Time Invariant Virtual Member Function Dispatching For C++ Evolvable Classes

Roger Voss

Aldus Corporation
411 First Avenue South
Seattle, WA 98104-2871 USA
Roger.Voss@Aldus.com
CompuServe: 73650,616

Abstract

Motivated to achieve the technical requirements of object-oriented component-based application architecture, and hoping to migrate substantial existing C++ source code into such future applications, Aldus Corporation has investigated the obstacles that prevent the C++ language from being applied successfully to this programming domain. At stake is this: can the C++ language be enhanced to support the next generation object-oriented application architecture - or will it prove necessary to abandon C++ and completely rewrite new applications in one of the more dynamic object-oriented languages, such as Objective-C, SmallTalk, Eiffel, etc? This paper attempts to address one major C++ obstacle: the need for class method protocol evolvability.

1. Introduction

An evolvable C++ base class is one that can be exported from a shared library and derived from or used in other components. Specifically, though, it must be possible to evolve the private implementation and public method protocol of such classes into new generations and then reissue recompiled versions of the shared libraries containing such classes into the field. Prior existing components that are still in use and that are dependent upon any evolved base classes should not be adversely impacted by new shared library releases.

Contemporary implementations of C++ are unsuitable to this purpose. It is possible, however, to invigorate the C++ language with a new underlying model of class method dispatch compatible to the requirement of base class evolvability. The scheme presented here for a new C++ method dispatching approach has sufficient flexibility yet still retains traditional C++ efficiency and relatively strong type security.

The description to be presented herein of a virtual member function dispatching technique for C++ evolvable classes, is a revisionary supplement to a previous

paper titled: ‘‘C++ Evolvable Base Classes Residing In Dynamic Linking Libraries’’ [Voss93]. The monograph here supersedes the portion of that paper which described a dispatching technique based on a cached, linear search look up of method selectors. That approach is described in sections of that paper titled: ‘‘Method Look up Tables’’; ‘‘Method Selector Caching’’; ‘‘Method Invocation’’; and ‘‘Method Name Mangling’’; those sections are supplanted by the system to be described herein.

The term ‘‘method’’ will henceforth be used in lieu of the phrase ‘‘virtual member function’’. ‘‘Methods’’ will be employed as being synonymous as class protocol actions; and ‘‘virtual member function’’ will be used so as to mean a class-specific method implementation.

The new method dispatching approach to be presented no longer employs cached, linear search look up, but instead uses a pair of index variables of static storage - which when used together directly specify an appropriate entry in a class `vtbl` (‘‘virtual table’’). One of these, the method selector index value, is determined at program execution time *(as opposed to compile time as with conventional C++ classes)*.

The timing and manner in which this is done can be described as *lazy resolution.* The index value is calculated when the class method is invoked from some particular location in program code for the first time. All such method invocations for evolvable classes thus incrementally resolve themselves as the application proceeds in execution. Once a given method selector index is resolved it remains so for the rest of the application run time session. Method dispatch time after the first invocation is invariant *(except with regard to virtual memory paging delays)*. Because method indices are resolved in a *lazy* manner, application startup time is not impeded by having to resolve all method invocation occurrences that exist in the application at one time *(or all those that exist in the paged-in application ‘‘working set’’)*. For very large applications this could be an important consideration.

It is also worth noting that method invocations that remain unexecuted throughout an application run time session are never resolved at all. Thus a user does not incur any delay penalty for having to do method resolution on unused portions of an application. With lazy resolution you pay for it if and when you use it.

Lastly, the instruction code for the method dispatch process does not occur in-line where the method is invoked but instead is centralized in a dispatch function. This permits *after-the-fact* customization of the method dispatch service so as to better tune an application to imposed memory consumption/execution speed constraints. The fastest method dispatch technique will consume the greatest amount of memory. A less memory consumptive dispatch approach will likely sacrifice invariant method dispatching for some form of dynamic method caching. This choice could be made by a programmer by simply specifying different compiler run time libraries to link to.

2. Unique Class-Method Identification

For the purpose of enabling class evolvability, a method will be uniquely distinguished from all other currently existing class methods, and those to yet exist in future base class generations, by a string name representation. *(With conventional C++ classes a method is identified by only a compile time constant offset into a class* `vtbl`*.)* This method string name representation will be very nearly the same as the *mangled name* generated by a C++ compiler for a virtual member function. *(The mangled name is the basis of insuring type safe linkage of C++ programs at static link time.)*

The name mangling procedure used by the compiler to generate a method name string is similar, though not exactly like what is used to generate the mangled name of a virtual member function. There are two differences: The class name embedded into a mangled method name is always the name of the class of the method's origin - the class that first introduced the method declaration, also known as the *introducer class*. Additionally, the string prefix `"@meth@"` is prepended to the name. A method name will not be able to possibly clash with the mangled names of virtual member functions *(which comprise the various class-specific implementations of a method)*. This distinction will be important for source-level debuggers.

At any time that a compiler must generate a reference that reliably identifies a class method, it will use the method string name - as opposed to, say, a `vtbl` offset constant or a function pointer. When compiling a given source translation unit and encountering a situation where a class method is referenced, the compiler will emit a pointer to a string constant. The string constant is of course the method name in mangled form. For the first occurrence of use, the compiler emits a forward declaration for the method name string constant. When compilation has reached the end of the source translation unit then the compiler will emit a declaration for the string constant that is of static storage. It is an anonymous declaration and does not have any external name visibility. A method name string constant declaration will thus occur no more than once per any source translation unit.

3. Method Invocation

There are three static variables needed to support the invocation of an evolvable class method. These three variables can be encapsulated as fields in a structure:

```
struct __MethodSelector {
   unsigned short   methodIndex;
   unsigned short   classIndex;
   char *       methodName;
   };
```

The first field, `methodIndex`, is used for indexing into a *`vtbl` class subsection*

during a method dispatch. The second field, `classIndex`, is used to select the appropriate *class subsection* indirection slot - which are contained internally in the class's `vtbl`. The third field, `methodName`, is used to specify the mangled method name for when the method dispatch is being resolved on the first method invocation occurrence. For a typical 32-bit system this data structure will consume 8 bytes of storage.

On CPUs where the preferred word alignment is 4 bytes, care must be taken to ensure that `__MethodSelector` is a packed data structure or else it might needlessly consume an additional 4 bytes. Of course, in using unsigned short ints, a limit is being imposed here that there can be no more that 65536 classes inclusive in an evolvable class's inheritance hierarchy nor more that a total of 65535 methods introduced by a class. *(The zero method index entry is always reserved for use by the method dispatching system.)*

For each different method invocation that is encountered in a source translation unit, the compiler synthesizes a forward declaration to a variable of type `__MethodSelector`. Here is an example forward declaration on behalf of the first encountered method invocation:

```
extern __MethodSelector __methSel_1;
```

When compilation reaches the end of a source translation unit, then an anonymous static variable of type `__MethodSelector` is synthesized for each different prior synthesized forward declaration, like so:

```
static __MethodSelector __methSel_1 = { 0, 2, __methNameStr_1 };
```

Thus for each different method that is invoked in a source translation unit there is a corresponding anonymous static variable of type `__MethodSelector`. This data structure will be referred to as a *vcall descriptor.* Its `methodIndex` field is initialized to zero and its `methodName` field is initialized with a pointer to the appropriate mangled method name string constant. The `classIndex` field is initialized with a compile time determined constant. In Listing 1 is an entire example sequence of synthesized declarations generated for handling method invocation occurrences in some hypothetical source translation unit.

