Controversy: The Case For Multiple Inheritance in C++

Jim Waldo, Hewlett-Packard Co.

ABSTRACT: Multiple inheritance is a difficult and complex feature added to C++ at release 2.0. Cargill argues that the addition was a step backward in that the feature adds complexity to the language without adding functionality. His basis for the latter half of this claim is that no example of multiple inheritance has been given which cannot be rewritten into a functionally equivalent example which uses single inheritance and aggregation instead of multiple inheritance.

I examine Cargill's arguments, and then sketch an example which uses multiple inheritance but which cannot be rewritten to be functionally equivalent by using single inheritance and aggregation. I then distinguish among three forms of inheritance, and argue that most attempts to give an example of multiple inheritance have attempted to use the one form which is the least likely to need that feature.

0. Introduction

Cargill has argued [1991] that multiple inheritance in C++ complicates the language without adding anything that can be shown to be of use. He backs up his claim by showing that all existing examples of the use of multiple inheritance in the current literature can be executed just as well without using multiple inheritance. His conclusion is that

> *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...*

This is not the first time that Cargill has argued that multiple inheritance is not a useful addition to the C++ programming language (other examples of the argument can be found in Cargill 1990a and Cargill 1990b).

The purpose of this article is threefold. First, I will attempt to clarify what the argument is by restating what I take Cargill to be saying. I will then sketch an example of a use of multiple inheritance that I believe does show that the feature is needed in the language to solve a certain set of problems. Finally, I will try to explain why convincing examples of multiple inheritance have not appeared before.

1. Cargill's Argument

Before attempting to sketch an example of a use of multiple inheritance in C++ that could not be programmed in some other way, I think it would be useful to look at just exactly what Cargill's argument is. The main argument is basically laid out as having three premises that lead to the conclusion quoted above. These premises are

1. Multiple inheritance in C++ is complicated. It is complicated to learn, write, and read;

2. Multiple inheritance is not (strictly) needed in C++, i.e., there is nothing that can be done with the feature that cannot be done without it;
3. If a feature is complicated and not needed in a language, it should not be a part of that language;

Therefore

4. Multiple inheritance should be removed from C++.

The first of these is stated overtly and is backed up in a section of Cargill's paper. The second is not stated but is the obvious conclusion of the other sections of Cargill's paper. The third premise of the argument is a general principle that is never overtly stated but is clearly argued for in the section of Cargill's paper on Programming Language Design.

Certainly, the main argument is valid; that is, if (1) through (3) are true, we must accept the conclusion. While the position that multiple inheritance is complicated to learn, write, and read could be disputed, I believe that if that were the only weakness of the argument I would probably side with Cargill. I will, therefore, accept (1) as true, and turn my attention to (2) and (3).

The second premise is itself the conclusion of an argument that has the form

5. No one has given a good reason for multiple inheritance in C++;
6. If no one has shown a reason to include a feature in a programming language, then that feature is not needed by the language;

Therefore,

7. Multiple inheritance is not needed in C++;

Statement (5) is proved by exhaustion. Cargill goes through the various examples and arguments given for including multiple inheritance in C++ and shows that the arguments are either flawed or the examples could be accomplished more easily using single inheritance and aggregation. (6) is again not explicitly stated, but seems to be one allied with (3) as a general principle of programming language design that Cargill tries to express in his section on that activity.

Having misspent my youth as a logician, I feel compelled to point out that the arguments that Cargill is putting forward are perilously

close to having the form of the *ad ignoratium,* or appeal to ignorance, fallacy, in which one concludes that a proposition is false because no one has shown it to be true. However, I take Cargill to be making a more interesting claim. Rather than claiming that multiple inheritance is not and never could be useful in writing a C++ program, he has made the weaker claim that no one has shown it to be useful. The interesting part of his claim is that he moves from that proposition to the claim that the feature should not be part of the language.

