Multiple Inheritance for C++

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Multiple Inheritance is the ability of a class to have more than one base class (super class). In a language where multiple inheritance is supported a program can be structured as a set of inheritance lattices instead of (just) as a set of inheritance trees. This is widely believed to be an important structuring tool. It is also widely believed that multiple inheritance complicates a programming language significantly, is hard to implement, and is expensive to run. I will demonstrate that none of these last three conjectures are true.

1 Introduction

This paper describes an implementation of a multiple inheritance mechanism for C++[4,6.] It provides only the most rudimentary explanation of what multiple inheritance is in general and what it can be used for†. The particular variation of the general concept implemented here is primarily explained in term of this implementation.

First a bit of background on multiple inheritance and C++ implementation technique is presented, then the multiple inheritance scheme implemented for C++ is introduced in three stages:

[1] The basic scheme for multiple inheritance, the basic strategy for ambiguity resolution, and the way to implement virtual functions.

[2] Handling of classes included more than once in an inheritance lattice; the programmer has the choice whether a multiply included base class will result in one or more sub-objects being created.

[3] Details of construction of objects, destruction of objects, and access control.

Finally, some the complexities and overheads introduced by this multiple inheritance scheme are summarized.

2 Multiple Inheritance

Consider writing a simulation of a network of computers. Each node in the network is represented by an object of class `Switch`, each user or computer by an object of class `Terminal`, and each communication line by an object of class `Line`. One way to monitor the simulation (or a real network of the same structure) would be to display the state of objects of various classes on a screen. Each object to be displayed is represented as an object of class `Displayed`. Objects of class `Displayed` are under control of a display manager that ensures regular update of a screen and/or data base. The classes `Terminal` and `Switch` are derived from a class `Task` that provides the basic facilities for co-routine style behavior. Objects of class `Task` are under control of a task manager (scheduler) that manages the real processor(s).

Ideally `Task` and `Displayed` are classes from a standard library. If you want to display a terminal class `Terminal` must be derived from class `Displayed`. Class `Terminal`, however, is already derived from class `Task`. In a single inheritance language, such as the original version of C++[4] or Simula67, we have only two ways of solving this problem: deriving `Task` from `Displayed` or deriving `Displayed`

† An earlier version of this paper was presented to the European UNIX Users' Group conference in Helsinki, May 1987. This paper has been revised to match the multiple inheritance scheme that was arrived at after further experimentation and thought. For more information see references 5 and 6.

from `Task`. Neither is ideal since they both create a dependency between the library versions of two fundamental and independent concepts. Ideally one would want to be able choose between saying that a `Terminal` is a `Task` *and* a `Displayed`; that a `Line` is a `Displayed` *but not* a `Task`; and that a `Switch` is a `Task` *but not* a `Displayed`.

The ability to express this using a class hierarchy, that is, to derive a class from more than one base class, is usually referred to as *multiple inheritance*. Other examples involve the representation of various kinds of windows in a window system[7] and the representation of various kinds of processors and compilers for a multi-machine, multi-environment debugger[1].

In general, multiple inheritance allows a user to combine independent (and not so independent) concepts represented as classes into a composite concept represented as a derived class. A common way of using multiple inheritance is for a designer to provide sets of base classes with the intention that a user creates new classes by choosing base classes from each of the relevant sets. Thus a programmer creates new concepts using a recipe like ‘‘pick an A and/or a B’’. In the window example, a user might specify a new kind of window by selecting a style of window interaction (from the set of interaction base classes) and a style of appearance (from the set of base classes defining display options). In the debugger example, a programmer would specify a debugger by choosing a processor and a compiler.

Given multiple inheritance and N concepts each of which might somehow be combined with one of M other concepts, we need N+M classes to represent all the combined concepts. Given only single inheritance, we need to replicate information and provide N+M+N*M classes. Single inheritance handles cases where N==1 or M==1. The usefulness of multiple inheritance for avoiding replication hinges on the importance of examples where the values of N and M are both larger than 1. It appears that examples with N>=2 and M>=2 are not uncommon; the window and debugger examples described above will typically have both N and M larger than 2.

3 C++ Implementation Strategy

Before discussing multiple inheritance and its implementation in C++ I will first describe the main points in the traditional implementation of the C++ single inheritance class concept.

An object of a C++ class is represented by a contiguous region of memory. A pointer to an object of a class points to the first byte of that region of memory. The compiler turns a call of a member function into an ‘‘ordinary’’ function call with an ‘‘extra’’ argument; that ‘‘extra’’ argument is a pointer to the object for which the member function is called.

