A Critique of the Inheritance Principles of C++

Markku Sakkinen University of Jyväskylä, Finland

ABSTRACT: Although multiple inheritance (MI) is already a feature of the C++ language, there is a debate going on about its good and bad sides. In this paper, I am defending MI. At the same time, many of the rules and principles of inheritance in current C++ seem to me to need improvements. The suggested modifications are relatively simple, do not introduce many new reserved words, and should not affect other parts of the language.

I do not find the current rules totally adequate even for single inheritance (SI). The problems lie in the meaning of access levels and in the redefinability of virtual functions. Additional inconsistencies in C++ virtual functions appear in so-called independent multiple inheritance (IMI), which is in principle the easy case of MI. The most difficult problems are caused by so-called fork-join inheritance (FJI), which is the most complicated kind of inheritance.

One essential cause of complexity is private inheritance because of its intransitive nature. The main idea suggested here is that, simply put, private inheritance should be implicitly "non-virtual" and public inheritance "virtual." This is actually a simplification of the language, at least from the programmer's if not from the implementor's point of view. The main rule has also been generalised to arbitrary combinations of private and public inheritance, with some restrictions on legal combinations. I sincerely think that the current C++ rules will be very harmful if programmers start developing complex class hierarchies in which FJI is applied. On the other hand, the new rules suggested here should behave consistently even in complex situations but demonstration is so far missing, of course. The same principles should be applicable to other object-oriented languages beside C++.

MI increases the complexity of the language in any case; Cargill [1991a] has therefore required good examples of its advantages to make it worthwhile. I think that Waldo's [1991a] example is convincing enough for IMI, but I join in the quest for equally good examples using FJI.

1. Introduction

This article is aimed at an audience that has some previous understanding of both object-oriented programming (OOP) and the C++ language (some parts will require a detailed knowledge of C++). A reader who thinks that the two are almost synonyms, or that OOP and Smalltalk-80 are almost synonyms, should probably read one of the good tutorials or books on OOP that are available today, first.

Be warned that I am a convinced opponent of C and all C-based languages for general-purpose programming [Sakkinen 1988, 1992]. In spite of this, I will make several constructive suggestions for improvements to C++ in this article (of course I would prefer the fundamental ideas to be adopted into inherently better languages). Secondly, I am currently a rather pure theoretician, not having done any real programming in any language for quite some time. I had some experience of C++ prior to Release 2.0, and while finishing this paper, have been able to test some questionable things on a Release 2.1 compiler (Hewlett-Packard).

The initial motivation for this article was to examine multiple inheritance (MI). There is still some debate on whether MI is at all necessary in object-oriented systems, and whether its true advantages outweigh the complexity that it introduces into a language, particularly C++. I do believe in MI, especially from a theoretical standpoint. It was therefore surprising to me that among the 30 or so participants of the ECOOP'91[1] workshop on "Types, inheritance and assignments", a majority answered "No" when asked if they would include MI in their next object-oriented language. Perhaps the answer was motivated mainly by implementation considerations—I forgot to ask about that.

In C++, multiple inheritance has already been defined and implemented; although every independent new implementer must suffer the tedium of MI. As there is no official standard for the language yet, Tom Cargill has been campaigning here [1991a] and elsewhere at least for a moratorium against MI. He has recently been countered by Jim Waldo [1991a]. Here I will try to give some further arguments in favour of MI.

Unless we agree with the verdict that multiple inheritance should be banished from C++, the next important question is *how* it should be defined. Is the current approach adequate? Originally I really liked it [Sakkinen 1989]. However, some problems have appeared [Baclawski 1990; Snyder 1991]. I will try to elaborate on these problems in the larger part of this paper. They proved to be more numerous and more complex than I had thought at the start of writing. Another surprise was that some problems already pertain to single inheritance (SI).

The plot of the rest of this article is as follows. I will try to summarise shortly the most relevant points of Cargill and Waldo on MI in Section 2, with some references to other literature. Section 6 draws some conclusions, finishing in optimism on the feasibility of multiple inheritance. At the end we have the obligatory acknowledgements and reference list. The beef of the hamburger is in the middle.

1. Fifth European Conference on Object-Oriented Programming, Geneva, Switzerland, July 15-19, 1991.

Section 3 treats issues that are relevant even to SI, especially ideas and problems of inheritance-related access control. Some of the points will not be too familiar to most readers; also, the distinction between private and public inheritance will be essential in the ensuing analysis of MI. Some language modification proposals will appear here.

Section 4 treats the simpler form of MI, so-called independent multiple inheritance. I will claim that the current C++ rules have a severe defect that affects already this case. Section 5 discusses the more complicated case of MI, so-called fork-join inheritance. It is essentially more complex than would appear from the typical literature examples containing four or five classes. Not surprisingly, here I will present the largest number of defects, problems, and solutions.

The most important points will be emphasised in the form of six theses and one rule in boldface. In fact, a hasty reader may look up just these points and perhaps the conclusions. The rule, which I believe to be the most significant single contribution of this paper, will be found in Subsection 5.4. It is followed by two restrictions.

The paper tries to be as self-contained as possible in its reasoning. However, there are more numerous and more detailed references to existing literature than may be conventional in this journal. Those readers who are not interested to compare the various papers in detail can safely ignore the references.

2. The *If* and the *How* of Multiple Inheritance

2.1 *If:* the main points of Cargill and Waldo

Cargill [1991a §4] describes the complexity of current C++ inheritance:

> *[There are] six variants of inheritance: a choice of three access levels for each inheritance relationship (public, protected or private), and another choice of whether or not each base class is virtual. The real expressive power of inheritance is delivered by just one of the six variants: public inheritance from a non-virtual base.*

I disagree with the last statement, and will argue against it in a later section. I agree about the complexity (see §3.2 about *protected* base classes), and so did Waldo.

Cargill goes on to show how multiple inheritance further increases the complexity of the language. We have no argument here: Bjarne Stroustrup himself has been the first to admit and explain the basic complications. Note that the choice between "virtual" and "nonvirtual" base class mentioned above was introduced because of MI.

The other main argument of Cargill [1991a] is that no convincing examples of MI had been published. I had wondered a little about that myself, but reasoned that such examples would tend to be too long for typical journal and conference papers. However, I had heard people complain how impossible it was to utilise two independent, extensive class libraries such as OOPS (currently NIHCL) [Gorlen 1987] for "foundation classes" and InterViews [Linton & Calder 1987] for the user interface, within the same software system using C++ without MI.

Waldo [1991a] suggests that the MI examples cited by Cargill are unconvincing mainly because they are based on *implementation inheritance*. This is a very noteworthy point in my opinion. He presents an example with *interface inheritance* and *"data inheritance"*, in outline. He convinces me at least that

1. the example is not artificial, but programmers can often be confronted with similar situations, and
2. the problem could be solved only in very contorted ways if MI were not available.

The true usefulness of multiple inheritance (in C++) has thus been demonstrated. However, to a sceptic this only means that MI cannot be dismissed off-hand; the tradeoff between its advantages and disadvantages still remains a matter of judgement.

Cargill's paper nevertheless makes a lot of sense, especially from its chosen viewpoint of practical programming. At several places it really does not criticise so much *multiple* inheritance, as the common tendency in OOP to apply inheritance even where simple aggregation would be more appropriate; I could not agree more.

2.2 *How:* the different cases

Waldo [1991a] seems to perceive Cargill as quite adamant against multiple inheritance; I have a more openminded impression. Cargill

[1991a §6] actually sketches a class structure from which MI could not be eliminated without serious distortion, and says:

> *If MI were widely used in this manner in real programs, my thesis would collapse.*

The main idea is that at least some virtual function of each base class is redefined in the derived class so that it calls a virtual function of *another* base class. Interestingly, Waldo's example is totally different from this structure.

We can distinguish between two main forms of MI. By *"independent multiple inheritance"* (IMI) we mean that parallel superclasses have no common ancestors, or that there is only one derivation path connecting a class to an non-immediate base class[2]. The opposite case we will call *"fork-join-inheritance"* (FJI), as coined in Sakkinen [1989] in analogy with the forking and joining of parallel processes. I have not seen any other compact term for this phenomenon in the literature.

