Fundamentals of Symbian C++/Compatibility

- Fundamentals of Symbian C++ Table of Contents

The Symbian Foundation ships a software developer kit (SDK) that enables both phone manufacturers and third-party developers to extend the platform. Accordingly, developers must be able to rely on the stability of the interfaces exposed in the SDK, and the foundation has a commitment to maintain a certain degree of compatibility between incremental releases of its software. The foundation also needs to be able to update the software components in order to maintain and improve them.

This article outlines the sorts of compatibility the Symbian Foundation defines, the rules for preserving it across releases and some techniques that help to ensure code is reasonably future-proof.

Forward and Backward Compatibility

A change in a software component is regarded as **backward compatible** if code that used the old version will still function with the new version.

The most obvious example of this is a library used by an application: a change to the library is backward compatible if the application will still work correctly when the old library is replaced with the new one:

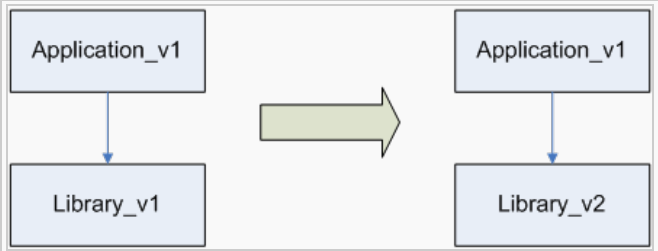

Illustration of backward compatibility

A change is **forward compatible** if code that is written to work with the new version will still function correctly with the old version.

Backward compatibility tends to be the primary goal when making incremental releases of a component, with forward compatibility a desirable extra. Forward compatibility is most often a concern when extending data formats, rather than when revising a C++ interface.

Source and Binary Compatibility

A change is **binary compatible** if the component using the changed interface will still run correctly without needing to be recompiled or re-linked. Binary compatibility is important when an interface is public and the developer cannot know who will use it, so it is not possible to ensure that all users are able to recompile. Symbian does not know who will use the interfaces it publishes to third parties, so tries to maintain binary compatibility.

A change is **source compatible** if the component using the changed interface need only recompile, and does not need to make any code changes. If an interface has a limited, known set of users who can easily rebuild their code, it may be appropriate to only maintain source compatibility.

It is normally harder to maintain binary compatibility than source compatibility. Many changes are source compatible but not binary compatible. For example, if you change the size of a class, any code that instantiates the class will need to be recompiled, because the code at the point of instantiation needs to know the class size.

On the other hand, most changes that are not source compatible are also not binary compatible, such as removing public methods or changing the number of parameters to a function. A few changes are binary compatible but source incompatible, such as changing the name of an exported function. Because Symbian OS uses link-by-ordinal rather than link-by-name methods, the client component will continue to work as long as it is not recompiled and when it is recompiled the source will need to be updated.

Preserving Source Compatibility

This section outlines how you can ensure that your code remains source compatible.

Adding Interface Elements

You can add new interface elements: classes, globals, statics, class member functions, class data members.

Removing Interface Elements

You cannot remove any published elements of the interface: classes, globals, statics, public member functions, public data. If users of your interface can derive from a class (the class supports external derivation), you cannot remove protected methods or data either.

Changing Elements

As a general rule, you can make changes that increase access, but not changes that reduce access. You can:

- Relax but not tighten access specifiers (private methods can be made protected)
- Grant but not withdraw friendship

- Widen but not narrow function inputs:
- A non-const input parameter can be made const.
- A class-type input parameter can be replaced with a base class.
- Narrow but not widen function outputs:
- A const return value can be made non-const.
- A class type return value can be replaced with a sub-class.

Preserving Binary Compatibility

This section outlines how you can ensure that your code maintains binary compatibility.

Library-Level Compatibility

Each function exported from a DLL is associated with an ordinal number, which is used by the linker to identify the function. The function ordinals are stored in the module definition (`.def`) file.

If the `.def` file list is re-ordered, say by adding a new export within the list, the ordinal number values will change and previously compiled code will be unable to locate the correct function. For example, adding a new function at the start of the list will shunt all the ordinals up one. Thus any component using ordinal 4 would be now be looking at what was previously ordinal 3. This breaks binary compatibility.

When adding a new export use, use `abld freeze` to update the `.def` file, which will ensure that the new export is appended to the file.

NONSHARABLE_CLASS

EABI `.def` files include additional exports beginning with `_ZTI` and `_ZTV` and containing the name of a class:

```
_ZTI17CMyClass @ 25 NONAME; #<TI>#
_ZTV17CMyClass @ 26 NONAME; #<VT>#
```

These are pointers to the RTTI and virtual function table for the class, and are needed to compile classes outside the library that derive from the class in question.

They can present a problem for maintaining binary compatibility because the compiler will generate them for internal classes as well as published classes. If the internal class is later removed, the export is also removed, which changes the ordinal number associated with subsequent exports, and breaks binary compatibility.

If you have to remove these entries, you can use the ABSENT keyword in the .def file to consume that ordinal slot and preserve the ordinal values for subsequent exports:

```
_ZTI17CMyClass @ 25 NONAME ABSENT; #<TI>#
_ZTV17CMyClass @ 26 NONAME ABSENT; #<VT>#
```

It is better, though, to avoid the generation of these exports for internal classes in the first place and, to effect this, classes that are not intended for derivation outside the DLL should be declared using the `NONSHARABLE_CLASS` macro:

```
NONSHARABLE_CLASS(CMyClass) : public CBase
  {
  ...
  };
```

This suppresses the generation of RTTI and virtual function table pointers for the relevant class.