4. `vtbl` Layout and Class IDs

4-1. Multiple inheritance complexity

The complicated aspect of the C++ virtual function dispatch - whether for conventional classes or evolvable classes - is the issue of accommodating multiple

```
// a synthesized declaration introduced prior to
// the beginning of each source translation unit
struct __MethodSelector {
   unsigned short  methodIndex;
   unsigned short  classIndex;
   char *       methodName;
   };


/*** beginning of actual .cpp source translation unit ***/

Menu::Menu(VString* itemName,MenuID id,BOOL largeMenu,BOOL isPopUp) :
            MenuItem(itemName,MenuItemID(id))
{
   // synthesized forward declarations for method invocations
         // Collection::Find()
   extern __MethodSelector __methSel_1;
         // Dictionary::AddKeyValuePair()
   extern __MethodSelector __methSel_2;

   fHMenu = HMENU(0);
   fLongMenu = largeMenu;
   fIsPopUp = isPopUp;

   fMenuItemID = (id == kNoResourceMenuID)
                                       ? ++gNextCreatedMenuID : id;
   fMenuItems = (LList*)NULL;

   if (gKnownMenus == NULL)
      gKnownMenus = new Dictionary;
   if (gKnownMenus != NULL)
   {
      DWORDCItem* theKey = new DWORDCItem(fMenuItemID);
      if (!gKnownMenus->Find(theKey))
         gKnownMenus->AddKeyValuePair(theKey,this);
   }
}

/*** ending of actual source translation unit ***/


// synthesized anonymous static variables
extern char __methNameStr_1[];
extern char __methNameStr_2[];

static __MethodSelector __methSel_1 = { 0, 3, __methNameStr_1 };
static __MethodSelector __methSel_2 = { 0, 5, __methNameStr_2 };

static char __methNameStr_1[] =
   "@meth@Collection@Find$qn5CItem";
static char __methNameStr_2[] =
   "@meth@Dictionary@AddKeyValuePair$qn5CItemt1";
```

Listing 1

inheritance. In the case of a multiple inheritance class hierarchy an object map may typically have more than one embedded `vptr` (“virtual table pointer”) that points to some sub portion of the overall class `vtbl`. The complexity of this problem is exacerbated for evolvable classes on account the exact location of a method’s virtual function pointer in a `vtbl` is not established until program execution time.

Due to differing derivation paths through multiple inheritance hierarchy graphs, it is necessary for a compiler to sometimes simply *reintroduce* a method into a derived class - instead of continue to associate the actual introducer base class with the method. Such a reintoduced method is treated as though it is introduced for the first time in the derived class and in effect creates a new method protocol in the derived class - which is semantically the same as the one in the original introducer base class. This will be done to establish different coherent *`vptr` views* into the `vtbl` *(where each view corresponds to a different object map derivation hierarchy subsection).* These different `vptr` views of the `vtbl` exist so as to enable potential upward casting of a class pointer to one of its base classes. Upward casting a multiple inheritance class pointer to a base class may cause it to then use a different relative location for a method protocol in the `vtbl` than some other possible cast.

There is also the more straightforward and more typical case, though, where some single `vptr` view into the `vtbl` will be used mutually for several potential upward casts. In this case an inherited method protocol must be assumed to have the same relative location in the `vtbl` for all relevant potential upward cast. This must hold true, too, for where the base classes are used standalone or are derived from in other inheritance hierarchies. In all these possible scenarios a method protocol must make use of the same method index value *(which translates to “same relative placement in the `vtbl`”).* Why? Because the vcall descriptor associated with a method invocation can potentially be used in conjunction to all manner of different `vptr`/`vtbl`/derivation-hierarchy combinations. The possibility of a class appearing in different derivation hierarchies and the potentiality of upward casting make that quite clear.

An abstraction will be introduced for the case of evolvable classes that is intended to help grapple with the complexities of multiple inheritance. For evolvable classes, one might view the `vtbl` organization as dynamic information - as its complexion is established at program execution time. However, it is also possible to cleave an evolvable class `vtbl` into two conceptual parts: one part would be compile-time static known information; the other part would be the remaining dynamic run time information. The ultimate intent in doing this is that it will then be possible to deal with multiple inheritance for evolvable classes using the same solution approaches as is done with conventional classes.

4-2. Two part `vtbl`

Taking this line of thought, we proceed to split the `vtbl` into two distinct conceptual halves. To describe the first half, imagine taking the vtbl of a conventional C++

class, which has several base classes in its inheritance graph and even has multiple inheritance present *(encompassing, say, the vexatious but inevitable diamond inheritance graph case),* and squeezing each class down to where it only introduces one method. This is a `vtbl` layout where every class present in the inheritance graph just introduces a single method. We now have a simple monolithic array of function pointers where each element in this array corresponds one-to-one to an individual class *(or node)* in the inheritance graph.

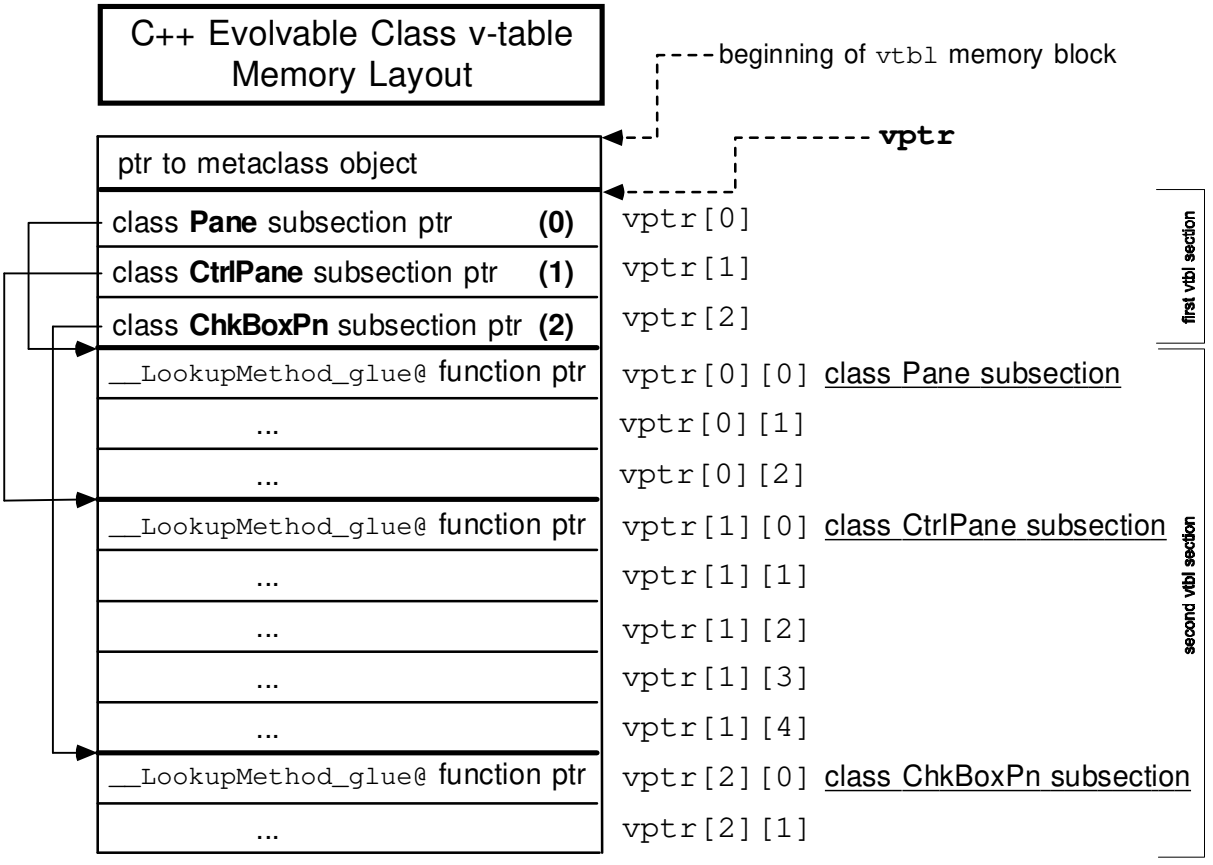
Now instead of containing pointers to virtual member functions, this monolithic array will instead contain pointers to *`vtbl` class subsections.* A class subsection is where virtual member function pointers will actually be located. A class subsection thus represents the protocol of methods introduced by its owning class. *(The concatenation of all the class subsections associated with some inheritance graph into a contiguous block of memory comprises the second conceptual half of the `vtbl` - more on it later.)* We now have a `vtbl` arrangement as portrayed in the schematic of Figure 1.