The real unstated principle behind Cargill's attacks on multiple inheritance in C++ appears to be a sort of Occam's razor applied to programming languages. Occam's razor, applied to metaphysics, states that one should not multiply entities needlessly, and is generally interpreted as holding that simpler systems are to be preferred to more complex ones. Applied to programming languages, this interpretation would hold that less is more, and that languages should only be made more complex if the added complexity allows one to do something with the language that could not be done without the feature being added.

While this principle sounds like one that we would all accept without question, I believe that it is far less absolute than most of us would like to admit. For example, I have seen introductory programming students struggle with the **for** loop construction in C. The construction is actually quite complex, having non-trivial entry and exit conditions, variable scoping conditions, and genuinely complex rules for determining the state of variables when the loop is terminated in some irregular fashion. Further, anything that can be done with a **for** loop can be done with a **while** loop. Therefore, by the principle of simplicity that I am ascribing to Cargill, the **for** loop is a genuine candidate for being banished from the ANSI C++ standard for exactly the same reasons as multiple inheritance. I take this to be an argument that is more damaging to the principle stated in (6) than to the **for** loop construction.

Whether Cargill's argument carries the logical weight it should, the fact remains that it is somewhat puzzling that no one seems to be able to give an example that uses multiple inheritance in C++ that is both convincing and understandable. One reason for this might well be that multiple inheritance is not the sort of thing that lends itself to examples. The real use of multiple inheritance is found in large systems, not small examples.

However, in the next section I will attempt to sketch an example that does require the use of multiple inheritance. The example will not be a full, compilable chunk of C++ but rather a part of a much larger system. It will also appear to be a rather odd use of inheritance, having little to do with the inheritance most C++ programmers are used to. The reasons for this will be discussed in the last section of this paper.

2. *An Example of Multiple Inheritance*

Before laying out some class definitions, let me begin by giving a brief explanation of the purpose of the example. In what follows, I will sketch a system with three sorts of objects. One is the usual C++ object, which exists when it is created and ceases to exist when it is freed (if it was created by a call to new) or when it goes out of scope (if it was created by entry into a scope).

The second sort of object is like the first but exists from one run of a program to another; it is a persistent object. In this design, persistence is treated as a type.[1] The notion of persistence being talked about is not terribly sophisticated, only allowing the saving of state into some entity in a file system. Indeed, we will not really talk about how the persistence works, appealing to magic in the details of the actual mechanism for saving and restoring the state.

The third sort of object can be thought of as a remote pointer to the first or second sort of object; it is what I will refer to as a surrogate for an object that exists in a different address space (perhaps on a different machine). The purpose of this object is to cloak the difference between local and remote objects. In the discussion of remoteness, I will assume that there is some underlying RPC mechanism that allows the sending of a message to an object in a different address space. While this is an interesting subject, it is not the subject of this paper, and so the details of how this is done will not be discussed here.[2]

1. I realize that this is not the only way to deal with persistence. Others, such as Atwood [private discussion], argue that the best approach to persistence is to consider it a storage class, analogous to **automatic.** This debate is only slightly relevant to the current debate, and would require a change in the language. The example outlined above may take the wrong approach, but can be implemented within the current definition of the language.
2. This is not a particularly novel approach to remote objects. See, for example, [Tiemann 1988], [Seliger 1990], or [Martin 1991].

The purpose of this system is to isolate the programmer from having to distinguish between purely local objects, persistent objects, and remote objects. Such isolation is not always possible; there will be times that the programmer will need to know what kind of object is being manipulated. But the overall goal is to allow the same code to manipulate all three.

We start, then, with a class definition for a persistent object, which will look something like the following:

```
class persist
{
 protected:
  char            *where,
  .
  .
  .
 public:
  virtual save() = 0;
  virtual restore() = 0;
}
```

This is, admittedly, a rather odd class definition, consisting of a string (which we can assume is the name of the file in which the persistent state is stored) and two virtual functions, which are not implemented at this level. There might be other data (offsets into the persistent file, whatever) that are used to help in the implementation, but exactly what such data would be is not of interest to this example. This class doesn't look particularly useful, and indeed in its current state it is not. But bear with me.

