C++ ABI for IA-64: Data Layout

Revised 9 September 1999

Revisions

[990811] Described member pointer representations, virtual table layout.

[990730] Selected first variant for empty base allocation; removed others.

General

In what follows, we define the memory layout for C++ data objects. Specifically, for each type, we specify the following information about an object O of that type:

- the *size* of an object, *sizeof*(O);
- the *alignment* of an object, *align*(O); and
- the *offset* within O, *offset*(C), of each data component C, i.e. base or member.

For purposes internal to the specification, we also specify the *data size* of an object, *dsize*(O), which intuitively is sizeof(O) minus the size of tail padding.

Definitions

The descriptions below make use of the following definitions:

alignment of a type T (or object X)
: A value A such that any object X of type T has an address satisfying the constraint that &X modulo A == 0.

empty class
: A class with no non-static data members, no virtual functions, no virtual base classes, and no non-empty non-virtual base classes.

nearly empty class
: A class, the objects of which contain only a Vptr.

polymorphic class
: A class requiring a virtual table pointer (because it or its bases have one or more virtual member functions or virtual base classes).

primary base class
: For a polymorphic class, the unique base class (if any) with which it shares the Vptr at offset 0.

POD Data Types

The size and alignment of C POD types is as specified by the base (C) ABI. Type bool has size and alignment 1. All of these types have data size equal to their size. (We ignore tail padding for PODs because the Standard does not allow us to use it for anything else.)

Member Pointers

A pointer to data member is an offset from the base address of the class object containing it, represented as a **`ptrdiff_t`**. It has the size, data size, and alignment of a **`ptrdiff_t`**.

A pointer to member function is a pair as follows:

`ptr`:
: For a non-virtual function, this field is a simple function pointer. (Under current base IA-64 psABI conventions, that 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*, represented as a **`ptrdiff_t`**.

It has the size, data size, and alignment of a class containing those two members, in that order. (For 64-bit IA-64, that will be 16, 16, and 8 bytes respectively.)

Non-POD Class Types

For non-POD class types C, assume that all component types (i.e. base classes and non-static data member types) have been laid out, defining size, data size, and alignment. Layout (of type C) is done using the following procedure.

I. **Initialization**

1. Initialize sizeof(C) to zero, align(C) to one, dsize(C) to zero.
2. If C is a polymorphic type:
    a. If C has a polymorphic base class, attempt to choose a primary base class B. It is the first non-virtual polymorphic base class, if any, or else the first nearly empty virtual base class. Allocate it at offset zero, and set sizeof(C) to sizeof(B), align(C) to align(B), dsize(C) to dsize(B).
    b. Otherwise, allocate the vtable pointer for C at offset zero, and set sizeof(C), align(C), and dsize(C) to the appropriate values for a pointer (all 8 bytes for IA-64 64-bit ABI).

II. **Non-Virtual-Base Allocation**

For each data component D (i.e. base or non-static data member) except virtual bases, first the non-virtual base classes in declaration order and then the non-static data members in declaration order, allocate as follows:

1. If D is not an empty base class, start at offset dsize(C), incremented if necessary to alignment align(type(D)). Place D at this offset unless doing so would result in two components (direct or indirect) of the same type having the same offset. If such a component type conflict occurs, increment the candidate offset by align(type(D)), and try again, repeating until success occurs (which will occur no later than sizeof(C) incremented to the required alignment).

    Update sizeof(C) to max (sizeof(C), offset(D)+sizeof(D)). Update align(C) to max (align(C), align(D)). If D is a base class (not empty in this case), update dsize(C) to offset(D)+dsize(D). If D is a data member, update dsize(C) to max (offset(D)+dsize(D), offset(D)+1).

2. If D is an empty base class, its allocation is similar to the first case above, except that additional candidate offsets are considered before starting at dsize(C). First, attempt to place D at offset zero. If unsuccessful (due to a component type conflict), proceed with attempts at dsize(C) as for non-empty bases. As for that case, if there is a type conflict at dsize(C) (with alignment updated as necessary), increment the candidate offset by align(type(D)), and try again, repeating until success occurs.

Once offset(D) has been chosen, update sizeof(C) to max (sizeof(C), offset(D)+sizeof(D)). Note that align(D) is 1, so no update of align(C) is needed. Similarly, since D is an empty base class, no update of dsize(C) is needed.

III. **Virtual Base Allocation**

Finally allocate any virtual base classes (except one selected as the primary base class in I-2a, if any) as we did non-virtual base classes in step II-1, in declaration order. Update sizeof(C) to max (sizeof(C), offset(D)+sizeof(D)). If non-empty, also update align(C) and dsize(C) as in II-1.