Cargill [1991a §8] briefly admits the need for IMI, motivated by multiple independent class libraries (cf. §2.1):

> *We may discover that MI is indeed useful, but that virtual base classes are unnecessary.*

He has later [1991b] paraphrased this unambiguously:

> *FJI is not useful and therefore the virtual base question is moot.*

Waldo does not take a stand on this issue in the article [1991a], but he has later [1991b] clarified his position to be rather opposite to Cargill:

> *[...] my point might show that virtual base classes are the only candidates for multiple inheritance.*

In IMI there is no difference between virtual and nonvirtual base classes. Therefore, Cargill's original statement could have been interpreted thus:

> *FJI is useful, but only with non-virtual base classes;*

at least out of context. This was certainly not the intention. Cargill does not give any semantic reasons against virtual base classes; his

2. This term is obviously used in a slightly more general meaning in Stroustrup [1989b].

goal is to reduce the complexity mentioned in the previous section. We have here a similar tradeoff situation as with MI in general. Neither Cargill's nor Waldo's example contains fork-join inheritance; thus the "no supporting evidence" argument remains valid. I will later try to suggest some reasons in favour of FJI, but they will not be very decisive.

Baclawski [1990] calls MI with non-virtual base classes 'multiple independent inheritance'. This includes both IMI as defined above and non-virtual FJI. Baclawski regards this as a peculiar variation of MI.

Another recent paper with some highly interesting points on MI—although I do not agree with all of them—is Snyder [1991]. It has been written with the purpose of describing the most essential properties of C++ in terms of a supposedly language-independent model. Snyder regards non-virtual FJI as an unusual corner case of the language; he did not bother to make his general model so complex that it could account for this situation. FJI with virtual base classes is not treated specifically in his article.

3. *Inheritance and Accessibility*

3.1 *Access Levels of Class Members*

There were originally only two alternative accessibility levels for both class members and base classes, `public` and `private` (and actually no explicit specifier yet for private accessibility). The intermediate level, `protected`, was added for members only in C++ Release 1.2 from AT&T.

C++ checks for name clashes between inherited and noninherited members (and between members inherited from different base classes) *before* applying access controls. I think that this is the wrong choice, the rationale in Ellis & Stroustrup [1990 §11.3c] notwithstanding:

> *Making a name public or private will not quietly change the meaning of a program from one legal interpretation to another.*

The flaw in this reasoning is that changing the access level of a class member is a modification of the class's *interface,* which should always be expected to affect the clients of the class and should not be done too lightly.

On the other hand, adding or renaming a private member is only a matter of the class's *implementation* and therefore should not concern clients at all. As things are in C++, private members of a class cannot be of any benefit to derived classes (and outside clients) but can cause harm to them. The same holds for members of private base classes. The special case of `virtual private` functions will be discussed in §3.4.

When `protected` class members were first introduced, their definition was simple and sensible [AT&T 1986]. A protected member `m` defined in a class `A` was accessible both to class `A` itself and to any class `B` directly derived from `A`; unless the derivation was private, the accessibility of `m` in `B` was the same as if `m` had been a protected member of `B` itself.

With Release 2.0, the meaning was subtly changed. Even if a protected member `m` of class `A` is accessible to some descendant class `C`, member functions of `C` are now allowed to access the `m` part of an object only if that object is statically known to belong to class `C` or a descendant of `C`.

The rationale in Ellis & Stroustrup [1991 p. 254] says:

> *[The original rule] would allow a class to access the base class part of an unrelated class (as if it were its own) without the use of an explicit cast. This would be the only place where the language allowed that.*

'Unrelated' in the quote seems queer since the classes have a common ancestor and only the common part would be accessed. Otherwise this argument makes some sense—I assume that mentioning explicit casts is not meant to imply that they could be used to bypass the new restriction.

The example on p. 255 has a class `Account` with derived classes `checking_account` and `AutoLoan_account`, and it is argued:

> [`Account`] has information common to all kinds of accounts, including the account balance, so a friend of `Account` can walk the list of all Accounts and tell [the balances]. [...] The member functions of `checking_accounts`, however, should not be able to access the balance in an `AutoLoan_account`. The restriction prevents that.

This also makes some sense, but not completely. If a friend function is supposed to be able to treat the balance in all possible classes derived from `Account` correctly, why should a member function of a derived class be unable to do the same? I would much more want to prevent other people's account *instances* from accessing the balances of my accounts (of whatever kind), but that is not possible in C++.

In my opinion, this complication is a display of paternalism contrary to the general philosophy of C++[3] (cf. §3.7). It can prevent some obscure programming errors, but also makes a potentially much larger number of perfectly sound and useful pieces of code illegal.

3.2 Modes (Access Levels) of Inheritance

There is an uncertainty on whether `protected` base classes should be possible in current C++. Ellis & Stroustrup [1990] is inconsistent on this point, with most clues leading to a negative answer. At least the HewlettPackard C++ translator/compiler does not accept `protected` base classes. On the contrary, Stroustrup [1991] makes it clear that protected base classes are indeed meant to be possible (in some future version?), and also describes their semantics.

Bjarne Stroustrup's books and articles try to make a clear conceptual distinction between public and private inheritance (derivation). For a long time, I was in doubt about how protected inheritance could logically fit into this picture. Now I think that one can regard protected inheritance as a restricted case of public inheritance, in that both these modes are *transitive*: any class in an inheritance hierarchy can access all its non-immediate base classes. For the purposes of this paper, it suffices to speak of public inheritance, and the results should be applicable to protected inheritance as well.

Private inheritance, in contrast, is *intransitive*: every class can access only its immediate base classes. This is the kind of inheritance that has been recommended in earlier work of Alan Snyder [1987]. It is akin to "incidental inheritance" in Sakkinen [1989], while the public inheritance of C++ corresponds to "essential inheritance."

Public inheritance is supposed to imply more or less an *is-a* relationship, which must be transitive. Already because of its intransitivity, private inheritance does not imply any *is-a* relationship. In private (single) inheritance, the role of the base class can be very similar to the role of a *representation* in CLU [Liskov et al. 1981].

C++ differs from the majority of object-oriented languages in that the data members of an object are directly contained in the object,

3. I would still prefer a more paternalistic general philosophy!

independently of their type. In most other languages, an object can contain only *pointers* to other objects; this is called "reference semantics". Because of this property, private inheritance is much less different from aggregation (i.e., a private base class from a data member) in C++ than in those other languages. One might suggest that private (and protected) inheritance be eliminated: that would simplify the language quite a bit.

I can agree with Baclawski [1990][4] and Cargill [1991a] that public inheritance is the more important case and should preferably have been the default. However, I cannot totally agree that private inheritance is *only* aggregation with some syntactic sugar, although I had suggested in Sakkinen [1989] that *incidental* inheritance could be replaced by aggregation and three simple constraints. Baclawski is wrong in claiming that private inheritance in C++ does not support late binding; I had not quite realised that earlier, either. In reality, a virtual function of a base class can be redefined even in a privately derived class—with less restrictions than I would like (§3.4)—and its invocations from other functions of the base class and the derived class will be late-bound.

Because `private` derivation does conserve this essential property of object-oriented inheritance, late-bound self-reference, I am at the end not willing to have it removed from C++, in spite of the complications. It is also such a traditional feature that its elimination would break too many pieces of existing software. `Protected` derivation could easily be omitted now as it is just being introduced. On the other hand, it adds only little complexity, and can probably be useful in some situations. It is also more orthogonal that the sets of possible access levels are the same for members and for base classes.

3.3 Some Definitions

We will now define some terms that will be needed a little later. The more generally used term 'inheritance' has already been used as a synonym of 'derivation', which is more common in C++ literature. The words *'ancestor'* and *'superclass'* will be used to mean immediate or non-immediate base class, and likewise the words *'descendant'* and *'subclass'* to mean immediate or non-immediate derived class. (These

4. Baclawski [1991] says that this assessment was originally made by Stroustrup himself.

meanings correspond to 'proper ancestor' and 'proper descendant' in Meyer [1988].)

The accessibility of ancestors to descendants is probably intuitively clear to those readers who know enough about C++ to have bothered to read this far. The following definitions will make our concepts precise enough. For the purposes of MI we need to think about the accessibility of *paths*, while the accessibility of (ancestor) classes would be sufficient for SI.

Let us regard the *inheritance graph* of a "closed"[5] collection of C++ classes as a labelled directed graph, where each edge is directed from derived class to base class and labelled with its access mode and sharability (virtual or non-virtual). An inheritance graph is always a directed acyclic graph (DAG), but *not* in general a lattice, contrarily to what is often incorrectly claimed in the object-oriented literature. We will call any path (sequence of nodes and adjoining edges) in this labelled DAG a *derivation path*.

