C++ ABI Closed Issues

Revised 17 August 1999

Issue Status

In the following sections, the ***class*** of an issue attempts to classify it on the basis of what it likely affects. The identifiers used are:

call	Function call interface, i.e. call linkage
data	Data layout
lib	Runtime library support
lif	Library interface, i.e. API
g	Potential gABI impact
ps	Potential psABI impact
source	Source code conventions (i.e. API, not ABI)
tools	May affect how program construction tools interact

A. Object Layout Issues

#	Issue	Class	Status	Source	Opened	Closed
A-1	Vptr location	data	closed	SGI	990520	990624
Summary: Where is the Vptr stored in an object (first or last are the usual answers).						

[990610 All] Given the absence of addressing modes with displacements on IA-64, the consensus is to answer this question with "first."

[990617 All] Given a Vptr and only non-polymorphic bases, which (Vptr or base) goes at offset 0?

- HP: Vptr at end, but IA-64 is different because no load displacement
- Sun: Vptr at 0 probably preferred
- g++: Vptr at end today

Tentative decision: Vptr always goes at beginning.

[990624 All] Accepted tentative decision. Rename, close this issue, and open separate issue (B-6) for Vtable layout.

#	Issue	Class	Status	Source	Opened	Closed
A-2	Virtual base classes	data	closed	SGI	990520	990624
Summary: Where are the virtual base subobjects placed in the class layout? How are data member accesses to them handled?						

[990610 Matt] With regard to how data member accesses are handled, the choices are to store either a pointer or an

offset in the Vtable. The concensus seems to be to prefer an offset.

[990617 All] Any number of empty virtual base subobjects (rare) will be placed at offset zero. If there are no non-virtual polymorphic bases, the first virtual base subobject with a Vpointer will be placed at offset zero. Finally, all other virtual base subobjects will be allocated at the end of the class, left-to-right, depth-first.

[990624 All] Define an empty object as one with no non-static, non-empty data members, no virtual functions, no virtual base classes, and no non-empty non-virtual base classes. Define a nearly empty object as one which contains only a Vptr. The above resolution is accepted, restated as follows:

Any number of empty virtual base subobjects (rare, because they cannot have virtual functions or bases themselves) will be placed at offset zero, subject to the conflict rules in A-3 (i.e. this cannot result in two objects of the same type at the same address). If there are no non-virtual polymorphic base subobjects, the first nearly empty virtual base subobject will be placed at offset zero. Any virtual base subobjects not thus placed at offset zero will be allocated at the end of the class, in left-to-right, depth-first declaration order.

#	Issue	Class	Status	Source	Opened	Closed
A-3	Multiple inheritance	data	closed	SGI	990520	990701
Summary: Define the class layout in the presence of multiple base classes.						

[990617 All] At offset zero is the Vptr whenever there is one, as well as the primary base class if any (see A-7). Also at offset zero is any number of empty base classes, as long as that does not place multiple subobjects of the same type at the same offset. If there are multiple empty base classes such that placing two of them at offset zero would violate this constraint, the first is placed there. (First means in declaration order.)

All other non-virtual base classes are laid out in declaration order at the beginning of the class. All other virtual base subobjects will be allocated at the end of the class, left-to-right, depth-first.

The above ignores issues of padding for alignment, and possible reordering of class members to fit in padding areas. See issue A-9.

[990624 All] There remains an issue concerning the selection of the primary base class (see A-7), but we are otherwise in agreement. We will attempt to close this on 1 July, modulo A-7.

[990701 All] This issue is closed. A full description of the class layout can be found in issue A-9. (At this time, A-7 remains to be closed, waiting for the Taligent rationale.)

#	Issue	Class	Status	Source	Opened	Closed
A-4	Empty base classes	data	closed	SGI	990520	990624
Summary: Where are empty base classes allocated? (An empty base class is one with no non-static data members, no virtual functions, no virtual base classes, and no non-empty non-virtual base classes.)						

[990624 All] Closed as a duplicate of A-3.

#	Issue	Class	Status	Source	Opened	Closed
A-5	Empty parameters	data	closed	SGI	990520	990701
Summary: When passing a parameter with an empty class type by value, what is the convention?						

