
Itanium C++ ABI (Revision: 1.83)

Contents

- [Acknowledgements](#)
- [Chapter 1: Introduction](#)
 - [1.1 Definitions](#)
 - [1.2 Limits](#)
 - [1.3 Namespace and Header](#)
 - [1.4 Scope of This ABI](#)
 - [1.5 Base Documents](#)
- [Chapter 2: Data Layout](#)
 - [2.1 General](#)
 - [2.2 POD Data Types](#)
 - [2.3 Member Pointers](#)
 - [2.4 Non-POD Class Types](#)
 - [2.5 Virtual Table Layout](#)
 - [2.6 Virtual Tables During Object Construction](#)
 - [2.7 Array Operator *new* Cookies](#)
 - [2.8 Initialization Guard Variables](#)
 - [2.9 Run-Time Type Information \(RTTI\)](#)
- [Chapter 3: Function Calling Conventions and APIs](#)
 - [3.1 Non-virtual Function Calling Conventions](#)
 - [3.2 Virtual Function Calling Conventions](#)
 - [3.3 Construction and Destruction APIs](#)
 - [3.4 Demangler API](#)
- [Chapter 4: Exception Handling](#)
- [Chapter 5: Linkage and Object Files](#)
 - [5.1 External Names \(a.k.a. Mangling\)](#)
 - [5.2 Vague Linkage](#)
 - [5.3 Unwind Table Location](#)
- [Appendix R: Revision History](#)

Acknowledgements

This document was developed jointly by an informal industry coalition consisting of (in alphabetical order) CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI. Additional contributions were provided by a variety of individuals.

Chapter 1: Introduction

In this document, we specify the Application Binary Interface for C++ programs, that is, the object code interfaces between user C++ code and the implementation-provided system and libraries. This includes the memory layout for C++ data objects, including both predefined and user-defined data types, as well as internal

compiler generated objects such as virtual tables. It also includes function calling interfaces, exception handling interfaces, global naming, and various object code conventions.

In general, this document is written as a generic specification, to be usable by C++ implementations on a variety of architectures. However, it does contain processor-specific material for the Itanium 64-bit ABI, identified as such. Where structured data layout is described, we generally assume Itanium psABI member sizes. An implementation for a 32-bit ABI would typically just change the sizes of members as appropriate (i.e. pointers and long ints would become 32 bits), but sometimes an order change would be required for compactness, and we note more substantive changes.

1.1 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 $\&X \bmod A == 0$.

base class of a class T

When this document refers to base classes of a class T, unless otherwise specified, it means T itself as well as all of the classes from which it is derived, directly or indirectly, virtually or non-virtually. We use the term *proper base class* to exclude T itself from the list.

base object destructor of a class T

A function that runs the destructors for non-static data members of T and non-virtual direct base classes of T.

complete object destructor of a class T

A function that, in addition to the actions required of a base object destructor, runs the destructors for the virtual base classes of T.

deleting destructor of a class T

A function that, in addition to the actions required of a complete object destructor, calls the appropriate deallocation function (i.e., `operator delete`) for T.

direct base class order

When the direct base classes of a class are viewed as an ordered set, the order assumed is the order declared, left-to-right.

diamond-shaped inheritance

A class has diamond-shaped inheritance iff it has a virtual base class that can be reached by distinct inheritance graph paths through more than one direct base.

dynamic class

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

empty class

A class with no non-static data members other than zero-width bitfields, no virtual functions, no virtual base classes, and no non-empty non-virtual proper base classes.

inheritance graph

A graph with nodes representing a class and all of its subobjects, and arcs connecting each node with its direct bases.

inheritance graph order

The ordering on a class object and all its subobjects obtained by a depth-first traversal of its inheritance graph, from the most-derived class object to base objects, where:

- No node is visited more than once. (So, a virtual base subobject, and all of its base subobjects, will be visited only once.)
- The subobjects of a node are visited in the order in which they were declared. (So, given `class A : public B, public C`, A is walked first, then B and its subobjects, and then C and its subobjects.)

Note that the traversal may be preorder or postorder. Unless otherwise specified, preorder (derived classes before their bases) is intended.

morally virtual

A subobject X is a *morally virtual* base of Y if X is either a virtual base of Y, or the direct or indirect base of a virtual base of Y.

nearly empty class

A class that contains a virtual pointer, but no other data except (possibly) virtual bases. In particular, it:

- has no non-static data members other than zero-width bitfields,
- has no direct base classes that are not either empty, nearly empty, or virtual,
- has at most one non-virtual, nearly empty direct base class, and
- has no proper base class that is empty, not morally virtual, and at an offset other than zero.

Such classes may be primary base classes even if virtual, sharing a virtual pointer with the derived class.

POD for the purpose of layout

In general, a type is considered a POD for the purposes of layout if it is a POD type (in the sense of ISO C++ [basic.types]). However, a POD-struct or POD-union (in the sense of ISO C++ [class]) with a bitfield member whose declared width is wider than the declared type of the bitfield is not a POD for the purpose of layout. Similarly, an array type is not a POD for the purpose of layout if the element type of the array is not a POD for the purpose of layout. Where references to the ISO C++ are made in this paragraph, the Technical Corrigendum 1 version of the standard is intended.



The ISO C++ standard published in 1998 had a different definition of POD types. In particular, a class with a non-static data member of pointer-to-member type was not considered a POD in C++98, but is considered a POD in TC1. Because the C++ standard requires that compilers not overlay the tail padding in a POD, using the C++98 definition in this ABI would prevent a conforming compiler from correctly implementing the TC1 version of the C++ standard. Therefore, this ABI uses the TC1 definition of POD.

primary base class

For a dynamic class, the unique base class (if any) with which it shares the virtual pointer at offset 0.

secondary virtual table

The instance of a virtual table for a base class that is embedded in the virtual table of a class derived from it.

thunk

A segment of code associated (in this ABI) with a target function, which is called instead of the target function for the purpose of modifying parameters (e.g. `this`) or other parts of the environment before transferring control to the target function, and possibly making further modifications after its return. A thunk may contain as little as an instruction to be executed prior to falling through to an immediately

following target function, or it may be a full function with its own stack frame that does a full call to the target function.

vague linkage

The treatment of entities -- e.g. inline functions, templates, virtual tables -- with external linkage that can be defined in multiple translation units, while the ODR requires that the program behave as if there were only a single definition.

virtual table (or vtable)

A dynamic class has an associated table (often several instances, but not one per object) which contains information about its dynamic attributes, e.g. virtual function pointers, virtual base class offsets, etc.

virtual table group

The primary virtual table for a class along with all of the associated secondary virtual tables for its proper base classes.

1.2 Limits

Various representations specified by this ABI impose limitations on conforming user programs. These include, for the 64-bit Itanium ABI:

- The offset of a non-virtual base subobject in the full object containing it must be representable by a 56-bit signed integer (due to RTTI implementation). This implies a practical limit of 2^{55} bytes on the size of a class.

1.3 Namespace and Header

This ABI specifies a number of type and function APIs supplemental to those required by the ISO C++ Standard. A header file named `cxxabi.h` will be provided by implementations that declares these APIs. The reference header file included with this ABI definition shall be the authoritative definition of the APIs.

These APIs will be placed in a namespace `__cxxabiv1`. The header file will also declare a namespace alias `abi` for `__cxxabiv1`. It is expected that users will use the alias, and the remainder of the ABI specification will use it as well.

In general, API objects defined as part of this ABI are assumed to be extern "C++". However, some (many?) are specified to be extern "C" if they:

- are expected to be called by users from C code, e.g. `longjmp_unwind`; or
- are expected to be called only implicitly by compiled code, and are likely to be implemented in C.

1.4 Scope of This ABI

1.4.1 Runtime Libraries

The objective of a full ABI is to allow arbitrary mixing of object files produced by conforming implementations, by fully specifying the **binary interface** of application programs. We do not fully achieve this objective.

There are two principal reasons for this:

- I. We start from the Itanium processor-specific ABI as the standard for the underlying C interfaces. At this time, however, the psABI does not attempt to specify the supported C library interfaces.
- II. More fundamental is the definition of the Standard C++ Library. As the standard interface makes heavy use of templates, most user object files will end up with embedded template instantiations. Vendors are allowed to use helper functions and data in their implementations of these templates, and quite reasonably do so, with the result that a typical user object file will contain references to such helper objects specific to the implementation where compiled. We have not attempted to constrain the interface at this level, because we do not consider doing so feasible at this time.

Notwithstanding these problems, because this ABI does completely specify the data model and certain library interfaces that inherently interact between objects (e.g. construction, destruction, and exceptions), it is our intent that interoperability of object files produced by different compilers be possible in the following cases:

- A program which uses only the standalone standard library interfaces (Chapter 18) does not depend on the problematic template features.
- Since the standard library headers for an implementation presumably match the interfaces of the standard library on that implementation, a program compiled with the target system's headers, even if a mixture of compilers is used, should function properly on that system.

Even these cases can fail if the compiler makes use of implementation-defined library interfaces to implement runtime functionality without explicit user reference, e.g. a software divide function. We can distinguish between:

- the standard support library, which provides interfaces required by the C++ Standard Library specification and the vendor header files required for it, as well as interfaces required by this ABI; and
- the implicit compiler support library, which provides other interfaces implicitly assumed by the compiler and used to implement either standard features or extensions.

An implementation shall place its standard support library in a DSO named `libcxa.so` on Itanium systems, or in auxiliary DSOs automatically loaded by it. It shall place implicit compiler support in a library separate from the standard support library, with any external names chosen to avoid conflicts between vendors (e.g. by including a vendor identifier as part of the names). This allows a program to function properly if linked with the target's standard support library and the implicit compiler support libraries from any implementations used to build components.

1.4.2 Export Templates

This ABI does not specify the treatment of export templates, as there are no working implementations to serve as models at this time. We hope to address this weakness in the future when implementation experience is available.