A derivation path will be called *transitively accessible* if it contains no edge labelled private (which includes the special case of a zero-length path), *intransitively accessible* if only the first edge is labelled private, and *inaccessible* otherwise. A *class* A is called accessible to a class B if there is at least one accessible path from B to A (note the direction: B is either A itself or a descendant of A), and inaccessible otherwise. When there is a danger of ambiguity, we can use the longer word 'inheritance- accessible' about classes.

What does this mean in practice? To any class B, B itself and its public (and protected) ancestors are transitively accessible. The direct private base classes of B and their public (and protected) ancestors are intransitively accessible to B (unless also transitively accessible by another path). The private ancestors of any ancestor class are inaccessible to B (unless accessible by another path). The distinction between accessible and inaccessible ancestor classes will be crucial in the sequel. The distinction between transitive and intransitive accessibility will be also be needed, but much less often.

The reader should convince him/herself that the accessibility gained by a descendant due to inheritance in C++ indeed corresponds to the above definition. We expressly exclude `friend` accessibility (which is intransitive like private inheritance) from these definitions,

5. For every class in the collection, all its ancestors must also be in the collection.

for instance because friend relationships do not affect late binding. The fact that a descendant may gain better access to an ancestor by being declared also a friend should just be remembered.

For every single class `C` we define the inheritance graph of `C` as consisting of all derivation paths whose first node is `C`. For every instance of `C` there is a *subobject graph* corresponding to the inheritance graph, where each subobject node is labelled with the name of its class. Each subobject contains the non-inherited non-static members of its class. All the graphs in Ellis & Stroustrup [1990 §10] are subobject graphs: the nodes are subobjects and not classes.

The correspondence between *paths* in the two graphs is one-to-one. However, a class in the inheritance graph may correspond to more than one node in the subobject graph, depending on the sharabilities. That is one of the main issues of §5. For that discussion we will need one more definition: the *complete subobject* corresponding to a node N in the subobject graph shall be the subgraph reachable from N, i.e. consisting of all paths whose first node is N.

3.4 Problems of Private Inheritance

There is a flaw in the access control principles that only affects virtual functions. Namely, a derived class can *redefine* any virtual function of any ancestor class, even those that it cannot *invoke*. We illustrate this with an example:

```
class Top {
public: /* or protected: */
      virtual void work();
      ...
};
class Middle: private Top {
      ...
};
class Bottom: /* any mode */ Middle {
      ...
      virtual void work();
      ...
};
```

Example 1.

All calls of `work` in member functions of `Top` without explicit class qualification will invoke `Bottom::f`, within a `Bottom` object. Private derivation thus does not isolate the classes `Top` and `Bottom` from each other.

I propose in a slightly oversimplified way:

Thesis 1: A descendant class must not be able to redefine a virtual function of an inaccessible ancestor class.

In the next subsection I will suggest the possibility of purely static overriding, however. It may seem that we have here only a question of taste instead of a true anomaly in the current C++ rules. Stroustrup [1991b] says that he did not want to be paternalistic. However, without this restriction it is not possible to avoid the "exponential yoyo problem" (§5.6).

There is an interesting consequence of Thesis 1. It obviously makes Example 1 illegal, or at least prevents late binding from `Top` to `Bottom::work`. That does not change if a redefinition of `work` is added to class `Middle`. Therefore, `Middle::work` cannot be virtually redefined further, according to the basic principle expressed in Ellis & Stroustrup [1990 p. 205]:

For virtual functions, [...] the same function is called independently of the static type of the pointer, reference, or name of the object for which it is called.

Corollary 1: A virtual function of class `C` whose original definition is inherited from an intransitively accessible ancestor class, must not be further redefined in descendants of `C`.

Thesis 1 was originally formulated thus: "A descendant class must not be able to redefine a virtual function that it cannot access." Under this rule it would not make sense to declare a member function `private virtual`. Bjarne Stroustrup convinced me that there can be a scenario in which the virtuality of a private function is useful, and that virtuality should be independent of the access level. He could *not* convince me that virtuality should be independent of derivation modes.

We thus split the accessibility of a virtual function into *invokability* and *redefinability*. The former is initially defined by the access level and the latter by the virtuality; for an inherited function, both are affected by the mode of derivation.

The usefulness of a private (instead of protected) virtual function is rather marginal. Let us examine a simple example, enhanced from Stroustrup's [1991b] by the addition of the intervening class `B`:

```
class A {
private:
      virtual void f();
public:
      virtual void g() { ... f(); ... }
      ...
};
class B: public A {
      // no redefinition of f
      ...
};
class C: public B {
      ...
      virtual void f();
      ...
};
```

Example 2.

Obviously the member functions of class `B` cannot invoke `f`; this restriction can be desired in some situations. There is no declaration in current C++, however, that would prevent class `C` from invoking `f`. On the other hand, `C::f` cannot invoke `A::f`, nor can a redefinition of g in class `B` invoke `f`, both of which would typically be wanted. The code of `A::f` should then instead be duplicated in `C::f` or `B::g`, which is against the object-oriented reuse principle.

3.5 A Proposed Solution

We must search deeper in the foundations for a totally consistent solution. It appears that C++ has no mechanism nor even term for handling a "family" of virtual functions, i.e. the "most base" function together with its all redefinitions. This is one of the most important concepts in Snyder's [1991] object model: he uses the term 'operation' in this specific meaning. One of the most difficult questions in that paper indeed is: "What are the operations?" Pointers to member functions are a partial answer (see §4.1, §4.2).

In current C++, every member[6] of a virtual function family is considered a completely independent function, except for late binding. Therefore also the invokability of a redefined function may be different from that of the original one. I suggest that a redefinition should *never* change the invokability of a virtual function. Of course, the mode of derivation may *lower* the invokability.

Thesis 2: The invokability of a redefinition of a virtual function in the redefining class should always be the same as that of the inherited function.

Note that according to Thesis 2, the function `f` in Example 2 would not be invokable even from class C, even though it is redefined there. This may sound surprising, but is in a way more consistent than the existing situation, where class `B` cannot invoke `f` although both its superclass and subclass can. Further, I really think that the cases in which a virtual function should be private are very rare.

There appears to be a nice way to satisfy the thesis while solving another, related problem. The meaning of virtual function declarations is currently quite contextdependent. An explicit `virtual` specifier can mean either that a new virtual function family is being defined or that an inherited virtual function is being redefined (this ambiguity was not allowed originally). The omission of `virtual` can mean either that an inherited virtual function is being redefined, an inherited non-virtual function is being statically overloaded, or a new non-virtual function is being defined. It would be much better to have a distinct keyword for the redefinition, as e.g. in Eiffel (cf. §4.2).

According to Thesis 2, the invokability of a virtual function redefinition would not be affected by the member access specifiers; this is similar to `friend` declarations. A syntactically distinguished virtual function redefinition would be elegant also from this viewpoint. In Example 1, the following could be added to the definition of class `Middle`:

```
    redefine void work();
```

6. Here 'member' is in its ordinary meaning, not in the somewhat confusing C++ meaning (component).

Allowing the static (non-virtual) overriding of an inaccessible virtual function could be possible as an extension to the current C++ rules; it only makes sense because of the other suggestions. The function `Bottom::work` in Example 1 would then be legal, but only a static overloading of `Top::work` and would not belong to the same family, although itself virtual. In contrast, it will be lightly suggested in §3.7 that the non-virtual overriding of *accessible* members should not be allowed—a restriction to the current rules.

It has been conceded in §3.4 that the scope of virtual redefinability of a member function can sometimes be larger than that of invokability. Probably more often one would like to restrict the scope of redefinability to be *smaller,* i.e. to prevent further redefinitions from some class downward but retain invokability. BETA [Madsen 1987] is one language that offers such a possibility. Let us modify Example 1 again a little, with some tentative syntax:

```
class Middle: public Top {
        ...
        freeze void work();
        ...
};
```

Note that the derivation from `Top` was changed to public. Class `Bottom` would then be able to invoke but not to redefine `work`.

Adding new keywords to C++ is always a bit dubious, or at least requires very good reasons. This is a general problem in languages in which keywords are in the same space as programmer-invented names. One could manage without new keywords by replacing `virtual`, `redefine`, and `freeze` in the suggestion by, say, `virtual >`, `virtual =`, and `virtual <`, respectively. Unfortunately, one reviewer was afraid that this alternative could cause trouble for parsing.

