Controversy: The Case Against Multiple Inheritance in C++

T.A. Cargill Consultant

ABSTRACT: Multiple inheritance (MI) is now part of C++. MI greatly complicates the language, burdening those who learn, write and read C++. The costs would be justified if MI enriched the language, making it easier to express programs. But the literature contains no convincing examples of MI solving programming problems. Still, the ANSI X3J16 standards committee for C++ has embraced MI by adopting Ellis & Stroustrup [1990] as a base document. Before imposing a standard that includes MI, the committee should justify the costs by publishing realistic sample programs that demonstrate a compelling need for MI.

1. Introduction

Originally C++ had single inheritance (SI): a derived class could have only a single base class, and a class inheritance hierarchy was a tree. Since the release of AT&T's C++ 2.0 (and compatible implementa-

tions) C++ has included multiple inheritance (MI): the number of base classes is unlimited, and in general, a class inheritance hierarchy is a directed acyclic graph (DAG).

Under MI, derived classes inherit all the members of all their base classes. The potential ambiguity arising from an identifier's use in more than one base class is resolved at compile time, usually by giving fully qualified names. A more subtle problem arises when an ancestor base class can be reached by more than one path through the DAG: should there be a unique shared instance of the base class or a distinct copy along each path? This issue is addressed by an additional inheritance mechanism: the virtual base class. Virtual ancestor base classes are shared; non-virtual bases are distinct. The semantics of MI in C++ and its possible implementations are discussed in depth in Stroustrup [1989b] and Ellis & Stroustrup [1990].

2. Programming Language Design

The primary purpose of programming languages is the writing of computer programs. Improvements in programming languages over the years have enabled us to program more effectively. Progress is made by identifying shortcomings in existing languages and experimenting with new languages or new language features. In this process, the perceived benefits of language features range from ease of learning for novice programmers to support for veterans maintaining aging code. Every feature added to a language incurs costs for programmers and implementors. The costs of each feature are weighed against its benefits, such as increased expressive power, safety or efficiency. Of course, the design process must consider not only each feature in isolation, but also the interactions between features.

In selecting language features, one simple criterion can be applied universally: it should be possible to use each feature to write practical programs that are improvements in some respect over the corresponding programs expressed without the feature. Programming language features should be useful for writing computer programs. I wish to apply this criterion to multiple inheritance in C++.

3. *The Popular Perception*

The popular perception is that MI is a valuable addition to C++. In books, magazines, and advertising copy, MI in C++ is described favorably, with phrases like

> learn how to create powerful hybrid
> classes using multiple inheritance

> a significant enhancement to the
> support of object-oriented programming

> one of the most important
> and fundamental changes to C++

[Weston 1990; Lippman 1990; Eckel 1989] Unfortunately, those offering such opinions on the virtues of MI have not offered convincing evidence of those virtues.

4. *The Costs of Multiple Inheritance*

Multiple inheritance in C++ is complicated. It is complicated to learn, write and read. Each of these contributes to the cost of using C++.

With only one inheritance mechanism C++ would be a complex language. But a newcomer today is faced with 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. Yet we must learn the complexity of all six variants' interactions with other language features, such as initialization, virtual functions, overloading and conversions.

The evidence that MI is difficult to program is found in most of the published attempts to demonstrate its merits. Wiener and Pinson set out to exhibit a practical use of MI, but create an incorrect program that happens to produce the intended output on some hardware [Wiener & Pinson 1989]. Most textbook examples of MI are correct

programs, but merely disguise aggregation as inheritance, for example [Dewhurst & Stark 1989; Pohl 1989; Stevens 1990]. It is unrealistic to believe that programmers at large will be more successful in using MI than the authors of text books.

Programs that use MI are hard to understand. For example, Shopiro describes the use of MI in the Iostream library [Shopiro 1989]. Shopiro's code, about one hundred source lines, has essentially the same architecture as Iostream, simplified to reveal its use of MI. I have encountered several programmers who attempted to read the paper. Their talents are probably above average in that they take the trouble to read *SIGPLAN Notices*. None managed to understand the code. In my own case perseverance paid off on the third reading. This measure may be unfair; more evidence, and of a more objective nature, would be welcome. However, I doubt that anyone would argue that MI is easy to read and comprehend. Skeptical readers should examine Shopiro [1989] for themselves.