1.5 Base Documents

A number of other documents provide a basis on which this ABI is built, and are occasionally referenced herein:

- [gABI] The **System V Application Binary Interface**, otherwise known as the *Generic ABI*. This document describes processor-independent object file formats and binary software interfaces for C under Unix. A somewhat out-of-date version is available from the SCO website, <http://www.caldera.com/developers/devspecs/>. A newer version, produced in conjunction with the next document, should be released in the future. Included by reference in this ABI.

- [psABI] The Intel **Unix System V Application Binary Interface, Itanium Processor Supplement**. This document describes Itanium processor-specific object file formats and binary software interfaces, primarily for C, under Unix. Available from the Intel Itanium software developer website, <http://developer.intel.com/design/ia-64/devinfo.htm>. Included by reference in this ABI.
- [SWCONV] The Intel **Itanium Software Conventions and Runtime Architecture Guide**. This document describes Itanium processor-specific binary software interfaces, notably including register usage, subprogram calling conventions, and stack unwind facilities, under all systems. Available from the Intel Itanium software developer website, <http://developer.intel.com/design/ia-64/devinfo.htm>. Included by reference in this ABI.
- [ABI-EH] The **C++ ABI for Itanium: Exception Handling**. Its Level II is considered an integral part of this document (Chapter 4). It also contains the base specification of unwind support for [psABI].
- [C++FDIS] The **Final Draft International Standard, Programming Language C++**, ISO/IEC FDIS 14882:1998(E). References herein to the "C++ Standard," or to just the "Standard," are to this document.

Chapter 2: Data Layout

2.1 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:

- *dsize(O)*: the *data size* of an object, which is the size of *O* without tail padding.
- *nvsiz(O)*: the *non-virtual size* of an object, which is the size of *O* without virtual bases.
- *nvalign(O)*: the *non-virtual alignment* of an object, which is the alignment of *O* without virtual bases.

2.2 POD Data Types

The size and alignment of a type which is a [POD for the purpose of layout](#) is as specified by the base (C) ABI. Type `bool` has size and alignment 1. All of these types have data size and non-virtual size equal to their size. (We ignore tail padding for PODs because the Standard does not allow us to use it for anything else.)

2.3 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 and alignment attributes of a `ptrdiff_t`. A NULL pointer is represented as -1.

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 Itanium psABI conventions, that is a pointer to a GP/function address pair.) For a virtual function, it is 1 plus the virtual table offset (in bytes) of the function, represented as a `ptrdiff_t`. The value zero represents a NULL pointer, independent of the adjustment field value below.

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 Itanium, that will be 16, 16, and 8 bytes respectively.)

2.4 Non-POD Class Types

For a class type C which is not a [POD for the purpose of layout](#), assume that all component types (i.e. proper base classes and non-static data member types) have been laid out, defining size, data size, non-virtual size, alignment, and non-virtual alignment. (See the description of these terms in [General](#) above.) Further, assume for data members that `nvsz==size`, and `nvaln==align`. Layout (of type C) is done using the following procedure.

I. Initialization

1. Initialize `sizeof(C)` to zero, `align(C)` to one, `dsz(C)` to zero.
2. If C is a dynamic class type:
 - a. Identify all virtual base classes, direct or indirect, that are primary base classes for some other direct or indirect base class. Call these *indirect primary base classes*.
 - b. If C has a dynamic base class, attempt to choose a primary base class B. It is the first (in direct base class order) non-virtual dynamic base class, if one exists. Otherwise, it is a nearly empty virtual base class, the first one in (preorder) inheritance graph order which is not an indirect primary base class if any exist, or just the first one if they are all indirect primaries.
 - c. If C has no primary base class, allocate the virtual table pointer for C at offset zero, and set `sizeof(C)`, `align(C)`, and `dsz(C)` to the appropriate values for a pointer (all 8 bytes for Itanium 64-bit ABI).



Case (2b) above is now considered to be an error in the design. The use of the first indirect primary base class as the derived class' primary base does not save any space in the object, and will cause some duplication of virtual function pointers in the additional copy of the base classes virtual table.

*The benefit is that using the derived class virtual pointer as the base class virtual pointer will often save a load, and no adjustment to the *this* pointer will be required for calls to its virtual functions.*

*It was thought that 2b would allow the compiler to avoid adjusting *this* in some cases, but this was incorrect, as the [virtual function call algorithm](#) requires that the function be looked up through a pointer to a class that defines the function, not one that just inherits it. Removing that requirement would not be a good idea, as there would then no longer be a way to emit all thunks with the functions they jump to. For instance, consider this example:*

```
struct A { virtual void f(); };
struct B : virtual public A { int i; };
```

```
struct C : virtual public A { int j; };
struct D : public B, public C {};
```

When B and C are declared, A is a primary base in each case, so although vcall offsets are allocated in the A-in-B and A-in-C vtables, no this adjustment is required and no thunk is generated. However, inside D objects, A is no longer a primary base of C, so if we allowed calls to C::f() to use the copy of A's vtable in the C subobject, we would need to adjust this from C to B::A*, which would require a third-party thunk. Since we require that a call to C::f() first convert to A*, C-in-D's copy of A's vtable is never referenced, so this is not necessary.*

II. Allocation of Members Other Than Virtual Bases

For each data component D (first the primary base of C, if any, then the non-primary, non-virtual direct base classes in declaration order, then the non-static data members and unnamed bitfields in declaration order), allocate as follows:

1. If D is a (possibly unnamed) bitfield whose declared type is τ and whose declared width is n bits:

There are two cases depending on `sizeof(τ)` and n :

- a. If `sizeof(τ)*8` $\geq n$, the bitfield is allocated as required by the underlying C psABI, subject to the constraint that a bitfield is never placed in the tail padding of a base class of C.

If `dsz(C) > 0`, and the byte at offset `dsz(C) - 1` is partially filled by a bitfield, and that bitfield is also a data member declared in C (but not in one of C's proper base classes), the next available bits are the unfilled bits at offset `dsz(C) - 1`. Otherwise, the next available bits are at offset `dsz(C)`.

Update `align(C)` to `max(align(C), align(τ))`.

- b. If `sizeof(τ)*8` $< n$, let T' be the largest integral POD type with `sizeof(T')*8` $\leq n$. The bitfield is allocated starting at the next offset aligned appropriately for T' , with length n bits. The first `sizeof(τ)*8` bits are used to hold the value of the bitfield, followed by $n - \text{sizeof}(\tau)*8$ bits of padding.

Update `align(C)` to `max(align(C), align(T'))`.

In either case, update `dsz(C)` to include the last byte containing (part of) the bitfield, and update `sizeof(C)` to `max(sizeof(C), dsz(C))`.

2. If D is not an empty base class or D is a data member:

Start at offset `dsz(C)`, incremented if necessary for alignment to `nalign(D)` for base classes or to `align(D)` for data members. 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 `nalign(D)` for base classes or by `align(D)` for data members and try again, repeating until success occurs (which will occur no later than `sizeof(C)` rounded up to the required alignment).

If D is a base class, this step allocates only its non-virtual part, i.e. excluding any direct or indirect virtual bases.

If D is a base class, update `sizeof(C)` to `max(sizeof(C), offset(D)+nvsize(D))`. Otherwise, if D is a data member, update `sizeof(C)` to `max(sizeof(C), offset(D)+sizeof(D))`.

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

3. If D is an empty proper base class:

Its allocation is similar to case (2) 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 `nvalign(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 `nvalign(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.

After all such components have been allocated, set `nvalign(C) = align(C)` and `nvsizesize(C) = sizeof(C)`. The values of `nvalign(C)` and `nvsizesize(C)` will not change during virtual base allocation. Note that `nvsizesize(C)` need not be a multiple of `nvalign(C)`.

III. Virtual Base Allocation

Finally allocate any direct or indirect virtual base classes (except the primary base class or any indirect primary base classes) as we did non-virtual base classes in step II-2 (if not empty) or II-3 (if empty), in inheritance graph order. Update `sizeof(C)` to `max (sizeof(C), offset(D)+nvsizesize(D))`. If non-empty, also update `align(C)` and `dsize(C)` as in II-2.

The primary base class has already been allocated in I-2b. Any indirect primary base class E of the current class C, i.e. one that has been chosen as the primary base class of some other base class (direct or indirect, virtual or non-virtual) of C, will be allocated as part of that other base class, and is not allocated here. If E is a primary base class of more than one other base, the instance used as its allocation in C shall be the first such in the inheritance graph order.

Consider:

```
struct R { virtual void r (); };
struct S { virtual void s (); };
struct T : virtual public S { virtual void t (); };
struct U : public R, virtual public T { virtual void u (); };
```

R is the primary base class for U since it is the first direct non-virtual dynamic base. Then, since an inheritance-order walk of U is { U, R, T, S } the T base is allocated next. Since S is a primary base of T, there is no need to allocate it separately. However, given:

```
struct V : public R, virtual public S, virtual public T {
    virtual void v ();
};
```

the inheritance-order walk of V is { V, R, S, T }. Nevertheless, although S is considered for allocation first as a virtual base, it is not allocated separately because it is a primary base of T, another base. Thus `sizeof (V) == sizeof (U)`, and the full layout is equivalent to the C struct:

```
struct X {
```

```

    R r;
    T t;
};

```

IV. Finalization

Round `sizeof(C)` up to a non-zero multiple of `align(C)`. If `C` is a POD, but not a POD for the purpose of layout, set `nvsized(C) = sizeof(C)`.

2.5 Virtual Table Layout

2.5.1 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 virtual tables. There may be multiple virtual tables for a particular class, if it is used as a base class for other classes. However, the virtual table pointers within all the objects (instances) of a particular most-derived class point to the same set of virtual tables.

A virtual table 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 virtual table 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 virtual table (i.e. the address contained in objects pointing to a virtual table) may not be the beginning of the virtual table. We call it the *address point* of the virtual table. The virtual table may therefore contain components at either positive or negative offsets from its address point.

2.5.2 Virtual Table Components and Order

This section describes the usage and relative order of various components that may appear in virtual tables. Precisely which components are present in various possible virtual tables is specified in the next section. If present, components are present in the order described, except for the exceptions specified.

