2;
else
 degree=3;
```

9. References

1. Ambler, S.W. (1998a*). Building Object Applications That Work: Your Step-By-Step Handbook for Developing Robust Systems with Object Technology*. New York: Cambridge University Press.
2. Ambler, S.W. (2000*). Writing Robust Java code* http://www.ambysoft.com/javaCodingStandards.pdf
3. Kanerva, J. (1997). *The Java FAQ*. Reading, MA: Addison Wesley Longman Inc.
4. Nagler, J. (1995). *Coding Style and Good Computing Practices*. http://wizard.ucr.edu/~nagler/coding_style.html
5. NPS (1996). *Java Style Guide*. United States Naval Postgraduate School. http://dubhe.cc.nps.navy.mil/~java/course/styleguide.html