To illustrate the language complexity of MI in general, and virtual base classes in particular, consider the following small program, adapted from Stroustrup [1989b].

```
class Top {
public:
virtual void f() { printf("Top::f()"); }
};

class Left : public virtual Top {
public:
        void g() { f(); }
};

class Right : public virtual Top {
public:
        void f() { printf("Right::f()"); }
};

class Bottom : public Left, public Right {
};

main()
{
```

```
        Bottom x;

        x.g();
        return 0;
    }
```

Class |Bottom| inherits from both |Left| and |Right|, which have a common virtual base class, |Top|, as shown in Figure 1. The virtual function |Top::f()| is redefined by |Right|. In |main()| the function |Left::g()| is invoked on |x|, an instance of |Bottom|. When |f()| is called from within |Left::g()|, should |Top::f()| or |Right::f()| be invoked?

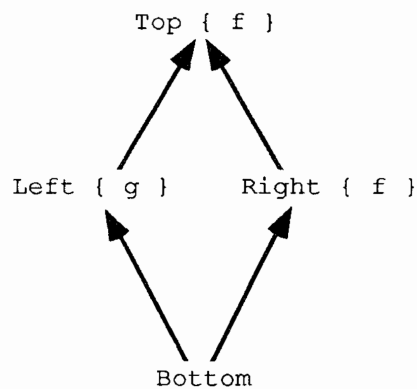

Figure 1: Inheritance hierarchy

The answer is |Right::f()|!!! (Triple exclamation marks appear in an earlier treatment of this phenomenon [Stroustrup 1989a], though not in its successor [Stroustrup 1989b].) If virtual base classes are used, determining which virtual function is called is much more complicated than under SI, where one need examine only classes that are reached by traversing up-paths and down-paths in the inheritance tree from the calling context. To understand the behavior of |Left::g()| we must examine the entire DAG reachable by traversing from any class derived from |Left| to any virtual base class of |Left|.

This example illustrates the complexity of part of the MI language mechanism in isolation. The complexity compounds when interactions

with other language features, such as initialization and conversion, are considered. We must weigh the complexity and costs of MI in C++ against its benefits in terms of its assistance in writing programs. The benefits should be manifest in examples of C++ programs that use MI.

5. *Examples of MI in the Literature*

Most of the published examples of MI in C++ are just aggregation disguised as inheritance. Aggregation can be expressed more directly and naturally using the language mechanism intended for that purpose. It is better to express simple aggregation by embedding member objects than by pretending it is the richer relationship of inheritance.

Pohl [1989] contains a typical example.

```
class tools {
public:
    int     cost();
// ...
};

class labor {
public:
    int     cost();
// ...
};

class parts {
public:
    int     cost();
// ...
};

class plans : public tools, public parts, public
  labor {
public:
    int     tot_cost() { return (parts::cost()
      + labor::cost()); }
// ...
};
```

Without detriment, class |plans| can be re-expressed without MI—indeed without any inheritance—as follows.

```
class plans {
public:
    tools   t;
    labor   l;
    parts   p;
    int     tot_cost() { return (p.cost()
     + l.cost()); }
// ...
};
```

This example is representative of most attempts to demonstrate MI for the benefit of novice C++ programmers. Some similar examples are rewritten in Cargill [1990a].

Gorlen et al. [1990] use MI twice. Their first example (p. 295) uses inheritance to open the scope of classes serving as modules (that is, all members are static). The code has the following form.

```
class A {
// static data
public:
static void f();
static void g();
// ...
};

class B {
// static data
public:
static void f();
// ...
};

class M : public A, public B {
public:
    void    h();
};

void M::h()
{
//  ...
    B::f(); // just f() would be ambiguous
    g();    // unambiguously A::g()
//  ...
}
```

Because class `M` inherits from both `A` and `B`, member functions of `A` and `B` can be called from `M`'s scope without name qualification, provided there is no ambiguity. Here MI is used to open other scopes and let the programmer write `g()` rather than `A::g()`. However, in general there is no saving. As Gorlen et al. themselves point out, when ambiguity arises scope resolution must be used to specify the function explicitly, e.g. `B::f()`. The call to `g()` is unambiguous only until a `g()` is added to `B`. It would be safer to write `A::g()` in the first place, making the inheritance redundant.

