C++ ABI for IA-64: Data Layout

Revised 6 October 1999

Contents

Revisions

[991006] Added RTTI proposal.

[990930] Updated to new vtable layout proposal.

[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, 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. The values can be positive or negative.

- *vcall offsets* are used to perform pointer adjustment for overridden virtual functions. These entries are allocated when the class is used as a virtual base. They are referenced by the virtual functions to find the necessary adjustment from the base to the derived class, if any. These values may 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 *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.

- *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.

- *Secondary 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).

The virtual pointer in the object points to the first virtual function pointer.

Virtual Table Order

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

1. If vcall offsets are required, they come first, with ordering as defined in categories 3 and 4 below.

2. If virtual base offsets are required, they come next, with ordering as defined in categories 3 and 4 below.

3. The offset to top field is next. It is present if the class has virtual functions. Question: Should we include the RTTI fields for classes with no virtual functions (only virtual bases), too?

4. The typeinfo pointer field is next. It is present if the class has virtual functions.

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

6. Virtual function pointers come next, in order of declaration of the corresponding member function in the class. They appear both for newly introduced functions and overridden functions.

7. The secondary vtables are last. They are laid out in the same order used for the bases themselves in the object.

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 virtual base classes.
- No virtual functions.

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

Category 1: Leaf

Structure:

- No inherited virtual functions.
- No virtual 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.
- Inherits virtual functions.

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).

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.)

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.

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, except that the vtable for a virtual base A also includes a vcall offset entry for each virtual function represented in A's primary vtable and the secondary vtables from A's non-virtual bases. The vcall offset entries are allocated from the inside out, in the same order as the functions appear in A's 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 is 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 at increasing negative offsets from the address point of the vtable in the 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: Duplicate base structure representation.*

- *B-2: Will contain one entry per return type for covariant returns.*

Vtables During Object Construction (open issue C-4)

In some situations, a special vtable called a construction vtable is used during the execution of base class constructors and destructors. These vtables are for specific cases of virtual inheritance.

During the construction of a class object, the object assumes the type of each of its base classes, as each base class subobject is constructed. RTTI queries in the base class constructor will return the type of the base class, and virtual calls will resolve to member functions of the base class rather than the complete class. Normally, this behavior is accomplished by setting, in the base class constructor, the object's vtable pointers to the addresses of the vtables for the base class.

However, if the base class has direct or indirect virtual bases, the vtable pointers have to be set to the addresses of construction vtables. This is because the normal base class vtables may not hold the correct virtual base index values to access the virtual bases of the object under construction, and adjustment addressed by these vtables may hold the wrong this parameter adjustment if the adjustment is to cast from a virtual base to another part of the object. The problem is that a complete object of a base class and a complete object of a derived class do not have virtual bases at the same offsets.

A construction vtable holds the virtual function addresses and the RTTI information associated with the base class and the virtual base indexes and the addresses of adjustor entry points with this parameter adjustments associated with objects of the complete class.

To ensure that the vtable pointers are set to the appropriate vtables during base class construction, a table of vtable pointers, called the VTT, which holds the addresses of construction and non-construction vtables is generated for the complete class. The constructor for the complete class passes to each base class constructor a pointer to the appropriate place in the VTT where the base class constructor can find its set of vtables. Construction vtables are used in a similar way during the execution of base class destructors.

Run-Time Type Information (RTTI)

The C++ programming language definition implies that information about types be available at run time for three distinct purposes:

a. to support the typeid operator,
b. to match an exception handler with a thrown object, and
c. to implement the dynamic_cast operator.

(c) only requires type information about polymorphic class types, but (a) and (b) may apply to other types as well; for example, when a pointer to an int is thrown, it can be caught by a handler that catches "int const*".

Deliberations

The following conclusions were arrived at by the attending members of the C++ IA-64 ABI group:

- The exact layout for type_info objects is dependent on whether a 32-bit or 64-bit model is supported.
- Advantage should be taken of COMDAT sections and symbol preemption: two type_info pointers point to equivalent types if and only if the pointers are equal.
- A simple dynamic_cast algorithm that is efficient in the common case of base-to-most-derived cast case is preferrable over more sophisticated ideas that handle deep-base-to-in-between-derived casts more efficiently at a slight cost to the common case. Hence, the original scheme of providing a hash-table into the list of base classes (as is done e.g. in the HP aC++ compiler) has been dropped.

