New Technical Notes
Macintosh

®

Developer Support

Multiple Inheritance and HandleObjects

Platforms & Tools **M.PT.HandleObjects**

Written by: Larry Rosenstein August 1990

This Technical Note answers a common question about MPW C++: "Why doesn't `HandleObject` support multiple inheritance?" It does this by giving a brief overview of how multiple inheritance is implemented in MPW C++.

What Are HandleObjects Anyway?

MPW C++ contains several extensions to "standard C++" for supporting Macintosh programming. One such extension is the built-in class `HandleObject`. An instance of any class descended from `HandleObject` is allocated as a handle in the heap. You refer to one of these instances as if it were a simple pointer; the compiler takes care of the extra dereference required because the object is really a handle.

A `HandleObject` is useful in Macintosh programming for the same reason a handle is useful. The use of handles helps prevent heap fragmentation. The nature of `HandleObject` imposes some restrictions on how you can use it in a program, however.

First, since each instance is allocated as a handle, it follows that all instances must be allocated on the heap. ("Native" C++ objects can be allocated on the stack or in the global space as well.) Consequently, you always declare variables, parameters, etc. to be pointers to the class. For example:

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

TSample  *aSampleInstance;        // 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 in this code seem to both declare objects. Actually, the first declaration is of a *pointer* to an object, not of the object itself.

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 a `HandleObject` instance variable, since the object might move if the heap is compacted. If you write

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

then x becomes invalid if the object moves. The solution in this case is to lock the object if there's a possibility of the heap being compacted. Instances of HandleObject are allocated with a call to _NewHandle, so you can use _HLock and _HUnlock to lock and unlock the object.

The third restriction is that you cannot use multiple inheritance with a HandleObject. The reason behind this restriction is not evident, however. 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 instances of these classes (see Figure 1), you would find that in each case the instance storage would contain four bytes for the C++ virtual table (vtable) and four bytes for the instance variable. Any code that accesses the instance variable (for example TBaseB::SetVarB) would do so using a fixed offset from the start of the object. (In this particular version of C++, this offset was 0; your offset may vary.)

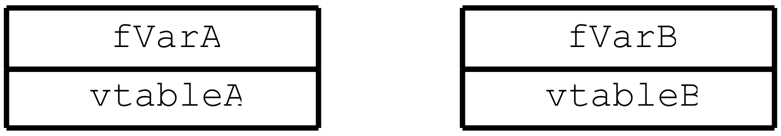

Figure 1–Layout of TBaseA and TBaseB Instances

Now suppose you define another class:

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

In this case, an instance of `TDerived` has the following layout:

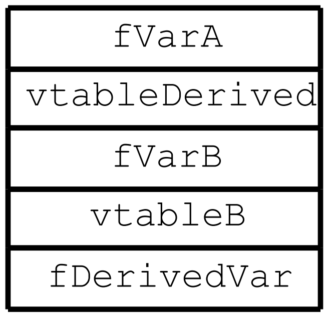

Figure 2–Layout of TDerived Instance

This is what you would expect. `TDerived` inherits from both `TBaseA` and `TBaseB`, and therefore instances 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 methods defined in `TBaseA` and `TBaseB`. Suppose you wanted to call the method `SetVarB`, using a `TDerived` object. The code for `SetVarB` is expecting to be passed a pointer to a `TBaseB` object (all methods 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.

So What About HandleObjects?

The next question is how this implementation imposes a restriction on a `HandleObject`. The answer is simple. Each method of a `HandleObject` class expects to be passed a handle to the object, instead of a pointer. But when multiple inheritance is used, the compiler sometimes has to pass a pointer to the middle of the object. It is not possible to create a valid handle that refers to the middle of another handle. (Creating a fake handle is a compatibility risk; besides, the pointer into the middle of the handle would be invalid if the handle is moved.)

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 that this restriction will disappear in the future. There are, however, two alternatives to consider:

Damn the Fragmentation, Full Speed Ahead

The main reason to use a `HandleObject` is to reduce the chance of fragmentation that would result from using a non-relocatable block. In a few applications, however, the memory allocation patterns are very predictable, and fragmentation might not be an issue. In those cases, you can use “native” C++ classes. (Don’t use the argument that 8 Mb machines are common, and virtual memory is here to stay so fragmentation isn't an issue at all. Data always expands to fill the available memory space, real or virtual.)

If you adopt this approach, you should read the article "Using C++ Objects in a Handle-Based World" by Andrew Shebanow in Issue 2 of *d e v e l o p*, April 1990. This article describes how you can use native C++ objects and minimize heap fragmentation, by overriding the way C++ normally allocates objects. The same techniques can be used to customize the way your program allocates certain objects.

"Doctor, It Hurts When I Do That…"

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 instance of `TBaseB` as an instance variable. It also implements the method `SetBaseB` to call the method by the same name in the `TBaseB` class. (In effect, `TSingleDerived` delegates part of its implementation to `TBaseB`.) The advantage of this approach is that it requires only single inheritance, yet you can still reuse the implementation of `TBaseB`.

The disadvantages are that `TSingleDerived` is not a subtype of `TBaseB`, which means that an instance of `TSingleDerived` cannot be used in a situation that requires a `TBaseB`. Also, `TSingleDerived` has to define a method that corresponds to each method in `TBaseB`. (You can, however, define these functions as inline and non-virtual, which eliminates any run-time overhead.)

By The Way…

You should realize that the multiple inheritance implementation previously described 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 method call 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 instances are allocated in the same way as normal C++ instance, but which only supports single inheritance. (By the way, 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.)

Conclusion

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
- "Using C++ Objects in a Handle-Based World," Andrew Shebanow, *d e v e l o p*, Issue 2, April 1990.
- "Multiple Inheritance for C++," Bjarne Stroustrup, *Proceedings EUUG Spring 1987 Conference*, Helsinki.