The simple monolithic array that makes up the first conceptual half of the `vtbl` can be determined at compile-time and is thus statically known information. In fact it could be given static storage and perhaps simply tacked onto an evolvable class's metaclass object - which also is of static storage.

The complication of evolvable classes is that the exact size of class subsections cannot be known at compile time. The number of entries in a class subsection can only be determined at run time by a query to the class's metaclass object. This fact does not adversely impact the monolithic array, or static half of the `vtbl`, that was just described. Regardless of how large individual class subsections end up actually being at run time, the first conceptual half of the `vtbl` will remain static in regard to its assumed complexion *(or layout).*

Well, perhaps that is a quasi truth. As evolvable base classes are derived from and new classes declared, the `vtbls` of these new classes will have their inherited static half accrue a new entry. This will come about through the process of recompiling shareable libraries that export these base classes and then reissuing the libraries *(where they now additionally export new derived classes).* Or through creating new shareable libraries or application modules that host the new derived classes. The net result is that it is fair to say that the static portion of a `vtbl` only evolves by being appended to - a situation no more perplexing than conventional single inheritance class hierarchies.

The `vptr(s)` that are placed in an object map during construction, and that point into a `vtbl`, will point only into the first half, or static portion, of the `vtbl`. A `vptr` will never point directly into the second half of the `vtbl` *(the dynamic portion).* An evolvable class `vptr` thus points to an array of class subsection pointers. In the


C++ Evolvable Class v-table Memory Layout
beginning of vtbl memory block
vptr
ptr to metaclass object
class Pane subsection ptr (0)
class CtrlPane subsection ptr (1)
class ChkBoxPn subsection ptr (2)
__LookupMethod_glue@ function ptr
...
...
__LookupMethod_glue@ function ptr
...
...
...
...
__LookupMethod_glue@ function ptr
...
vptr[0]
vptr[1]
vptr[2]
vptr[0][0] class Pane subsection
vptr[0][1]
vptr[0][2]
vptr[1][0] class CtrlPane subsection
vptr[1][1]
vptr[1][2]
vptr[1][3]
vptr[1][4]
vptr[2][0] class ChkBoxPn subsection
vptr[2][1]
first vtbl section
second vtbl section

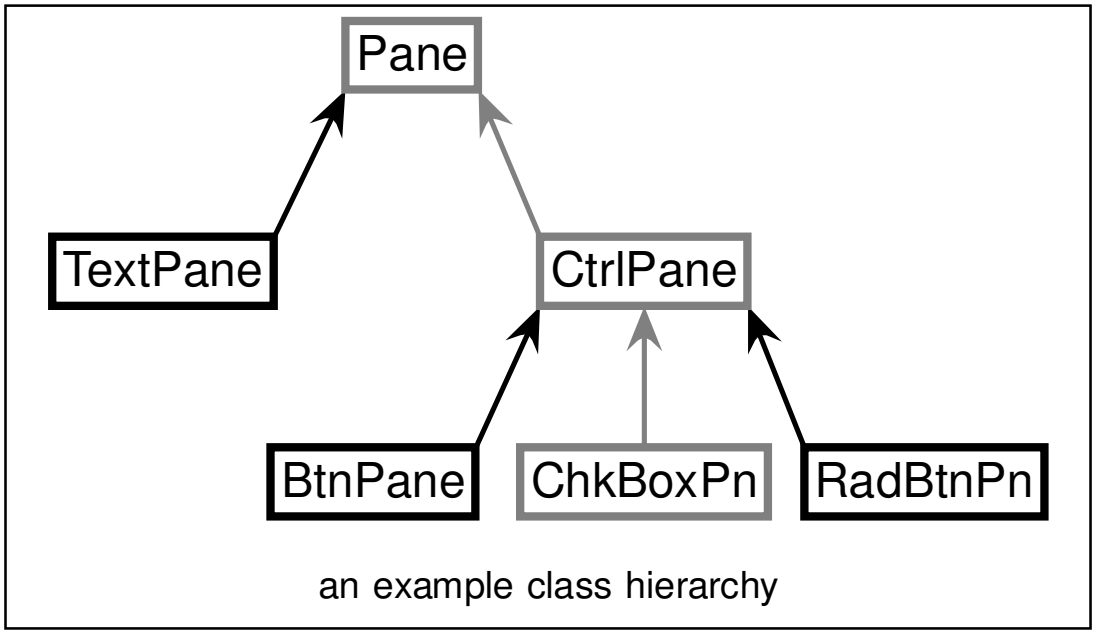

Pane
TextPane
CtrlPane
BtnPane
ChkBoxPn
RadBtnPn
an example class hierarchy

Figure 1

case of where a multiple inheritance object map possesses several `vptrs`, each `vptr` will point to a *windowed* view of the overall monolithic array. Each `vptr` enjoys a view into the monolithic array that accommodates the upward cast *(or subset hierarchy of base classes)* that a given `vptr` represents.

4-3. Class IDs

To dereference some virtual member function in order to complete a method invocation, it is first necessary to dereference the appropriate class subsection pointer. The *class ID index* value facilitates this. The class ID index is the value of the `classIndex` field contained in the vcall descriptor *(recall that a unique vcall desciptor is associated with each and every method invocation).* The method index contained in the vcall descriptor can then be used to dereference into the class subsection so that the appropriate virtual function pointer is selected. A method index value is class subsection relative. This property is why a given vcall descriptor can be used in combination to `vptrs` of differering origins.

Class IDs are thus the statically known indices of class subsection pointers, which are in turn elements of the monolithic array comprising the `vtbl` static half. The process of establishing class IDs occurs incrementally during compilation as declarations of classes comprising an inheritance graph are encountered. Therefore, when a class method call is encountered during compilation, the relevant class ID will be known by the compiler and the associated `classIndex` field can be initialized with the appropriate compile time computed constant. *(Keep in mind that the method index is initialized to zero at compile time and will be determined at program execution time.)*

It is possible that a class that appears in different derivation hierarchies will, as a consequence, be known by different class ID values. That is okay, though, on account `vptr` windowing will set up relative views into `vtbls` that amend for this. A given `vptr` value will set the base position within the overall monolithic array that a class ID index value will be applied against. Object maps containing multiple `vptrs` are simply some of the joys of C++ multiple inheritance. Class ID indirection and `vptr` windowing combine to make the situation tractable for evolvable classes.