- The GNU egcs development team has implemented an idea of this ABI group to accelerate dynamic_cast operations by a-posteriori checking a "likely outcome". The interface of std::__dynamic_cast therefore keeps the src2dst_offset hint.

- std::__extended_type_info is dropped.

Place of emission

It is probably desirable to minimize the number of places where a particular bit of RTTI is emitted. For polymorphic types, a similar problem occurs for virtual function tables, and hence the information can be appended at the end of the primary vtable for that type. For other types, they must presumably be emitted at the location where their use is implied: the object file containing the typeid, throw or catch.

Basic type information (such as for "int", "bool", etc.) can be kept in the run-time support library. Specifically, this proposal is to place in the run-time support library type_info objects for the following types:

- void*, void const*; and
- X, X* and X const*, for every X in: bool, wchar_t, char, unsigned char, signed char, short, unsigned short, int, unsigned int, long, unsigned long, long long, unsigned long long, float, double, long double.

(Note that various other type_info objects for class types may reside in the run-time support library by virtue of the preceding rules; e.g., that of std::bad_alloc.)

The typeid operator

The typeid operator produces a reference to a std::type_info structure with the following public interface:

```
struct std::type_info {
   virtual ~type_info();
   bool operator==(type_info const&) const;
   bool operator!=(type_info const&) const;
   bool before(type_info const&) const;
   char const* name() const;
};
```

Assuming that after linking and loading only one type_info structure is active for any particular type symbol, the equality and inequality operators can be written as address comparisons: to type_info structures describe the same type if and only if they are the same structure (at the same address). In a flat address space (such as that of the IA-64 architecture), the before() member is also easily written in terms of an address comparison. The only additional piece of information that is required is the NTBS that encodes the name. The type_info structure itself can hold a pointer into a read-only segment that contains the text bytes.

Matching throw expressions with handlers

When an object is thrown a copy is made of it and the type of that copy is TT. A handler that catches type HT will match that throw if:

- HT is equal to TT except that HT may be a reference and that HT may have top-level cv qualifiers (i.e., HT can be "TT cv", "TT&" or "TT cv&"); or
- HT is a reference to a public and unambiguous base type of TT; or
- HT has a pointer type to which TT can be converted by a standard pointer conversion (though only public, unambiguous derived-to-base conversions are permitted) and/or a qualification conversion.

This implies that the type information must keep a description of the public, unambiguous inheritance relationship of a type, as well as the const and volatile qualifications applied to types.

The dynamic_cast operator

Although dynamic_cast can work on pointers and references, from the point of view of representation we need only to worry about polymorphic class types. Also, some kinds of dynamic_cast operations are handled at compile time and do not need any RTTI. There are then three kinds of truly dynamic cast operations:

- dynamic_cast, which returns a pointer to the complete lvalue,
- dynamic_cast operation from a base class to a derived class, and
- dynamic_cast across the hierarchy which can be seen as a cast to the complete lvalue and back to a sibling base.

RTTI layout

1. The RTTI layout for a given type depends on whether a 32-bit or 64-bit mode is in effect.

2. Every vtable shall contain one entry describing the offset from a vptr for that vtable to the origin of the object containing that vptr (or equivalently: to the vptr for the primary vtable). This entry is directly useful to implement dynamic_cast, but is also needed for the other truly dynamic casts. This entry is located two words ahead of the location pointed to by the vptr (i.e., entry "-2").

