PASSING LARGE DYNAMIC OBJECTS OUT OF FUNCTION FRAME BOUNDARIES: THE TEMPORARY LINKED LIST METHOD

Ahsan J. Sharafuddin, Nathan Ida and James E. Grover
Department of Electrical Engineering
The University of Akron
Akron, OH 44325

Abstract: **It is a well known fact that passing objects *into* functions by reference (in C++) is more efficient than passing by value. Passing objects, created within functions, *out of* functions by reference could also be more efficient if it wasn't sure to cause memory-leak in the system. Programmers therefore resort to "pass-by-reference, return-by-value". Here we first briefly look at why it is important to be able to return by reference, particularly for large dynamic objects in mathematical applications, and then present a new scheme for doing so. The new method can attain efficiency comparable to that of 'method of reference counting', but without the undesirable side-effects of the latter.**

INTRODUCTION:

Objects can be passed into and out of functions in two ways in C++:

(i) A copy of the object is made.
(ii) The address of the object is passed.

Here we are not considering sharing global memory with the rest of the program -- a phenomenon considered a deviation from object oriented paradigm.

The first method is called *pass by value*. Since a copy of the object is made, an object passed into a function can be changed inside the function without affecting the original.

There are two variations of the second method -- one when a pointer is explicitly taken, the other when a reference to the object is used. In both cases only the address of the object is transferred. *Pass by reference* is clearly the preferred method of transferring objects because it does not involve time-consuming operations of creating and initializing new objects.

Passing objects *into* functions by reference is easy and simple because the objects passed to the function are created outside the function boundaries. But returning objects, created within the function boundaries, by reference may be disastrous because of the following reasons. First, since local objects created on the stack are deleted as soon as they go out of scope, outside the function a reference to such an object points to an invalid location in memory. Secondly, if instead the object is created in heap memory (free store) and a reference is returned, the problem of how the memory occupied by such an object ever get reclaimed arises. This is a 'memory leak' situation [1][2].

In this article we will consider a matrix class as an examples and examine why it is important to be able to return matrix objects, created within matrix member functions, by reference. We will also introduce a new scheme that will enable us to *"pass-by-reference, return-by-reference"*. Finally we will compare the efficiency of this method with other methods of parameter passing.

WHY RETURN BY REFERENCE? :

To prove our point let us focus our attention on a simple statement:

```
A = B + C;
```

where A, B and C are matrices of the same size. Since the size of a matrix may not be known at compile-time, the memory allocation for these matrices must be done at run-time. That is, A, B and C are also dynamic objects. There are two operations involved in the statement -- addition (+) and assignment (=). We also assume that both of these operations are carried out by member functions of the matrix class. Now let us look at the sequence of operations (including *hidden* activities carried out by compilers) corresponding to the statement -- first, if we return by value and second, if we *could* return by reference.

`A = B + C;` Pass By Reference, Return By *Value*

1) Create a new object `'result'` on the stack,	Constructor call
2) calculate B+C and store in `result`,	
3) create a temporary object, say `temp1`, and copy result to `temp1`,	Copy-Constructor call
4) destroy result,	Destructor call
5) exit the addition function,	
6) throw away contents of A,	Same as a destructor call
7) copy `temp1` to A,	& a copy-constructor call
8) copy A to `temp2`,	Copy-Constructor call
9) exit the assignment function,	
10) destroy `temp1` and `temp2`,	2 destructor calls.

Here we assumed that the assignment function returns a copy of the object by value to make multiple assignments in single statements possible. We therefore have the equivalent of

1 Constructor call
3 Copy-Constructor calls
4 Destructor calls.

`A = B + C;` Pass By Reference, Return By *Reference*

1) Create an object on the heap pointed to by the pointer `'result'`,	Constructor call
2) calculate B+C and store in object pointed to by `result`,	
3) return `result` by reference,	
4) exit the addition function,	
5) throw away contents of A,	Same as a destructor call
6) make A point to `result`,	
7) return A by reference and exit the assignment function.	

Here we assume that a mechanism for returning objects by reference exists. We will later see that such a mechanism would also make step 6 above an obvious choice instead of copying object pointed to by `result` into A element-by-element. In the above sequence we have the equivalent of

1 Constructor call
1 Destructor call.

Therefore, being able to return by reference would save us 3 copy-constructor calls and 3 destructor calls in the single statement `A = B + C`. How much of a savings is this?

The time taken to execute a destructor call is not proportional to the size of the object being destroyed. But the execution-time of a copy-constructor is proportional to the size of the object. In physics or engineering applications involving finite difference or finite element analysis we frequently encounter very large matrices. For a 100x100 matrix, for example, a single call to the copy-constructor involves transferring 100,000 bytes of data (assuming 80-bit floating point representation). Saving three copy-constructor calls therefore amounts to a saving of 300,000 bytes of data transfer for such matrices in a simple statement like `A = B + C`. Thus, for large dynamic objects, returning by reference can result in substantial efficiency improvement.