Equally useless looking is the second class I will define, which is the base class for surrogates to remote objects:

```
class remote
{
 protected:
  objid_t         object,
  .
  .
  .
};
```

This class contains an **objid_t,** which is used to locate and identify the object for which this is a surrogate. The exact nature of this type

would depend on the underlying RPC system used. Note that the class contains no public data or function members. By itself, this is not a very interesting class.

To get something that is of some interest, we need a class that does something. For the purpose of this example, even this class will not be very interesting—I will use the old standard of an employee class. Employees will be characterized by an employee name and an employee number. Again, remember that all I am trying to do is to sketch an example of multiple inheritance, not an interesting or useful example.

While the employee class may not be very interesting, the way I get to the definition of the class is not the standard way. I begin by defining what I will call the employee interface class, which looks like:

```
class employee_if
{
  public:
   virtual
   char*
         get_name() = 0;
   virtual
   void
         set_name(
         char    *new_name
         ) = 0;
   virtual
   int
         get_num() = 0;
   virtual
   void
         set_num(
         int         new_num
         ) = 0;
};
```

The reason for calling this the employee interface class is now clear. All this class does is define the set of calls that can be made on classes that derive from it—it establishes an interface between the rest of the world and objects that are of this class. However, there is no data associated with instances of this class, and there is no implementation of any of the functions in the interface supplied by this class.

Now I can define some classes that actually do some work. As might be expected, I will define three classes of employee objects. The

first will be just like a standard C++ object, in that it will be within a particular address space and will not persist beyond the time the program is running. I will call this class the temporary employee (in all senses of the term):

```
class employee_t :
   public employee_if
{
   char    *name;
   int       number;
  public:
   employee_if();
   ~employee_if();
 };
```

This class contains only two functions in addition to those that were declared in the **employee_if** class; a constructor and a destructor for the instances of the class[3]. The implementation of the class will, of course, have to supply code for all of the virtual functions in the **employee_if** class.

The second class we will define is our first use of multiple inheritance. This class defines the set of persistent instances of employees. These are instances that can be written to long term storage and read back from that store. The idea for such objects is that they are only restored when needed; if they are never accessed they are never restored. Such a class would be defined as

```
class employee_persist :
        public employee_if,
        public persist
{
  employee_t        *local_inst;
 public:
  employee_t(
       char      *in_store
       );
  ~employee_t();
};
```

3. Of course, if this were a real chunk of code rather than an example, there would be other functions declared, such as a copy constructor and assignment function. While such calls would be needed, they are not needed to show the requirement for multiple inheritance. Making this into a genuinely useful C++ class is an exercise left to the reader.

The persistent employee class contains all of the functions in the employee interface class and the **save** and **restore** functions of the persistent class. It also contains its own constructor and destructor. Implementations of all of these functions will need to be supplied; no implementation is or can be inherited.

The idea behind the **employee_persist** class is that the implementation of all of the functions inherited from the **employee_if** class will first look at the **local_inst** pointer. If that pointer is non-null, no restoration from persistent store is needed, and the function simply calls the same function in the local instance. If the pointer is null, the restore function is called. This function will bring in the employee data from the appropriate file and construct a local instance, with a pointer to that instance put into the **local_inst** private data member. When this restoration is complete, the implementation of the function on the local instance is called.

The destructor for the persistent version of the employee should call the **save** function that will pickle the current state of the **local_inst** and save it to persistent storage. Other functions might also call the **save** function; for example, if the class is being used in a situation where it would be bad to lose any changes, the **save** function could be called as part of the implementation of the set functions that change the data.

The final part of the example constructs a class of employees that live, potentially, outside of the address space of the current application (and perhaps on another machine). The idea for this class is much like that found in the persistent class; the class structure looks like

```
class employee_remote :
  public employee_if,
  public remote
{
 public:
  employee_remote(
       objid_t   far_emp
       );
  ~employee_remote();
};
```

Note that this class adds no new functions to the employee interface class other than a constructor and a destructor (like the local employee case) and adds no new data members.