[990623 SGI] We propose that no parameter slot be allocated to such parameters, i.e. that no register be used, and that no space in the parameter memory sequence be used. This implies that the callee must allocate storage at a unique address if the address is taken (which we expect to be rare).

[990624 All] In addition to the address-taken case, care is required if the object has a non-trivial copy constructor. HP

observes that in (some?) such cases, they perform the construction at the call site and pass the object by reference.

[990625 SGI -- Jim] I understand that the Standard explicitly allows elimination of even non-trivial copy construction in some cases. Is this one of them? Where should I look? Also, of course, varargs processing for elided empty parameters would need to be careful.

I have opened a new issue (C-7) for passing copy-constructed parameters by reference. Since doing so would turn an empty value parameter into a non-empty reference parameter, this issue can ignore such cases.

[990701 All] An empty parameter will not occupy a slot in the parameter sequence unless:

1. its type is a class with a non-trivial copy constructor; or
2. it corresponds to the variable part of a varargs parameter list.

Daveed and Matt will pursue the question of when copy constructors may be ignored for parameters with the Core committee, and if they identify cases where the constructors may clearly be omitted, those (empty) parameters will also be elided.

#	Issue	Class	Status	Source	Opened	Closed
A-7	Vptr sharing with primary base class	data	closed	HP	990603	990729
Summary: It is in general possible to share the virtual pointer with a polymorphic base class (the *primary* base class). Which base class do we use for this?						
Resolution: Share with the first non-virtual polymorphic base class, or if none with the first nearly empty virtual base class.						

[990617 All] It will be shared with the first polymorphic non-virtual base class, or if none, with the first nearly empty polymorphic virtual base class. (See A-2 for the definition of *nearly empty*.)

[990624 All] HP noted that Taligent chooses a base class with virtual bases before one without as the primary base class), probably to avoid additional "this" pointer adjustments. SGI observed that such a rule would prevent users from controlling the choice by their ordering of the base classes in the declaration. The bias of the group remains the above resolution, but HP will attempt to find the Taligent rationale before this is decided.

[990729 All] Close with the agree resolution. If a convincing Taligent rationale is found, we can reconsider.

#	Issue	Class	Status	Source	Opened	Closed
A-8	(Virtual) base class alignment	data	closed	HP	990603	990624
Summary: A (virtual) base class may have a larger alignment constraint than a derived class. Do we agree to extend the alignment constraint to the derived class? (An alternative for virtual bases: allow the virtual base to move in the complete object.)						

[990623 SGI] We propose that the alignment of a class be the maximum alignment of its virtual and non-virtual base classes, non-static data members, and Vptr if any.

[990624 All] Above proposal accepted. (SGI observation: the size of the class is rounded up to a multiple of this alignment, per the underlying psABI rules.)

#	Issue	Class	Status	Source	Opened	Closed
A-9	Sorting fields as allowed by [class.mem]/12	data	closed	HP	990603	990624
Summary: The standard constrains ordering of class members in memory only if they are not separated by an access clause. Do we use an access clause as an opportunity to fill the gaps left by padding?						
Resolution: See separate writeup of Data Layout.						

[990610 all] Some participants want to avoid attempts to reorder members differently than the underlying C struct ABI rules. Others think there may be benefit in reordering later access sections to fill holes in earlier ones, or even in base classes.

[990617 all] There are several potential reordering questions, more or less independent:

1. Do we reorder whole access regions relative to one another?
2. Do we attempt to fill padding in earlier access regions with initial members from later regions?
3. Do we fill the tail padding of non-POD base classes with members from the current class?
4. Do we attempt to fill interior padding of non-POD base classes with later members?

There is no apparent support for (1), since no simple heuristic has been identified with obvious benefits. There is interest in (2), based on a simple heuristic which might sometimes help and will never hurt. However, it is not clear that it will help much, and Sun objects on grounds that they prefer to match C struct layout. Unless someone is interested enough to implement and run experiments, this will be hard to agree upon. G++ has implemented (3) as an option, based on specific user complaints. It clearly helps HP's example of a base class containing a word and flag, with a derived class adding more flags. Idea (4) has more problems, including some non-intuitive (to users) layouts, and possibly complicating the selection of bitwise copy in the compiler.