IV. **Finalization**

Round sizeof(C) up to a non-zero multiple of align(C).

Virtual Table Layout

General

A *virtual table* (*vtable*) is a table of information used to dispatch virtual functions, to access virtual base class subobjects, and to access information for runtime type identification (RTTI). Each class that has virtual member functions or virtual bases has an associated set of vtables. There may be multiple vtables for a particular class, if it is used as a base class for other classes. However, the vtable pointers within all the objects (instances) of a particular most-derived class point to the same set of vtables.

A vtable consists of a sequence of offsets, data pointers, and function pointers, as well as structures composed of such items. We will describe below the sequence of such items. Their offsets within the vtable are determined by that allocation sequence and the natural ABI size and alignment, just as a data struct would be. In particular:

- Offsets are of type **`ptrdiff_t`** unless otherwise stated.
- Data pointers have normal pointer size and alignment.
- Function pointers remain to be defined. One possibility is that they will be <function address, GP address> pairs, with pointer alignment.

In general, what we consider the address of a vtable (i.e. the address contained in objects pointing to a vtable) may not be the beginning of the vtable. We call it the *address point* of the vtable. The vtable may therefore contain components at either positive or negative offsets from its address point.

Components

Each vtable consists of the following components:

- *Base offsets* are used to access the virtual bases of an object, or non-virtual bases when pointer adjustment is required for some overriden virtual function. Such an entry is a displacement to a virtual base subobject from the location within the object of the vtable pointer that addresses this vtable. These entries are only necessary if the class directly or indirectly inherits from virtual base classes, or if it overrides a virtual function defined in a base class and if adjustment from the base to the derived class is needed. The values can be positive or negative.

- The *offset to top* holds the displacement to the top of the object from the location within the object of the vtable pointer that addresses this vtable, as a **`ptrdiff_t`**. A negative value indicates the vtable pointer is part of an embedded base class subobject; otherwise it is zero. The offset provides a way to find the top of the object from any base subobject with a vtable pointer. This is necessary for dynamic_cast in particular.

  The virtual pointer points to the "offset to top" field.

- The *typeinfo pointer* points to the typeinfo object used for RTTI. All entries in each of the vtables for a given class point to the same typeinfo object.

- The *duplicate base information pointer* points to a table used to perform runtime disambiguation of duplicate base classes for dynamic casts. The entries in the vtables for a given class do not necessarily all point to the same table of duplicate base information.

- *Virtual function pointers* are used for virtual function dispatch. Each pointer holds either the address of a virtual function of the class (or the address of a secondary entry point that performs certain adjustments before transferring control to a virtual function.) In the case of shared library builds, a virtual function pointer entry contains a pair of components (each 64 bits in the 64-bit IA-64 ABI): the value of the target GP value and the actual function address. That is, rather than being a normal function pointer, which points to such a two-component descriptor, a virtual function pointer entry is the descriptor.

- *Base vtables* are copies of the vtables for base classes of the current class (copies in the sense that they have the same layout, though virtual function pointers may point to overriding functions).

Virtual Table Order

A virtual table's components are laid out in the following order, analogous to the corresponding object layout.

1. If virtual base offsets are required, they come first, with ordering as defined in categories 3 and 4 below. *Question:* Is the characterization of which bases need offsets above adequate? *Note:* The non-virtual cases need to be reflected in the descriptions below of categories 3 and 4.

2. The vtable address point points here, i.e. this is the address of the vtable contained in an object's vptr.

3. The offset to top field is next (therefore at the vtable address), if required (always if there is a virtual function declared in the class).

4. The typeinfo pointer field is next. It is always present.

5. The duplicate base information pointer comes next, if required.

6. Virtual function pointers come next, in order of declaration of the corresponding member function in the class. They appear only if there is no suitable entry in one of the base vtables, i.e. if the return type is overridden or if it is not declared in a base. *Question:* Is this a correct characterization?

7. The base vtables are last, except for the primary base class vtable, which is a subtable of the derived class vtable. They are in order of base class declaration, with the vtables for a virtual base class coming after the vtable for the first base class defining it. *Question:* Can we just use the first vtable for the virtual base class embedded in another of the base vtables?

Virtual Table Construction

In this section, we describe how to construct the vtable for an class, given vtables for all of its base classes. To do so, we divide classes into several categories, based on their base class structure.

Category 0: Trivial

Structure:

- No inherited base classes.
- No virtual functions.