The virtual functions inherited from the **employee_if** class would be implemented in the **employee_remote** class as RPC calls to some actual employee object. That object would receive the RPC call would be determined by the object identifier that is the only data member of instances of this class, that was assigned as part of the constructor for the object.

One of the ways in which this example differs from others given of multiple inheritance in C++ is that none of the derived classes listed above inherit any code from a base class. Indeed, the functions in the base classes are all explicitly defined as empty.

This is not an artifact of the example being merely sketched. No code can be given to implement the persistent base class, as the two functions that constitute that class are dependent on the data that is added by the derivation. There are no functions to implement in the **remote** class; that class simply provides a way of storing the data needed to give the underlying RPC a handle to contact a remote object. The **employee_if** class implements no functions, since that class has no data associated with it.

Sharing code among these classes was not the reason for constructing this particular class hierarchy. The reason for this hierarchy is to share code that *manipulates* instances of these classes. With the above hierarchy, for example, I can write a single **print_employee_name** function, that will work on all three kinds of employee objects: local, persistent, and remote. In the same way, we can pass lists of pointers to **employee_if** objects to functions that sort by name or employee number and then list the names of the employees in that order. In short, we have provided a way of writing code that handles three kinds of employees at the same time when the differences between those employees are hidden (as it should be) by the encapsulation offered by the **employee_if** abstraction.

In the same way, I could write routines that work on all persistent objects, whether they be employee objects or, say, inventory objects. These could be ways to force save the objects, or ways or doing batch restores. The main point is that whatever else these objects are, there are certain functions that require only that they be persistent. For those functions, being derived from the **persist** class is enough to allow the object to be passed into and be manipulated by them.

It is this reuse of code that manipulates the objects that keeps the example from being reducible by the Cargillian technique of aggrega-

tion to an equivalent example that uses single inheritance. The signature of a function that prints all employee names would be

```
void
print_emp_name(
  employee_if     *to_print
);
```

to allow the printing of names of local, persistent, and remote employees. The signature of a routine that forced the saving of a persistent object would be something like

```
void
force_save(
  persist       *to_save
);
```

To allow a single object to have its address passed into either routine requires that we use multiple inheritance.

3. *Multiple Inheritance for Multiple Inheritances*

What is really going on in this example is that we are using different kinds of inheritance for different purposes. The C++ language doesn't help us here, for all inheritance in C++ appears to be of a single kind—a derived class inherits from its base class, and that's all there is to it. Unfortunately, this hides a distinction between at least three different kinds of inheritance, all of which look like they are simple cases of a class deriving from another class.

In the C++ literature, most examples of inheritance are examples of what I will call implementation inheritance. Implementation inheritance is characterized as the relationship a derived class has with its base class when some of the functions of the derived class are delegated to functions that have been implemented in the base class. The derived class inherits code from the base class.

This sort of inheritance is one that most programmers from outside the object-oriented paradigm can immediately appreciate. It's easy to see the advantages implementation inheritance gives you, because you can produce a new object without writing very much code. So it isn't

surprising that most of the books that attempt to teach C++ center on this sort of inheritance.

Unfortunately, implementation inheritance is probably the most difficult kind of inheritance to use if one wishes to give an example in which multiple inheritance is needed. The actual implementation of some code is tied rather closely to the particulars of some class, and can usually only be reused if the class derived from the base class differs only in a fairly small way from the original class. Put another way, implementation inheritance works well only in cases when the derived class is a subset of the base class, i.e., it differs from the base class only by being more restrictive. The implementation of functions in the base class that are irrelevant to the restriction can then be reused (inherited), because the things that make the derived class different from the base are irrelevant to those implementations.