- *Virtual call (vcall) offsets* are used to perform pointer adjustment for virtual functions that are declared in a virtual base class or its subobjects and overridden in a class derived from it. These entries are allocated in the virtual table for the virtual base class that is most immediately derived from the base class containing the overridden virtual function declaration. They are used to find the necessary adjustment from the virtual base to the derived class containing the override, if any. When a virtual function is invoked via a virtual base, but has been overridden in a derived class, the overriding function first adds a fixed offset to adjust the `this` pointer to the virtual base, and then adds the value contained at the `vcall` offset in the virtual base to its `this` pointer to get the address of the derived object where the function was overridden. These values may be positive or negative. These are first in the virtual table if present, ordered as specified in categories 3 and 4 of Section 2.5.3 below.
- *Virtual Base (vbase) offsets* are used to access the virtual bases of an object. Such an entry is added to the derived class object address (i.e. the address of its virtual table pointer) to get the address of a virtual base

class subobject. Such an entry is required for each virtual base class. The values can be positive or negative.



However, in classes sharing a virtual table with a primary base class, the vcall and vbase offsets added by the derived class all come before the vcall and vbase offsets required by the base class, so that the latter may be laid out as required by the base class without regard to additions from the derived class(es).

- The *offset to top* holds the displacement to the top of the object from the location within the object of the virtual table pointer that addresses this virtual table, as a `ptrdiff_t`. It is always present. The offset provides a way to find the top of the object from any base subobject with a virtual table pointer. This is necessary for `dynamic_cast<void*>` in particular.



In a complete object virtual table, and therefore in all of its primary base virtual tables, the value of this offset will be zero. For the secondary virtual tables of other non-virtual bases, and of many virtual bases, it will be negative. Only in some construction virtual tables will some virtual base virtual tables have positive offsets, due to a different ordering of the virtual bases in the full object than in the subobject's standalone layout.

- The *typeinfo pointer* points to the typeinfo object used for RTTI. It is always present. All entries in each of the virtual tables for a given class must point to the same typeinfo object. A correct implementation of typeinfo equality is to check pointer equality, except for pointers (directly or indirectly) to incomplete types. The typeinfo pointer is a valid pointer for polymorphic classes, i.e. those with virtual functions, and is zero for non-polymorphic classes.
- The virtual table address point points here, i.e. this is the virtual table address contained in an object's virtual pointer. This address must have the alignment required for pointers.
- *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.

The form of a virtual function pointer is specified by the processor-specific C++ ABI for the implementation. In the specific case of 64-bit Itanium shared library builds, a virtual function pointer entry contains a pair of components (each 64 bits): 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.

The order of the virtual function pointers in a virtual table is the order of declaration of the corresponding member functions in the class. There is an entry for any virtual function declared in a class, whether it is a new function or overrides a base class function, unless it overrides a function from the primary base, and conversion between their return types does not require an adjustment. (In the case of this exception, the primary base and the derived class share the virtual table, and can share the virtual function entry because their 'this' and result type adjustments are the same.) If a class has an implicitly-defined virtual destructor, its entries come after the declared virtual function pointers.

When a derived class and its primary base share a virtual table, the virtual function entries introduced by the derived class follow those for the primary base, so that the layout of the primary base's embedded virtual table is the same as that of its standalone virtual table. In particular, if the derived class overrides a base class virtual function with a different (covariant) return type, the entry for the derived class comes after the primary base's embedded virtual table in declaration order, and is the entry used for calls from the derived class without adjustment. The entry in the embedded primary virtual table points to a routine that adjusts the result pointer before returning.

The entries for virtual destructors are actually pairs of entries. The first destructor, called the complete object destructor, performs the destruction without calling `delete()` on the object. The second destructor,

called the deleting destructor, calls `delete()` after destroying the object. Both destroy any virtual bases; a separate, non-virtual function, called the base object destructor, performs destruction of the object but not its virtual base subobjects, and does not call `delete()`.

Following the primary virtual table of a derived class are *secondary virtual tables* for each of its proper base classes, except any primary base(s) with which it shares its primary virtual table. These are copies of the virtual tables for the respective base classes (copies in the sense that they have the same layout, though the fields may have different values). We call the collection consisting of a primary virtual table along with all of its secondary virtual tables a *virtual table group*. The order in which they occur is the same as the order in which the base class subobjects are considered for allocation in the derived object:

- First are the virtual tables of direct non-primary, non-virtual proper bases, in the order declared, including their secondary virtual tables for non-virtual bases in the order they appear in the standalone virtual table group for the base. (Thus the effect is that these virtual tables occur in inheritance graph order, excluding primary bases and virtual bases.)
- Then come the virtual base virtual tables, also in inheritance graph order, and again excluding primary bases (which share virtual tables with the classes for which they are primary).

2.5.3 Virtual Table Construction

In this section, we describe how to construct the virtual table for an class, given virtual tables for all of its proper 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 virtual table, and an object of such a class contains no virtual pointer.

Category 1: Leaf

Structure:

- No inherited virtual functions.
- No virtual base classes.
- Declares virtual functions.

The virtual table contains offset-to-top and RTTI fields followed by virtual function pointers. There is one function pointer entry for each virtual function declared in the class, in declaration order, with any implicitly-defined virtual destructor pair last.

Category 2: Non-Virtual Bases Only

Structure:

- Only non-virtual proper base classes.
- Inherits virtual functions.


The class has a virtual table for each proper base class that has a virtual table. The secondary virtual table for a base class B has the same contents as the primary virtual table for B, except that:


- The offset-to-top and 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 proper base class `Base`, and a derived class `Derived` for which we are constructing this set of virtual tables, we shall refer to the virtual table for `Base` as `Base-in-Derived`. The virtual pointer of each base subobject of an object of the derived class will point to the corresponding base virtual table in this set.

The primary virtual table for the derived class contains entries for each of the functions in the primary base class virtual table, replaced by new overriding functions as appropriate. Following these entries, there is an entry for each virtual function declared in the derived class (in declaration order) for which one of the following two conditions holds:

- The virtual function does not override any function already appearing in the virtual table.
- The virtual function overrides a function (or functions) appearing in the virtual table, but the return type of the overrider is substantively different from the return type of the function(s) already present. If the return types are different, they are both pointer-to-class types, or both reference-to-class types. Let `B` and `D` denote the classes, where `D` is derived from `B`. The types are substantively different if `B` is a morally virtual base of `D` or if `B` is not located at offset zero in `D`.

 *The primary virtual table can be viewed as two virtual tables accessed from a shared virtual table pointer.*

 *A benefit of replicated virtual function entries (i.e., entries that appear both in the primary virtual table and in a secondary virtual table) is that they reduce 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 virtual table 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. Replication saves two 'this' adjustments for each virtual call to an overridden function originally introduced by a non-primary proper base class.*

Category 3: Virtual Bases Only

Structure:

- Only virtual base classes (but those may have non-virtual bases).
- The virtual base classes are neither empty nor nearly empty.

The class has a virtual table for each virtual base class that has a virtual table. These are all secondary virtual tables, because there are no empty or nearly empty base classes to be primary, and they are constructed from copies of the base class full object virtual tables according to the same rules as in Category 2, except that the virtual table for a virtual base `A` also includes a vcall offset entry for each virtual function represented in `A`'s primary virtual table and the secondary virtual tables from `A`'s non-virtual bases.

The vcall offsets in the secondary virtual table for a virtual base `A` are ordered as described next. We describe the ordering from the entry closest to the virtual table address point to that furthest. Since the vcall offsets precede the virtual table address point, this means that the memory address order is the reverse of that described.

- If virtual base `A` has a primary base class `P` sharing its virtual table, `P`'s vcall offsets come first, in the same order they would appear if `P` itself were the virtual base.
- Next come vcall offsets for each virtual function declared in `A`, in declaration order. Note that even for an overriding virtual function with covariant return types, only one vcall offset is present, as it can be shared

by both virtual table entries.

- Finally come vcall offsets for virtual functions declared in non-virtual bases of A other than P. These bases are considered in inheritance graph preorder, and the vcall offsets for multiple functions declared in one of them are in declaration order.

If the above listing of vcall offsets includes more than one for a particular virtual function signature, only the first one (closest to the virtual table address point) is allocated. That is, an offset from primary base P (and its non-virtual bases) eliminates any from A or its other bases, an offset from A eliminates any from the non-primary bases, and an offset from a non-primary base B of A eliminates any from the bases of B.

Note that there are no vcall offsets for virtual functions declared in a virtual base class V of A and never overridden within A or its non-virtual bases. Calls to such functions will use the vcall offset in V's virtual table.

The class also has a virtual table that is not copied from the virtual base class virtual tables. This virtual table is the primary virtual table of the class and is addressed by the virtual table pointer at the top of the object, which is not shared because there are no nearly empty virtual bases to be primary. It holds the following function pointer entries, following those of any primary base's virtual table, in the virtual functions' declaration order:

- Entries for virtual functions introduced by this class, i.e. those not declared by any of its bases.
- Entries for overridden virtual functions from the base classes, called replicated entries because they are already in the secondary virtual tables of the class.

The primary virtual table 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, direct or indirect. The entries are in the reverse of the inheritance graph order. That is, the entry for the leftmost virtual base is closest to the address point of the virtual table.


Category 4: Complex


Structure:

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

The rules for constructing virtual tables of the class are a combination of the rules from Categories 2 and 3, and can generally be determined inductively. The differences are mostly due to the fact that virtual base classes can now have (nearly empty) primary bases:

- If virtual base A has a primary virtual base class P sharing its virtual table, P's vbase and vcall offsets come first in the primary virtual table, in the same order they would appear if P itself were the virtual base, and those from A that do not replicate those from P precede them.
- As for the non-virtual base case, virtual function pointer entries from the derived class introductions occur only after the entries from the primary base class. There are entries for overridden virtual functions from the primary base class only if the result types are different (covariant). For purposes of this case, the two types are considered different if one of them is a non-primary or virtual base class of the other.