3. Every vtable shall contain one entry pointing to an object derived from std::type_info. This entry is located at the word preceding the location pointed to by the vptr (i.e., entry "-1").

   std::type_info contains just two pointers:
   - its vptr
   - a pointer to a NTBS representing the name of the type

   The possible derived types are:
   - std::__fundamental_type_info
   - std::__pointer_type_info
   - std::__reference_type_info
   - std::__array_type_info
   - std::__function_type_info
   - std::__enum_type_info
   - std::__class_type_info
   - std::__ptr_to_member_type_info

4. std::fundamental_type_info adds no fields to std::type_info;

5. std::__pointer_type_info adds two fields (in this order):

   - a word describing the cv-qualification of what is pointed to (e.g., "int volatile*" should have the "volatile" bit set in that word)

   - a pointer to the std::type_info derivation for the unqualified type being pointed to

   Note that the first bits should not be folded into the pointer because we may eventually need more qualifier bits (e.g. for "restrict"). The bit 0x1 encodes the "const" qualifier; the bit 0x2 encodes "volatile".

6. std::__reference_type_info is similar to std::__pointer_type_info but describes references.

7. std::__array_type_info and std::__function_type_info do not add fields to std::type_info (these types are only produced by the typeid operator; they decay in other contexts). std::__enum_type_info does not add fields either.

8. std::__class_type_info introduces a variable length structure. The variable part that follows consists of a sequence of base class descriptions having the following structure:

```
struct std::__base_class_info {

   std::type_info *type; /* Null if unused. */
   std::ptrdiff_t offset;
```

```
        int is_direct: 1;
        int is_floating: 1; /* I.e., virtual or base of virtual subobject. */
        int is_virtual: 1; /* Implies is_floating. */
        int is_shared: 1; /* Implies is_floating and the virtual subobject
                             appears on multiple derivation paths. */
        int is_accessible: 1;
        int is_ambiguous: 1;
    };
```

(Ed. note: to avoid endianness confusion, the above should be defined in terms of a flags field and masks.)

9. The std::__ptr_to_member_type_info type adds two fields to std::type_info:
    - a pointer to a std::__class_type_info (e.g., the "A" in "int A::*")
    - a_pointer to a std::type_info corresponding to the member type (e.g., the "int*" in "int A::*")

std::type_info::name()

The NTBS returned by this routine is the mangled name of the type.

The dynamic_cast algorithm

Dynamic casts to "void cv*" are inserted inline at compile time. So are dynamic casts of null pointers and dynamic casts that are really static.

This leaves the following test to be implemented in the run-time library for truly dynamic casts of the form "dynamic_cast(v)": (see [expr.dynamic_cast] 5.2.7/8)

- If, in the most derived object pointed (referred) to by v, v points (refers) to a public base class sub-object of a T object [note: this can be checked at compile time], and if only one object of type T is derived from the sub-object pointed (referred) to by v, the result is a pointer (an lvalue referring) to that T object.
- Otherwise, if v points (refers) to a public base class sub-object of the most derived object, and the type of the most derived object has an unambiguous public base class of type T, the result is a pointer (an lvalue referring) to the T sub-object of the most derived object.
- Otherwise, the run-time check fails.

The first check corresponds to a "base-to-derived cast" and the second to a "cross cast". These tests are implemented by std::__dynamic_cast:

```
void* std::__dynamic_cast ( void *sub,
                            std::__class_type_info *src,
                            std::__class_type_info *dst,
                            std::ptrdiff_t src2dst_offset);
/* sub: source address to be adjusted; nonnull, and since the
 *      source object is polymoprhic, *(void**)sub is a vptr.
 * src: static type of the source object.
 * dst: destination type (the "T" in "dynamic_cast(v)").
 * src2dst_offset: a static hint about the adjustment needed
 *    on sub; since this adjustment cannot be 1, 2 or 3,
 *    those special values mean:
 *       1: no hint
 *       2: src is not a public base of dst
 *       3: src is a multiple public base type but never a
 *          virtual base type
 *    otherwise, the src type is a unique public nonvirtual
 *    base type of dst at offset -src2dst_offset from the
 *    origin of dst.
 */
```

The exception handler matching algorithm

Since the RTTI related exception handling routines are "personality specific", no interfaces need to be specified in this document (beyond the layout of the RTTI data).

External Names (a.k.a. Mangling)

<To be specified.>

Please send corrections to Jim Dehnert.