Such a class has no associated vtable, and its objects contain no vptr.

Category 1: Leaf

Structure:

- No inherited base classes.
- Declares virtual functions.

The vtable contains an RTTI field followed by virtual function pointers. There is one function pointer entry for each virtual function declared in the class.

Category 2: Non-Virtual Bases

Structure:

- Only non-virtual base classes.
- Base classes are trivial or leaf (they have no base classes).
- No adjustment is required in any overridden virtual functions.

If none of the base classes have vtables, a vtable is constructed for the class according to the Category 1 rules.

Otherwise, the class has a vtable for each base class that has a vtable. The class's vtables are constructed from copies of the base class vtables. The entries are the same, except:

- The RTTI fields contain information for the class, rather than for the base class.
- The function pointer entries for virtual functions inherited from the base class and overridden by this class are replaced with the addresses of the overriding functions (or the corresponding adjustor secondary entry points).
- At negative offsets, offsets to the base classes are generated if used by adjustor secondary entry points.

For a base class **`Base`**, and a derived class **`Derived`** for which we are constructing this set of vtables, we shall refer to the vtable for **`Base`** as **`Base-in-Derived`**. The vptr of each base subobject of an object of the derived class will point to the corresponding base vtable in this set.

The vtable copied from the primary base class is also called the primary vtable; it is addressed by the vtable pointer at the top of the object. The other vtables of the class are called secondary vtables; they are addressed by vtable pointers inside the object.

Following the function pointer entries that correspond to those of the primary base class, the primary vtable holds the following additional entries at its tail:

- Entries for virtual functions introduced by this class.
- Entries for overridden virtual functions not already in the vtable. (These are also called replicated entries because they are already in the secondary vtables of the class.) *Question:* Are these really needed?

The primary vtable, therefore, has the base class functions appearing before the derived class functions. The primary vtable can be viewed as two vtables accessed from a shared vtable pointer.

Note: Another benefit of replicating virtual function entries is that it reduces the number of this pointer adjustments during virtual calls. Without replication, there would be more cases where the this pointer would have to be adjusted to access a secondary vtable prior to the call. These additional cases would be exactly those where the function is overridden in the derived class, implying an additional thunk adjustment back to the original pointer. Thus replication saves two adjustments for each virtual call to an overridden function introduced by a non-primary base class.

Category 3: Virtual Bases Only

Structure:

- Only virtual base classes.
- Base classes are not empty or nearly empty.
- Base classes are trivial or leaf (they have no base classes).

The class has a vtable for each virtual base class that has a vtable. These are all secondary vtables and are constructed from copies of the base class vtables according to the same rules as in Category 2. The vtable pointers of the virtual base subobjects within the object address these vtables.

The class also has a vtable that is not copied from the virtual base class vtables. This vtable is the primary vtable of the class and addressed by the vtable pointer at the top of the object, which is not shared. It holds the following function pointer entries:

- Entries for virtual functions introduced by this class.
- Entries for overridden virtual functions. (These are also called replicated entries, because they are already in the secondary vtables of the class.)

The primary vtable also has virtual base offset entries to allow finding the virtual base subobjects. There is one virtual base

offset entry for each virtual base class. For a class that inherits only virtual bases, the entries are in the reverse order in which the virtual bases appear in the class declaration, that is, the entry for the leftmost virtual base is closest to the address point of the vtable.

Category 4: Complex

Structure:

- None of the above, i.e. directly or indirectly inherits both virtual and non-virtual base classes, or at least one nearly empty virtual base class.

The rules for constructing vtables of the class are a combination of the rules from Categories 2 and 3, and for the most part can be determined inductively. However the rules for placing virtual base offset entries in the vtables requires elaboration.

The primary vtable has virtual base offset entries for all virtual bases directly or indirectly inherited by the class. Each secondary vtable has entries only for virtual bases visible to the corresponding base class. The entries in the primary vtable are ordered so that entries for virtual bases visible to the primary base class are placed closest to the vtable address point, i.e. at higher addresses, while entries for virtual bases only visible to this class are further from the vtable address point, i.e. at lower addresses.

For virtual bases only visible to this class, the entries are in the reverse order in which the virtual bases are encountered in a depth-first, left-to-right traversal of the inheritance graph formed by the class definitions. Note that this does not follow the order that virtual bases are placed in the object.

Issues:

- *A-6: RTTI representation.*
- *A-6: Duplicate base structure representation.*
- *B-2: Will contain one entry per return type for covariant returns.*

External Names (a.k.a. Mangling)

<To be specified.>

Please send corrections to Jim Dehnert.