[990624 all] We will not do (1), (2), or (4). We will do (3). Specifically, allocation will be in modified declaration order as follows:

1. Vptr if any, and the primary base class per A-7.
2. Any empty base classes allocated at offset zero per A-3.
3. Any remaining non-virtual base classes.
4. Any non-static data members.
5. Any remaining virtual base classes.

Each subobject allocated is placed at the next available position that satisfies its alignment constraints, as in the underlying psABI. This is interpreted with the following special cases:

1. The "next available position" after a non-POD class subobject (base class or data member) with tail padding is at the beginning of the tail padding, not after it. (For POD objects, the tail padding is not "available.")
2. Empty classes are considered to have alignment and size 1, consisting solely of one byte of tail padding.
3. Placement on top of the tail padding of an empty class must avoid placing multiple subobjects of the same type at the same address.

After allocation is complete, the size is rounded up to a multiple of alignment (with tail padding).

[990722 all] The precise placement of empty bases when they don't fit at offset zero remains imprecise in the above. Accordingly, a precise layout algorithm is described in a separate writeup of Data Layout.

[990729 all] The layout writeup was accepted, with the first choice for empty base placement. That is, if placement at offset zero doesn't work, it will be placed like a normal base/member. The concensus was that this won't happen often, and such bases will often overlap with the preceding tail padding or following components anyway. Jim will modify the writeup accordingly.

#	Issue	Class	Status	Source	Opened	Closed
A-10	Class parameters in registers	call	closed	HP	990603	990710
Summary: The C ABI specifies that structs are passed in registers. Does this apply to small non-POD C++ objects passed by value? What about the copy constructor and `this` pointer in that case?						

[990701 all] A separate issue (C-7) deals with cases where a non-trivial copy constructor is required; we ignore those cases here. Our conclusion is that, without a non-trivial copy constructor, we need not be concerned about the class object moving in the process of being passed, and there is no need to use a mechanism different from the base ABI C struct mechanism. At the same time, if we do use the underlying C struct mechanism, the user has complete control of the passing technique, by choosing whether to pass by value or reference/pointer.

Therefore, except in cases identified by issue C-7 for different treatment, class parameters will be passed using the underlying C struct protocol.

#	Issue	Class	Status	Source	Opened	Closed
A-11	Pointers to member functions	data	closed	Cygnus	990603	990812
Summary: How should pointers to member functions be represented?						
Resolution: As a pair of values, described below.						

[990729 All] Jason described the g++ implementation, which is a three-member struct:

1. The adjustment to *this*.
2. The Vtable index plus one of the function, or -1. (Zero is a NULL pointer.)
3. If (2) is an index, the offset from the full object to the member function's Vtable. If -1, a pointer to the function (non-virtual).

A concern about covariant returns was raised. It was observed that, given our decision to use distinct Vtable entries for distinct return types, no further concern is required here. Others will describe their representations. IBM has an alternative, but it is believed to be patented by Microsoft.

[990805 All] It is agreed that a two-element struct will be used for a pointer to a member function, with elements as follows:

`ptr`:
: For a non-virtual function, this field is a simple function pointer. (Under current base IA-64 psABI conventions, this is a pointer to a GP/function address pair.) For a virtual function, it is 1 plus twice the Vtable offset of the function. The value zero is a NULL pointer.

`adj`:
: The required adjustment to *this*.

Although we agreed to close this, SGI suggests a minor modification. Since the Vtable offset of a virtual function will always be even, we suggest that it not be doubled before adding 1. This is because shifts are more restricted on many processors than other integer ALU operations (shifters are large structures), so an XOR or NAND will often be cheaper than a right shift.

[990812 All] Close this issue with the suggested modification.

#	Issue	Class	Status	Source	Opened	Closed
A-12	Merging secondary vtables	data	closed	Sun	990610	990805
Summary: Sun merges the secondary Vtables for a class (i.e. those for non-primary base classes) with the primary Vtable by appending them. This allows their reference via the primary Vtable entry symbol, minimizing the number of external symbols required in linking, in the GOT, etc.						
Resolution: Concatenate the Vtables associated with a class in the same order that the corresponding base subobjects are allocated in the object.						