4-4. The `vtbl` dynamic half

Much has by now been said of the static half of the `vtbl`. Discussion now turns to the dynamic half of the `vtbl`. The dynamic half of the `vtbl` is established at run time by the metaclass object of the owning class. The metaclass object is of static storage and has a static constructor. The metaclass constructor will be responsible for determining the size of the dynamic part of the `vtbl`, allocating a block of memory for it, and performing necessary initialization.

The dynamic half of the `vtbl` is the concatenation of all class subsections into a single block. The metaclass constructor determines the size of this block by

querying each base class metaclass object for the number of methods its class introduces. The count of methods per subsection is incremented by one to allow for the zero index position of the subsection, which is reserved *(recall that vcall descriptor method index values start out as zero and will hit this slot position when first executed).* The sum of all class subsection counts multiplied by the size of a code pointer yields the required size in bytes of the overall aggregate block. The metaclass constructor, after allocating this block from the free store, will initialize each pointer slot with a pointer to a special method dispatch binding routine. The metaclass constructor will also need to initialize the static half of the `vtbl` by placing a pointer to the appropriate class subsection into each entry in the monolithic array.

4-5. Retrieving the metaclass pointer

One last detail about the `vtbl` is that the monolithic array comprising the static half actually has a *dummy* entry as its first entry. This entry does not contain a pointer to any class subsection but instead holds a pointer to the class's metaclass object. Any `vptrs` into the `vtbl` *(such as the "base `vptr`")* will be adjusted to point past this dummy entry. Refer again to the `vtbl` schematic in Figure 1. It will be possible to retrieve the metaclass pointer by subtracting the size of a data pointer from the *base `vptr`* address and dereferencing the resulting *pointer-to-a-metaclass-pointer* address.

4-6. Applying a *delta* to *`this`*

In section 10.8c of *The Annotated C++ Reference Manual* [Ellis90] titled, "Multiple Inheritance and Virtual Functions", there is a discussion of two techniques for applying a *delta* against the `this` pointer for when a virtual function - which was introduced in some multiple inherited base class - is being invoked. *(The `this` pointer must sometimes be adjusted to point into the object map at a subsection that will resemble an object map layout that the base class expects.)* The first technique of adjusting the `this` pointer suggest implanting a delta value next to the relevant virtual function pointer(s) within the `vtbl`:

```
struct __vtbl_entry {
    void (*fct)();
    int delta;
};
```

However, the method dispatching proposal being described here will favor the second recommendation for applying delta to `this`. In this second approach, a pointer to a *thunk* is placed into a normal `vtbl` entry *(in the context of this proposal it would be a class subsection entry)*. The thunk is a small piece of code that applies the necessary delta adjustment on the `this` pointer and then proceeds to invoke the associated virtual function. Using this technique the `vtbl` entries are always exactly the same under all conditions - they are just simple pointers to code:

```
struct __vtbl_entry {
   void (*fct)();
};
```

This technique is favored on account keeping the `vtbl` class subsection entries as just simple pointers to code under all circumstances greatly simplifies the overall implementation. This is much more true for this evolvable class method dispatching proposal than is the case with just conventional C++ classes.

5. Method Call Code Generation and Conventions

(Refer to Appendix A for an alternative approach to method dispatch code generation that is conducive to CFront C++ implementations.)

For an Intel 80386 CPU, the compiler would generate code for evolvable method invocation as in the examples below for the `Collection::Find()` and the `Dictionary::AddKeyValuePair()` method call occurrences taken from the source code example in Listing 1:

```
                ;
                ;         if (!gKnownMenus->Find(theKey))
                ;
001F FF 75 FC         PUSH  DWORD PTR [EBP-4]            ; theKey argument
0022 8B 1D 00000000r  MOV   EBX,DWORD PTR FLAT:@Menu@gKnownMenus
0028 53               PUSH  EBX                          ; this object pointer
0029 8B 53 04         MOV   EDX,DWORD PTR [EBX+4]        ; vptr (windowed)
002C 8B 4B 04         MOV   ECX,DWORD PTR [EBX+4]        ; base vptr (non-windowed)
002F B8 0000000Cr     MOV   EAX,offset FLAT:___methSel_1 ; ptr to __methSel_1
0034 E8 00000000e     CALL  NEAR PTR __DispatchMethod@   ; call to dispatcher
0039 83 C4 08         ADD   ESP,8
003C 0F B7 C0         MOVZX EAX,AX
003F 0B C0            OR    EAX,EAX
0041 75 1C            JNE   short @18
                ;
                ;            gKnownMenus->AddKeyValuePair(theKey,this);
                ;
0043 56               PUSH  ESI                          ; this argument
0044 57               PUSH  EDI                          ; theKey argument
0045 8B 1D 00000000r  MOV   EBX,DWORD PTR FLAT:@Menu@gKnownMenus
004B 53               PUSH  EBX                          ; this object pointer
004C 8B 53 04         MOV   EDX,DWORD PTR [EBX+4]        ; vptr (windowed)
004F 8B 4B 04         MOV   ECX,DWORD PTR [EBX+4]        ; base vptr (non-windowed)
0052 B8 0000000Cr     MOV   EAX,offset FLAT:___methSel_2 ; ptr to __methSel_2
0057 E8 00000000e     CALL  NEAR PTR __DispatchMethod@   ; call to dispatcher
005C 83 C4 0C         ADD   ESP,12
005F            @18:
```

Listing 2

The arguments to a method are first pushed onto the stack; the object `this` pointer is pushed; the class `vptr` is moved into the `EDX` register; the *`base vptr`* is moved into the `ECX` register; a pointer to the associated vcall descriptor static variable is moved into the `EAX` register; and then a call is made to the `__DispatchMethod@` compiler helper function.

The `__DispatchMethod@` compiler helper function is a generic dispatch service for evolvable class method calls. For the 80386 CPU, the register trio, `EDX:ECX:EAX`, is used to pass arguments to the method dispatch service. Passing these dispatch arguments in registers is preferable to pushing them onto the stack in this case. The various pointers that will be dereferenced during the dispatch process will already be laying in registers ready for use; access to stack memory is minimized; and most importantly: there is no need to readjust the stack in any manner before completing the method call dispatch. If these arguments were pushed onto the stack, they would have to be removed from the stack before jumping into a virtual member function body *(which would not expect them to be on the stack)*. Doing such would mean having to temporarily pop off the return address of the function call and then pushing it back on. This all amounts to a lot of hand waving that can be easily dispensed with by simply using `EDX:ECX:EAX` registers to pass arguments to the dispatch routine.

