T E C H N O T E :
On Multiple Inheritance & HandleObjects

By Larry Rosenstein (DTS Emeritus)
Revised by Peter Gontier & Chris Forden
gurgle@apple.com
Apple Developer Technical Support (DTS)

This Technote answers a common question about MPW C++: Why doesn't `HandleObject` support multiple inheritance? To answer that question, this Note provides a brief overview of how multiple inheritance is implemented in MPW C++.

This Technote is addressed primarily to C++ developers who are concerned about memory fragmentation.

About HandleObjects

Beginning with Version 3.0, MacApp switched the focus of its object memory management from a handle-based system to a pointer-based system. This change substantially improved execution speed, specifically because pointer-based objects avoid compaction delays.

Accordingly, Apple recommends using `malloc` or the standard `operator new` for allocating small objects.

Technote 1009 - Release 1.0 © Apple Computer, Inc. 10/24/95

For large objects, handles have some significant advantages. For one thing, they minimize RAM usage by avoiding fragmentation. Also, some developers need to continue using handles in their existing code.

Handles under Copland

Plans for Copland, the advanced Mac OS designed to supercede System 7.x, call for handles to become less important in the Mac's system software and API. An improved implementation of virtual memory (VM) will alleviate the effects of fragmentation for objects larger than the size of a VM page (4K bytes) while increasing the duration of heap compaction due to page-swapping between disk and RAM.

Note
The items called "vtable" in this Technote are actually pointers to the vtable, which resides elsewhere in memory. The more recent reference *The Annotated C++ Reference Manual* uses the term "vptr".

Using HandleObjects

MPW C++ contains several extensions to standard C++ for supporting Macintosh programming. One such extension is the built-in class `HandleObject`. Objects of any class descended from `HandleObject` are allocated as handles in the heap. Your program may refer to one of these objects as if it were a simple pointer; the compiler takes care of the extra dereference required.

A `HandleObject` is useful in Macintosh programming for the same reason that a handle is useful. The use of handles helps prevent heap fragmentation, which is critical on Macintosh computers that use small amounts of memory. If you need to write an app that is small — i.e., less than 100K — you need to consider using `HandleObjects`.

The nature of `HandleObject` imposes some restrictions, however, on how you can use it in a program. These restrictions include:

- heap allocation
- handle manipulation

Technote 1009 - Release 1.0 © Apple Computer, Inc. 10/24/95

- multiple inheritance.

Heap Allocation

Because each object is allocated as a handle, all objects must be allocated on the heap. (“Native” C++ objects can be allocated on the stack or in the static space as well.) Consequently, you always declare variables and parameters as pointers to an object of the class. For example:

```
class TSample: public HandleObject {
public:
    …
    long  fData;
};

TSample  *aSampleObject; // Legal
TSample anotherSample; // Results in a compile-time error
```

The error message the compiler generates in this case is:

```
Can’t declare a handle/pascal object: anotherSample
```

At first, this message might seem strange because the last two lines seem to both declare objects. Actually, the first declaration is of a *pointer* to an object, not of the object itself.

Handle Manipulation

The second restriction is that you must follow the usual rules for manipulating handles. In particular, you have to be careful about creating pointers to `HandleObject` data members, since the object might move if the heap is compacted. If you write

```
long *x = & (aSampleObject -> fData);
```

then x becomes “stale,” i.e., it has a valid address but doesn’t point to where the programmer intends, if the object moves. The solution is to lock the object if there is a possibility that the heap may be compacted. Objects of `HandleObject` are allocated with a call to `NewHandle`, so you can use `HLock` and `HUnlock` (along with an appropriate type cast) to lock and unlock the object.

Multiple Inheritance

The third restriction is that you cannot use multiple inheritance with a `HandleObject`. The reason behind this restriction is not obvious. To understand the reason, you must look at the implementation of multiple inheritance.