Class-Level Compatibility

Data members

The amount of freedom you have to change data members while preserving binary compatibility depends on how well hidden the data members are.

If the class has a private non-exported constructor, and is created using a static factory function such as `NewL()`, then you can change the size of the class, adding, removing and moving public data members.

You cannot move any public data members, and this of course means that you cannot remove any private data members that precede the public members without supplying a dummy value to keep the public data at the same offset.

Any private data accessed through public inline functions is effectively public from the point of view of binary compatibility. So given a class declaration such as this:

```
CMyClass : public CBase
    {
public:
    static CMyClass* NewL();
    inline TSomePublishedData PublicData() { return iEffectivelyPublicData; }
    ...
private:
    CMyClass();
    TSomePrivateData iPrivateData;
```

```
    TSomePublishedData iEffectivelyPublicData;
    };
```

the data member `iEffectivelyPublicData` becomes public from the point of view of compatibility, This means that not only can it not be removed or changed, but it cannot be moved. This means that increasing the size of `TSomePrivateData` is now also a BC break, because it moves the data members that follow it.

If it is possible to allocate the class from outside the code being changed, you cannot change the size of the class without breaking binary compatibility. So, for example, the following types of classes cannot change size:

- Classes supporting external derivation
- C classes with public C++ constructors
- T classes.

Also, if you allow external derivation, protected data is effectively public from the point of view of compatibility.

Virtual functions

The amount of freedom you have to change data members while preserving binary compatibility depends primarily on whether you permit external derivation.

If you do not allow derivation from outside the code being changed, you should hide construction of the virtual function table for your class by giving it a private non-exported constructor and allowing the class to be created only through a static factory function such as `NewL`. If you do this, you can:

- Add new virtual functions,
- Remove or re-order any virtual functions if you can be sure they cannot be called by users of your interface.

If you want to allow external derivation, you should declare and export a constructor. You will not be able to add, remove, re-order or override any previously inherited virtual functions without breaking binary compatibility.

Future-Proofing Techniques

Make Class Data and Methods Private Where Possible

For compatibility purposes, `protected` methods may as well be `public`. Make class methods `private` if possible. Avoid `public` data or `protected` data that is accessible to users of the interface.

Avoid Inline Functions

Avoid inline functions in almost all circumstances, and never use inline functions to access data members.

Avoid Default Parameters

Default parameters are effectively inline data, so use function overloading instead.

For example, consider a function declaration such as this:

```
IMPORT_C void MyFunction(TInt aDefaultedParameter = 0);
```

Code built against this that does not supply a value will have the default value zero built into the client. If the default is changed, the client will still supply zero to the function – it will not be updated to supply the new default, and the default behavior will not change.

If the code uses function overloading, then the default behavior can be changed:

```
IMPORT_C void MyFunction();
IMPORT_C void MyFunction(TInt aCustomisedBehaviour);
//in the cpp
const TInt KTheDefault = 0;
EXPORT_C void CMyClass:MyFunction()
    {
    MyFunction(KTheDefault);
    }
```

Now code that is not interested in customized behavior calls the overload with no parameters, and this implements the default behavior – whatever we have defined that to be.

External Derivation

Allowing client code outside of your control to derive from your class greatly limits the changes you can make. Do not permit it by default, or just because you think it might be useful.

If the class derives from `CBase` and is *not* intended to be externally derivable, it is good practice to make its C++ constructor `private` and wrap it inside a static `NewL` method, because this enables you to change its size.

If your design does support external derivation, there are some techniques that can help to future-proof the design:

Override all virtual functions

Override all virtual functions whether you need to or not. If you do not need to, just call the base class implementation directly. This ensures that your class will have virtual function table entries for all the virtual functions and enables you to extend the overridden functions later on.

Provide room for more data

Give the class at least 4 bytes of spare member data and ensure that object creation is allowed to leave: then you can use this to extend the class data without increasing its size.

Provide room for more virtual functions

In the past, classes have included spare virtual functions, to reserve slots in the virtual function table. Symbian no longer recommends this approach because it is very difficult to guarantee that it will work. Instead, use the `Extension_()` mechanism implemented in `CBase`:

```
class CBase
    {
protected:
    IMPORT_C virtual TInt Extension_(TUint aExtensionId, TAny*& a0, TAny* a1);
    };
```

To add an extension, allocate a new UID and use it to select the new function or return an extension interface, and wrap the call with an exported or inline function so clients can call something more meaningful.

For the extension mechanism to work, all classes in the inheritance hierarchy must override it, and they must all call to their base if they do not understand the UID supplied.

Data Compatibility

Data compatibility refers to the ability of one version of a software component to understand data created by a different version of it.

For example, suppose a contacts application creates and stores contact data using a specific format and the new version of the software extends the format to include some extra fields. It is obviously desirable for the new version to understand the old format, so the user can import an address book created using the old software: this is a requirement for backward data compatibility.

Similarly, it is desirable for data created by the new application to be usable by the old application, for example by discarding the additional fields. This would allow a user who has created some contacts using the new application, backed up their data and then lost their phone to go back to an old phone and restore the data to it. This is a requirement for forward data compatibility.

The most common technique to help maintain data compatibility is versioning: tagging data elements with the version of the data format they belong to. Then a component only parses the elements it is interested in and can ignore the rest.

Behavioral Compatibility

A change is behaviorally compatible if the documented behavior of the old version of the interface is not changed. Suppose a function returns an error code, and the documentation explains that the code may be `KErrNone` or `KErrNoMemory`. If the function is extended so that it may also return `KErrNotFound`, this is technically a behavioural compatibility break.