Consider a simple class `A`†:

```
class A {
    int a;
    void f(int i);
};
```

An object of class A will look like this

```
----------------
|    int a; |
----------------
```

No information is placed in an `A` except the integer `a` specified by the user. No information relating to (non-virtual) member functions is placed in the object.

A call of the member function `A::f`:

```
A* pa;
pa->f(2);
```

is transformed by the compiler to an ‘‘ordinary function call’’:

† In most of this paper data hiding issues are ignored to simplify the discussion and shorten the examples. This makes some examples illegal. Changing the word `class` to `struct` would make the examples legal, as would adding `public` specifiers in the appropriate places.

```
f__F1A(pa,2);
```

Objects of derived classes are composed by concatenating the members of the classes involved:

```
class A { int a; void f(int); };
class B : A { int b; void g(int); };
class C : B { int c; void h(int); };
```

Again, no ‘‘housekeeping’’ information is added, so an object of class `C` looks like this:

```
-----------------
|    int a; |
|    int b; |
|    int c; |
-----------------
```

The compiler ‘‘knows’’ the position of all members in an object of a derived class exactly as it does for an object of a simple class and generates the same (optimal) code in both cases.

Implementing virtual functions involves a table of functions. Consider:

```
class A {
    int a;
    virtual void f(int);
    virtual void g(int);
    virtual void h(int);
};

class B : A { int b; void g(int); };
class C : B { int c; void h(int); };
```

In this case, a table of virtual functions, the `vtbl`, contains the appropriate functions for a given class and a pointer to it is placed in every object. A class `C` object looks like this:

```
-----------------
|    int a; |   vtbl:
|       vptr .........> -----------------
|    int b; |   |     A::f  |
|    int c; |   |     B::g  |
-----------------   |     C::h  |
            -----------------
```

A call to a virtual function is transformed into an indirect call by the compiler. For example,

```
C* pc;
pc->g(2);
```

becomes something like:

```
(*(pc->vptr[1]))(pc,2);
```

A multiple inheritance mechanism for C++ must preserve the efficiency and the key features of this implementation scheme.

4 Multiple Base Classes

Given two classes

```
class A { ... };
class B { ... };
```

one can design a third using both as base classes:

```
class C : A , B { ... };
```

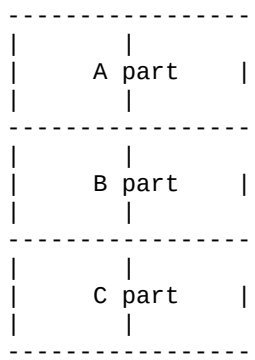
This means that a `C` is an `A` and a `B`. One might equivalently† define `C` like this:

† Except for possible side effects in constructors and destructors (access to global variables, input operations, output operations, etc.).

```
class C : B , A { ... };
```

4.1 Object Layout

An object of class `C` can be laid out as a contiguous object like this:



Accessing a member of classes `A`, `B` or `C` is handled exactly as before: the compiler knows the location in the object of each member and generates the appropriate code (without spurious indirections or other overhead).

4.2 Member Function Call

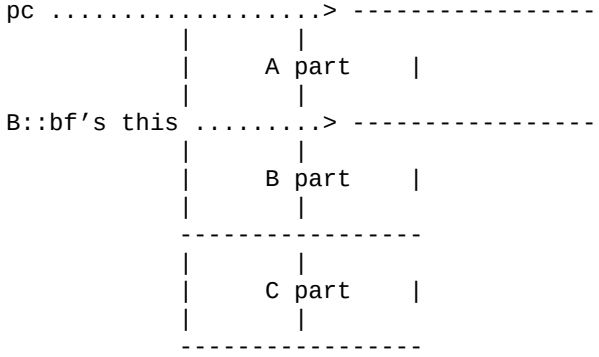
Calling a member function of `A` or `C` is identical to what was done in the single inheritance case. Calling a member function of `B` given a `C*` is slightly more involved:

```
C* pc;
pc->bf(2);  // assume that bf is a member of B
        // and that C has no member named bf
        // except the one inherited from B
```

Naturally, `B::bf()` expects a `B*` (to become its `this` pointer). To provide it, a constant must be added to pc. This constant, `delta(B)`, is the relative position of the `B` part of `C`. This delta is known to the compiler that transforms the call into:

```
bf__F1B((B*)((char*)pc+delta(B)),2);
```

The overhead is one addition of a constant per call of this kind. During the execution of a member function of `B` the function's `this` pointer points to the `B` part of `C`:



Note that there is no space penalty involved in using a second base class and that the minimal time penalty is incurred only once per call.