3.6 Virtual Base Classes

Although the problems of `virtual` base classes would logically belong to the section on fork-join multiple inheritance, they are so central in the discussion that we may note some points already here. To begin, I think it has been an unfortunate choice to overload the term 'virtual' with this meaning, remotely related to the original one; for instance 'shared' would have been a better word. Recall that the meaning is [Ellis & Stroustrup 1990 p. 200]:

> *A single sub-object of the virtual base class is shared by every [derived]*[7] *class that specified the base class to be virtual.*

As explained by Stroustrup [1987, 1989] and Ellis & Stroustrup [1990 §10], virtual base classes are a little more difficult and more costly to implement than non-virtual ones. Even then they are subject to the additional restriction that a pointer to a virtual base class cannot be cast into a pointer to a derived class. Casts from base to derived class are, however, extremely risky in C++ even in the cases where they are allowed, and should be avoided. The reason is the lack of run-time type information in objects—in my opinion, insufficient object orientation.

Stroustrup admits that writing virtual functions can be trickier in the presence of virtual base classes. The example in e.g. Ellis & Stroustrup [1990 p. 201–202] shows that one must often write another, protected and non-virtual function for descendant classes to call; otherwise the function in some base classes may get invoked more than once. Cargill [1991a] dislikes the complexity caused by possible "sideways" redefinitions of virtual functions with virtual base classes. We will treat this question in §5.5–5.6 and find that FJI with non-virtual base classes can actually lead to much more anomalous situations.

The virtuality of base classes must be considered in object construction, too. It is written in Ellis & Stroustrup [1990 §12.6.2]:

> *A complete object* is an object that is not a sub-object representing a base class. Its class is said to be the *most derived* class for the object. All sub-objects for virtual base classes are initialized by the constructor of the most derived class.

This is obviously necessary in some situations, but in this formulation it is a too sweeping requirement, which may cause the effects of virtual derivation to propagate too far. It should be refined. Consider:

7. It actually reads 'base' in the book, but that must be a simple clerical mistake.

```
class A {
public:
      A(int);
      ...
};
class B : public virtual A {
public:
      B(int i) : A(i) { ... }
      ...
};
```

Example 3.

Now all constructors of every class derived from B directly or indirectly, even by single inheritance, must explicitly invoke the constructor of A! The rule also requires an ugly exception to the accessibility rules of class members: the constructors of a virtual base class are accessible both to the virtually derived class and all its descendants, ignoring all access specifiers.

The effect of declaring a base class virtual is too global also in the sense that there is only one subobject within a complete object that corresponds to *all* occurrences of the same class as a virtual base. Although this seems simple and logical at first, it is less logical in some more complicated inheritance graphs. I will suggest an essential modification to this principle in §5.4.

Thesis 3: The effects of declaring a derivation virtual should not propagate too far in the subobject graph.

3.7 Non-virtual overriding and overloading

The philosophy of name scoping between superclasses and subclasses in C++ is the same as in Simula™: the scopes are regarded as if they were lexically nested. Variables and functions of the derived class can thus non-virtually override (hide) those of the base class.

This philosophy originated when there was yet no mechanism similar to the access modes of C++, and was then a sensible way to decrease unnecessary interference between super- and subclasses. However, now that the different access modes exits, it would be more natural to regard the accessible members of a base class to be in the *same* scope as the members of the subclass itself.

In consequence, I suggest that the overriding of accessible non-virtual base class members should not be allowed in a derived class; it is not very useful but can cause treacherous errors. Especially the difference between virtually and non-virtually overridden member functions is quite subtle. The overloading of non-virtual member functions would of course not be forbidden if the types of the explicit arguments are different.

The previous suggestion is more or less tentative; the following one is more serious. The hiding rule has been extended (I suppose in Release 2.0) so that defining or redefining a member *function* in a class also hides any inherited, overloaded member functions with the same name. This is another paternalistic rule (cf. §3.1) that should be retracted. It tries to prevent some errors but causes much more nuisance to perfectly good programming.

Ellis & Stroustrup [1990] give two examples as a rationale for the rule (§13.1, p. 310–312). The first example is based on the thinking that a client of a derived class should not need to be aware of the public functions of public base classes. In my opinion, this is contrary to the very idea of public inheritance. A work-around for accessing a hidden inherited function is presented:

```
class X1 {
public:
      void f(int);
};

// chain of derivations X2 .. X8

class X9 : public X8 {
public:
      void f(double);
      void f(int i) { X8::f(i); }
};
```

Example 4.

Having to write a lot of such auxiliary functions can be an unnecessary pain in the neck for programmers. If the hiding rule were kept as the default, at least one should have some more convenient means to escape it, such as:

```
reveal void f (int);
reveal f;
```

The latter declaration would reveal all overloaded, inherited variants of `f`. Even better, the hiding rule should not be the default, but applicable by explicit `hide` declarations analogous to the above `reveal` declarations.

The second example of Ellis & Stroustrup [1990 p. 312] is based on an assignment operator:

```
struct B {
      void operator= (int i)
      ...
};
```

If this operator were not automatically hidden in subclasses of `B` by the default assignment operator (if nothing else), it would probably cause an incomplete assignment when applied to a subclass object. Here the hiding rule tries to protect against sloppy programming, but succeeds only half way: an invocation through a pointer of type `B*` will cause the "incomplete" operation to be performed anyway. The operator should absolutely be declared virtual in the first place and redefined in the appropriate derived classes.

The above kind of `reveal` declaration would also be a better way than the existing one [Ellis & Stroustrup 1990 §11.3] for restoring the original accessibility of selected members of private (or protected) base classes. A small but admitted defect of the current method is that overloaded member functions with the same name cannot be treated separately. To take the book's example:

```
class X {
private:
      f(int);
public:
      f();
};
class Y : private X {
public:
      X::f;    // error
};
```

Example 5.

There is no way to restore the accessibility of `X::f()` to `public` in `Y` because `X::f(int)` is `private`.

4. *Independent Multiple Inheritance*

4.1 *Problems in Current C++*

Simplifying things a little, we can say that the prime conceptual problem of multiple inheritance are *horizontal* name clashes, i.e. those between parallel superclasses (base classes). Independent MI is a simple and unproblematic case *in principle:* any such name clashes can be regarded as purely accidental and resolved by explicit class qualification.

Name clashes between *data members* indeed cause no big trouble in C++: one must just explicitly qualify the member name with the appropriate class name when referring to it in a derived class. Unfortunately, name clashes between *function members* cannot always be handled adequately. Let us consider an example from Stroustrup [1991 13.8]: the two totally unrelated classes `Window` and `Cowboy` both happen to have a function called draw, but their meanings are obviously quite different.

```
class Window {
      // ...
      virtual void draw();
};
class Cowboy {
      // ...
      virtual void draw();
};
class CowboyWindow : public Window, public Cowboy {
      // ...
};
```

Example 6.

There is a similar example in Snyder [1991 Fig. 8b]. Snyder sees here an anomaly in C++: that there is no way to denote either `Window`'s or `Cowboy`'s draw in the lexical context of `CowboyWindow`, in a way that would not suppress virtuality and would be resilient to a later evolution of the class hierarchy. Indeed, a simple class qualification (`Cowboy::draw` or `Window::draw`) may need to be changed if `draw` is later redefined in `CowboyWindow`, or if some new class redefining `draw` is interposed in the inheritance graph between `Cowboy` and `CowboyWindow` (or `Window` and `CowboyWindow`).

Actually there *is* a solution to this problem, by using "pointers to members"[8]—a difficult new feature which happens to be discussed in Snyder [1991] as well. Specifically, if `cw` is an instance of `Cowboy-Window`, one could use

```
(cw.*(&Cowboy::draw)) ()
```

to invoke the first function, and

```
(cw.*(&Window::draw)) ()
```

to invoke the second function. As Ellis & Stroustrup [1990, p. 157] confess, "the syntax isn't the most readable one can imagine", but it should work exactly as Snyder wanted. Even this solution does not work for *non-virtual* functions. However, if the suggestion of §3.7 to forbid nonvirtual overriding were accepted, ordinary class qualification would suffice for them.

This whole example would have been illegal C++ and therefore moot according to the rules of Stroustrup [1989 p. 379]: a horizontal name conflict between *virtual* functions was required to be resolved by a redefinition in the derived class. This rule has obviously been lifted, sensibly enough.