The subset relation holding between a derived class and its base class is a characteristic of single inheritance. Multiple inheritance allows (and, in fact, generally requires) that the derived class be a superset of any of its base classes. In a single inheritance system, if A is derived from B you can be sure that A is just a more restrictive kind of B. In a multiple inheritance system, if A is derived from B and C you cannot be sure that A is just a more restrictive kind of B. In fact, you know that it isn't—it is both a B and a C. So it is not surprising that any code implemented for something that is only a B would not work well for something that is both a B and a C.

A second sort of inheritance is the reason for the **employee_if** class in the example. This sort of inheritance is often called interface inheritance[4], because the reason for using this sort of inheritance is to allow the same functional interface to be presented by all objects that are members of classes that derive from that class.

The **employee_if** class can be thought of as a class that exists purely for the purpose of interface inheritance. It has no data associated with it, and there is no implementation of any of the functions that are the only real meat of the class. In effect, the class is a contract between any class that derives off of it and the rest of the world, saying that since the class derives off of the **employee_if** class, it thereby contracts to allow the calling of any and all of those functions.

4. There are a number of authors who identify this sort of inheritance with this name; see, for example, Dewhurst and Stark [1989].

The final kind of inheritance is what I will label data inheritance; it is the kind of inheritance used in the **remote** class in the example. This is sort of the flip-side of interface inheritance—while the latter gives functional interfaces without any data or implementation, the pure case of data inheritance allows the derivation of a new class that shares only data members with no implication that the functions that can be called on instances of such a derived class or the behavior of those instances will have anything in common with the base. In our example, the remote class was defined only so that all remote objects, no matter what their behavior or interface, could inherit the same data layout, that in turn would allow the RPC mechanism to work.

These last two sorts of inheritance, interface inheritance and data inheritance, are far more likely than implementation inheritance to require support for multiple inheritance in the language. Each of these is of limited use by itself, but gains power when allowed to be part of an inheritance graph. These other sorts of inheritance allow the specification of relationships between classes of objects that are more subtle than those that are given by the use of implementation inheritance, and allow the same code to use different sorts of objects based on their similarities.

The reason that no examples of multiple inheritance have been given thus far has much to do with the obvious power of implementation inheritance. Implementation inheritance is an obvious win, for it allows new classes of objects to be created that only need to have code written for a small part of their functionality. Showing examples of implementation inheritance is a good way to win over programmers who are unfamiliar with object oriented programming to the paradigm, for they see that they can do more with less code.

No such obvious savings are seen when interface or data inheritance is used. It is only when looking at the larger system, the part of the system that uses the objects of the classes that are created, that the payoff of these other kinds of inheritance become obvious. Since this payoff requires seeing the use of multiple inheritance within the context of a full system, and since the gains in such systems come about only when the systems themselves become large or complex, it is difficult to give examples that are both easily understood and obviously useful that use these other forms of inheritance.

Once we understand this, it is not surprising that no simple example of multiple inheritance has been given showing a need for the fea-

ture. However, to argue that because an example is lacking the feature should not be put into the language misses the distinction between the various forms of inheritance. Accepting such an argument is tantamount to saying that the language should contain only implementation inheritance. This would weaken the language significantly.

To say that a language feature should be taken out if no one can provide a simple example of the feature's use is justified if the language is being designed to support the construction of examples. However, for a language like C++, which has been designed for production work, the lack of a simple example does not show that the feature is not needed. It only shows that the feature may not be simple.

References

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

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

T.A. Cargill, Controversy: The Case Against Multiple Inheritance in C++, *Computing Systems 4.1* (1991)

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

Bruce Martin, The Separation of Interface and Implementation in C++, *Proceedings of the USENIX C++ Conference, Washington, D.C.,* April 1991.

Robert Seliger, Extended C++, *Proceedings of the USENIX C++ Conference, San Francisco,* April 1990.

Michael D. Tiemann, Wrappers: Solving the RPC Problem in GNU C++, *Proceedings of the USENIX C++ Conference, Denver,* October 1988.

[received Mar. 13, 1991; accepted Mar. 29, 1991]