4.3 Ambiguities

Consider potential ambiguities if both `A` and `B` have a public member `ii`:

```
class A { int ii; };
class B { char* ii; };
class C : A, B { };
```

In this case `C` will have two members called `ii`, `A::ii` and `B::ii`. Then

```
C* pc;
pc->ii;     // error: A::ii or B::ii ?
```

is illegal since it is ambiguous. Such ambiguities can be resolved by explicit qualification:

```
pc->A::ii;  // C's A's ii
pc->B::ii;  // C's B's ii
```

A similar ambiguity arises if both `A` and `B` have a function `f()`:

```
class A { void f(); };
class B { int f(); };
class C : A, B { };

C* pc;
pc->f();    // error: A::f or B::f ?

pc->A::f(); // C's A's f
pc->B::f(); // C's B's f
```

As an alternative to specifying which base class in each call of an `f()`, one might define an `f()` for `C`. `C::f()` might call the base class functions. For example:

```
class C : A, B {
    int f() { A::f(); return B::f(); }
};

C* pc;
pc->f();    // C::f is called
```

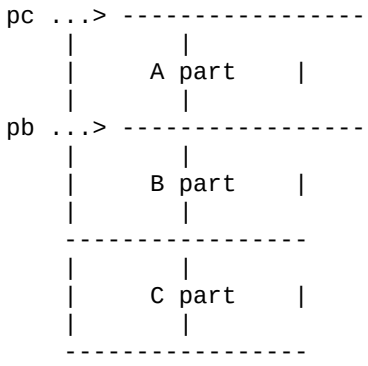
This solution usually leads to cleaner programs; it localizes the specification of the meaning of the name for objects of a derived class to the declaration of the derived class.

4.4 Casting

Explicit and implicit casting may also involve modifying a pointer value with a delta:

```
C* pc;
B* pb;
pb = (B*)pc;    // pb = (B*)((char*)pc+delta(B))
pb = pc;    // pb = (B*)((char*)pc+delta(B))
pc = pb;    // error: cast needed
pc = (C*)pb;    // pc = (C*)((char*)pb-delta(B))
```

Casting yields the pointer referring to the appropriate part of the same object.



Comparisons are interpreted in the same way:

```
pc == pb;   // that is, pc == (C*)pb
        //          or equivalently (B*)pc == pb

        // that is, (B*)((char*)pc+delta(B)) == pb
        //          or equivalently pc == (C*)((char*)pb-delta(B))
```

Note that in both C and C++ casting has always been an operator that produced one value given another rather than an operator that simply reinterpreted a bit pattern. For example, on almost all machines `(int).2` causes code to be executed; `(float)(int).2` is not equal to `.2`. Introducing multiple inheritance as described here will introduce cases where `(char*)(B*)v!=(char*)v` for some pointer type `B*`. Note, however, that when `B` is a base class of `C`, `(B*)v==(C*)v==v`.

4.5 Zero Valued Pointers

Pointers with the value zero cause a separate problem in the context of multiple base classes. Consider applying the rules presented above to a zero-valued pointer:

```
C* pc = 0;
B* pb = 0;
if (pb == 0) ...
pb = pc;    // pb = (B*)((char*)pc+delta(B))
if (pb == 0) ...
```

The second test would fail since `pb` would have the value `(B*)((char*)0+delta(B))`.

The solution is to elaborate the conversion (casting) operation to test for the pointer-value `0`:

```
C* pc = 0;
B* pb = 0;
if (pb == 0) ...
pb = pc;        // pb = (pc==0)?0:(B*)((char*)pc+delta(B))
if (pb == 0) ...
```

The added complexity and run-time overhead are a test and an increment.

5 Virtual Functions

Naturally, member functions may be virtual:

```
class A { virtual void f(); };
class B { virtual void f(); virtual void g(); };
class C : A , B { void f(); };

A* pa = new C;
B* pb = new C;
C* pc = new C;
```

```
pa->f();
pb->f();
pc->f();
```

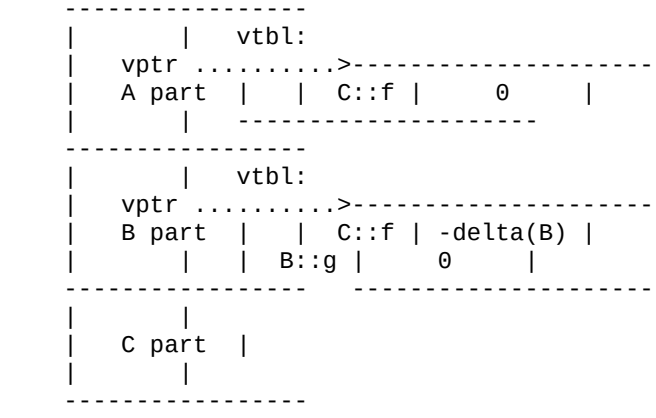
All these calls will invoke `C::f()`. This follows directly from the definition of `virtual` since class `C` is derived from class `A` and from class `B`.