[990701 Michael Lam] Michael will check what the Sun ABI treatment is and report back.

[990729 All] A separate issue raised in conjunction with A-7 is whether to include Vfunc pointers in the primary Vtable for functions defined only in the base classes and not overridden. If the primary and secondary Vtables are concatenated, this is no longer an issue, since all can be referenced from the primary Vptr.

[990805 All] All of the Vtables associated with a class will be concatenated, and a single external symbol used (to be identified as part of the mangling issue F-1). The order of the tables will be the same as the order of base class subobjects in an object of the class, i.e. first the primary Vtable, then the non-virtual base classes in declaration order, and finally the virtual base classes in depth-first declaration order.

#	Issue	Class	Status	Source	Opened	Closed
A-13	Parameter struct field promotion	call	closed	SGI	990603	990701
Summary: It is possible to pass small classes either as memory images, as is specified by the base ABI for C structs, or as a sequence of parameters, one for each member. Which should be done, and if the latter, what are the rules for identifying "small" classes?						
Resolution: No special treatment will be specified by the ABI.						

[990701 all] Define no special treatment for this case in the ABI. A translator with control over both caller and callee may choose to optimize.

#	Issue	Class	Status	Source	Opened	Closed
A-14	Pointers to data members	data	closed	SGI	990729	990805
Summary: How should pointers to data members be represented?						
Resolution: Represented as one plus the offset from the base address.						

[990729 SGI] We suggest an offset from the base address of the class, represented as a **`ptrdiff_t`**.

[990805 All] Such pointers are represented as one plus the offset from the base address of the class, as a **`ptrdiff_t`**. NULL pointers are zero.

B. Virtual Function Handling Issues

#	Issue	Class	Status	Source	Opened	Closed
B-2	Covariant return types	call	closed	SGI	990520	990722

Summary: There are several methods for adjusting the 'this' pointer of the returned value for member functions with covariant return types. We need to decide how this is done. Return thunks might be especially costly on IA64, so a solution based on returning multiple pointers may prove more interesting.

Resolution: Provide a separate Vtable entry for each return type.

[990610 Matt] One possibility is to have two Vtable entries, which might point to different functions, different entrypoints, or a real entrypoint and a thunk. Another is to return two result pointers (base/derived), and have the caller select the right one.

[990715 All] Daveed presented his multiple-return-value scheme, including an example that involved virtual base classes, return values that are pointers to nonpolymorphic classes, and other equally horrible things.

Consensus: we need to get the horrible cases correct, but speed only matters in the simple case. The simple case: class B has a virtual function f returning a B1* and class D has a virtual function f returning a D1*, where all four classes are polymorphic, B is a primary base of D, and B1 is a primary base of D1. (The really important case is where B1 is B and D1 is D, but that simplification doesn't make any difference.)

Jason: Would the usual multiple-entry-point scheme work just as well? That is, would it be just as fast as Daveed's scheme in the simple case, and still preserve enough information for the more complicated cases? It appears so, but we don't have a proof. Jason will try to provide one.

[990716 Cygnus -- Jason] Proof? You always know what types a given override must be able to return, and you know how to convert from the return type to those base types. You know from the entry point which type is desired. Seems pretty straightforward to me.

[990716 Cygnus -- Jason] The alternative I was talking about yesterday goes something like this:

When we have a non-trivial covariant return situation, we create a new entry in the vtable for the new return type. The caller chooses which vtable entry to use based on the type they want.

This could be implemented several ways, at the discretion of the vendor:

1. Multiple entry points to one function, with an internal flag indicating which type to return.

2. Thunks which intercept the function's return and modify the return value. Note that unlike the case of calling virtual functions, for covariant returns we always know which adjustments will be needed, so we don't have to pay for a long branch. We do, however, lose the 1-1 correspondence between calls and returns, which apparently affects performance on the Pentium Pro.

3. Function duplication.

The advantage of this approach to the complex case is that we don't have to do a dynamic_cast when faced with multiple levels of virtual derivation. It is also strictly simpler; Daveed's model already requires something like this in cases of multiple inheritance.

Of course, we can always mix and match; we could choose to only do this in cases of virtual inheritance, or use Daveed's proposal and do this only in cases of repeated virtual inheritance. In that case, the multiple returns would just be an optimization for the single virtual inheritance case.