In my opinion, the real conceptual fault appears first when one would like to redefine one of the inherited `draw` functions. Namely, it is impossible to redefine `Cowboy::draw` and `Window::draw` separately in `CowboyWindow`: a redefinition will *unify* the functions. The situation is the same whether the functions are virtual or not. This is absolutely wrong if we assume that name clashes between independent superclasses really are accidental. In Example 6 it is impossible to imagine a function that could be a sensible common specialisation of the two draw functions. An analogous situation in Snyder [1991 Fig. 8a] is only called "challenging" there, in the sense of its modelling being non-obvious.

Thesis 4: The language must not force functions inherited from mutually unrelated ancestors to be unified in a common descendant class.

It is interesting to note that Ellis & Stroustrup [1990 §10.11c] recognise this problem, but do not seem to consider it important. They ac-

8. Again a term that I do not really like, because those are offsets rather than pointers.

tually write:

The semantics of this concept are simple, and the implementation is trivial; the problem seems to be to find a suitable syntax.

Syntax concerns seem a poor excuse for leaving the problem unsolved, especially as the syntax of C++ is not so wonderful anyway.

The anomaly of the above becomes still more obvious if we think of cases in which the two `draw` functions are *not* of exactly the same type. If they differ only in their return type, then it is not possible to redefine either of them in `CowboyWindow`, as far as I can infer from Ellis & Stroustrup [1990]! If they differ in the types of arguments, then they remain separate even in `C`, but if only one is redefined the other becomes hidden (§3.7).

4.2 Different solutions

Stroustrup [1991 §13.8] presents a work-around for redefining the `draw` functions in Example 6. `CowboyWindow` cannot be derived directly from `Cowboy` and `Window`, but auxiliary intermediate classes and functions are needed. This is Stroustrup's solution, with trivial type errors corrected:

```
// ...
class WWindow : public Window {
      virtual void win_draw() = 0;
      void draw() { win_draw(); }
};
class CCowboy : public Cowboy {
      virtual void cow_draw() = 0;
      void draw() { cow_draw(); }
};
class CowboyWindow : public Window, public Cowboy {
      // ...
      void win_draw();
      void cow_draw();
};
```

Example 7.

This method would work just as well even if the two `draw` functions had different result types. However, it looks a little awkward, adding complexity to the class structure. The awkwardness becomes worse if we suppose that the classes `WWindow` and `CCowboy` should be reusable, and that there may be more than one pair of colliding functions that could be redefined. In order not to require every derived class to redefine every function, win_draw and cow_draw should actually not be defined as *pure virtual* (= 0), but rather like this:

```
    virtual void win_draw() { Window::draw(); }
```

More importantly, programming tools such as class browsers probably cannot tell the programmer that in order to get a redefinition of Window::draw in classes derived from CowboyWindow, it is the function win_draw that must be redefined.

One way to avoid the problem of mixing up unrelated functions would be the mechanism of *"titles"* [Sakkinen 1990]. I had originally thought it out earlier, and noted when the "pointer to function member" concept was added to C++ that there was a strong similarity. A bit paradoxically, although families of virtual functions are not well defined in C++ (§3.5), pointers to them now exist. As Snyder [1991 p. 13] puts it:

Pointers to class function members correspond exactly to operations in our model of C++: such pointers cannot distinguish between individual methods for the same operation [...]

Simplifying to the most essential for this case, a title would have a meaning lying between a class qualification and a pointer to function member. To try some concrete syntax, A..f would mean "the most specific overriding of function A::f". The most important difference to *(&A::f) would be that this construct could be used also in redefinitions. In the above case, one would use just A..f if redefining the function lexically within the definition of a derived class C, and C::A..f outside class definitions.

Using the title could automatically take into account even non-virtual overridings (previous subsection). If all the suggestions in §3.5 were realised, titles could be needed for the opposite purpose: to disambiguate *vertical* name clashes between virtual functions already in single inheritance. Member function pointers are also adequate for that task, though, because the need would appear only in invocations, not definitions. Neither of these two uses would be relevant if non-virtual overriding of accessible members were forbidden (§3.7).

Finally, in spite of what was said above, in some cases one might *want* to unify two functions inherited from different superclasses. This could be done by equating their titles (possible syntax is left to the reader's imagination), even if the original names were different. Of course, the argument and result types of both functions should be identical.

Note that in Eiffel™ [Meyer 1988 §11.2], inherited *features* (Eiffel terminology, equivalent to 'members' in C++ terms) with identical names can be redefined separately:

```
class Cowboy feature draw ... end
class Window feature draw ... end
class CowboyWindow inherit
  Cowboy rename draw as cow_draw redefine cow_draw;
  Window rename draw as win_draw redefine win_draw;
... end
```

Example 8.

The suggestion for C++ mentioned in Ellis & Stroustrup [1990 §10.11c] is totally analogous to this. The advantage of my "title" solution is that it would not be necessary to invent new names for the overriding functions; although even in Examples 7 and 8, *both* were renamed only for the sake of symmetry.

4.3 Private Multiple Inheritance

The following section will present quite a lot of complications that are caused by the presence of private inheritance in C++, in contrast to most other object-oriented languages. Before that, let us have one more argument in favour of private inheritance.

As explained in Stroustrup [1991 §12.2.5], a typical simple use of IMI is to have one public base class and one private base class which serves as the implementation. The very first example of multiple inheritance in Meyer [1988 §10.4.1], already entitled "The marriage of convenience", is like this. If we disregard genericity, the example defines the class FIXED_STACK by inheriting both the *deferred* (Eiffel term for 'abstract') class STACK, which defines the interface, and the ordinary class ARRAY, which is used for the implementation. All features of STACK are exported by FIXED_STACK, corresponding to public derivation in C++, while no features of ARRAY are exported, corresponding to private derivation.

This example has often been frowned upon, but actually there is only one defect in it: in Eiffel nothing prevents an object of type FIXED_STACK from being assigned to a variable of type ARRAY, after which all ARRAY routines can be directly invoked. Here the private derivation of C++ has an advantage over Eiffel: in C++ it would not be possible for clients to implicitly convert a pointer to FIXED_STACK into a pointer to ARRAY. Unfortunately, an explicit cast is always possible; but explicit casts can cause even catastrophic effects in C++, so a wise programmer *writes* them only when necessary and reads them in existing code as warning signs.

5. *Fork-Join Inheritance*

5.1 The Positive Side

I tend to believe that many situations really demand FJI (with virtual base classes). On a conceptual level, such situations arise whenever we have several mutually independent classifications of the same domain. For instance: A vehicle is either a land, water, air, or amphibious vehicle; in another classification it is either private or public (not in the C++ sense!); in a third one it is powered by wind, man, animal, or engine. Any instance of a vehicle is nevertheless *one* vehicle, thus `vehicle` as a non-virtual base class would make no sense. This example is perhaps not absolutely convincing, as it might be modelled adequately without using inheritance at all.

An interesting programming style has been suggested by Paul Johnson [1990] as *fine grain inheritance,* although I have not seen examples clearly showing its advantages. Very briefly, it means that every class should define a minimal coherent set of features, and a typical, conventionally designed class should be broken into a number of smaller classes related by (multiple) inheritance. Fine grain inheritance obviously requires a high degree of FJI with virtual public base classes.

One very simple example of public fork-join inheritance is presented by Stroustrup [1991 §6.5.1]:

```
class link { ... };
class task: public link { ... };
class displayed: public link { ... };
class satellite: public task, public displayed { ...
};
```

Example 9.

With this definition there will be two separate `link` subobjects in each `satellite` object. If link were declared a virtual base class of `task` and `displayed`, there would be only one `link` subobject.

In Sakkinen [1989], I criticised Eiffel because, contrarily to C++, its rules would not guarantee the *integrity* of `link` subobjects in the above case. Depending on how the class `satellite` were defined, part of the components of `link` might well be shared and the rest duplicated. This happens on purpose in the "intercontinental drivers" example of Meyer [1988 §11.6.2]. The same danger seems to exist in several other languages that support MI.

Bertrand Meyer [1990] in turn has criticised the C++ principles on an issue slightly different from subobject integrity. According to his reasoning, there is no sense in having `task` and `displayed` decide about the sharing or duplication of `link`, since it does not affect them but only `satellite`. I continue to disagree with Meyer even on this point: it can be very important for the class task to know whether it has a `link` part to itself or is prepared to share it with any other, unknown class derived from `link`.

