Syntax Proposal for Generation Versionable Classes

Martin J. O'Riordan
Project Lead for C++

Written : 2-February-1990
Revised : 22-February-1990
Revised : 6-July-1990

Introduction

A versionable class is a class for which there is an immutable and a mutable component. The immutable component may never be altered, whereas the mutable component may be altered with each revision of an implementation. What makes a versionable class unique, is its ability to allow clients to see and use only the immutable component, independently of the mutable component. A C++ 'class' allows the client to see the whole definition, and all parts must be present in order to compile.

I wish to describe our system for separate compilation of both parts as being the separation of 'interface' and 'implementation'. A further revision of this idea, is that of "Generation Versioning", where the interface may go through several generations during the lifetime of a type. The simple interface/implementation model does not address this, but interface generations does.

A generation versionable class may be described by three components. The initial interface, or generation zero interface definition. A sequence of extensions to this interface which describe the generations of the type. And finally an implementation component.

Only the initial interface and the implementation component are compulsory. All modules must see the interface component, but only the implementor of the type need see the implementation component. All 'clients' of the type will generate "Type Initialisation" code which allows the hidden implementation component to be bound to the complete type. The 'implementor' of the type will generate the code necessary to describe these component bindings required by the clients.

Language Extensions

Extensions to the initial interface may be described between the initial interface and the implementation. Since the sequence of extensions is critical to the correct behaviour of the type, sequencing information is necessary. Sequencing information ensures that the compiler can detect inconsistant versioning of the interface. It also permits the client to specify interface to a specific version, thus avoiding name conflicts presented by subsequent versions of the versionable class.

I propose the addition of three new keywords to the language to describe this metaphor. These are as follows :-

```
interface
implementor
revision
```

The syntax would be modified to reflect the use of these new words thus :-

```
version-head:
        interface
        implementor
        revision [ expression ]

class-key:
        version-head          // 'class' implied
        version-head class
        version-head struct
        class
        struct
        union

base-specifier:
        version-head_opt class-name
        virtual access-specifier_opt version-head_opt class-name
        access-specifier virtual_opt version-head_opt class-name
```

The 'expression' in the revision dimension must be a compile time constant expression. Declarations of revisions must preserve an ascending sequence of revisions after the interface. The first revision is revision '1', and subsequent revisions increase in value by one from the previous revision. The compiler can thus determine the correct visibility rules, and detect missing or out of sequence revisions. No revisions are permitted after the implementor class has been declared.

Scope of Definition

The whole collection of components describes the whole type.

All members of all the fragments are considered to exist in the same scope, as if they were all present in the same class definition. Similarly, all bases introduced in all components are considered as part of a single class declaration, and obey the same rules for ambiguity and visibility.

Mapping of a generation versioned class is maintained by the compiler and is thus not subject to arbitrary conventions. This mapping of members, and indeed of introduced bases, is done in strict order of declaration, enabling the correct coexistance of older and newer versions of applications using the type.

A client may make use of the initial interface, and any number of correctly sequenced revisions. Visibility and determination of ambiguity and overloading is a function of how much of the whole class definition has been seen by the time the client declaration appears. If the revision unspecified type is used between revisions, then only the information declared to that point is used. Alternatively, the programmer of a client type using a versionable class, may declare dependence upon a specific version of the class, by explicitly using one of the 'version-head' forms of declaration. With these two methods, generation versioning may be done within a module as easily as between modules or DLLs.

The 'implementor' part defines the end of the type and signals two activities. Firstly, it announces that all knowledge about the type is now present, and the compiler must generate the necessary support code and data for the versionable type. The compiler may also make use of its exact knowledge of the type to generate more optimal code than is possible when only a partial definition is present. Secondly, an implementor component indicates that no more revisions to the type may be declared.

Consequences and Restrictions

Virtual Base Classes

Only the 'interface' class may introduce a **virtual** base class, since it is the most derived class which constructs and maps virtual classes. If this information is not always visible, a virtual base class introduced later, will fail for two reasons. Firstly, it will not be initialised by the ultimately derived class, and secondly, it will not share with the same virtual base class introduced in classes derived from the original 'interface' class.

Thus it would be an error to introduce a virtual base class at either the generation revision classes, or at the implementor class. This applies to both direct virtual base classes, and indirect virtual base classes.

This restriction does not apply to the interface class.

However, a base class introduced at any revision, or in the implementor, may itself be a versionable class.

Special Member Functions

The compiler can generate certain special functions on the behalf of a type in the absense of explicit user provided declarations of these special functions. The functions in question are specifically :-

The Default Constructor

The Copy Constructor

The Assignment Operator

The Destructor

The 'sizeof' Operator

These are normally supplied by the compiler as implicit inlines. However, with versionable classes, they must have external linkage, so that alternative definitions may be provided by future revisions. Of course, they obey the normal rules of inlinability when explicitly provided.