Since we don't seem to care about the performance of anything but single nonvirtual inheritance, it seems simpler not to bother with multiple returns.

The remaining question is how to handle the case of nontrivial nonvirtual inheritance: do we use multiple slots or have the caller do the adjustment? My inclination is to have the caller adjust.

WRT patents, the idea of having the function return the base-most class and having the caller adjust is parallel to the patented Microsoft scheme whereby they pass the base-most class as the 'this' argument to virtual functions, but the word

'return' does not appear anywhere in the patent, so it seems safe.

[990722 All] The group was generally agreed that the simplicity of multiple entries in the vtable outweighed any space/performance advantage of more complex schemes (e.g. the method Daveed described on 15 July). Discussion focussed on whether it is worthwhile to eliminate some of the entries in cases where they are unnecessary because the caller knows the required conversion, namely when the return type has a unique non-virtual subobject of the original return type.

Agreement was reached to avoid the complication of eliminating some of the Vtable entries. Thus, the Vtable will have one entry for each accessible return type of a covariant virtual function. These may be implemented in a variety of ways, e.g. duplicated functions, separate entrypoints, or stubs, and the ABI need not specify the choice. The location of the Vtable entries is part of the separate Vtable layout issue B-6.

#	Issue	Class	Status	Source	Opened	Closed
B-3	Allowed caching of vtable contents	call	closed	HP	990603	990805
Summary: The contents of the vtable can sometimes be modified, but the concensus is that it is nonetheless always allowed to "cache" elements, i.e. to retain them in registers and reuse them, whenever it is really useful. However, this may sometimes break "beyond the standard" code, such as code loading a shared library that replaces a virtual function. Can we all agree when caching is allowed?						
Resolution : Caching is allowed.						

[990604 HP -- Christophe] Mike (Ball) gave me what I believe is an excellent definition of when caching is allowed. I'd like him to present it.

[990805 All] Christophe explained that the rule is simply that, within a call to a member function of the class, the class Vtable may not be modified. Between such calls, no assumption may be made. With this observation, the issue is closed.

[990812 All] The rule is even simpler. Once a program changes the type of a pointer's target, the pointer is invalidated, and its value may not be reused. Therefore, a code sequence which repeatedly refers to the same pointer value is invalid if the pointee's vtable has been changed.

#	Issue	Class	Status	Source	Opened	Closed
B-4	Function descriptors in vtable	data	closed	HP	990603	990805
Summary: For a runtime architecture where the caller is expected to load the GP of the callee (if it is in, or may be in, a different DSO), e.g. HP/UX, what should vtable entries contain? One possibility is to put a function address/GP pair in the vtable. Another is to include only the address of a thunk which loads the GP before doing the actual call.						
Resolution : The Vtable will contain a function address/GP pair.						

[990624 All] Note that putting GP in the Vtable prevents putting it in shared memory. See B-7.

[990805 All] It was decided that special representations to accomodate shared memory would be expensive and therefore undesirable. Therefore, the decision is to put the function address/GP pair in the vtable, avoiding the cost of an extra indirection in using it.

#	Issue	Class	Status	Source	Opened	Closed
B-7	Objects and Vtables in shared memory	data	closed	HP	990624	990805
Summary: Is it possible to allocate objects in shared memory? For polymorphic objects, this implies that the Vtable must also be in shared memory.						
Resolution : No special representation is useful in support of shared memory.						

[990624 All] Note that putting GP in the Vtable prevents putting it in shared memory. This interacts with B-4.

[990624 HP -- Cary] For a C++ object to be placed into shared memory, its vtable pointer must be valid in all processes that are sharing that object.

1. If the vtable can be placed in text, that would be fine, but the vtable contains function pointers (or descriptors) that require runtime relocation, so it must be in data.

2. We can place the vtables in shared memory, but only if the function pointers/descriptors are valid in all processes. The entry point addresses, which refer to shared text, should be shareable, but the gp values may not be identical for all processes. (RTTI pointers are also an issue, and could be solved by putting the RTTI information in shared memory as well.)