5.1 Implementation

On entry to `C::f`, the `this` pointer must point to the beginning of the `C` object (and not to the `B` part). However, it is not in general known at compile time that the `B` pointed to by `pb` is part of a `C` so the compiler cannot subtract the constant `delta(B)`. Consequently `delta(B)` must be stored so that it can be found at run time. Since it is only used when calling a virtual function the obvious place to store it is in the table of virtual functions (`vtbl`). For reasons that will be explained below the delta is stored with each function in the `vtbl` so that a `vtbl` entry will be of the form†:

```
struct vtbl_entry {
    void    (*fct)();
    int delta;
};
```

An object of class `C` will look like this:




```
pb->f();    // call of C::f:
        // register vtbl_entry* vt = &pb->vtbl[index(f)];
        // (*vt->fct)((B*)((char*)pb+vt->delta))
```

Note that the object pointer may have to be adjusted to point to the correct sub-object before looking for the member pointing to the `vtbl`. Note also that each combination of base class and derived class has its own `vtbl`. For example, the `vtbl` for `B` in `C` is different from the `vtbl` of a separately allocated `B`. This implies that in general an object of a derived class needs a `vtbl` for each base class plus one for the derived class. However, as with single inheritance, a derived class can share a `vtbl` with its first base so that in the example above only two `vtbl`s are used for an object of type `C` (one for `A` in `C` combined with `C`'s own plus one for `B` in `C`).

Using an `int` as the type of a stored delta limits the size of a single object; that might not be a bad thing.

† The AT&T C++ 2.0 implementation uses three fields in this structure, but only the two shown here are used by the virtual function call mechanism; see reference 3.

5.2 Ambiguities

The following demonstrates a problem:

```
class A { virtual void f(); };
class B { virtual void f(); };
class C : A , B { void f(); };

C* pc = new C;

pc->f();

pc->A::f();
pc->B::f();
```

Explicit qualification ‘‘suppresses’’ `virtual` so the last two calls really invoke the base class functions. Is this a problem? Usually, no. Either `C` has an `f()` and there is no need to use explicit qualification or `C` has no `f()` and the explicit qualification is necessary and correct. Trouble can occur when a function `f()` is added to `C` in a program that already contains explicitly qualified names. In the latter case one could wonder why someone would want to both declare a function virtual and also call it using explicit qualification. If `f()` is virtual, adding an `f()` to the derived class is clearly the correct way of resolving the ambiguity.

The case where no `C::f` is declared cannot be handled by resolving ambiguities at the point of call. Consider:

```
class A { virtual void f(); };
class B { virtual void f(); };
class C : A , B {  };        // error: C::f needed

C* pc = new C;
pc->f();    // ambiguous

A* pa = pc; // implicit conversion of C* to A*
pa->f();    // not ambiguous: calls A::f();
```

The potential ambiguity in a call of `f()` is detected at the point where the virtual function tables for `A` and `B` in `C` are constructed. In other words, the declaration of C above is illegal because it would allow calls, such as `pa->f()`, which are unambiguous *only* because type information has been ‘‘lost’’ through an implicit coercion; a call of `f()` for an object of type `C` is ambiguous.

6 Multiple Inclusions

A class can have any number of base classes. For example,

```
class A : B1, B2, B3, B4, B5, B6 { ... };
```

It illegal to specify the same class twice in a list of base classes. For example,

```
class A : B, B { ... }; // error
```

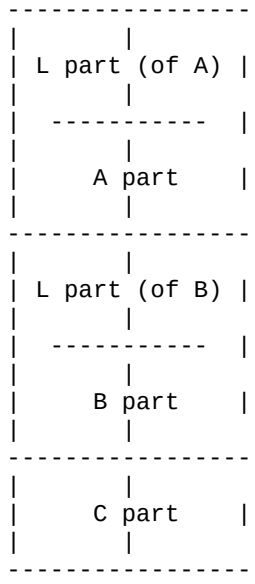
The reason for this restriction is that every access to a B member would be ambiguous and therefore illegal; this restriction also simplifies the compiler.

6.1 Multiple Sub-objects

A class may be included more than once as a base class. For example:

```
class L { ... };
class A : L { ... };
class B : L { ... };
class C : A , B { ... };
```

In such cases multiple objects of the base class are part of an object of the derived class. For example, an object of class `C` has two `L`’s: one for `A` and one for `B`:



This can even be useful. Think of `L` as a link class for a Simula-style linked list. In this case a `C` can be on both the list of `A`s and the list of `B`s.