The 'sizeof' operator is special. Since the actual size is only known to modules where the 'implementor' clause is visible, most clients will be unaware of the actual size. For this reason, the compiler must synthesise an externally linked constant or function to determine the actual size of the type. This does not provide the programmer with the facility to define or override the sizeof operator.

Allocation and Storage Class

Since actual size is not known at compile time for most clients, the allocation of static or automatic objects involving versionable types is very difficult. Indeed, using a C++ to C translator, it may not be possible to do so without using indirection and allocating the versionable component on the free-store. However, all modules which see the 'implementor' clause may freely allocate static and automatic objects involving versionable components.

In many cases, with compiler help, it is possible to manage the dynamically sized automatic versionable objects. This can be done by first allocating all of the automatic objects of known size on the stack in the usual way, and also the non-versionable components of the versionable classes. The non-versionable parts include the original 'interface' part, which is completely immutable. After this has been done, the compiler may generate code to determine and reserved space for the versionable components. The constructors for the versionable classes may then initialise the object to refer to the versionable parts.

Static objects are generally much harder, and to implement correctly would require specialised help from the linker and run-time loader. The linker would need to know how to allocate the fixed parts of static objects, and then reserve a variant amount of space, which the loader could determine when the program is started. This could require special OS support in some environments.

I believe that it is an acceptable restriction to disallow the allocation of static or automatic versionable objects, when the implementor clause is not visible.

Name Space Conflicts

If a client type is described using the original version of a versionable class, and a newer revision of that versionable class is provided, then the object code and runtime for the client will continue to perform exactly as before (unless the client has embedded some form of size dependency). However, if the client recompiles using the newer version of the declaration for the versionable class, it is possible that the new revision will contain names which conflict with names the client already has. This 'after-the-fact' name conflict could present considerable problems to the client code. Perhaps much code would need changing, depending upon the particular nature of the conflict.

This proposal presents two possible ways of resolving the problem. Firstly, the client may place the dependent type declaration before the revision declaration which presents the conflict. Since only information to that point is visible at the time, the mechanism will resolve all of the name space according to the information presented so far.

For example :-

```
interface class Ver { ... };
revision[ 1 ] class Ver { ... };

class Dependent : Ver { ... };

revision[ 2 ] class Ver { ... };
```

Thus, the dependent class is not aware of 'revision[2]' and later, and the conflict presented by 'revision[2] may be avoided.

This may not always be feasible, especially if the new revision is in the same header file as the original and previous revisions. The second solution permits the client to specify in the declaration of the dependent type, precisely which revision they are interested in.

Thus :-

```
interface class Ver { ... };
revision[ 1 ] class Ver { ... };
revision[ 2 ] class Ver { ... };

class Dependent : revision[1]Ver { ... };
```

Similarly, members and object declarations may be made in a revision dependent way.

Both of these approaches have the problem that the client type may not avail of improvements and extensions provided by the new revision.

However, another possible way of circumventing the name conflict is to use another proposed C++ language extension "Renaming". Renaming would permit the programmer to specify alternate names for the base class members presenting the conflict.

A Possible Mechanism

There are several methods for implementing this type of construction. Given specialised compiler help, very optimal representations may be developed, involving no indirection, and run-time constant determination of member displacements. This is generally not possible, and what I intend to outline, is a portable implementation, which is still relatively small in cost, and easily implemented using current C++ translator technology.

Reusing the Virtual Base Mechanism

The whole mechanism can be described and maintained after a fashion very similar to the virtual base mechanism already implemented in todays C++ translators and compilers. The virtual base mechanism allows for a constant component, and a movable component. For versionable classes, the constant component is the original interface, and the movable and in this case, resizable component is the composite of the revisions and implementor parts.

However, unlike virtual bases, the versionable component is not shared with other versionable parts (unless of course it is a virtual base class also). The same mechanisms for member selection and virtual function calling is retained. But the visibility and ambiguity rules are as for non-virtual bases. Thus there is a very simple and orthogonal 'fit' with existing C++. The immutable part (the interface) contains a pointer to the mutable part, and this pointer is initialised by the most derived class, just as the pointers to the virtual base classes are in the existing C++ translators/compilers.

All client components will make external requests for information regarding the position and size of the actual object. This includes accessing a virtual function table of arbitrary length and version. The client modules will never generate a virtual function table, or any of the special purpose functions described earlier.

The implementor module will automatically generate the set of special functions if required, and the virtual function table. In addition, it is the responsibility of the implementor module to supply the necessary external constants for performing 'this' adjustments, and size computations for the versionable type.