The question that I had forgotten to pose in admiring the MI principles of C++ [Sakkinen 1989] was: While C++ is in this respect more disciplined than e.g. Eiffel, does even the choice between virtual and non-virtual derivation independently of access mode, as allowed by C++, make sense conceptually and semantically? We will investigate this question, based in part on the analysis of Baclawski [1990].

In the following subsections, we suppose that a most derived class `C` is derived from several base classes by FJI. We examine how the subobject graph of `C` (§3.3) is related to its inheritance graph in various situations, and what restrictions are needed to guarantee the consistency and semantic feasibility of the inheritance structure.

5.2 Accessible Base Classes

Let us examine Example 9 further. The base class `link` is supposed to implement lists of objects: here a scheduler list of tasks and a display list. The derivations from `link` are therefore non-virtual, resulting in two distinct link subobjects in a satellite.

The inheritance in the example is public, which should imply transitive *is-a* relationships (§3.2). Since a satellite is a task and a task is a link, a satellite should also be a link. However, there are *two* distinct links in a satellite, thus one cannot say that a satellite *is-a* link. Indeed the rules of C++ will not allow us to convert a pointer of type satellite* directly to type link*, although such conversions are always possible in public *single* inheritance.

There will hopefully be no argument that one of the ubiquitous FJI examples in the literature of semantic databases is correctly modelled in C++ as follows.

```
class Person { ... };
class Student: public virtual Person { ... };
class Employee: public virtual Person { ... };
class StudentEmployee: public virtual Student, public
 virtual Employee { ... };
```

Example 10.

Of course the virtuality or non-virtuality of the immediate bases of StudentEmployee does not matter unless further classes are derived from it. The important thing to note is that the subobject structure from the viewpoint of StudentEmployee would not become different if its direct bases were made private. The ancestors would only become hidden from clients.

On these grounds I postulate a little imprecisely the following

Thesis 5: Accessible fork-join inheritance should always be virtual.

Note the more general qualification 'accessible' instead of 'public or protected'. In the pure case (i.e., no private derivations in the inheritance hierarchy) the thesis is sufficient and means that the subobject graph shall be isomorphic to the inheritance graph. In particular, an object shall contain exactly one subobject corresponding to each ancestor class. A precise formulation that is valid also for mixed derivations can be based on the definitions of §3.3: If class A is accessible to class B over more than one derivation path, an instance of B shall contain one complete subobject of class A that is common to all those derivation paths. (An instance can be either a complete object or a complete subobject.)

Now we have gotten into a conflict with Stroustrup's example, which looked perfectly natural at first sight. At least link simply cannot be a virtual base class there, according to the intended semantics.

The only way to both preserve the inheritance graph and conform to the rule is to make `link` a private base class (of at least one of its immediate descendants). In fact, that would be reasonable also because otherwise it would be much too easy e.g. to put tasks on the display list.

Pondering Example 9 a little more, we note that it is questionable to make `link` a base class at all, instead of a data member. If a task, for example, should happen to need more than one link, it would not be possible to use inheritance. Declaring data members of type `link` in `task` and `displayed` would cause different problems: there would be no way for the `link` objects to refer to the objects in which they are contained. A suggested new feature of C++ that was obviously designed especially for cases like this, is *template class* [Ellis & Stroustrup 1990 14; Stroustrup 1991 §8]. The class `link` could be declared as a template class taking a type name (here `task`, `displayed`) as its template argument.

One unconventional property of Eiffel is that *direct repeated inheritance* [Meyer 1988 §11.6.1] is allowed: one class may appear more than once in the immediate ancestor list of another class. This property is often regarded as suspicious, but the possibility of public FJI with nonvirtual bases conceptually contains it as a special case, except for the subobject integrity problem in Eiffel (5.1). To see that, simply imagine that the classes `task` and `displayed` in Example 9 declare no new members, but act simply as naming aids for the two `satellite` subobjects. Because of the renaming facility, such auxiliary classes are not necessary in Eiffel.

5.3 Inaccessible Base Classes

The special case of FJI where all derivations are either public or protected proved to be relatively simple in the previous subsection. The opposite pure case where all derivations are private is also simple. Private inheritance is intransitive, i.e. derived classes can access only their immediate bases. I postulate the counterpart to the thesis of the previous subsection, again as an imprecise slogan:

Thesis 6: Inaccessible fork-join inheritance should never be virtual.

In the pure case, this means that if A is an ancestor of B, an instance of class B shall contain a separate subobject of class A for each derivation path from B to A. In other words, the subobject graph shall be a tree. Obviously, totally private non-virtual FJI is no more complicated to understand and implement than private IMI. It would therefore need no very ambitious example of its utility in order to escape "Cargill's razor".

In the mixed case, Thesis 6 needs more adjustment than Thesis 5, because even a private path is accessible if its length is one. The complete formulation will be deferred to the following subsection but we refine the statement a bit here. As mentioned in §3.6, the virtuality of base classes is too strong or too global in current C++. What I want to achieve is that two separate subobjects cannot have a "hidden" common sub-subobject. Thus: If there is an inaccessible derivation path from class B to class A, the A subobject of an instance of B corresponding to that path shall be totally disjoint from all other A subobjects of that instance.

Combining the theses 5 and 6 we conclude that an explicit `virtual` declaration of a base class becomes superfluous. We could thus simplify C++ by omitting the `virtual` specifier for base classes. However, this holds only in principle; pragmatic needs will be suggested in §5.7.

5.4 Mixed Cases

How do we get from the inheritance graph of a class C to the corresponding subobject graph in the general case, in which virtual (public or protected) and non-virtual (private) derivations can be arbitrarily combined? The non-transitivity of private inheritance makes the problem difficult.

I present a general rule, however, which I believe to result in consistent and understandable object structures even in very large and complicated class hierarchies.

Rule: Let *P* and *Q* be two derivation paths from class B to class A, having no common nodes except the end points. The paths *P* and *Q* will correspond to the same complete A subobject in an instance of B if and only if both are accessible. Otherwise the paths will give rise to two disjoint complete subobjects.

One consequence of the rule is that no class can access more than one subobject corresponding to each ancestor class. Multiple class qualifications (like `B::A::x`) will therefore never be necessary in order to uniquely denote inherited members. Stroustrup [1989] noted that such multiple qualifications would be useful under the current C++ rules, but they have not been introduced into the language for this purpose. Instead, when the nesting of class definitions was made meaningful (causing nested scopes) in Release 2.1 [Ellis & Stroustrup §9.7], this syntax was employed to refer to such nested definitions from outside.

In current C++ there are no restrictions on the legal inheritance graph of a class, except acyclicity. There can be at most one edge directly connecting any two nodes (classes), but that is a normal requirement for proper graphs. It appears that with the new rules as presented so far we can manage without additional restrictions on the basic graph structure. However, we will need further restrictions on the labelling (access modes) to assure consistency. The previous subsections already effectively removed the virtuality labels.

A problem point in the rule is revealed by the following anomalous example.

```
class A {};
class B : public virtual A {};
class C : private B, public virtual A {};
class D : public virtual C {};
```

Example 11.

The classes A and C are accessible to class D; even according to Thesis 1, virtual functions of A can be redefined in D, and such redefinitions are effective also for the C subobject of an instance of D. Because B is accessible to C, the redefinitions propagate also to the B subobject. On the other hand, class B is inaccessible to D; by Thesis 1 D should not be able to affect the virtual functions effective for its B subobject.

The contradiction is solved by the following

Restriction 1: If a class C has both a transitively accessible and an intransitively accessible derivation path to the same ancestor class, no further classes must be derived from C.

5.5 Virtual Functions with Virtual Base Classes

The late binding of virtual functions, often called "method lookup" in object-oriented literature, becomes complicated in the fork-join case. Let us first consider the case with virtual base classes, since according to my suggestions late binding would be relevant only there.

The little difficulty that was mentioned in §3.6 can be illustrated by augmenting Example 10 a little. This is equivalent to the example in Ellis & Stroustrup [1990 p. 201–202]:

```
class Person {
public:
      virtual void earn();
      ...
};
class Student: public virtual Person {
public:
      virtual void earn();
      ...
};
class Employee: public virtual Person {
public:
      virtual void earn();
      ...
};
class StudentEmployee: public virtual Student,
  public virtual Employee {
public:
      virtual void earn();
      ...
};
```

Example 12.