Implementing Multiple Inheritance

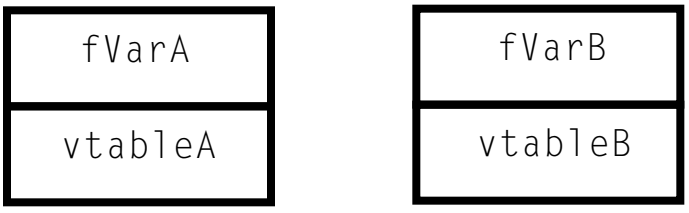
To understand how multiple inheritance is implemented, one needs a simple example. Suppose you define two classes as follows:

```
class TBaseA {
public:
    virtual void SetVarA(long newValue);
            long fVarA;
    …
};

class TBaseB {
public:
    virtual void SetVarB(long newValue);
            long fVarB;
    …
};
```

If you were to look at objects of these classes (see Figure 1), you would find that in each case the object storage would contain four bytes for the C++ virtual table (`vtable`) and four bytes for the data member. Any code that accesses the data members (for example, `TBaseB::SetVarB`) would do so using a fixed offset from the start of the object. (In the particular version of C++, this offset was 0; your offset may vary.) Figure 1 shows the layout of `TBaseA` and `TBaseB` objects.

Figure 1 Layout of TBaseA and TBaseB objects



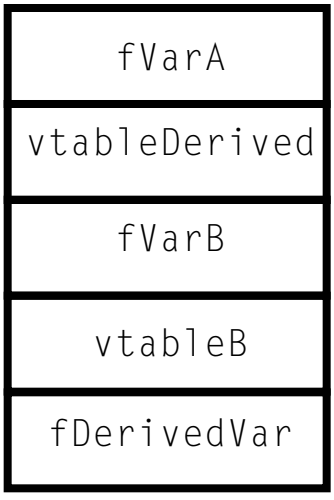
Now suppose you define another class:

```
class TDerived: public TBaseA, public TBaseB {
public:
    virtual void SetDerivedVar(long newValue);
            long fDerivedVar;
    …
};
```

In this case, an object of `TDerived` has the following layout, as shown in Figure 2:

Technote 1009 - Release 1.0 © Apple Computer, Inc. 10/24/95

Figure 2 Layout of TDerived object



This is what you would expect. `TDerived` inherits from both `TBaseA` and `TBaseB`, and therefore objects of `TDerived` contain a part that is a `TBaseA` and a part that is a `TBaseB`. In addition, the virtual table `vtableDerived` includes the tables for both `TBaseA` and `TDerived`.

`TDerived` also inherits the virtual member functions defined in `TBaseA` and `TBaseB`. Suppose you wanted to call `SetVarB`, using a `TDerived` object. The code for `SetVarB` expects to be passed a pointer to a `TBaseB` object (all member functions are passed a pointer to an appropriate object as an implicit parameter), and refers to `fVarB` by a fixed offset from that pointer. Therefore, to call `SetVarB` using a `TDerived` object, C++ passes a pointer to the middle of the object; specifically it passes a pointer to the part of the object that represents a `TBaseB`.

This gives you a very basic idea of how C++ implements multiple inheritance. For more details, read "Multiple Inheritance for C++" by Bjarne Stroustrup in *Proceedings EUUG Spring 1987 Conference*, Helsinki.

Impact on HandleObjects

Each member function of a `HandleObject` class expects to be passed a handle to the object, instead of a pointer; when multiple inheritance is used, the compiler sometimes has to pass a pointer to the middle of the object.

Pointers into the middle of an object, even though (and especially because) they are implicit in this case, nevertheless present the same problem as pointers to object data members (as described earlier). The object's handle could be moved during heap compaction, rendering the pointer "stale."

Designing a new implementation of multiple inheritance that is compatible with a `HandleObject`, as well as the rest of C++, is a big undertaking. For that reason, it is unlikely this restriction will disappear in the future. There are, however, two alternatives to consider:

Ignore Fragmentation

For the majority of today's machines and applications, the main reason to use `HandleObject` is for purposes of compatibility with code that expects handle objects. However, another valid reason is to reduce the chance of fragmentation that would result from using non-relocatable blocks.

But even in applications for which fragmentation would otherwise be a critical concern, memory allocation patterns may be very predictable; fragmentation is less of an issue when all allocated blocks are of similar sizes.