The in-line code generation size of an evolvable class method call is 11 to 13 bytes greater than would be the case for a non-virtual member function call for a conventional C++ class. The method dispatching routine, `__DispatchMethod@`, *(implementation in Listing 3 below)* represents another 17 bytes of code - its overhead only occurs once in a program module. Removing the dispatch code from in-line results in a net savings of 12 bytes per method call. For large applications with hundreds or thousands of method calls this could result in a significant reduction of overall application code size. Perhaps even more importantly, a non-in-line dispatch service permits the compiler implementer to supply several different kinds of method dispatch; it could even permit the compiler user to create and link to a customized method dispatcher.

```
                ; Parameters passed in registers on entry:
                ;
                ;       EDX         :       vptr (windowed vptr)
                ;       ECX         :       base vptr (non-windowed vptr)
                ;       EAX         :       ptr to vcall descriptor
                ;
                ; EDX:ECX:EAX registers are preserved on function exit
                ;
0000            __DispatchMethod@ proc near
0000 51            PUSH  ECX                          ; preserve ECX
0001 0F B7 48 02 MOVZX ECX,WORD  PTR [EAX+2]        ; move class index into ECX
0005 8B 1C 8A    MOV   EBX,DWORD PTR [EDX][ECX*4] ; get vtbl class subsection offset
0008 0F B7 08    MOVZX ECX,WORD  PTR [EAX]          ; move method index into ECX
000B 8B 1C 8B    MOV   EBX,DWORD PTR [EBX][ECX*4] ; place virtual function ptr in EBX
000E 59           POP   ECX
000F FF E3        JMP   EBX                          ; jump to the virtual function
0011            __DispatchMethod@ endp
```

Listing 3

The flexibility of method dispatching being encapsulated in a function is nice because it permits application developers more choice as to the tradeoffs of memory consumption versus execution speed. For maximum speed, full size v-tables could be allocated - permitting direct indexing and thus time invariant method dispatching. *(The C++ `vtbl` approach is sometimes referred to as "static caching" on account every existing virtual member function has a function pointer residing in a `vtbl` entry ready to be invoked.)* For very large programs it may be preferable to use some form of global method selector caching *(where the number of entries in the global cache is something less than the total number of virtual member functions)* coupled with method look-up on cache miss [André92]. A dynamic scheme of either global or local caching would reduce the memory requirement for cache tables.

There is also the possibility of method dispatchers that offer various kinds of debugging assistance. Breakpointing of method calls, Eiffel-style *preconditions* and *postconditions* on objects [Meyer92], and dispatching-related diagnostic messages could be centralized in a special debugging version of `__DispatchMethod@`.

By making the method dispatcher a non-in-line function, it is only necessary to link an executable module to a different version of the compiler's method dispatch run time service in order to employ a different dispatch strategy.

6. *Lazy Resolution*

The most interesting novelty of the method dispatching approach described herein - other than compatibility to the goal of class protocol evolvability - is the manner in which it permits *lazy resolution* of a method invocation on its first occurrence. Lazy resolution is made possible by two conventions: **1)** the first method slot entry (the zero offset entry in a `vtbl` class subsection) is reserved for a special function pointer and **2)** a pointer to the `__MethodSelector` static structure variable, or vcall descriptor, is made available in a register *(the `EAX` register on the Intel 80386 CPU)* to the method dispatch service.

6-1. Method Dispatch Blow-By-Blow

The `methodIndex` field of the vcall descriptor is initialized to zero at compile time. When the method call that it is associated with is first executed, the method dispatch service will dereference to and invoke the zero-entry function pointer located in the `vtbl` class subsection. By convention the first entry of a class subsection is reserved. A function pointer to a method binding routine called, `__LookupMethod_glue@`, is placed in this entry.

Actually all class subsection slots are at first set to contain the `__LookupMethod_glue@` function pointer. This is done so that a virgin `vtbl` can be used in conjunction to an already resolved vcall descriptor, i.e., a vcall descriptor that has a `methodIndex` field that has been resolved to a non zero value. Resetting a `vtbl` to have all slots contain the `__LookupMethod_glue@` function pointer

would cause all the virtual function pointers to be reestablished in the `vtbl`*. Such a method code "hot updating" feature is an exceptionally intriguing capability.*

The `__LookupMethod_glue@` function gets invoked with the same contents in the `EDX:ECX:EAX` register trio as does the `__DispatchMethod@` function *(the* `__DispatchMethod@` *function preserves the* `EDX:ECX:EAX` *register trio)*. The `__LookupMethod_glue@` function is essentially an assembly language glue routine *(refer to its implementation in Listing 4 below)* that itself in turn calls the C++ implemented function, `__LookupMethod()`. The `__LookupMethod_glue@` routine, by being implemented in assembly language, is able to access the arguments passed to it in the `EDX:ECX:EAX` register trio.

```
                ; Parameters passed in registers on entry:
                ;
                ;       EDX         :       vptr (windowed vptr)
                ;       ECX         :       base vptr (non-windowed vptr)
                ;       EAX         :       ptr to vcall descriptor
                ;
                ; EDX:EAX registers are preserved on function exit
                ;
0011            __LookupMethod_glue@ proc near
0011 52            PUSH  EDX                        ; save vptr
0012 50            PUSH  EAX                        ; save vcall descriptor ptr

                                                    ; push arguments to function call:
0013 83 E9 04      SUB   ECX,4                      ;  (deref base vptr by -4 offset)
0016 FF 31         PUSH  DWORD PTR [ECX]            ;    metaclass ptr
0018 50            PUSH  EAX                        ;    ptr to vcall descriptor
0019 52            PUSH  EDX                        ;    vptr
001A E8 00000000e CALL  NEAR PTR ___LookupMethod
001F 83 C4 0C      ADD   ESP,12                     ; cut back function call args

0022 58            POP   EAX                        ; get vcall descriptor ptr
0023 5A            POP   EDX                        ; get vptr

0024 EB DA         JMP   __DispatchMethod@          ; complete the method call dispatch
0026            __LookupMethod_glue@ endp
```

Listing 4

The `__LookupMethod_glue@` function dereferences the *base* `vptr` in `ECX` in order to retrieve the metaclass object pointer - which is then pushed onto the stack. Then the `EDX` and `EAX` register values are also pushed onto the stack. All three arguments have now become conventional C++ function stack arguments. The `__LookupMethod_glue@` function then invokes the `__LookupMethod()` C++ function. Here is the C++ source code prototype of `__LookupMethod()`:

```
extern "C" void __LookupMethod(  __vtbl_entry **windowed_vptr,
                                 __MethodSelector *pMethSel,
                                 __MetaClass_Super *pMetaClass);
```

Now this function - implemented conveniently in a high level language - does the necessary look up *(the metaclass object is called on to perform the method look up operation)* of the method being invoked and as a side effect assigns a valid `vtbl`

method index value into the vcall descriptor static structure. The method call can now be dispatched to completion. As a final step `__LookupMethod_glue@` calls the `__DispatchMethod@` function to complete the method invocation *(with the `EDX:EAX` register pair holding the same values as before - the original `ECX` argument is now superfluous)*. This time the method call will dispatch directly to the appropriate virtual member function.

To reiterate, the `__LookupMethod()` function, via the pointer to a vcall descriptor static structure variable it is passed as an argument, sets the `methodIndex` field of this structure to a `vtbl` class subsection index value - it is the side effect of calling this function. It also puts a virtual member function pointer into the `vtbl` *(if it is not already in the `vtbl` due to some other previous method call that resulted in invoking the same virtual member function)*. Consequently, the next time the method call is executed, the method dispatching service will directly invoke the virtual member function pointer that is definitely now in the `vtbl` and that is dereferenced by the `methodIndex` and `classIndex` values.