 *For an S-as-T virtual table, the vbase offset entries from the primary virtual table for T are replaced with appropriate offsets given the completed hierarchy.*

 *Consider the following inheritance hierarchy:*


```
struct S { virtual void f() };
struct T : virtual public S {};
struct U : virtual public T {};
struct V : public T, virtual public U {};
```

T's virtual table contains a virtual base offset for S. U's virtual table contains virtual base offsets for S and T. V's virtual table contains virtual base offsets for S, U, and T (in reverse inheritance graph preorder), where the vbase offset for T is for the virtual base of U, not for the non-virtual direct base of V.

Consider in addition:

```
struct W : public T {};
```

T is a primary base class for W. Therefore, its virtual base offset for S in its embedded T-in-W virtual table is the only one present.

 *The above-described virtual table group layout would allow all non-virtual secondary base class virtual tables in a group to be accessed from a virtual pointer for one of them, since the relative offsets would be fixed. (Since the primary virtual table could end up being embedded, as the primary base class virtual table, in another virtual table with additional virtual pointers separating it from its secondary virtual tables, this observation is not true of the primary virtual table.) However, since construction virtual table groups may be organized differently (see below), an implementation may not depend on this relationship between secondary virtual tables. This tradeoff was made because the space savings resulting from not requiring construction virtual tables to occur in complete groups was considered more important than potential sharing of virtual pointers.*

2.6 Virtual tables During Object Construction

2.6.1 General

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

During the construction of a class object, the object assumes the type of each of its proper 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. RTTI queries, dynamic casts and virtual calls of the object under construction statically converted to bases of the base under construction will dynamically resolve to the type of the base under construction. Normally, this behavior is accomplished by setting, in the base class constructor, the object's virtual table pointers to the addresses of the virtual tables for the base class.

However, if the base class has direct or indirect virtual bases, the virtual table pointers have to be set to the addresses of construction virtual tables. This is because the normal proper base class virtual tables may not hold the correct virtual base index values to access the virtual bases of the object under construction, and adjustment addressed by these virtual tables 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 proper base class and a complete object of a derived class do not have virtual bases at the same offsets.

A construction virtual table holds the virtual function addresses, offset-to-top, and RTTI information associated with the base class, and virtual base offsets and addresses of adjustor entry points with their parameter adjustments associated with objects of the complete class.

To ensure that the virtual table pointers are set to the appropriate virtual tables during proper base class construction, a table of virtual table pointers, called the VTT, which holds the addresses of construction and non-

construction virtual tables is generated for the complete class. The constructor for the complete class passes to each proper base class constructor a pointer to the appropriate place in the VTT where the proper base class constructor can find its set of virtual tables. Construction virtual tables are used in a similar way during the execution of proper base class destructors.

i *When a complete object constructor is constructing a virtual base, it must be wary of using the vbase offsets in the virtual table, since the possibly shared virtual pointer may point to a construction virtual table of an unrelated base class. For instance, in*

```
struct S {};
struct T: virtual S {};
struct U {};
struct V: virtual T, virtual U {};
```

the virtual pointers for T and V are in the same place. When V's constructor is about to construct U, that virtual pointer points to a virtual table for T, and therefore cannot be used to locate U.

2.6.2 VTT Order

An array of virtual table addresses, called the **VTT**, is declared for each class type that has indirect or direct virtual base classes. (Otherwise, each proper base class may be initialized using its complete object virtual table group.)

The elements of the VTT array for a class D are in this order:

1. *Primary virtual pointer*: address of the primary virtual table for the complete object D.
2. *Secondary VTTs*: for each direct non-virtual proper base class B of D that requires a VTT, in declaration order, a sub-VTT for B-in-D, structured like the main VTT for B, with a primary virtual pointer, secondary VTTs, and secondary virtual pointers, but without virtual VTTs.

i *This construction is applied recursively.*

3. *Secondary virtual pointers*: for each base class X which (a) has virtual bases or is reachable along a virtual path from D, and (b) is not a non-virtual primary base, the address of the virtual table for X-in-D or an appropriate construction virtual table.

X is reachable along a virtual path from D if there exists a path X, B1, B2, ..., BN, D in the inheritance graph such that at least one of X, B1, B2, ..., or BN is a virtual base class.

The order in which the virtual pointers appear in the VTT is inheritance graph preorder.

i *There are virtual pointers for direct and indirect base classes. Although primary non-virtual bases do not get secondary virtual pointers, they do not otherwise affect the ordering.*

Primary virtual bases require a secondary virtual pointer in the VTT because the derived class with which they will share a virtual pointer is determined by the most derived class in the hierarchy.

Secondary virtual pointers may be required for base classes that do not require secondary VTTs. A virtual base with no virtual bases of its own does not require a VTT, but does require a virtual pointer entry in the VTT.

4. *Virtual VTTs*: For each proper virtual base classes in inheritance graph preorder, construct a sub-VTT as in (2) above.



The virtual VTT addresses come last because they are only passed to the virtual base class constructors for the complete object.

Each virtual table address in the VTT is the address to be assigned to the respective virtual pointer, i.e. the address of the first virtual function pointer in the virtual table, not of the first vcall offset.



It is required that the VTT for a complete class D be identical in structure to the sub-VTT for the same class D as a subclass of another class E derived from it, so that the constructors for D can depend on that structure. Therefore, the various components of its VTT are present based on the rules given, even if they point to the D complete object virtual table or its secondary virtual tables. That is, secondary VTTs are present for all bases with virtual bases (including the virtual bases themselves, which have their secondary VTTs in the virtual VTT section), and secondary virtual pointers are present for all bases with either virtual bases or virtual function declarations overridden along a virtual path. The only exception is that a primary non-virtual base class does not require a secondary virtual pointer.

Parts (1) and (3) of a primary (not secondary, i.e. nested) VTT, that is the primary and secondary virtual pointers, are used for the final initialization of an object's virtual pointers before the full-object initialization and later use, and must therefore point to the main virtual table group for the class. Those bases which do not have secondary virtual pointers in the VTT have their virtual pointers explicitly initialized to the main virtual table group by the constructors (see [Subobject Construction and Destruction](#)).

The virtual pointers in the secondary VTTs and virtual VTTs are used for subobject construction, and may always point to special construction virtual tables laid out as described in the following subsections. However, it will sometimes be possible to use either the full-object virtual table for the subclass, or its secondary virtual table for the full class being constructed. This ABI does not specify a choice, nor does it specify names for the construction virtual tables, so the constructors must use the VTT rather than assuming that a particular construction virtual table exists.

For example, suppose we have the following hierarchy:

```
class A1 { int i; };
class A2 { int i; virtual void f(); };
class V1 : public A1, public A2 { int i; };
    // A2 is primary base of V1, A1 is non-polymorphic
class B1 { int i; };
class B2 { int i; };
class V2 : public B1, public B2, public virtual V1 { int i; };
    // V2 has no primary base, V1 is secondary base
class V3 {virtual void g(); };
class C1 : public virtual V1 { int i; };
    // C1 has no primary base, V1 is secondary base
class C2 : public virtual V3, virtual V2 { int i; };
    // C2 has V3 primary (nearly-empty virtual) base, V2 is secondary base
class X1 { int i; };
class C3 : public X1 { int i; };
class D : public C1, public C2, public C3 { int i; };
    // C1 is primary base, C2 is secondary base, C3 is non-polymorphic
```

Then the VTT for D would appear in the following order, where indenting indicates the sub-VTT structure, and asterisks () indicate that construction virtual tables instead of complete object virtual tables are required.*

```
// 1. Primary virtual pointer:
[0] D has virtual bases (complete object vptr)
```

```

// 2. Secondary VTTs:
[1] C1 * (has virtual base)
[2]   V1-in-C1 in D (secondary vptr)

[3] C2 * (has virtual bases)
[4]   V3-in-C2 in D (primary vptr)
[5]   V2-in-C2 in D (secondary vptr)
[6]   V1-in-C2 in D (secondary vptr)

// 3. Secondary virtual pointers:
// (no C1-in-D -- primary base)
[7]   V1-in-D (V1 is virtual)
[8] C2-in-D (preorder; has virtual bases)
[9]   V3-in-D (V3 is virtual)
[10]  V2-in-D (V2 is virtual)
// (For complete object D VTT, these all can point to the
// secondary vtables in the D vtable, the V3-in-D entry
// will be the same as the C2-in-D entry, as that is the active
// V3 virtual base in the complete object D. In the sub-VTT for
// D in a class derived from D, some might be construction
// virtual tables.)

// 4. Virtual VTTs:
// (V1 has no virtual bases).
[11] V2 * (V2 has virtual bases)
[12]  V1-in-V2 in D * (secondary vptr, V1 is virtual)
      (A2 is primary base of V1)
// (V3 has no virtual bases)

```

If A2 is a virtual base of V1, the VTT will contain more elements (exercise left to the astute reader).

2.6.3 Construction Virtual Table Layout

The construction virtual tables for a complete object are emitted in the same object file as the virtual table. So the virtual table structures for a complete object of class C include, in no particular order:

- the main virtual table group for C, i.e. the virtual table to which the primary virtual pointer (at offset 0) points, along with its proper base class virtual tables in order of allocation, including virtual base class virtual tables;
- any construction virtual tables required for non-virtual and virtual bases; and
- the VTT array for C, providing location information for the above.

The VTT array is referenced via its own mangled external name, and the construction virtual tables are accessed via the VTT array, so the latter do not have external names.

2.6.4 Construction Virtual Table entries

The construction virtual table group for a proper base class subobject B (of derived class D) does not have the same entries in the same order as the main virtual table group for a complete object B, as described in [Virtual Table Layout](#) above. Some of the base class subobjects may not need construction virtual tables, which will therefore not be present in the construction virtual table group, even though the subobject virtual tables are present in the main virtual table group for the complete object.