6.2 Naming

Assume that class `L` in the example above has a member `m`. How could a function `C::f` refer to `L::m`? The obvious answer is ‘‘by explicit qualification’’:

```
void C::f() { A::m = B::m; }
```

This will work nicely provided neither `A` nor `B` has a member `m` (except the one they inherited from `L`). If necessary, the qualification syntax of C++ could be extended to allow the more explicit:

```
void C::f() { A::L::m = B::L::m; }
```

6.3 Casting

Consider the example above again. The fact that there are two copies of `L` makes casting (both explicit and implicit) between `L*` and `C*` ambiguous, and consequently illegal:

```
C* pc = new C;
L* pl = pc;       // error: ambiguous
pl = (L*)pc;      // error: still ambiguous
pl = (L*)(A*)pc;  // The L in C's A
pc = pl;          // error: ambiguous
pc = (L*)pl;      // error: still ambiguous
pc = (C*)(A*)pl;  // The C containing A's L
```

I don’t expect this to be a problem. The place where this will surface is in cases where `A`s (or `B`s) are handled by functions expecting an `L`; in these cases a `C` will not be acceptable despite a `C` being an `A`:

```
extern f(L*);   // some standard function

A aa;
C cc;

f(&aa);     // fine
f(&cc);     // error: ambiguous
f((A*)&cc); // fine
```

Casting is used for explicit disambiguation.

7 Virtual Base Classes

When a class C has two base classes A and B these two base classes give rise to separate sub-objects that do not relate to each other in ways different from any other A and B objects. I call this *independent multiple inheritance*. However, many proposed uses of multiple inheritance assume a dependence among base classes (for example, the style of providing a selection of features for a window described in §2). Such dependencies can be expressed in term of an object shared between the various derived classes. In other words, there must be a way of specifying that a base class must give rise to only one object in the final derived class even if it is mentioned as a base class several times. To distinguish this usage from independent multiple inheritance such base classes are specified to be virtual:

```
class AW : virtual W { ... };
class BW : virtual W { ... };
class CW : AW , BW { ... };
```

A single object of class W is to be shared between AW and BW; that is, only one W object must be included in CW as the result of deriving CW from AW and BW. Except for giving rise to a unique object in a derived class, a `virtual` base class behaves exactly like a non-virtual base class.

The ‘‘virtualness’’ of W is a property of the derivation specified by AW and BW and not a property of W itself. Every `virtual` base in an inheritance DAG refers to the same object.

A class may be both a normal and a virtual base in an inheritance DAG:

```
class A : virtual L { ... };
class B : virtual L { ... };
class C : A , B { ... };
class D : L, C { ... };
```

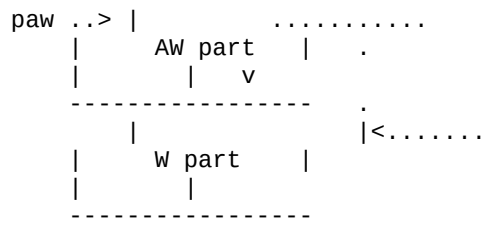
A D object will have two sub-objects of class L, one virtual and one ‘‘normal.’’

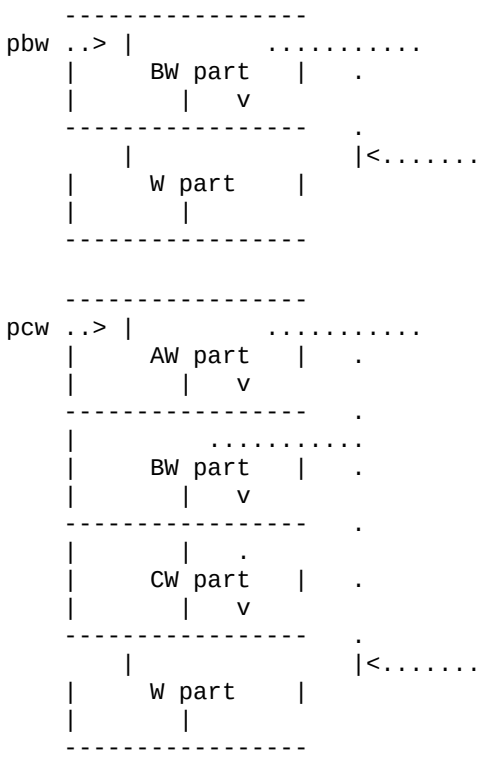
Virtual base classes provide a way of sharing information within an inheritance DAG without pushing the shared information to an ultimate base class. Virtual bases can therefore be used to improve locality of reference. Another way of viewing classes derived from a common virtual base is as alternative and composable interfaces to the virtual base.