Gorlen et al. also use MI to construct a class whose objects can be placed on two linked lists (p. 297), in the fashion described in Ellis & Stroustrup [1990, p. 199]. In the member functions of the derived class casts are used to select one of two auxiliary linked-list base classes. Ignoring the risks of using casts unnecessarily, such casts are equivalent to selection within an aggregate. A simpler equivalent class may be created using SI and aggregation, as shown in Cargill [1990b].

Weston uses MI to add an `ostream` (for debugging) to a text edit window [Weston 1990, p. 270]. The `ostream` could equally well be incorporated by aggregation as a member object. Indeed, using aggregation simplifies the cumbersome initialization code in the MI version.

As mentioned above, Shopiro describes the use of MI in Iostream [Shopiro 1989]. This is the most convincing example of MI published to date. An input-output (I/O) stream inherits from both an input and an output stream, each of which in turn inherits from a buffer management virtual base class. The weakness is that the I/O duality vanishes as soon as an operation is applied to an I/O stream. Any operation has the effect of selecting the input or the output half of the I/O stream. An alternative architecture that uses single inheritance and explicit aggregation is described in Cargill [1990a]. Moreover, the version of Iostream in the (December 1990) draft X3J16 library does not use MI [Schwarz 1990].

In summary, every published example of MI in C++ can be written just as easily without MI.

6. General Transformations

That all published MI programs can be transformed into equally simple SI programs might tempt one to look for a general procedure for eliminating MI. Strictly, such a procedure is trivial, but provides no insight. Since Cfront converts C++ to C, it could easily translate any C++ program into C++ without inheritance. But, as with the general algorithm for replacing **goto** by **while**, this approach would make no attempt to conserve structure. It therefore has no bearing on the question of writing real programs with or without MI.

In all but one of the examples above MI is eliminated by replacing inheritance with aggregation. But a general MI-eliminating transformation would have to do more than merely replace inheritance with aggregation. The MI in the following program schema cannot be converted to aggregation without serious distortion.

```
class B1 {
public:
virtual void f1();
};

class B2 {
public:
virtual void f2();
};

class M : public B1, public B2 {
public:
        void f1();
        void f2();
};

void M::f1()
{
    ...
    f2();
    ...
}
```

```
void M::f2()
{
    ...
    f1();
    ...
}
```

In this schema the virtual functions `M::f1()` and `M::f2()` are redefined such that they mutually depend on each other. This schema cannot easily be expressed using only aggregation and single inheritance. The schema demonstrates the expressive power of MI, but only in a hypothetical setting. If MI were widely used in this manner in real programs, my thesis would collapse. However, this style of MI is not found in any of the published examples.

7. MI in Other Languages

A popular thesis is that MI has proven its worth in other languages and therefore C++ was incomplete without it. Such feature-by-feature reasoning between languages is spurious. The effectiveness of features depends on how they interact with other features as much as on their intrinsic properties.

Examples of MI in Eiffel [Meyer 1988] do not map directly into C++ because of other features in Eiffel. The scope system of Eiffel means that inheritance is needed for access to the equivalent of a public static member in C++. But, as discussed above, C++'s regular scope resolution is a simpler solution than inheritance. MI is also used in Eiffel in combination with generic classes. A proposal for adding generic classes to C++ ("parameterized types") has been adopted by the ANSI committee. At present, the interaction between MI and the forthcoming generics in C++ is no more than the subject of speculation. (Of course, I question the wisdom of standardizing extensions before better understanding the basis.)

MI is used in CLOS [Bobrow et al. 1988] for creating "mix-in" classes. However, the type system and binding mechanisms of CLOS are radically different from those of C++. The run-time method selection algorithm of CLOS is comparable with the compile-time overloading resolution algorithm of C++. Ellis and Stroustrup observe that mix-ins do not work in C++ [Ellis & Stroustrup 1990, p. 202].

Curry and Ayers [1984] discuss the use MI in the Traits language for the Xerox Star workstation. Their conclusions about MI are mixed. MI was used many times, but not to much advantage:

> Although [. . .] many of the classes used multiple subclassing, very few used multiple subclassing in an intrinsic or unavoidable way; minor rearrangements of the traits graph, not costly either in program logic or in class data space, could eliminate the multiple inheritance.

A comprehensive study of the use of MI in other languages, with attempts to map programs into C++, might yield further insight into the interaction between MI and the type and scope mechanisms of C++.