The *values* of some construction virtual table entries will differ from the corresponding entries in either the main virtual table group for B or the virtual table group for B-in-D, primarily because the virtual bases of B will be at different relative offsets in a D object than in a standalone B object, as follows:

1. Virtual base class offsets reflect the positions of the virtual base classes in the full D object.

2. Similarly, vcall offsets reflect the relative positions of the overridden and overriding classes within the complete object D.
3. The offset-to-top fields refer to B (and B's in particular will therefore be zero).
4. The RTTI pointers point to B's RTTI.
5. Only functions in B and its base classes are considered for virtual function resolution.

2.7 Array Operator `new` Cookies

When operator `new` is used to create a new array, a cookie is usually stored to remember the allocated length (number of array elements) so that it can be deallocated correctly.

Specifically:

- No cookie is required if the array element type `T` has a trivial destructor (12.4 [class.dtor]) and the usual (array) deallocation function (3.7.3.2 [basic.stc.dynamic.deallocation]) function does not take two arguments.

(Note: if the usual array deallocation function takes two arguments, then it is a member function whose second argument is of type `size_t`. The standard guarantees (12.5 [class.free]) that this function will be passed the number of bytes allocated with the previous array `new` expression.)

- No cookie is required if the `new` operator being used is `::operator new[](size_t, void*)`.
- Otherwise, this ABI requires a cookie, setup as follows:
 - The cookie will have size `sizeof(size_t)`.
 - Let `align` be the maximum alignment of `size_t` and an element of the array to be allocated.
 - Let `padding` be the maximum of `sizeof(size_t)` and `align` bytes.
 - The space allocated for the array will be the space required by the array itself plus `padding` bytes.
 - The alignment of the space allocated for the array will be `align` bytes.
 - The array data will begin at an offset of `align` bytes from the space allocated for the array.
 - The cookie will be stored in the `sizeof(size_t)` bytes immediately preceding the array data.

These rules have the following consequences:

- *The array elements and the cookie are all aligned naturally.*
- *Padding will be required if `sizeof(size_t)` is smaller than the array element alignment, and if present will precede the cookie.*

Given the above, the following is pseudocode for processing `new(ARGS) T[n]`:

```

if T has a trivial destructor (C++ standard, 12.4/3)
    padding = 0
else if we're using ::operator new[](size_t, void*)
    padding = 0
else
    padding = max(sizeof(size_t), alignof(T))

p = operator new[](n * sizeof(T) + padding, ARGS)
p1 = (T*) ( (char *)p + padding )

if padding > 0
    *( (size_t *)p1 - 1) = n

for i = [0, n)
    create a T, using the default constructor, at p1[i]
```

```
return p1
```

2.8 Initialization Guard Variables

Associated with each function-scope static variable requiring runtime construction is a guard variable which is used to guarantee that construction occurs only once. Its name is mangled based on the mangling of the guarded object name, to allow distinct instances of the function (e.g. due to inlining) to use the same guard.

The size of the guard variable is 64 bits. The first byte (i.e. the byte at the address of the full variable) shall contain the value 0 prior to initialization of the associated variable, and 1 after initialization is complete. Usage of the other bytes of the guard variable is implementation-defined.

See [Section 3.3.2](#) for the API for references to this guard variable.

2.9 Run-Time Type Information (RTTI)

2.9.1 General

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 dynamic 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*".

It is intended that two type_info pointers point to equivalent type descriptions if and only if the pointers are equal. An implementation must satisfy this constraint, e.g. by using symbol preemption, COMDAT sections, or other mechanisms.



Note that the full structure described by an RTTI descriptor may include incomplete types not required by the Standard to be completed, although not in contexts where it would cause ambiguity. Therefore, any cross-references within the RTTI to types not known to be complete must be weak symbol references.

2.9.2 Place of Emission

It is desirable to minimize the number of places where a particular bit of RTTI is emitted. For dynamic class types, a similar problem occurs for virtual function tables, and hence the RTTI descriptor should be emitted with the primary virtual table for that type. For other types, they must be emitted at the location where their use is implied: the object file containing the typeid, throw or catch.

Basic type information (e.g. for "int", "bool", etc.) will be kept in the run-time support library. Specifically, the run-time support library should contain type_info objects for the types X, X* and X const*, for every X in: void, 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.)

2.9.3 The typeid Operator

The typeid operator produces a reference to a std::type_info structure with the following public interface (18.5.1):

```
namespace std {
  class type_info {
  public:
    virtual ~type_info();
    bool operator==(const type_info &) const;
    bool operator!=(const type_info &) const;
    bool before(const type_info &) const;
    const char* name() const;
  private:
    type_info (const type_info& rhs);
    type_info& operator= (const type_info& rhs);
  };
}
```

After linking and loading, only one std::type_info structure is accessible via the external name defined by this ABI for any particular complete type symbol (see [Vague Linkage](#)). Therefore, except for direct or indirect pointers to incomplete types, the equality and inequality operators can be written as address comparisons when operating on those type_info objects: two type_info structures describe the same type if and only if they are the same structure (at the same address). However, in the case of pointer types, directly or indirectly pointing to incomplete class types, a more complex comparison is required, described below with the RTTI layout of pointer types.

The name() member function returns the address of an NTBS, unique to the type, containing the [mangled name](#) of the type. It has a [mangled name](#) defined by the ABI to allow consistent reference to it, and the [Vague Linkage](#) section specifies how to produce a unique copy.

In a flat address space (such as that of the Itanium architecture), the operator==, operator!=, and before() members are easily implemented in terms of an address comparison of the name NTBS.

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.

2.9.4 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<void cv*>, which returns a pointer to the complete lvalue,
- dynamic_cast operation from a proper 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.

The most common kind of dynamic_cast is base-to-derived in a singly inherited hierarchy.

2.9.5 RTTI Layout

1. The class definitions below are to be interpreted as implying a memory layout following the class layout rules for the host ABI. They specify data members only, except for the Standard-specified member functions of the std::type_info class given below, and do not imply anything about the member functions of these classes. Virtual member functions of these classes may only be used within the target systems' respective runtime libraries. The data members must be laid out exactly as specified.

2. Every virtual table shall contain one entry describing the offset from a virtual pointer for that virtual table to the origin of the object containing that virtual pointer (or equivalently: to the virtual pointer for the primary virtual table). This entry is directly useful to implement `dynamic_cast<void cv*>`, but is also needed for the other truly dynamic casts. This entry is located two words ahead of the location pointed to by the virtual pointer (i.e., entry "-2"). This entry is present in all virtual tables, even for classes having virtual bases but no virtual functions.
3. Every virtual table shall contain one entry that is a pointer to an object derived from `std::type_info`. This entry is located at the word preceding the location pointed to by the virtual pointer (i.e., entry "-1"). The entry is allocated in all virtual tables; for classes having virtual bases but no virtual functions, the entry is zero.

We add one pointer to the `std::type_info` class in addition to the virtual table pointer implied by its virtual destructor:

```
class type_info {
    ... // See section 2.9.3
private:
    const char *__type_name;
};
```

- The class will contain a virtual pointer before the explicit members.
- `__type_name` is a pointer to a NTBS representing the mangled name of the type.

The possible derived types are:

- `abi::__fundamental_type_info`
- `abi::__array_type_info`
- `abi::__function_type_info`
- `abi::__enum_type_info`
- `abi::__class_type_info`
- `abi::__si_class_type_info`
- `abi::__vmi_class_type_info`
- `abi::__pbase_type_info`
- `abi::__pointer_type_info`
- `abi::__pointer_to_member_type_info`

4. `abi::__fundamental_type_info` adds no data members to `std::type_info`;
5. `abi::__array_type_info` and `abi::__function_type_info` do not add data members to `std::type_info` (these types are only produced by the `typeid` operator; they decay in other contexts).
`abi::__enum_type_info` does not add data members either.
6. Three different types are used to represent user type information:
 - a. `abi::__class_type_info` is used for class types having no bases, and is also a base type for the other two class type representations.

```
class __class_type_info : public std::type_info {}
```

This RTTI class may also be used for incomplete class types when referenced by a pointer RTTI, in which case it must be prevented from preempting the RTTI for the complete class type, for instance by emitting it as a static object (without external linkage).

Two `abi::__class_type_info` objects can always be compared, for equality (i.e. of the types represented) or ordering, by comparison of their name NTBS addresses. In addition, complete class RTTI objects may also be compared for equality by comparison of their `type_info` addresses.

- b. For classes containing only a single, public, non-virtual base at offset zero (i.e. the derived class is dynamic iff the base is), class `abi::__si_class_type_info` is used. It adds to `abi::__class_type_info` a single member pointing to the `type_info` structure for the base type, declared `"__class_type_info const *__base_type"`.

```
class __si_class_type_info : public __class_type_info {
public:
    const __class_type_info *__base_type;
};
```

- c. For classes with bases that do not satisfy the `__si_class_type_info` constraints, `abi::__vmi_class_type_info` is used. It is derived from `abi::__class_type_info`:

```
class __vmi_class_type_info : public __class_type_info {
public:
    unsigned int __flags;
    unsigned int __base_count;
    __base_class_type_info __base_info[1];

    enum __flags_masks {
        __non_diamond_repeat_mask = 0x1,
        __diamond_shaped_mask = 0x2
    };
};
```

- `__flags` is a word with flags describing details about the class structure, which may be referenced by using the `__flags_masks` enumeration.
 - 0x01: class has non-diamond repeated inheritance
 - 0x02: class is diamond shaped

These flags refer to both direct and indirect bases. The type of the `__flags` field is defined by each psABI, but must be at least 16 bits. For the 64-bit Itanium ABI, it will be unsigned int (32 bits).

- `__base_count` is a word with the number of direct proper base class descriptions that follow. The type of the `__base_count` field is defined by each psABI. For the 64-bit Itanium ABI, it will be unsigned int (32 bits).
- `__base_info[]` is an array of base class descriptions -- one for every direct proper base. Each description is of the type:

```
struct abi::__base_class_type_info {
public:
    const __class_type_info *__base_type;
    long __offset_flags;

    enum __offset_flags_masks {
        __virtual_mask = 0x1,
        __public_mask = 0x2,
        __offset_shift = 8
    };
};
```

The `__base_type` member points to the RTTI for the base type.