7.1 Representation

The object representing a virtual base class W object cannot be placed in a fixed position relative to both AW and BW in all objects. Consequently, a pointer to W must be stored in all objects directly accessing the W object to allow access independently of its relative position. For example:

```
AW* paw = new AW;
BW* pbw = new BW;
CW* pcw = new CW;
```





A class can have an arbitrary number of virtual base classes.

One can cast from a derived class to a virtual base class, but not from a virtual base class to a derived class. The former involves following the virtual base pointer; the latter cannot be done given the information available at run time. Storing a ‘‘back-pointer’’ to the enclosing object(s) is non-trivial in general and was considered unsuitable for C++ as was the alternative strategy of dynamically keeping track of the objects ‘‘for which’’ a given member function invocation operates. The uses of such a back-pointer mechanism did not seem to warrant the added implementation complexity, the extra space required in objects, and the added run-time cost of initialization.

The information necessary to cast from a virtual base class to a derived class is part of what is needed to perform a virtual call of a function declared in the virtual base and overridden by the derived class. The absence of the ‘back-pointer can thus be compensated for by defining and calling appropriate virtual functions:

```
class B { virtual void f(); ... };
class D : virtual B { void f(); ... };

void g(B* p)
{
    // instead of casting B* to a D* here call f()
}
```

Since explicit casting is often bets avoided this technique actually leads to better code in many cases.

7.2 Virtual Functions

Consider:

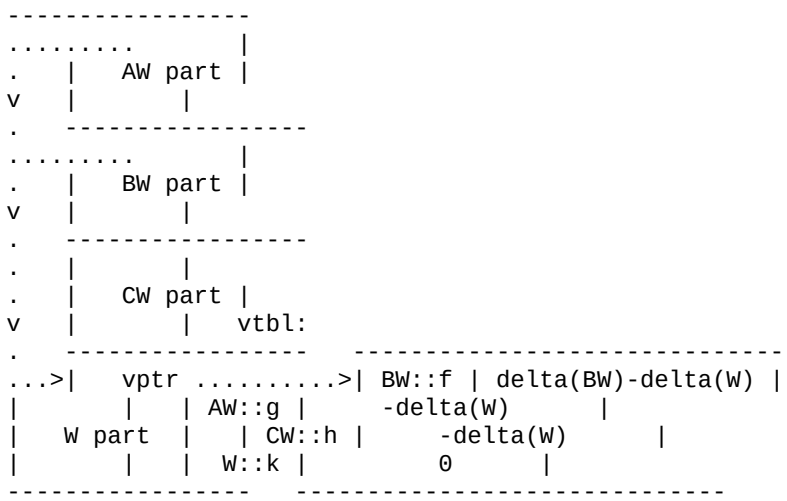
```
class W {
    virtual void f();
    virtual void g();
    virtual void h();
    virtual void k();
    ...
};

class AW : virtual W { void g(); ... };
class BW : virtual W { void f(); ... };
class CW : AW , BW { void h(); ... };

CW* pcw = new CW;

pcw->f();       // BW::f()
pcw->g();       // AW::g()
pcw->h();       // CW::h()
((AW*)pcw)->f();    // BW::f();
```

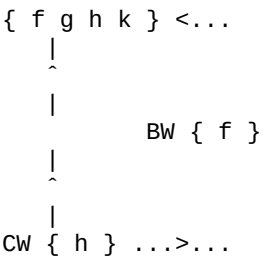
A `CW` object might look like this:



In general, the delta stored with a function pointer in a `vtbl` is the delta of the class defining the function minus the delta of the class for which the `vtbl` is constructed.

If `W` has a virtual function `f` that is re-defined in both `AW` and `BW` but not in `CW` an ambiguity results. Such ambiguities are easily detected at the point where `CW`'s `vtbl` is constructed.

The rule for detecting ambiguities in a class lattice, or more precisely a directed acyclic graph (DAG) of classes, is that there all re-definitions of a virtual function from a virtual base class must occur on a single path through the DAG. The example above can be drawn as a DAG like this:



Note that a call ‘‘up’’ through one path of the DAG to a virtual function may result in the call of a function (re-defined) in another path (as happened in the call `((AW*)pcw)->f()` in the example above).