8. Virtual Base Classes

Inheritance from multiple independent class libraries is often cited as the case for MI in C++. C++ can be significantly simplified by the elimination of virtual base classes and still be sufficient for independent multiple inheritance. Virtual base classes are only meaningful when the different paths through an inheritance DAG are designed as a whole. Virtual bases cannot be exploited when multiply inheriting from independently developed libraries.

There is therefore perhaps a technical middle ground to explore. We may discover that MI is indeed useful, but that virtual base classes are unnecessary. For example, MI is used in the C++ Booch Components [Booch & Vilot 1990]. Few details have been published, but it appears to make no use of virtual base classes. Most of the complexity of MI in C++ is due to virtual bases. Determining that the benefits of virtual bases do not justify their costs would be progress, even if the larger question of MI in general were left unresolved.

9. Programming Language Research

Critics may observe that I was among the first to call for MI in C++ [Cargill 1986]. In that paper I was calling for an opportunity to conduct research on the question of MI in C++. I believe that research in programming languages must involve programming experiments. Aesthetically elegant ideas do not always work well in practice. For exam-

ple, Algol 60's call-by-name was mathematically sound, but impractical, and is not found in modern programming languages. When I called for investigation of MI in C++ my concluding remark on the subject was:

> Of course, the success of multiple inheritance cannot be guaranteed without practical experience, but it is certainly worth pursuing.

At that time I hoped that MI would appear in an experimental implementation, where programmers could explore and evaluate the idea, but that did not happen. The first working implementations were closely followed by compiler products and the ANSI committee's adoption of MI in a base document. We must question the diligence of the C++ technical community in conducting our research.

10. Conclusion

The evidence to date is that multiple inheritance is not useful in writing C++ programs. It should not become part of the ANSI C++ standard before convincing examples of its use are published. If multiple inheritance is a mistake, programmers will pay the price of using an unnecessarily complicated language for years to come.

Acknowledgements

My thanks to Carol Meier and Rob Pike for their suggestions about this paper.

References

D. G. Bobrow et al., *The Common Lisp Object System,* Technical Report 88P002R, X3J13 Committee, ANSI, 1988.

G. Booch, M. Vilot, The Design of the Booch Components, *Proceedings OOPSLA/ECOOP 90, Sigplan Notices,* 25(10), October 1990.

T. A. Cargill, Pi: A Case Study in Object-Oriented Programming, *Proceedings OOPSLA 86, Sigplan Notices,* 21(11), November 1986.

T. A. Cargill, Does C++ Really Need Multiple Inheritance?, *Proceedings of the USENIX C++ Conference, San Francisco,* April 1990a.

T. A. Cargill, We Must Debate Multiple Inheritance, *C++ Journal,* 1(2), Fall 1990b.

G. A. Curry, R.A. Ayers, Experience with Traits in the Xerox Star Workstation, *IEEE Transactions on Software Engineering,* SE-10(5), September 1984.

S. C. Dewhurst, K. T. Stark, *Programming in C++,* Prentice Hall, 1989.

B. Eckel, *Using C++,* Osborne McGraw-Hill 1989.

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

K. E. Gorlen, S. M. Orlow, P. S. Plexico, *Data Abstraction and Object-Oriented Programming in C++,* Wiley, 1990.

S. B. Lippman, C++: How Release 2.0 Differs from Release 1.2, *The C++ Journal,* 1(1), Summer 1990.

B. Meyer, *Object-Oriented Software Construction,* Prentice-Hall, 1988.

I. Pohl, *C++ for C Programmers,* Benjamin Cummings, 1989.

J. S. Schwarz, Personal Communication, December 1990.

J. E. Shopiro, An Example of Multiple Inheritance in C++: A Model of the Iostream Library, *Sigplan Notices,* 24(12), December 1989.

A. Stevens, *Teach Yourself C++,* MIS Press, 1990.

B. Stroustrup, The Evolution of C++: 1985 to 1989, *Computing Systems,* 2(3), Summer 1989a.

B. Stroustrup, Multiple Inheritance for C++, *Computing Systems,* 2(4), Fall 1989b.

D. Weston, *Elements of C++ Macintosh Programming,* Addison-Wesley, 1990.

R. S. Wiener, L. J. Pinson, A Practical Example of Multiple Inheritance in C++, *Sigplan Notices,* 24(9), September 1989.

[Submitted Jan. 8, 1991; revised Feb. 1, 1991; accepted Feb. 4,1991]