All but the lower 8 bits of `__offset_flags` are a signed offset. For a non-virtual base, this is the offset in the object of the base subobject. For a virtual base, this is the offset in the virtual table of the virtual base offset for the virtual base referenced (negative).

The low-order byte of `__offset_flags` contains flags, as given by the masks from the enumeration `__offset_flags_masks`:

- 0x1: Base class is virtual
- 0x2: Base class is public

Note that the resulting structure is variable-length, with the actual size depending on the number of trailing base class descriptions.

7. `abi::__pbase_type_info` is a base for both pointer types and pointer-to-member types. It adds two data members:

```
class __pbase_type_info : public std::type_info {
public:
    unsigned int __flags;
    const std::type_info *__pointee;

    enum __masks {
        __const_mask = 0x1,
        __volatile_mask = 0x2,
        __restrict_mask = 0x4,
        __incomplete_mask = 0x8,
        __incomplete_class_mask = 0x10
    };
};
```

- `__flags` is a flag word describing the cv-qualification and other attributes of the type pointed to (e.g., "int volatile*" should have the "volatile" bit set in that word); and
- `__pointee` is a pointer to the `std::type_info` derivation for the unqualified type being pointed to.

Note that the `__flags` bits should not be folded into the pointer to allow future definition of additional flags. It contains the following bits, and may be referenced using the flags defined in the `__masks` enum:

- 0x1: `__pointee` type has const qualifier
- 0x2: `__pointee` type has volatile qualifier
- 0x4: `__pointee` type has restrict qualifier
- 0x8: `__pointee` type is incomplete
- 0x10: class containing `__pointee` is incomplete (in pointer to member)


When the `abi::__pbase_type_info` is for a direct or indirect pointer to an incomplete class type, the incomplete target type flag is set. When it is for a direct or indirect pointer to a member of an incomplete class type, the incomplete class type flag is set. In addition, it and all of the intermediate `abi::__pointer_type_info` structs in the chain down to the `abi::__class_type_info` for the incomplete class type must be prevented from resolving to the corresponding `type_info` structs for the complete class type, possibly by making them local static objects. Finally, a dummy class RTTI is generated for the incomplete type that will not resolve to the final complete class RTTI (because the latter need not exist), possibly by making it a local static object.

Two `abi::__pbase_type_info` objects can always be compared for equality (i.e. of the types represented) or ordering by comparison of their name NTBS addresses. In addition, unless either or both have either of the incomplete flags set, equality can be tested by comparing the `type_info` addresses.

8. `abi::__pointer_type_info` is derived from `abi::__pbase_type_info` with no additional data members.
9. The `abi::__pointer_to_member_type_info` type adds one field to `abi::__pbase_type_info`:

```
class __pointer_to_member_type_info : public __pbase_type_info {
public:
    const abi::__class_type_info *__context;
};
```

- `__context` is a pointer to an `abi::__class_type_info` corresponding to the class type containing the member pointed to (e.g., the "A" in "int A::*")

 *Note that this ABI requires elsewhere that a virtual table be emitted for a dynamic type in the object where the first non-inline virtual function member is defined, if any, or everywhere referenced if none. Therefore, an implementation should include at least one non-inline virtual function member and define it in the library, to avoid having user code inadvertently preempt the virtual table. Since the Standard requires a virtual destructor, and it will rarely be called, it is a good candidate for this role.*

2.9.6 `std::type_info::name()`

The null-terminated byte string returned by this routine is the [mangled name](#) of the type.

2.9.7 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<T>(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 subobject of a `T` object [note: this can be checked at compile time], and if only one object of type `T` is derived from the subobject 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 subobject 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` subobject 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 `abi::__dynamic_cast`:

```
extern "C"
void* __dynamic_cast ( const void *sub,
                      const abi::__class_type_info *src,
                      const abi::__class_type_info *dst,
                      std::ptrdiff_t src2dst_offset);
/* sub: source address to be adjusted; nonnull, and since the
```

```

*      source object is polymorphic, *(void**)sub is a virtual
pointer.
* src: static type of the source object.
* dst: destination type (the "T" in "dynamic_cast<T>(v)").
* src2dst_offset: a static hint about the location of the
*   source subobject with respect to the complete object;
*   special negative values are:
*       -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.
*/

```



Rationale:

- *A simple `dynamic_cast` algorithm that is efficient in the common case of base-to-most-derived cast case is preferable to more sophisticated ideas that handle deep-base-to-in-between-derived casts more efficiently at a slight cost to the common case. Hence, an earlier scheme of providing a hash-table into the list of base classes (as is done e.g. in the HP aC++ compiler) was dropped.*
- *For similar reasons, we only keep direct base information about a class type. Indirect base information can be found by chasing `type_info` pointers (and care should be taken to determine ambiguous base class types).*
- *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". This is the purpose of the `src2dst_offset` hint. An implementation is free to always pass -1 (no hint), or to always ignore the hint in `__dynamic_cast`.*

2.9.8 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).

Chapter 3: Function Calling Conventions and APIs

In general, the calling conventions for C++ in this ABI follow those specified by the underlying processor-specific ABI for C, whenever there is an analogous construct in C. This chapter specifies exceptions required by C++-specific semantics, or by features without analogues in C. It also specifies the APIs of a variety of runtime utility routines required to be part of the support library of an ABI-conforming implementation for use by compiled code. In addition, reference is made to the separate description of [exception handling](#) in this ABI, which defines a large number of runtime utility routine APIs.

3.1 Non-Virtual Function Calling Conventions

3.1.1 Value Parameters

In general, C++ value parameters are handled just like C parameters. This includes class type parameters passed wholly or partially in registers. However, in the special case where the parameter type has a non-trivial copy

constructor or destructor, the caller must allocate space for a temporary copy, and pass the resulting copy by reference (below). Specifically,

- Space is allocated by the caller for the temporary. If there is no non-trivial copy constructor or destructor, it is in the normal parameter-passing space, i.e. in the parameter registers or on the stack, and the constructor is called if necessary. Otherwise, it is allocated on the stack or heap.
- The caller constructs the parameter in the space allocated, using a simple copy to the parameter space (parameter registers or stack) if there is no non-trivial copy constructor or destructor.
- The function is called, passing the parameter value (if there is no non-trivial copy constructor or destructor), or its address (if there is one).
- The caller calls any non-trivial destructor for the parameter after returning (at the end of the containing expression).
- If necessary (e.g. if the parameter was allocated on the heap), the caller deallocates space after return and destruction.

3.1.2 Reference Parameters

Reference parameters are handled by passing a pointer to the actual parameter.

3.1.3 Empty Parameters

Empty classes will be passed no differently from ordinary classes. If passed in registers the NaT bit must not be set on all registers that make up the class.

The contents of the single byte parameter slot are unspecified, and the callee may not depend on any particular value. On Itanium, the associated NaT bit must not be set if the parameter slot is associated with a register.

3.1.4 Return Values

In general, C++ return values are handled just like C return values. This includes class type results returned in registers. However, if the return value type has a non-trivial copy constructor or destructor, the caller allocates space for a temporary, and passes a pointer to the temporary as an implicit first parameter preceding both the this parameter and user parameters. The callee constructs the return value into this temporary.

A result of an empty class type will be returned as though it were a struct containing a single char, i.e. `struct S { char c; };`. The actual content of the return register is unspecified. On Itanium, the associated NaT bit must not be set.

3.1.5 Constructor Return Values

Constructors return void results.

3.2 Virtual Function Calling Conventions

3.2.1 Foundation

This section sketches the calling convention for virtual functions, based on the above virtual table layout. *See also the [ABI examples](#) document for motivating examples and potential implementations.*

We explain, at a high level, what information must be present in the virtual table for a class A which declares a virtual function *f* in order that, given an pointer of type *A**, the caller can call the virtual function *f*. This section does not specify exactly where that information is located (see above), nor does it specify how to convert a pointer to a class derived from A to an *A**, if that is required.

When this section uses the term *function pointer* it is understood that this term may refer either to a traditional function pointer (i.e., a pointer to a GP/address pair) or a GP/address pair itself. Which of these alternatives is actually used is specified elsewhere in the ABI, but is independent of the description in this section.

Throughout this section, we assume that A is the class for which we are creating a virtual table, B is the most derived class in the hierarchy, and C is the class that contains *C::f*, the unique final overrider for *A::f*. This section specifies the contents of the *f* entry in the A-in-B virtual table. (If A is primary base in the hierarchy, then the A-in-B virtual table will be shared with the derived class virtual table -- but the contents of the A portion of that virtual table will still be as specified here.)

In all cases, the *non-adjusting entry point* for a virtual function expects the 'this' pointer to point to an instance of the class in which the virtual function is defined. In other words, the non-adjusting entry point for *C::f* will expect that its 'this' pointer points to a C object.

3.2.2 Virtual Table Components

For each virtual function declared in a class C, we add an entry to its virtual table if one is not already there (i.e. if it is not overriding a function in its primary base). In particular, a declaration which overrides a function inherited from a secondary base gets a new slot in the primary virtual table. We do this to avoid useless adjustments when calling a virtual function through a pointer to the most derived class.

The content of this entry for class A is a function pointer, as determined by one of the following cases. Recall that we are dealing with a hierarchy where B is most derived, A is a direct (or indirect) base of B defining *f*, and C contains the unique final overrider *C::f* of *A::f*.

1. A = C

(In this case, we are creating either the primary virtual table for A, or the A-in-B secondary virtual table.)

The virtual table contains a function pointer pointing to the non-adjusting entry point for *A::f*.

2. A != C

In this case, we are creating the A-in-B secondary virtual table.

The virtual table contains a pointer to an entry point that performs the adjustment from an *A** to a *C**, and then transfers control to the non-adjusting entry point for *C::f*.

When a class is used as a virtual base, we add a vcall offset slot to the beginning of its virtual table for each of the virtual functions it provides, whether in its primary or secondary virtual tables. Derived classes which override these functions may use the slots to determine the adjustment necessary.