7.3 Using virtual bases

Programming with virtual bases is trickier than programming with non-virtual bases. The problem is to avoid multiple calls of a function in a virtual class when that is not desired. Here is a possible style:

```
class W {
    // ...
protected:
    _f() { my stuff }
    // ...
public:
    f() { _f(); }
    // ...
};
```

Each class provides a protected function doing ‘‘its own stuff’’, `_f()`, for use by derived classes and a public function `f()` as the interface for use by the ‘‘general public.’’

```
class A : public virtual W {
    // ...
protected:
    _f() { my stuff }
    // ...
public:
    f() { _f(); W::_f(); }
    // ...
};
```

A derived class `f()` does its ‘‘own stuff’’ by calling `_f()` and its base classes’ ‘‘own stuff’’ by calling their `_f()`s.

```
class B : public virtual W  {
    // ...
protected:
    _f() { my stuff }
    // ...
public:
    f() { _f(); W::_f(); }
    // ...
};
```

In particular, this style enables a class that is (indirectly) derived twice from a class `W` to call `W::f()` once only:

```
class C : public A, public B, public virtual W {
    // ...
protected:
    _f() { my stuff }
    // ...
public:
    f() { _f(); A::_f(); B::_f(); W::_f(); }
    // ...
};
```

Method combination schemes, such as the ones found in Lisp systems with multiple inheritance, were considered as a way of reducing the amount of code a programmer needed to write in cases like the one above. However, none of these schemes appeared to be sufficiently simple, general, and efficient enough to warrant the complexity it would add to C++.

8 Constructors and Destructors

Constructors for base classes are called before the constructor for their derived class. Destructors for base classes are called after the destructor for their derived class. Destructors are called in the reverse order of their declaration.

Arguments to base class constructors can be specified like this:

```
class A { A(int); };
class B { B(int); };
class C : A , virtual B {
    C(int a, int b) : A(a) , B(b) { ... }
};
```

Constructors are executed in the order they appear in the list of bases except that a virtual base is always constructed before classes derived from it.

A virtual base is always constructed (once only) by its 'most derived' class. For example:

```
class V { V(); V(int); ... };
class A : virtual V { A(); A(int); ... };
class B : virtual V { B(); B(int); ... };
class C : A, B { C(); C(int); ... };

    V v(1); // use V(int)
    A a(2); // use V(int)
    B b(3); // use V()
    C c(4); // use V()
```

9 Access Control

The examples above ignore access control considerations. A base class may be `public` or `private`. In addition, it may be `virtual`. For example:

```
class D
    : B1        // private (by default), non-virtual (by default)
    , virtual B2    // private (by default), virtual
    , public B3 // public, non-virtual (by default)
    , public virtual B4 {
    // ...
};
```

Note that a access or `virtual` specifier applies to a single base class only. For example,

```
class C : public A, B { ... };
```

declares a public base `A` and a private base `B`.

A virtual class that is accessible through a path is accessible even if other paths to it use private derivation.

10 Overheads

The overhead in using this scheme is:

[1] One subtraction of a constant for each use of a member in a base class that is included as the second or subsequent base.

[2] One word per function in each `vtbl` (to hold the delta).

[3] One memory reference and one subtraction for each call of a virtual function.

[4] One memory reference and one subtraction for access of a base class member of a virtual base class.

Note that overheads [1] and [4] are only incurred where multiple inheritance is actually used, but overheads [2] and [3] are incurred for each class with virtual functions and for each virtual function call even when multiple inheritance is not used. Overheads [1] and [4] are only incurred when members of a second or subsequent base are accessed ''from the outside''; a member function of a virtual base class does not incur special overheads when accessing members of its class.

This implies that except for [2] and [3] you pay only for what you actually use; [2] and [3] impose a

minor overhead on the virtual function mechanism even where only single inheritance is used. This latter overhead could be avoided by using an alternative implementation of multiple inheritance, but I don't know of such an implementation that is also faster in the multiple inheritance case and as portable as the scheme described here.

Fortunately, these overheads are not significant. The time, space, and complexity overheads imposed on the compiler to implement multiple inheritance are not noticeable to the user.

11 But is it Simple to Use?

What makes a language facility hard to use?

[1] Lots of rules.

[2] Subtle differences between rules.

[3] Inability to automatically detect common errors.

[4] Lack of generality.

[5] Deficiencies.

The first two cases lead to difficulty of learning and remembering, causing bugs due to misuse and misunderstanding. The last two cases cause bugs and confusion as the programmer tries to circumvent the rules and ''simulate'' missing features. Case [3] causes frustration as the programmer discovers mistakes the hard way.

The multiple inheritance scheme presented here provides two ways of extending a class's name space:

[1] A base class.

[2] A virtual base class.

These are two ways of creating/specifying a new class rather than ways of creating two different kinds of classes. The rules for using the resulting classes do not depend on how the name space was extended:

[1] Ambiguities are illegal.