WHY NOT *REFERENCE COUNT?* :

A method called *reference counting* can be used to keep track of how many objects are currently pointing to a particular data-structure. Since the reference count (an integer variable) must be associated with the data rather than with any particular object, it is allocated in heap memory as part of the data-structure. When an object is copied to another under this scheme, the objects are not given their own copies of the data, instead the reference count of the data-structure is increased and the newly assigned to object is made to point to the common data-structure. On the other hand, when a particular object goes out of scope, the destructor decrements the reference-count and only deletes the data if no other object is pointing to it. Reference counting eliminates the element-by-element copying of data in the system. In some application it can result in significant memory saving and substantial increases in speed.

But having more than one object point to the same data is not a good idea, because any change one object makes to the data will be reflected in the other objects with unpredictable results. In math-intensive applications this can be a potential threat to data-integrity, which may be extremely difficult to debug. It has therefore been suggested that reference counting is not always the best solution to the problem of passing and returning large objects [3]. Although it can be useful in 'cloning' objects.

THE NEW SCHEME:

Now we will present a simple idea that will enable us to return objects by reference.

Let each matrix object have three pointers as data-members, as shown in fig.1, and a list of functions to access and operate on the actual matrix data.

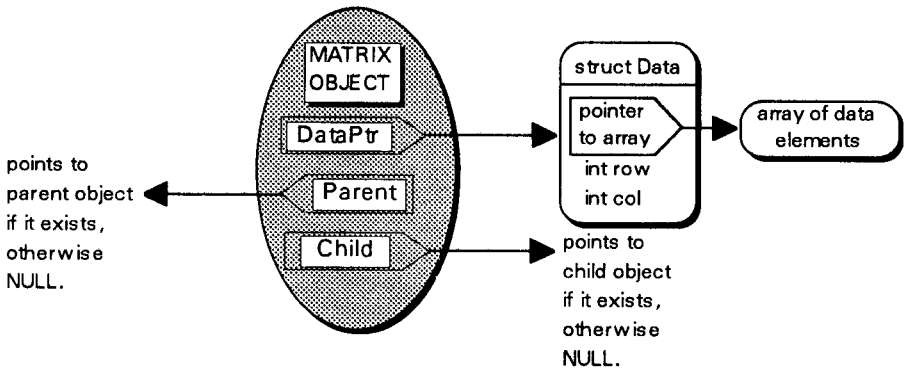

Fig. 1. Representation of matrix class.

In subsequent diagrams we will omit the `DataPtr` for simplicity. Also notice that in the following discussion *parent* and *child* have nothing to do with inheritance of classes (which we are not using), but only describe relationship between objects attached to each other.

In any member function that must return a new object, the object is created in the heap and `Child` points to the newly created object (this view will be slightly modified later). The self-pointer, `'this'`, is passed to the constructor when child is constructed so that the child's `Parent` pointer can point to the appropriate object. The function then computes the result and returns it by reference in child. The destructor on the other hand must check to see if the object being destroyed has a child -- if it does, the child is also destroyed. This ensures that no memory leak occurs in the system. Notice that any object created within member function to return by reference is created as a child, whereas any object created in the client program is a *regular* object (`Parent==NULL`). Child objects are meant to be used by member functions only.

The copy-constructor and the assignment operator must be modified to deal efficiently with this new situation as follows. These functions must first check if the argument is a regular (`Parent==NULL`) object. If it is (e.g., for the statement `A = B;`), a new copy is made. But if it is not (e.g., in `A = B + C;`), then the argument is *detached* from its parent and associated with the new object being created or assigned to. (Detaching an object from its parent is the same as assigning `NULL` to both of its `Parent` pointer and parent's `Child` pointer). This ensures that no two objects point to the same data and also prevents unnecessary copying of data.

Let us now go back to the member function that returns `Child` by reference. Suppose the function is the addition operator. Fig. 2 gives a graphical view of the operations corresponding to the statement `A = B+C`.

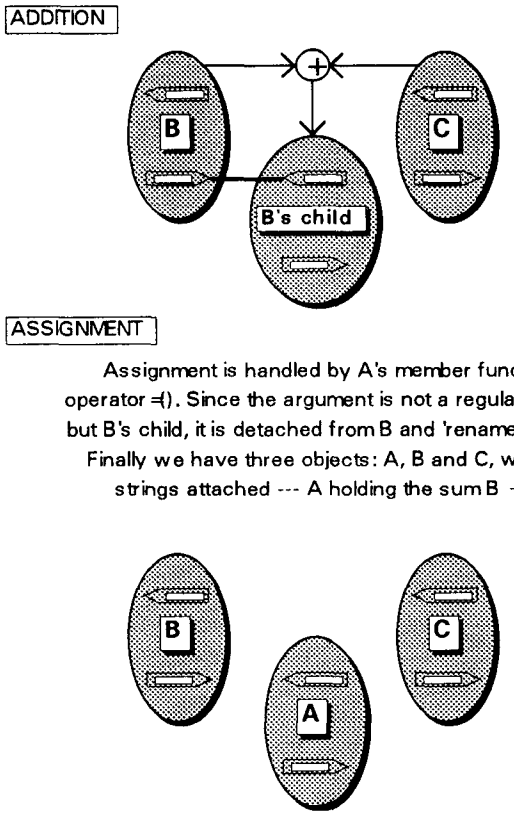

Fig. 2. Graphical representation of `A = B + C;`

The parent-child relationship of objects essentially forms a *doubly-linked-list* with two element -- a head (parent) and a tail (child). But the list need not have only two elements. To accommodate more complicated expressions, a member function must add its return object at the end of such a list instead of making it a direct child.