3.2.3 Callee

For each direct or indirect base A of C that is not a morally virtual base of C, the compiler must emit, in the same object file as the code for *C::f*, an *A-adjusting entry point* for *C::f*. This entry point will expect that its *this* pointer points to an *A**, and will convert it to a *C** (which merely requires adding a constant offset) before transferring control to the non-adjusting entry point for *C::f*.

For each direct or indirect virtual base V of C such that V declares f , the compiler must emit, in the same object file as the code for $C::f$, a *V-adjusting entry point* for $C::f$. This entry point will expect that its `this` pointer points to the unique virtual V subobject of C . (Note that there may in general be multiple V subobjects of C , but that only one of them will be virtual.) This entry point must load the `vcall` offset corresponding to f located in the virtual table for V obtained via its `this` pointer, extract the `vcall` offset corresponding to f located in that virtual table, and add this offset to the `this` pointer. (Note that, as specified in the data layout document, when V is used as a virtual base, its virtual table contains `vcall` offsets for every virtual function declared in V or any of its bases.) Then, this entry point must transfer control to the non-adjusting entry point.

For each morally virtual base M of C such that M is **not** a virtual base (and therefore must be a subobject of a virtual base V), and such that M declares f , the compiler must emit, in the same object file as the code for $C::f$, an *M-adjusting entry point* for $C::f$. This entry point will expect that its `this` pointer points to an M^* , and will convert it to a V^* (a fixed offset), where V is the nearest virtual base to M along the inheritance path from C to M . Then, it will convert the V^* to a C^* by using the `vcall` offset stored in the V 's virtual table.

3.2.4 Caller

When calling a virtual function f , through a pointer of static type B^* , the caller

- Selects a (possibly improper) subobject A of B such that A declares f . (In general, A may be the same as B .) (Note that A need not define f ; it may contain either a definition of f , or a declaration of f as a pure virtual function.)
- Converts the B^* to point to this subobject. (Call the resulting pointer `'a'`.)
- Uses the virtual table pointer contained in the A subobject to locate the function pointer through which to perform the call.
- Calls through this function pointer, passing `'a'` as the `this` pointer.

(Note that in general it will be optimal to select the class which contained the final overrider (i.e., C) as the class to which the B^ should be converted. This class is always a satisfactory choice, since it is known to contain a definition of f . In addition, if the dynamic type of the object is B , then $C::f$ will be the function ultimately selected by the call, which means that C 's virtual table will contain a pointer to the non-adjusting entry point, meaning that no additional adjustments to the `this` pointer will be required.)*

However, there may be cases in which choosing a different base subobject could be superior. For example, if there is an alternate base D which also declares f , and a pointer to the D subobject is already available, then it may be better to use the D subobject rather than converting the B^ to a C^* , in order to avoid the cost of the conversion.)*

3.2.5 Implementation

Note that the ABI only specifies the multiple entry points for a virtual function and its associated thunks; how those entry points are provided is unspecified. An existing compiler which uses thunks with a different means of adjusting the virtual table pointers can be made compliant with this ABI by only adding the `vcall` offsets -- the thunks need not use them. A more efficient implementation would be to emit all of the thunks immediately before the non-adjusting entry point to the function. Another might emit a new copy of the function for each entry point; this is a quality of implementation issue. See further discussion of implementation in the [ABI examples](#) document.

3.2.6 Pure Virtual Function API

An implementation shall provide a standard entry point that a compiler may reference in virtual tables to indicate a pure virtual function. Its interface is:

```
extern "C" void __cxa_pure_virtual ();
```

This routine will only be called if the user calls a non-overridden pure virtual function, which has undefined behavior according to the C++ Standard. Therefore, this ABI does not specify its behavior, but it is expected that it will terminate the program, possibly with an error message.

3.3 Construction and Destruction APIs

This section describes APIs to be used for the construction and destruction of objects. This includes:

- [3.3.1 Subobject Construction and Destruction](#)
- [3.3.2 One-time Construction API](#)
- [3.3.3 Array Construction and Destruction API](#)
- [3.3.4 Controlling Object Construction Order](#)
- [3.3.5 DSO Object Destruction API](#)

3.3.1 Subobject Construction and Destruction

The complete object constructors and destructors find the VTT, described in Section 2.6, Virtual Tables During Object Construction, via its mangled name. They pass the address of the subobject's sub-VTT entry in the VTT as a second parameter when calling the base object constructors and destructors. The base object constructors and destructors use the addresses passed to initialize the primary virtual pointer and virtual pointers that point to the classes which either have virtual bases or override virtual functions with a virtual step (have vcall offsets needing adjustment).

If a constructor calls constructors for base class subobjects that do not need construction virtual tables, e.g. because they have no virtual bases, the construction virtual table parameter is not passed to the base class subobject constructor, and the base class subobject constructors use their complete object virtual tables for initialization.

If a class has a non-virtual destructor, and a deleting destructor is emitted for that class, the deleting destructor must correctly handle the case that the `this` pointer is `NULL`. All other destructors, including deleting destructors for classes with a virtual destructor, may assume that the `this` pointer is not `NULL`.

Suppose we have a subobject class D that needs a construction virtual table, derived from a base B that needs a construction virtual table as part of D , and possibly from others that do not need construction virtual tables. Then the sub-VTT and constructor code for D would look like the following:

```
// Sub-VTT for D (embedded in VTT for its derived class X):
static vtable *__VTT__1D [1+n+m] =
{ D primary vtable,
  // The sub-VTT for B-in-D in X may have further structure:
  B-in-D sub-VTT (n elements),
  // The secondary virtual pointers for D's bases have elements
  // corresponding to those in the B-in-D sub-VTT,
  // and possibly others for virtual bases of D:
  D secondary virtual pointer for B and bases (m elements) };

D ( D *this, vtable **ctorvtbls )
{
  // (The following will be unwound, not a real loop):
  for ( each base A of D ) {

      // A "boring" base is one that does not need a ctorvtbl:
```

```

    if ( ! boring(A) ) {
        // Call subobject constructors with sub-VTT index
        // if the base needs it -- only B in our example:
        A ( (A*)this, ctorvtbls + sub-VTT-index(A) );

    } else {
        // Otherwise, just invoke the complete-object constructor:
        A ( (A*)this );
    }
}

// Initialize virtual pointer with primary ctorvtbls address
// (first element):
this->vptr = ctorvtbls+0;          // primary virtual pointer

// (The following will be unwound, not a real loop):
for ( each subobject A of D ) {

    // Initialize virtual pointers of subobjects with ctorvtbls
    // addresses for the bases
    if ( ! boring(A) ) {
        ((A*)this)->vptr = ctorvtbls + 1+n + secondary-vptr-index(A);
        // where n is the number of elements in the sub-VTTs

    } else {
        // Otherwise, just use the complete-object vtable:
        ((A *)this)->vptr = &(A-in-D vtable);
    }
}

// Code for D constructor.
...
}

```

A test program for this can be found in the [ABI Examples](#) document.

3.3.2 One-time Construction API

As described in [Section 2.8](#), function-scope static objects have associated guard variables used to support the requirement that they be initialized exactly once, the first time the scope declaring them is entered. An implementation that does not anticipate supporting multi-threading may simply check the first byte (i.e., the byte with lowest address) of that guard variable, initializing if and only if its value is zero, and then setting it to a non-zero value.

However, an implementation intending to support automatically thread-safe, one-time initialization (as opposed to requiring explicit user control for thread safety) may make use of the following API functions:

```
extern "C" int __cxa_guard_acquire ( __int64_t *guard_object );
```

Returns 1 if the initialization is not yet complete; 0 otherwise. This function is called before initialization takes place. If this function returns 1, either `__cxa_guard_release` or `__cxa_guard_abort` must be called with the same argument. The first byte of the `guard_object` is not modified by this function.

A thread-safe implementation will probably guard access to the first byte of the `guard_object` with a mutex. If this function returns 1, the mutex will have been acquired by the calling thread.

```
extern "C" void __cxa_guard_release ( __int64_t *guard_object );
```

Sets the first byte of the guard object to a non-zero value. This function is called after initialization is complete.

A thread-safe implementation will release the mutex acquired by `__cxa_guard_acquire` after setting the first byte of the guard object.

```
extern "C" void __cxa_guard_abort ( __int64_t *guard_object );
```

This function is called if the initialization terminates by throwing an exception.

A thread-safe implementation will release the mutex acquired by `__cxa_guard_acquire`.



The following is pseudo-code showing how these functions can be used:

```
if (obj_guard.first_byte == 0) {
    if ( __cxa_guard_acquire (&obj_guard) ) {
        try {
            ... initialize the object ...;
        } catch (...) {
            __cxa_guard_abort (&obj_guard);
            throw;
        }
        ... queue object destructor with __cxa_atexit() ...;
        __cxa_guard_release (&obj_guard);
    }
}
```

An implementation need not include the simple inline test of the initialization flag in the guard variable around the above sequence. If it does so, the cost of this scheme, when run single-threaded with minimal versions of the above functions, will be two extra function calls, each of them accessing the guard variable, the first time the scope is entered.

An implementation supporting thread-safety on multiprocessor systems must also guarantee that references to the initialized object do not occur before the load of the initialization flag. On Itanium, this can be done by using a `ld1.acq` operation to load the flag.

The intent of specifying an 8-byte structure for the guard variable, but only describing one byte of its contents, is to allow flexibility in the implementation of the API above. On systems with good small lock support, the second word might be used for a mutex lock. On others, it might identify (as a pointer or index) a more complex lock structure to use.

3.3.3 Array Construction and Destruction API

An ABI-compliant system shall provide several runtime routines for use in array construction and destruction. They may be used by compilers, but their use is not required. The required APIs are:

```
extern "C" void * __cxa_vec_new (
    size_t element_count,
    size_t element_size,
    size_t padding_size,
    void (*constructor) ( void *this ),
    void (*destructor) ( void *this ) );
```

Equivalent to

```
__cxa_vec_new2(element_count, element_size, padding_size, constructor,
    destructor, &::operator new[], &::operator delete[])
```

```
extern "C" void * __cxa_vec_new2 (
    size_t element_count,
```



```

size_t element_size,
size_t padding_size,
void (*constructor) ( void *this ),
void (*destructor) ( void *this ),
void* (*alloc) ( size_t size ),
void (*dealloc) ( void *obj ) );

```

Given the number and size of elements for an array and the non-negative size of prefix padding for a cookie, allocate space (using `alloc`) for the array preceded by the specified padding, initialize the cookie if the padding is non-zero, and call the given constructor on each element. Return the address of the array proper, after the padding.