[2] Rules for use of members are what they were for single inheritance.

[3] Visibility rules are what they were for single inheritance.

[3] Initialization rules are what they were for single inheritance.

Violations of these rules are detected by the compiler.

In other words, the multiple inheritance scheme is only more complicated to use than the existing single inheritance scheme in that

[1] You can extend a class's name space more than once (with more than one base class).

[2] You can extend a class's name space in two ways rather than in only one way.

In addition, care must be taken to take the sharing into account when programming member functions of classes with virtual base classes; see section 7 above.

This appears minimal and constitutes an attempt to provide a formal and (comparatively) safe set of mechanisms for observed practices and needs. I think that the scheme described here is ''as simple as possible, but no simpler.''

A potential source of problems exists in the absence of ''system provided back-pointers'' from a virtual base class to its enclosing object.

In some contexts, it might also be a problem that pointers to sub-objects are used extensively. This will affect programs that use explicit casting to non-object-pointer types (such as `char*`) and ''extra linguistic'' tools (such as debuggers and garbage collectors). Otherwise, and hopefully normally, all manipulation of object pointers follows the consistent rules explained in §4, §7, and §8 and is invisible to the user.

12 Conclusions

Multiple inheritance is reasonably simple to add to C++ in a way that makes it easy to use. Multiple inheritance is not too hard to implement, since it requires only very minor syntactic extensions, and fits naturally into the (static) type structure. The implementation is very efficient in both time and space. Compatibility with C is not affected. Portability is not affected.

13 Acknowledgements

In 1984 I had a long discussion with Stein Krogdahl from the University of Oslo, Norway. He had devised a scheme for implementing multiple inheritance in Simula using pointer manipulation based on addition and subtaction of constants. Reference 2 describes this work. Tom Cargill, Jim Coplien, Brian Kernighan, Andy Koenig, Larry Mayka, Doug McIlroy, and Jonathan Shopiro supplied many valuable suggestions and questions.

14 References

[1] Tom Cargill:
PI: A Case Study in Object-Oriented Programming.
OOPSLA'86 Proceedings, pp 350-360, September 1986.
[2] Stein Krogdahl:
An Efficient Implementation of Simula Classes with Multiple Prefixing.
Research Report No. 83 June 1984,
University of Oslo, Institute of Informatics.
[3] Stan Lippman and Bjarne Stroustrup:
Pointers to Members in C++
Proc. USENIX C++ Conference, Denver, October 1988.
[4] Bjarne Stroustrup:
The C++ Programming Language.
Addison-Wesley, 1986.
[5] Bjarne Stroustrup:
What is ''Objet-Oriented Programming?''.
Proc. ECOOP,
Springer Verlag Lecture Notes in Computer Science, Vol 276, June 1987.
[6] Bjarne Stroustrup:
The Evolution of C++: 1985-1989.
USENIX Computing Systems Vol 2 no 3, Fall 1989.
[7] Daniel Weinreb and David Moon:
Lisp Machine Manual.
Symbolics, Inc. 1981.

14.1 Appendix

The original multiple inheritance design as presented to the EUUG conference in Helsinki in May 1987 contained a notion of delegation†. Briefly, a user was allowed to specify a pointer to some class among the base classes in a class declaration and the object thus designated would be used exactly as if it was an object representing a base class. For example:

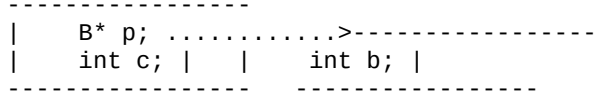
```
class B { int b; void f(); };
class C : *p { B* p; int c; };
```

The `:*p` meant that members of the object pointed to by `p` could be accessed exactly as if they were members of class `C`:

```
void f(C* q)
{
    q->f();     // that is, q->p->f()
}
```

An object of class `C` looked something like this after `C::p` has been initialized:

† Gul Agha: *An Overview of Actor languages*. SIGPLAN Notices, pp58-67, October 1986.



This concept looked very promising for representing structures that require more flexibility than is provided by ordinary inheritance. In particular, assignment to a delegation pointer could be used to reconfigure an object at run-time. The implementation was trivial and the run-time and space efficiency ideal. Unfortunately, every user of this mechanism suffered serious bugs and confusion. Because of this the delegation mechanism was not included in C++.

I suspect the fundamental problem with this variant of the notion of delegation was that functions of the delegating class could not override functions of the class derived to. Allowing such overriding would be at odds with the C++ strategy of static type checking and ambiguity resolution (consider the case of an object of a class with virtual functions delegated to by several objects of different classes).