Consider the statement `A = B * B.transpose();` The `transpose()` member function does not change the matrix B, but only returns the transpose of B in B's child. The multiplication operator, also B's member function, finds its argument (`B.transpose()`) to be its own child, it therefore creates a *grand child* and returns the product in it (Fig. 3).

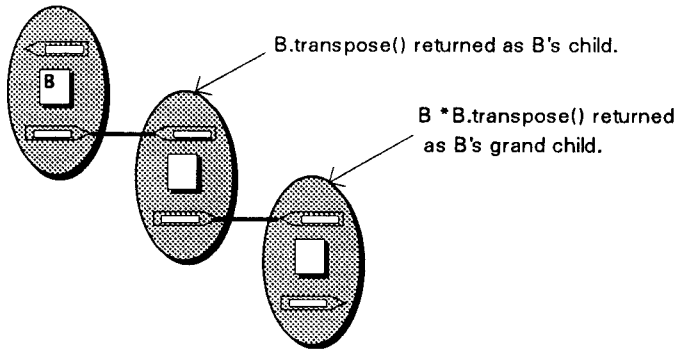

Fig. 3. B and associated objects just before the assignment in `A = B * B.transpose();`

During the assignment operation B's grand child is detached from B's child and 'renamed' as A, but B's child, although has served its purpose, remains in memory. This is not a memory-leak situation, because it will be deallocated as soon as B goes out of scope. But to keep memory usage to a minimum, especially when we are dealing with large objects, one can give the assignment operator (and copy-constructor) the responsibility of cleaning up of such objects. In fact a generalized clean-up routine can also be written to be called from member functions just before returning an object.

Notice that in fig. 2 the argument to the function `operator+()` is the object C, which is not part of any list, therefore no clean-up is required. On the other hand, in fig. 3 the argument to the function `operator*()` is `B.transpose()`, which is, being B's child, part of B's list and can be removed from the list before the function returns.

CONCLUSION:

The major part of the new scheme presented here essentially involves manipulating linked-lists. Since linked lists are used to store objects for temporary (internal or local) use, we call it the method of temporary linked list. Once implemented with appropriate clean-up routine, the scheme proves to be very efficient in passing large objects created within member functions.

Fig. 4 compares the time of execution of the statement `A = B + C;` for 100x100 matrices for different methods of parameter passing. It is interesting to note that the time taken by 'pass-by-reference, return-by-reference' is exactly the same as the time taken by 'reference count' method. But the former does not have the undesirable side-effects of the latter (discussed earlier). Better speed improvements may be observed for more complicated expressions.

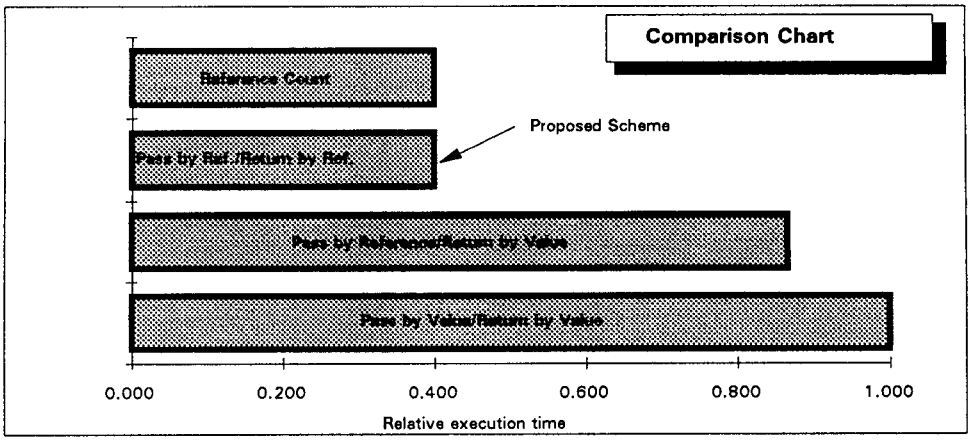

Fig. 4 Relative speed of execution of the statement `A = B + C;` for 100x100 matrices using different methods of parameter passing. (Compiler: Turbo C++ 3.0).

When used with inheritance hierarchies (currently under investigation), the method may prove to be useful in more important ways.

REFERENCE:

[1] L. S. Tang, "C++'s Destructors Can Be Destructive", ACM SIGPLAN Notices, vol 26, no 10, pp 44-52, October 1991.

[2] Scott Meyers, Effective C++: 50 Specific Ways to Improve Your Programs and Designs, Addison-Wesley Publishing Co., Inc., 1992.

[3] Bruce Eckel, Using C++, Osborne McGraw-Hill, 1989.