6-2. Method dispatch recap

The first time that a method call is executed it must be resolved. All subsequent executions of the method call are invariant in execution time *(of course CPU instruction caching and virtual memory paging can cause wide variation in any application instruction sequence execution time)*. A method call, once resolved, remains resolved for the remaining duration of application execution *(although the possibility exist for metaclass objects to reset `vtbls` to initial conditions and thus provide a method code "hot updating" capability).* Method calls that exist in application code, but that are never executed in a particular run time session, are never resolved. The cost of method call resolution is paid only when, or if, the call is first exercised. Effectively, the execution time overhead of method call resolution is spread throughout an application's execution progress. This behavior should lessen potential negative impact upon application start-up time stemming from C++ evolvable class implementation support.

7. Undispatchable Method Calls

The down side of lazy resolution is that it is possible to attempt a method call that subsequently cannot be resolved and thus obstructs completing the method dispatch. The `__LookupMethod()` function will have to throw an exception to the application in the event that this occurs. To prevent such an undesirable scenario arising, class programmers should never remove virtual member function declarations when evolving classes into new generations. Obsolete methods should remain present in new class generations in order to provide backward compatibility to old program code that may still be "in the field" and still in use along side new program code. This rule can only be enforced through programmer convention under the lazy resolution approach.

Perhaps in the future C++ development systems and persistent databases of application class objects *(where class hierarchy and metaclass information exist in the database as schema information)* can become wedded together. The act of compiling a new generation of a class might result in it being verified against existing database class schema information. Doing so could catch situations where a programmer has inadvertently removed a class method and the compiler could issue a diagnostic message. The AT&T Grail C++ development environment [Murray92] [Koenig92] would appear to be headed in a technology direction that may ultimately permit this manner of class evolvability safety verification once that system is fully wedded to database technology.

7-1. Weighing some alternatives

Conceivably some other approach might be to attempt resolution of all application method calls at application load time. If any method call was determined to be unresolvable, then the application launch could be aborted. This would tend to minimize the consequences faced by a user in such an event *(and certainly give the application programmer less to have to hassle with).* Though the user would be left with the mystery of trying to figure out which application component has a C++ class generation that is no longer in compliance with its prior class generations. This mystery could be clarified by a sufficiently helpful error message diagnostic. *(However, the same could be said of the lazy resolution approach - the difference being a matter of chronology as to when the diagnostic message is encountered.)*

The obvious penalty here, though, is of having to incur the total accumulative cost of all method call resolution at one time. For very large object-oriented applications this could perhaps lead to major delays in initiating application execution. Too, it would require resolving every method call whether it is actually later executed or not - you would pay for it even if you never actually used it *(for significant applications we are talking about a matter of hundreds or thousands of method calls).* One other slight advantage of the *all up-front* approach is that after method call resolution is completed, any method symbol table data structures that were used during the resolving process can be freed. An incremental approach, such as lazy resolution, must retain such data structures for the duration of the program's execution.

An approach that resolves method calls when individual pages *(or segments)* of application code are loaded on demand presents no real advantage over the advocated lazy resolution technique either. In fact this approach bears much closer kinship with lazy resolution than the brute force, all up-front approach mentioned in the previous two paragraphs. It has the additional requirement, though, that the operating system provide a feature that enables applications to hook the process of loading pages *(or segments)* of their code into memory. *(Lazy resolution is not dependent on any special operating system feature so is thus highly portable.)* Too, this approach could likewise possibly encounter an unresolvable method call when well into the application's execution progress - the application here too would have to be thrown an exception resulting from an undispatchable method call.

Given the use of an incremental method-call-resolution approach, the problem of undispatchable method calls can be viewed in much the same vein as divide by zero exceptions that possibly can result from an application's numeric calculations. Programmers attempt to build fire walls for data input screening so as to prevent such exceptions arising. None-the-less, prudent programmers may also enable their applications to gracefully recover in the event a divide by zero exception still manages to happen. *(This is particularly wise for very large complex applications that may have a wide range of numeric calculations appearing in diverse areas of the program.)* Hopefully programmers will exert due diligence when evolving their classes into new generations. And hopefully they will imbue their application with graceful recovery *(or even graceful shutdown)* in the event an undispatchable method call exception is thrown. As always, the market place will no doubt sift out those applications that are lackadaisical in this regard.

8. Method Look Up In Finer Detail

Let us return now to the discussion of how method call dispatching takes place. We left off in the dispatch explanation where the C++ function `__LookupMethod()` is invoked by an unresolved method call. There are several steps that the `__LookupMethod()` function will have to go through in resolving a method call:

- The method will be searched for via the metaclass method look up service *(described in the paper preceding this one [Voss93] on subject of C++ evolvable classes)*. If the method cannot be found in the class's derivation hierarchy then the application is thrown an undispatchable method call exception. Otherwise, the found virtual member function pointer is copied into and retained in a local variable for use at a later stage.

- The class name component, after extraction from the mangled method name string, must be hashed into an application-global class name symbol table. *(This symbol table is managed privately by the method call dispatching service.)* All metaclass objects will have hashed their class's class name strings into the class symbol table when they are constructed *(at application start up time - fortunately the number of classes is much fewer than the number of methods or number of method calls)*. Consequently hashing a class name upon a method call will result in collision. The symbol table entry for a class will be a pointer to a class's method name registration table. *(Refer to the schematic of the class name symbol table in Figure 2.)*