Abandoning Multiple Inheritance

The other alternative is to give up multiple inheritance. In most cases, this isn't as difficult as it sounds. The typical way you would do this is with a form of delegation. For example, you could rewrite the class `TDerived` as:

```
class TSingleDerived: public TBaseA {
public:
    virtual void SetDerivedVar(long newValue);
            void SetBaseB(long newValue);
            long fDerivedVar;
            TBaseB fBaseBPart;
    …
};
```

In this case `TSingleDerived` inherits only from `TBaseA`, but includes an object of `TBaseB` as an data member. It also implements the virtual member function `SetBaseB` to call the function by the same name in the `TBaseB` class. (In effect, `TSingleDerived` delegates part of its implementation to `TBaseB`.)

There are advantages and disadvantages to this approach. The advantage is that it requires only single inheritance, yet you can still reuse the implementation of `TBaseB`. The disadvantage is that `TSingleDerived` is not a sub-class of `TBaseB`, which means that an object of `TSingleDerived` cannot be used in a situation that requires a `TBaseB`. Also, `TSingleDerived` has to define a member function that corresponds to each function in `TBaseB`. (You can, however, define these functions as inline and non-virtual, which eliminates any run-time overhead.)

Caveat

You should realize that the multiple inheritance implementation described here costs some extra space, compared to a simpler implementation that does not support multiple inheritance (e.g., the implementation used for a `HandleObject`). Each `vtable` is twice as large, and each virtual member function takes about 24 bytes, compared to 14. This is true even if you do not take advantage of multiple inheritance. For this reason, MPW C++ also contains a built-in class called `SingleObject`, whose objects can be allocated in the same way as normal C++ objects, but which only supports single inheritance.

Note
The third class built into MPW C++, `PascalObject`, uses Object Pascal's run-time implementation, which takes the least amount of space, but the most execution time. ◆

Managing Many Handles

If you're writing large, high-end applications, you may need to manage thousands of objects. The Macintosh Memory Managers slow down when required to deal with so many handles. If you're dealing with many handles, here are some important points to keep in mind:

- Keep the objects locked except just after calling `WaitNextEvent`. Then, if some predetermined amount of time (perhaps one minute, for example) has

Technote 1009 - Release 1.0 © Apple Computer, Inc. 10/24/95

elapsed since the user interacted with the application, unlock everything and compact the heap.

- When you compact the heap, do it incrementally, i.e. do a little bit of compaction, then check to see if you have used up more than 50,000 microseconds in the process. When you have used up that much time, call `EventAvail` to check if there is now an event that needs processing. If such an event has arrived, return to your main event loop to process it.
- Because the Modern Memory Manager's `CompactMem` routine really compacts the entire application heap even if you ask it to compact just a little bit of it, use `NewPtr` instead. Ask for a moderate-sized block (e.g. "`NewPtr(40000)`"), and check the time to see if you need to call `EventAvail`. Then ask for another block.
- After compacting memory with `NewPtr`, dispose of the compaction pointers, and lock your handles.
- When you can't allocate a pointer, you're done.

This approach will cause purgeable handles to get purged, so if you don't want that to happen, create and manage a list of purgeable handles. Call `HNoPurge` on this list prior to a `NewPtr`-based compaction, then call `HPurge` on it after the compaction.

Summary

You cannot use a `HandleObject` with multiple inheritance because of the way multiple inheritance is implemented in MPW C++. Your alternatives are to give up one or the other: You can either use native C++ objects and let the objects fall where they may, or give up multiple inheritance and use a form of delegation.

Further Reference

- MPW C++ Reference Manual
- "Multiple Inheritance for C++," Bjarne Stroustrup, *Proceedings EUUG Spring 1987 Conference*, Helsinki.

■ Ellis, Margaret A. and Bjarne Stroustrup, *The Annotated C++ Reference Manual*, (a.k.a "*ARM* "), Addison-Wesley, 1990, ISBN 0-201-51459-1, pp.217-237

Change History

This Technote was originally written in August, 1990.

It was revised in July, 1995, to include information on managing multiple handles and updated again with new information in October, 1995.