Suppose that each version of earn has to call the inherited version(s) in addition to doing its "own job", as is very common. At least the classes Student and Employee must then define a separate, nonvirtual protected function, say own_earn, that does the "own job", and every virtual function must call all relevant non-virtual functions:

```
   void Student::earn()   {Person::earn(); own_earn();}
   void Employee::earn()   {Person::earn(); own_earn();}
```

```
void StudentEmpoyee:earn()   {
      Person::earn(); Student::own_earn();
      Employee::own_earn(); own_earn();
      }
```

For functions with a non-void result type (even in the example we would more naturally have `virtual int earn()`), there is the additional problem of how to combine the results of all the different functions.

There is a slightly erroneous rule in Ellis & Stroustrup [1990 p. 235]:

> *To avoid ambiguous function definitions, all redefinitions of a virtual function from a virtual base class must occur on a single path through the inheritance structure.*

The rule could actually make sense, but it does not seem to describe current C++ correctly. Taken literally, it would make even Example 12 illegal. The intent evidently was:

> *[...], if a virtual function from a virtual base class is redefined on more than one path through the inheritance structure, there must be one redefinition that dominates all others.*

An *advantage* of the unlimited scope of redefinability of virtual functions in current C++ is that there is always a class where such a dominating redefinition can be done if an ambiguity must be resolved: at least the most derived class. Obeying Thesis 1, there could well be cases in which there is no single most derived class in which the virtual functions of a given ancestor class can be redefined. That would happen in Example 12 if the base classes of `StudentEmployee` were `private`. To avoid that, we must make an explicit

Restriction 2: In the inheritance graph of a class C, for any ancestor class A that is accessible to several descendants, there must be one among them to which all others are accessible.

The principle of dominance leads to the "sideways" inheritance mentioned in §3.6. Suppose that the redefinitions of `earn` are removed from `Student` and `StudentEmployee` in Example 12. Calls of `earn` in an instance of `StudentEmployee` will then always be resolved to `Employee:earn`, even when they are issued from functions of `Stu-`

dent or by clients using a pointer of type `Student*`. Ellis & Stroustrup [1990 §10.10c] says:

> *A call to a virtual function through one path in an inheritance structure may result in the invocation of a function redefined on another path. This is an elegant way for a base class to act as a means of communication between sibling classes [...]*

Cargill [1991a §4] sees the disadvantages as more important (referring to a similar example):

> *To understand the behavior of* [`Student::earn()`] *we must examine the entire DAG reachable by traversing from any class derived from* [`Student`] *to any virtual base class of* [`Student`].

I must admit that both the advantages and the disadvantages of this method lookup scheme look important. However, since a *complete object* (§3.6) is primarily regarded as one single object, it seems logical that method lookup always begins this way, from the most derived class (i.e., the root of the complete object).

5.6 Virtual Functions with Non-Virtual Base Classes

It has appeared to me that the method lookup in current C++ is in some ways more problematic in FJI with *non-virtual* base classes. In fact, I have not succeeded to find this case explicitly described in Ellis & Stroustrup [1990], nor in other books and papers! It is in most respects like independent multiple inheritance. Note that this situation can occur only in existing C++, not with my new rules.

Consider Example 12, modified so that all derivations are non-virtual. Let us pretend that this could be sensible, ignoring the too obvious real-world situation that a student employee is only one person. If `earn` were not redefined in `Student` and `StudentEmployee`, no direct "sideways inheritance" as with virtual derivation could occur. On the other hand, any invocation of `StudentEmployee:earn` would be a compile-time error; either `Student::earn` or `Employee::earn` would have to be selected statically.

Suppose again from now on that the redefinitions of `earn` are there. With non-virtual derivation we do not need the non-virtual auxiliary functions in every class, and StudentEmployee;earn need not worry about Person::earn. The situation thus looks much simpler:

```
void Student::earn() {Person::earn();
  /* then they own stuff */ }
void Employee::earn() {Person::earn();
  /* then they own stuff */ }
void StudentEmployee::earn()
      {Student::earn(); Employee::earn();
  /* then they own stuff */ }
```

Of course, if the result type of earn were non-void, there would still be the problem of result combination, as in the virtual case.

However, think about the case that earn is invoked by a member function of Person. This call then comes from the Person sub-subobject of either the Student or the Employee subobject, but it will cause Person::earn to be invoked on *both* Person parts! This is much more insidious sideways inheritance than in virtual derivation.

The effects can get even more interesting. Suppose that Person::earn calls another virtual function work of Person, and work gets redefined similarly to earn. It is easy to see that one call of earn from Person will cause work to be invoked twice on both Person subobjects! We might call this anomaly *"the exponential yoyo problem"*—the explanation follows.

The suggestive name *"yoyo problem"* was coined by Taenzer et al. [1989] to describe the following situation, translated from the terminology of Objective-C® into that of C++: Whenever virtual functions invoke other virtual functions of the same object (*this), the method lookup starts from the most specific class of the actual instance and proceeds upward in the inheritance hierarchy until a definition is found. The flow of control can therefore oscillate arbitrarily up and down and be difficult to follow if the hierarchy is deep.

The yoyo problem was identified in an environment with single inheritance. It obviously becomes more complex with multiple inheritance, as explained in the quote from Cargill in §5.5. Our new problem is clearly similar to the yoyo problem; it is exponential in the sense that the number of invocations gets multiplied by the number of inheritance branches on each down-and-up trip. However, the new case is clearly erroneous, whereas the original yoyo problem means only difficulties in understanding and debugging software.

The exponential yoyo problem does not appear naturally with virtual base classes. To see that, let us return to Example 12, including all function redefinitions. Let there be another virtual function `work` in class `StudentEmployee`, which is again redefined in all the other classes just like `earn`. If `Person::earn` invokes `work`, the invocation will be late-bound to `StudentEmployee::work`. That in turn will cause `Person::work` and each `own_work` to be called exactly once; no surprises. In order to cause multiple invocations, at least one of the `own_earn` functions must be expressly written to call `work`.

5.7 Further Considerations

It was noted already in §3.6 that the liability of invoking the constructor for a virtual base class should not extend farther down in the inheritance graph than is logically necessary. Within the rules proposed so far in this paper, the following would be adequate: For every class `B` from which there are at least two *disjoint* accessible paths to an ancestor `A`, the accessible `A` subobject must be initialised directly by the constructor(s) of `B`; any initialisation of `A` specified by classes between `A` and `B` on the derivation paths will be ignored.

Alternatively, we could make this even simpler and specify that the initialisation caused by that immediate base class shall prevail which is mentioned first in the base list of `B`. This rule could be accompanied by a general ability of descendants to redefine the initialisation of any accessible non-immediate ancestor.

There are important exceptions to Thesis 5. First, there can sometimes be semantic reasons for public or protected inheritance with guaranteedly unshared subobjects. Second, the pragmatic viewpoint should not be ignored either. Presumably even in large inheritance graphs designed by programmers who know how to exploit the advantages of MI, there will be relatively few accessibly derived fork-join structures. Therefore one would not like to have the overhead of virtuality in all derivations.

Taking into account these factors and the tradition of the language, it is probably wisest to keep the `virtual` keyword but enhance its implications so that the rules prescribed in this paper will not be violated. Primarily, if class `A` is an immediate non-virtual base of `B`, it shall be illegal for any descendant class of `A` (including `B` itself!) to have an accessible path to `A` both over `B` and over some other immediate descendant. Even a `private virtual` declaration can then make sense, to assure that `B` remains a descendant of `A` independently of changes in its other inheritance relationships.

As a somewhat different situation, one might like to assure that a certain set of classes derived from a common abstract base class form a *taxonomy,* i.e. that they are both exhaustive and mutually exclusive on every derivation level. Baclawski [1990] says about the mutual exclusion of subclasses:

> *Such a constraint cannot normally be enforced by a programming language [...] It is curious that taxonomies are often cited as a motivation for inheritance, yet few systems offer the means of constraining a set of types to be a taxonomy.*

In the class dictionaries of the Demeter™ framework [Lieberherr et al. 1991], inheritance hierarchies are *forced* to be taxonomies.

A taxonomy does not completely prevent later fork-join inheritance, if multiple independent taxonomies of the same base class are allowed, as in the very rudimentary vehicle example of §5.1. The C++ Demeter system, contrarily to the Flavors Demeter system, at the time of writing does not allow such multiple taxonomies, nor FJI in general [Lieberherr 1991].