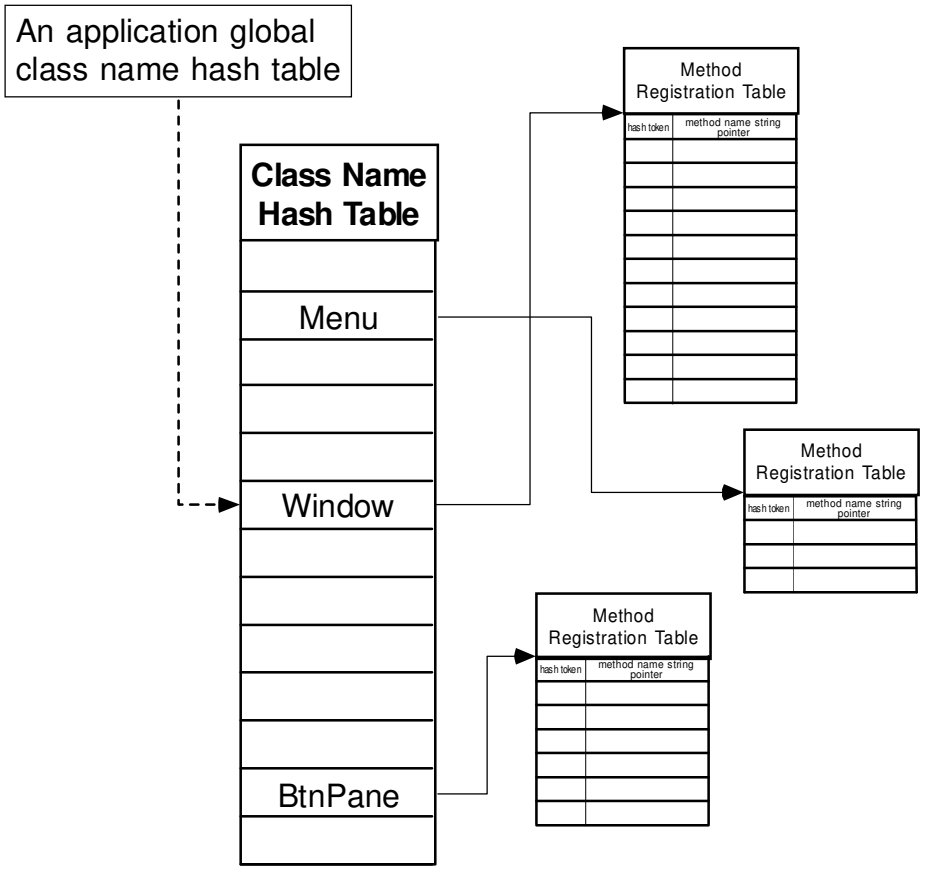


Figure 2

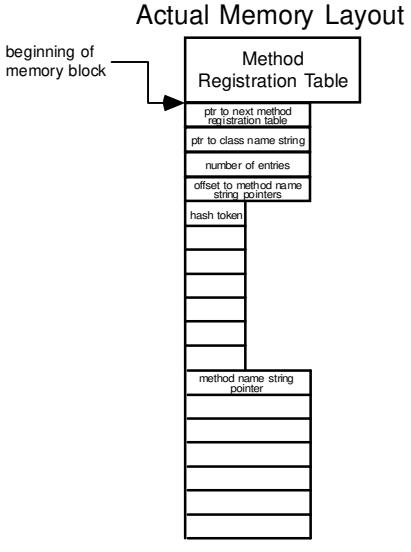


Figure 3

- The mangled method name must now have a hash value computed for it and this hash value is searched for in the method name registration table. If a method-name-hash-token collision occurs, then that entry is further compared to see if it is the same as the mangled method name string. If it exactly matches then this index position in the registration table is the index value that will be copied to the `methodIndex` field of the vcall decsriptor static structure. *(The memory layout of a typical method registration table is illustrated in Figure 3.)*

- If an exact match is not found in the search then the first available zero entry in the registration table will be taken. The hash value of the method name and a pointer to the method name string are copied into this entry - making it a non-zero entry.

  The class's metaclass object knows the number of entries that should be permissible in the method registration table. Thus it dynamically allocates this table and places a pointer to it in the class symbol table entry that the class name string hashes to.

- The class ID index value obtained from the `classIndex` field of the vcall descriptor is now used in conjunction to the `vptr` to dereference into the `vtbl` and obtain a pointer to the class subsection. *(Refer to figure 1 to see the layout schematic for a typical evolvable class `vtbl`.)*

  The virtual member function pointers that implement the method protocol of the class will be incrementally placed into its `vtbl` class subsection. A class's `vtbl` subsection has enough function pointer slots to account for every method protocol that is introduced by the class *(this number is known by a class's metaclass object and a metaclass object can be queried to obtain it - refer to the evolvable class paper [Voss93] preceding this one for more detail on this)*.

- The virtual member function pointer that was previously obtained and saved away must now be copied into its slot in the `vtbl` class subsection. The class ID index and method index values can now be used to dereference the appropriate `vtbl` slot in order to do this. Conceivably it may already be in the `vtbl` but to blindly copy it in again will do no harm. *(To "blindly copy" would be mandatory if metaclasses support resetting the `vtbl` to enable a method code "hot update" feature.)*

- Finally, `__LookupMethod()` can exit so that the method call dispatch can be completed.

9. Run Time Type Identification

One of the most recent and talked about C++ extension proposals is that of *run time type identification (RTTI)* [Stroustrup92]. The author's attention and efforts toward C++ evolvable classes subject were initiated at the beginning of 1992. The author's

development of ideas and subsequent papers on evolvable classes proceeded independent of efforts underway then toward investigating RTTI for C++. However, the author now concedes the possibility of synergy in the underlying implementation of RTTI and evolvable class support. Likely synergy of these two was even alluded to at some extent in the preceding evolvable class paper [Voss93]. That paper also expressed the opinion that RTTI would be highly desirable in conjunction to evolvable classes where these language capabilities are applied in concert in programming the Object-Oriented Component Application Architecture (OOCAA)[1].

None-the-less, the proposal for C++ method dispatching presented herein has been framed in the context of assuming RTTI support to not be present. Yet due to the dynamic nature of evolvable classes the implementation must rely upon compiler generated metaclass information. RTTI leads to another variation of requiring compiler generated metaclass *(or reflective)* information. If one is augmenting a compiler that already supports RTTI to also support evolvable classes, then it would be logical to merge and/or share the metaclass requirements of the two.

The author also concedes that the details of this proposal might stand need for revision if indeed RTTI support is already present in the particular C++ compiler being augmented. The validity of much of this proposal should probably hold up - such as the vcall descriptor mechanism and the concept of lazy resolution. The specific device of the class ID index that appears in the vcall descriptor may come under scrutiny, though.

10. Conclusion

A technique for implementing a time invariant virtual member function dispatch for C++ evolvable classes has been presented. The focus was to enable evolvable classes with an invariant dispatch that is highly comparable in efficiency to the dispatch of virtual functions for conventional classes. Economy in memory space, execution time, and compiler implementation complexity were of the utmost importance in attempting to arrive at this proposal. Concepts such as *lazy resolution* of method calls and non-in-lining of the method dispatch service have been adopted with the intent of striking a fair balance between all these concerns.

At the same time it was desired to introduce a certain amount of new freedom in C++ program configuration. Base class evolvability is certainly one of the most powerful expressions of new C++ flexibility. However, making the evolvable class method dispatcher a function callable service opens up additional degrees of freedom in: program performance tuning; method call debugging, *message traffic* monitoring, object state validation; and other feature extensions such as a class *pose as* capability as found in Objective C [Pinson91].

The author is also aware of existing C++ system designs that could capitalize on the method dispatching/binding technique described here to better implement object-

oriented data base concepts. Currently attention has been focused on placing object data into data base management. The next step will be to implement C++ compilers that compile C++ method code into a data base retrievable file structure. The methods of a class object could then begin to be treated as data as well. *Hot update* to class protocol could be performed by loading new method implementations into the data base *(where the data base identity of replaced method(s) remains the same),* and then cause an application's metaclass objects to reset `vtbl` entries for these methods. The application would then immediately begin executing with the new class behavior supplied by reinstalled methods.

Using the *method-as-data, hot updating* technique, it is also possible to imagine an application that configures itself by retrieving class objects *(and their methods)* from a network topology of object servers [Smith90]. Such an application would self-configure depending upon what a user has expressed a desire or intent to accomplish and what *object personalities* are at the application's disposal. Again, the dispatching scheme that has been presented would endow sufficient flexibility to undertake such an architecture vision and at the same time couple it to a rather efficient C++ run time.

Acknowledgements

I would like to acknowledge Peter Kukol and Tanj Bennett of Borland for their valuable critique and input toward my efforts of describing an evolvable class implementation for the C++ language. They have donated much valuable time and effort in reviewing my ideas, digesting my exhaustive email on the subject, and preparing their own proposal documents relating to this important C++ language issue. I would also like to cite credit to Peter Kukol for coining the term *vcall descriptor* which I have borrowed for use in this document. Peter Kukol also suggested the CFront compatible dispatch approach detailed in Appendix A. Lastly, I must also express my appreciation to my employer, Aldus Corporation, for recognizing the importance of this issue to the computer software development community at large and supporting my efforts toward popularizing the concern of C++ evolvable classes.