3. We can place the vtables in private memory, provided they are at the same address in all processes.

One way or another, we need a way of ensuring that a pointer from shared memory to private memory is valid in all processes, which means that we will need a means to ensure that certain shared library data segments can get mapped at the same address in all processes that load those certain libraries.

My wild idea a few years ago was to put the vtables in shared memory (by allocating and building them at load time, as Taligent did), and store a shared library index in place of the gp value in each function descriptor. Each process would have its own table of gp values, indexed by this shared library index, but the index space would be managed system-wide. The C++ runtime library would have been responsible for allocating a new index for each unique C++ shared library loaded on the system, then storing the process-local copy of the gp pointer in the appropriate slot of the table.

[990628 SGI -- Jim] Note a further problem with vtables in shared memory (Cary's point 2). If a virtual function comes from another DSO, it may be pre-empted differently in different programs. Hence, the function pointer itself is a problem even if the GP isn't.

[990701 All] An extensive discussion boiled down to a few points:

- The primary issue is objects in shared memory -- vtables aren't interesting in themselves, but rather because putting the object in shared memory implies having the vtable at the same address in all sharing processes.

- Many of us have a few customers asking for this. It is not clear just how extensive a facility they need, or how automatic it needs to be. We should attempt to gauge the need.

- Noone thinks we should penalize the non-shared case for the rare instances of shared demand.

- It is questionable whether we can define an ABI mechanism which will work on all of our systems, but we'd like not to preclude OS-specific extensions to do this if we can't.

- One possible approach would be an API allowing a user to place an object in shared memory, and then "install" it by setting its vtable pointers, possibly to copies also placed in the same shared memory.

- A more automatic approach would be something which allocated certain objects/vtables to shared memory, gave up at link time if not all pointers were internal to the object being linked, attempted to place the relevant segments at the same runtime addresses to allow sharing, and gave up on sharing if this was not possible. Such an approach would perhaps still require some care on the part of the user to prevent problematic runtime situations.

These ideas are very fuzzy. Participants should think about the need and possibilities and attempt to identify more concrete approaches.

[990805 All] It was determined (largely based on consideration by Jason) that the only practical approach to putting objects in shared memory is to force the objects, Vtables, functions, etc. to the same addresses in the various processes involved. If this is done, data representation issues are irrelevant. Therefore, this issue is closed as moot.

Note that the base psABI defines a flag, EF_IA_64_ABSOLUTE, which forces an executable object to the addresses specified in ELF, so at least one method of representing this is already available.

C. Object Construction/Destruction Issues

#	Issue	Class	Status	Source	Opened	Closed
C-7	Passing value parameters by reference	call	closed	All	990624	990805
Summary: It may be desirable in some cases where a type has a non-trivial copy constructor to pass value parameters of that type by performing the copy at the call site and passing a reference.						
Resolution : Whenever a class type has a non-trivial copy constructor, pass value parameters of that type by performing the copy at the call site and passing a reference.						

[990701 All] Daveed and Matt will attempt to pin down the copy requirements with the Core committee, i.e. when a non-trivial copy constructor may be elided. The relevant Standard requirement is 12.8/15, and there is an open defect report related to this question. For cases where the ctor may not be elided, we expect to perform the copy at the call site, and pass a reference.

[990729 All] Matt will produce a clear proposal for when the ABI will elide the constructor (and therefore pass the class object like a normal C struct), based on the Standard's exceptions.

[990805 All] There are no cases where a non-trivial copy constructor can be simply elided for all instances of a particular parameter. Therefore, we shall use the consistent convention that, if a value parameter's (class) type has a non-trivial copy constructor, the caller will allocate space for it, perform the copy, and pass a reference.

Note that the standard does allow the caller, if the value being passed is a temporary, to construct the temporary directly into the parameter memory and elide the copy constructor call.

#	Issue	Class	Status	Source	Opened	Closed
C-8	Returning classes with non-trival copy constructors	call	closed	All	990625	990722
Summary: How do we return classes with non-trivial copy constructors?						
Resolution: The caller allocates space, and passes a pointer as an implicit first parameter (prior to the implicit *this* parameter).						

D. Exception Handling Issues

E. Template Instantiation Model Issues

F. Name Mangling Issues

G. Miscellaneous Issues

H. Library Interface Issues

Please send corrections to Jim Dehnert.