Frameworks such as Demeter that require the whole class structure to be defined before compilation may evidently cause additional overhead for the incremental addition of new derived classes. On the other hand, they can allow a much higher degree of *customisation* [Lea 90] and consequent run-time efficiency than more conventional ways of object-oriented software development. In particular, in such a framework it would not be necessary for programmers to specify non-virtual derivation for merely pragmatic reasons. Another advantage of Demeter is the automatic generation and propagation of member functions for classes. This could probably be extended to generate such typical virtual function patterns as presented in §5.5 for FJI.

6. *Summary and Conclusions*

The discussion that this article is continuing began from the question *whether* C++ should support multiple inheritance or not. I now think that there is sufficient evidence to answer "Yes" to this question, at least for *independent* multiple inheritance. Therefore it appeared more important for me to study *how* MI should work.

During this work it became apparent to me that *private* inheritance, rather a speciality of C++, is a major cause of complexity. It also became apparent that private inheritance is semantically such an important and powerful tool that the complexity should be tolerated.

I found that there are some subtle inconsistencies in the current inheritance principles of C++ that affect already single inheritance. The most important flaws could be summarised by saying that private inheritance is not private enough. Independent multiple inheritance exposes at least one further effect: the unpreventable unification of member functions because of accidental name equality. Fortunately, all these flaws can be corrected by a small modification to the language.

Fork-join multiple inheritance was known to be the really complex case: it had caused the distinction between virtual and non-virtual base classes that was one of the main targets of Cargill's critique. I believe having made plausible enough that, on conceptual grounds, public (more precisely: accessible) inheritance should implicitly be "virtual" and private (more precisely: inaccessible) inheritance "non-virtual", and the language could therefore be simplified. Implementation reasons may make this principle too expensive for public inheritance in the cases when it is not actually needed. However, I suggested some possibilities for optimisation.

The "exponential yoyo problem" is presented in §5.6 as a *reductio ad absurdum* to show that at least the combination of non-virtual (duplicating) FJI with unrestricted redefinability of virtual functions, allowed by the current C++ principles, is unsound.

Although the suggestions of this paper would make FJI simpler and more logical, it still remains a complicated thing for both implementors and users of the language. The main suggestions should be applicable to several other object-oriented languages as well. We still need good examples of FJI to convince Tom Cargill and many others that it is worth the trouble. A major reason for the lack of good published examples of MI is that such examples tend to be large and complicated.

Very similarly, most of the problems and defects in the inheritance principles of C++ only become apparent when one studies inheritance graphs essentially more complex than those in the textbooks and reference manuals. Such complex situations have obviously not been sufficiently considered when the language has been designed. I also suspect that the design of multiple inheritance as well as several other features has been excessively driven by implementation considerations. In my opinion C++ is not sufficiently object-oriented; I hope to expound that in Sakkinen [1992].

I am now more optimistic about multiple inheritance than ever: it is a good thing in principle, *and* it can be done right; it just is not so simple as many of us have often thought. The current rules of C++ must urgently be revised, although a programmer who is convinced about the theses and other suggestions of this article can realise part of them by programming discipline. This holds for Thesis 1, Thesis 2 in most cases and Thesis 5 in simple cases; thesis 4 can be achieved by Stroustrup's work-around. However, C++ programmers should for the time being probably avoid MI as far as possible in order to guard against later trouble. Complex combinations of public and private, virtual and non-virtual fork-join inheritance are especially dangerous.

Acknowledgements

This research has been funded by the Academy of Finland, project 1061120 (Object-oriented languages and methods).

Several persons have helped me in discussions and correspondence, and by sending literature that I would not have had available otherwise. These include at least Bjarne Stroustrup (in spite of everything!), Kenneth Baclawski, Douglas Lea, Peter Grogono, Juha Vihavainen, Timothy Budd, Ian Holland, Antero Taivalsaari, Karl Lieberherr, Daniel Edelson, Alan Snyder, Tom Cargill, Peter Salus, Jim Waldo, and Jorma Kyppö. The comments of the anonymous reviewers were helpful, and have been taken into account.

The discussions in some Usenet newsgroups have been very stimulating and useful. The most important groups have been, of course, ‘comp.lang.c++’ and ‘comp.std.c++’ (although I have not had the time to follow these regularly any more), as well as ‘comp.object’. The occasions to give a guest lecture largely about the theme of this paper both at the University of Tartu (Estonia) and at the University of Kuopio further helped me to clarify my thinking and expression.

References

AT&T C++ Translator Release 1.2 Addendum to the Release Notes, Murray Hill, NJ: AT&T Bell Laboratories, 1986.

K. Baclawski, The Structural Semantics of Inheritance, manuscript (submitted for publication), Boston, MA: Northeastern University, 1990.

K. Baclawski, private communication, 1991.

T. Cargill, Controversy: The Case Against Multiple Inheritance in C++, *Computing Systems,* 4(1): 69–82, Winter 1991a.

T. Cargill, private communication, 1991b.

M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual,* Reading, MA: Addison-Wesley, 1990.

K. E. Gorlen, An Object-Oriented Class Library for C++ Programs, *Software—Practice and Experience,* 17(8): 503–512, August 1987.

P. Johnson, Fine Grain Inheritance and the Sibling-Supertype Rule, manuscript, Great Baddow, England: GEC-Marconi, 1990.

D. Lea, Customization in C++, *Proceedings of the 1990 USENIX C++ Conference,* pages 301–314, 1990.

K. J. Lieberherr, private communication, 1991.

K. J. Lieberherr, P. Bergstein, and I. Silva-Lepe, From objects to classes: algorithms for optimal object-oriented design, *Software Engineering Journal,* 6(4): 205–228, July 1991.

M A. Linton and P. Calder, The Design and Implementation of InterViews, *USENIX C++ Workshop Proceedings and Additional Papers,* pages 256–268, 1987.

B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Schiefler, and A. Snyder, *CLU Reference Manual,* New York, NY: Springer-Verlag, 1981.

O. L. Madsen, Block Structure and Object Oriented Languages, *Research Directions in Object-Oriented Programming* (B. Shriver and P. Wegner, Eds.), pages 113–128, Cambridge, MA: MIT Press, 1987.

B. Meyer, *Object-Oriented Software Construction,* Hemel Hempstead, England: Prentice Hall, 1988.

B. Meyer, postings on Usenet (comp.lang.eiffel and comp.object), 1990.

M. Sakkinen, On the darker side of C++, *ECOOP '88 Proceedings* (S. Gjessing and K. Nygaard, Eds.), pages 162–176, Berlin and Heidelberg: Springer-Verlag, 1988.

M. Sakkinen, Disciplined inheritance, *ECOOP '89 Proceedings* (S. Cook, Ed.), pages 39–56, Cambridge, England: Cambridge University Press, 1989.

M. Sakkinen, Between classes and instances, aided by titles, manuscript, Jyväskylä, Finland: University of Jyväskylä, 1990.

M. Sakkinen, The darker side of C++ revisited, manuscript in preparation, 1992.

A. Snyder, Inheritance and the Development of Encapsulated Software Systems, *Research Directions in Object-Oriented Programming* (B. Shriver and P. Wegner, Eds.), pages 165–188, Cambridge, MA: MIT Press, 1987.

A. Snyder, Modeling the C++ Object Model: An Application of an Abstract Object Model, *ECOOP '91 Proceedings* (P. America, Ed.), pages 1–20, Berlin and Heidelberg: Springer-Verlag 1991.

B. Stroustrup, *The C++ Programming Language,* Reading, MA: Addison-Wesley, 1986.

B. Stroustrup, Multiple Inheritance for C++, *EUUG Spring '87 Conference Proceedings,* pages 189–207, 1987.

B. Stroustrup, Multiple Inheritance for C++, *Computing Systems,* 2(4): 367–395, Fall 1989.

B. Stroustrup, *The C++ Programming Language, Second Edition,* Reading, MA: Addison-Wesley, 1991a.

B. Stroustrup, private communication, 1991b.

D. Taenzer, M. Ganti, and S. Podar, Problems in Object-Oriented Software Reuse, *ECOOP '89 Proceedings* (S. Cook, Ed.), pages 25–38, Cambridge, England: Cambridge University Press, 1989.

J. Waldo, The Case for Multiple Inheritance in C++, *Computing Systems,* 4(2): 157ff., Spring 1991a.

J. Waldo, private communication, 1991b.

[submitted Oct. 21, 1991; revised Dec. 6, 1991; accepted Dec. 23, 1991]