References

[Voss93] Roger D. Voss ‘‘C++ Evolvable Base Classes Residing In Dynamic Link Libraries,’’ To appear in *C++ Journal,* Vol. 2, No. 4 (1993).

[Ellis90] Margaret A. Ellis, Bjarne Stroustrup, *The Annotated C++ Reference Manual,* Addison-Wesley, Reading, MA, 1990.

[André92] Pascal André, Jean-Claude Royer, ‘‘Optimizing Method Search with Lookup Caches and Incremental Coloring,’’

Conference Proceedings of OOPSLA '92, October 18-22, 1992.

[Meyer92] Bertrand Meyer, *Eiffel: The Language,* Prentice Hall, New York, NY, 1992.

[Murray92] Robert B. Murray, ''A Statically Typed Abstract Representation for C++ Programs,'' *Usenix C++ Conference Proceedings,* USENIX Association (1992).

[Koenig92] Andrew Koenig, ''Space-Efficient Trees in C++,'' *Usenix C++ Conference Proceedings,* USENIX Association (1992).

[Stroustrup92] Bjarne Stroustrup, ''Run Time Type Identification for C++,'' *Usenix C++ Conference Proceedings,* USENIX Association (1992).

[Pinson91] Lewis J. Pinson, Richard S. Wiener, *Objective-C: Object-Oriented Programming Techniques,* Addison-Wesley, Reading, MA, 1991.

[Smith90] K. Stuart Smith, Arunodaya Chatterjee, ''A C++ Environment for Distributed Application Execution'' and ''Resource Allocation in Distributed Object-Oriented Systems: A Proposed Solution,'' Microelectronics and Computer Technology Corporation, Austin, TX (1990), MCC Technical Report Number: ACT-ESP-015-91.

Suggested Reading

Andrew J. Palay, ''C++ in a Changing Environment,'' *Usenix C++ Conference Proceedings,* USENIX Association (1992).

Marion Bucko, Larry K. Raper, ''OS/2 2.0 Technical Library System Object Model Guide and Reference,'' IBM Corporation, 1991, Document Number 10G6309.

Richard Johnson, Murugappan Palaniappan, ''MetaFlex: A Flexible Metaclass Generator,'' To appear in *Conference Proceedings of ECOOP '93,* July 26-30, 1993. (In meantime contact: Aldus Corporation, 411 First Avenue South, Seattle, WA 98104-2871.)

[1]Object-Oriented Component Application Architecture is a modular/dynamic approach to application design. Such an application will typically be built out of a foundation of shared libraries that export base classes. Such shared libraries *(DLLs under Windows and OS/2)* will usually consist of: application framework, graphics engine, data base/document engine, and various other shared, general services. The goal of the OOCAA approach is that it should be possible to upgrade and replace these shared libraries on an individual basis without adversely effecting existing applications and other shared libraries that are already deployed into the user base.

Another aspect of the OOCAA design approach is that applications will be extensible via ‘‘dropping in’’ new object-oriented components. This variety of executable component will technically be a shared library but will provide implementation classes that are derived from base classes that are in the domain of the application proper. The extending classes of these add-on components will essentially graft onto the overall class inheritance hierarchy at application load time and application run time. Again, if the base classes that these classes inherit from are evolved in new shared library releases, then extension components should not be adversely effected. In short, OOCAA is a microcosm of object-oriented operating system attributes but applied at the level of the individual application.

Appendix A

The method dispatching code generation description presented in the main body of this paper is oriented toward native C++ compiler implementations. It assumes that the C++ compiler back-end generates object code directly for the intended target CPU. The 80386 CPU is used as a standard reference CPU given its widespread use. To get maximum efficiency it is often desirable to precisely control object code generation for C++ language features. C++ exception handling is an example where native C++ compilers hold a distinct advantage over CFront-based compilers in this regard.

However, the evolvable class method dispatching code generation can also be easily expressed in high level C language, but with a bit more verbosity and hence a little less efficiency than that presented in the main body of this paper. Not only would this approach be amendable to CFont-based C++ compilers, it would also represent a shortcut route to implementing evolvable class support in native C++ compilers. This would be because all modifications could be made in the compiler front-end - no changes to the compiler back-end would be necessary.

This alternative technique is easily and briefly presented in the source code example below - which in turn is based on the source code example taken from Listing 1. In this approach, the `__DispatchMethod()` function receives its arguments passed on the stack instead of in registers. It returns the dereferenced virtual function pointer as its function result. The virtual function pointer result is immediately dereferenced and executed with the method call arguments passed to it on the call stack.

```
// synthesized declarations introduced prior to
// the beginning of each source translation unit
typedef struct __MethodSelector_tag {
   unsigned short  methodIndex;
   unsigned short  classIndex;
   char *       methodName;
   } __MethodSelector; // typedefing is for benefit of C

typedef struct __vtbl_entry_tag {
   void (*fct)();
   } __vtbl_entry;     // typedefing is for benefit of C

typedef int (*__vFunc)(void*,...);

// an ANSI C language prototype of the method dispatching routine
extern __vFunc __DispatchMethod( __vtbl_entry **_windowed_vptr,
                                 __vtbl_entry **_base_vptr,
                                 __MethodSelector *_vcall_descriptor_ptr );
```

```
/*** beginning of actual .cpp source translation unit ***/

Menu::Menu(VString* itemName,MenuID id,BOOL largeMenu,BOOL isPopUp) :
            MenuItem(itemName,MenuItemID(id))
{
   fHMenu = HMENU(0);
   fLongMenu = largeMenu;
   fIsPopUp = isPopUp;

   fMenuItemID = (id == kNoResourceMenuID) ? ++gNextCreatedMenuID : id;
   fMenuItems = (LList*)NULL;

   if (gKnownMenus == NULL)
      gKnownMenus = new Dictionary;
   if (gKnownMenus != NULL)
   {
      DWORDCItem* theKey = new DWORDCItem(fMenuItemID);
//    if (!gKnownMenus->Find(theKey))
      extern __MethodSelector __methSel_1;       // Collection::Find()
      if ( ! ((int)(__DispatchMethod(
                     *(((__vtbl_entry***)(Collection*)gKnownMenus) + 1),
                     *(((__vtbl_entry***)(Dictionary*)gKnownMenus) + 1),
                     &__methSel_1)
                        (gKnownMenus,theKey)))                    ) {

//       gKnownMenus->AddKeyValuePair(theKey,this);
         extern __MethodSelector __methSel_2;    // Dictionary::AddKeyValuePair()
         ((void)(__DispatchMethod(
                     *(((__vtbl_entry***)(Dictionary*)gKnownMenus) + 1),
                     *(((__vtbl_entry***)(Dictionary*)gKnownMenus) + 1),
                     &__methSel_2)
                        (gKnownMenus,theKey,this)));
   }
}

/*** ending of actual source translation unit ***/



// synthesized anonymous static variables
extern char __methNameStr_1[];
extern char __methNameStr_2[];

static __MethodSelector __methSel_1 = { 0, 3, __methNameStr_1 };
static __MethodSelector __methSel_2 = { 0, 5, __methNameStr_2 };

static char __methNameStr_1[] =
   "@meth@Collection@Find$qn5CItem";
static char __methNameStr_2[] =
   "@meth@Dictionary@AddKeyValuePair$qn5CItemt1";
```