If `alloc` throws an exception, rethrow the exception. If `alloc` returns `NULL`, return `NULL`. If the constructor throws an exception, call `destructor` for any already constructed elements, and rethrow the exception. If the destructor throws an exception, call `std::terminate`.

The constructor may be `NULL`, in which case it must not be called. If the `padding_size` is zero, the destructor may be `NULL`; in that case it must not be called.

Neither `alloc` nor `dealloc` may be `NULL`.

```

extern "C" void * __cxa_vec_new3 (
    size_t element_count,
    size_t element_size,
    size_t padding_size,
    void (*constructor) ( void *this ),
    void (*destructor) ( void *this ),
    void* (*alloc) ( size_t size ),
    void (*dealloc) ( void *obj, size_t size ) );

```

Same as `__cxa_vec_new2` except that the deallocation function takes both the object address and its size.

```

extern "C" void __cxa_vec_ctor (
    void *array_address,
    size_t element_count,
    size_t element_size,
    void (*constructor) ( void *this ),
    void (*destructor) ( void *this ) );

```

Given the (data) address of an array, not including any cookie padding, and the number and size of its elements, call the given constructor on each element. If the constructor throws an exception, call the given destructor for any already-constructed elements, and rethrow the exception. If the destructor throws an exception, call `terminate()`. The constructor and/or destructor pointers may be `NULL`. If either is `NULL`, no action is taken when it would have been called.

```

extern "C" void __cxa_vec_dtor (
    void *array_address,
    size_t element_count,
    size_t element_size,
    void (*destructor) ( void *this ) );

```

Given the (data) address of an array, the number of elements, and the size of its elements, call the given destructor on each element. If the destructor throws an exception, rethrow after destroying the remaining elements if possible. If the destructor throws a second exception, call `terminate()`. The destructor pointer may be `NULL`, in which case this routine does nothing.

```

extern "C" void __cxa_vec_cleanup (
    void *array_address,
    size_t element_count,
    size_t element_size,
    void (*destructor) ( void *this ) );

```

Given the (data) address of an array, the number of elements, and the size of its elements, call the given destructor on each element. If the destructor throws an exception, call `terminate()`. The destructor pointer may be `NULL`, in which case this routine does nothing.

```
extern "C" void __cxa_vec_delete (
    void *array_address,
    size_t element_size,
    size_t padding_size,
    void (*destructor) ( void *this ) );
```

If the `array_address` is `NULL`, return immediately. Otherwise, given the (data) address of an array, the non-negative size of prefix padding for the cookie, and the size of its elements, call the given destructor on each element, using the cookie to determine the number of elements, and then delete the space by calling `::operator delete[](void *)`. If the destructor throws an exception, rethrow after (a) destroying the remaining elements, and (b) deallocating the storage. If the destructor throws a second exception, call `terminate()`. If `padding_size` is 0, the destructor pointer must be `NULL`. If the destructor pointer is `NULL`, no destructor call is to be made.



The intent of this function is to permit an implementation to call this function when confronted with an expression of the form `delete[] p` in the source code, provided that the default deallocation function can be used. Therefore, the semantics of this function are consistent with those required by the standard. The requirement that the deallocation function be called even if the destructor throws an exception derives from the resolution to DR 353 to the C++ standard, which was adopted in April, 2003.

```
extern "C" void __cxa_vec_delete2 (
    void *array_address,
    size_t element_size,
    size_t padding_size,
    void (*destructor) ( void *this ),
    void (*dealloc) ( void *obj ) );
```

Same as `__cxa_vec_delete`, except that the given function is used for deallocation instead of the default delete function. If `dealloc` throws an exception, the result is undefined. The `dealloc` pointer may not be `NULL`.

```
extern "C" void __cxa_vec_delete3 (
    void *array_address,
    size_t element_size,
    size_t padding_size,
    void (*destructor) ( void *this ),
    void (*dealloc) ( void *obj, size_t size ) );
```

Same as `__cxa_vec_delete`, except that the given function is used for deallocation instead of the default delete function. The deallocation function takes both the object address and its size. If `dealloc` throws an exception, the result is undefined. The `dealloc` pointer may not be `NULL`.

```
extern "C" void __cxa_vec_ctor (
    void *dest_array,
    void *src_array,
    size_t element_count,
    size_t element_size,
    void (*constructor) (void *destination, void *source),
    void (*destructor) (void *));
```

Given the (data) addresses of a destination and a source array, an element count and an element size, call the given copy constructor to copy each element from the source array to the destination array. The copy constructor's arguments are the destination address and source address, respectively. If an exception occurs, call the given destructor (if non-`NULL`) on each copied element and rethrow. If the destructor

throws an exception, call `terminate()`. The constructor and or destructor pointers may be `NULL`. If either is `NULL`, no action is taken when it would have been called.

3.3.4 Controlling Object Construction Order

3.3.4.1 Motivation

The only requirement of the C++ Standard with respect to file scope object construction order is that file scope objects in a single object file are constructed in declaration order. However, building large programs sometimes requires careful attention to construction ordering for objects in different object files, and a number of vendors have provided extra-lingual facilities to control it. This ABI does not require an implementation to support this capability, but it specifies such a facility for those implementations that do.

This facility only controls construction order within a singled linked object (executable or DSO). Construction order between linked objects is determined by the initialization ordering specified in the base ABI.

3.3.4.2 Source Code API

A user may specify the construction priority with the pragma:

```
#pragma priority ( <priority> )
```

The `<priority>` parameter specifies a 32-bit signed initialization priority, with lower numbers meaning earlier initialization. The range of priorities `[MIN_INT .. MIN_INT+1023]` is reserved to the implementation. The pragma applies to all file scope variables in the file where it appears, from the point of appearance to the next priority pragma or the end of the file. Objects defined before any priority pragmas have a default priority of zero, as do initialization actions specified by other means, e.g. `DT_INIT_ARRAY` entries. For consistency with the C++ Standard requirements on initialization order, behavior is undefined unless the priorities appearing in a single file, including any default zero priorities, are in non-decreasing numeric (non-increasing priority) order.

Initialization entries with the same priority from different files (or from other sources such as link command options) will be executed in an unspecified order.

3.3.4.3 Object File Representation

Initialization priority is represented in the object file by elements of a target-specific section type, **SHT_IA_64_PRIORITY_INIT**, with section ID `0x79000000` on Itanium, and section name `.priority_init`, and attributes allowing writing but not execution. The elements are structs:

```
typedef struct {
    ElfXX_Word    pi_pri;
    ElfXX_Addr    pi_addr;
} ElfXX_Priority_Init;
```

The field `pi_addr` is a function pointer, as defined by the base ABI (a pointer to a function descriptor on Itanium). The function takes a single unsigned `int` priority parameter, which performs some initialization at priority `pi_pri`. The priority value is obtained from the signed `int` in the source pragma by subtracting `MIN_INT`, so the default priority is `-MIN_INT`. The section header field `sh_entsize` is 8 for ELF-32, or 16 for ELF-64.



An implementation may initialize as many (or as few) objects of the same priority as it chooses in a single such initialization function, as long as the sequence of such initialization entries for a given file preserves the source code order of objects to be initialized.

3.3.4.4 Runtime Library Support

Each implementation supporting priority initialization shall provide a runtime library function with prototype:

```
void __cxa_priority_init ( ElfXX_Priority_Init *pi, int cnt );
```

It will be called with the address of a `cnt`-element (sub-)vector of the priority initialization entries, and must call each of them in order. It will be called with the GP of the initialization entries.

3.3.4.5 Linker Processing

The only required static linker processing is to concatenate the `SHT_IA_64_PRIORITY_INIT` sections in link order, which, given equal section IDs, section names, and section attributes as specified above, is the default behavior specified by the generic ABI for unknown section types.



Given minimum static linker processing, an implementation supporting priority initialization would need to include bracketing files in the link command that (1) label the ends of the `SHT_IA_64_PRIORITY_INIT` section, and (2) provide initial and final `DT_INIT_ARRAY` entries. The initial `DT_INIT_ARRAY` entry would need to sort the `SHT_IA_64_PRIORITY_INIT` section and call `__cxa_priority_init` to run the constructors with negative priority (in the source). The final `DT_INIT_ARRAY` entry would need to call `__cxa_priority_init` to run the constructors with non-negative priority. Other `DT_INIT_ARRAY` entries would thus run at the proper point in the priority sequence.

A more ambitious linker implementation could sort the `SHT_IA_64_PRIORITY_INIT` section at link time and fabricate the code to call `__cxa_priority_init` at the beginning and end. At the extreme, it could even include other `DT_INIT_ARRAY` entries in the `SHT_IA_64_PRIORITY_INIT` sequence at the appropriate places and emit exactly one call to `__cxa_priority_init`, with no other entries in the `DT_INIT_ARRAY` section.

3.3.5 DSO Object Destruction API

3.3.5.1 Motivation

The C++ Standard requires that destructors be called for global objects when a program exits in the opposite order of construction. Most implementations have handled this by calling the C library `atexit` routine to register the destructors. This is problematic because the 1999 C Standard only requires that the implementation support 32 registered functions, although most implementations support many more. More important, it does not deal at all with the ability in most implementations to remove DSOs from a running program image by calling `dlopen` prior to program termination.

The API specified below is intended to provide standard-conforming treatment during normal program exit, which includes executing `atexit`-registered functions in the correct sequence relative to constructor-registered destructors, and reasonable treatment during early DSO unload (e.g. `dlopen`).

3.3.5.2 Runtime Data Structure

The runtime library shall maintain a list of termination functions with the following information about each:

- A function pointer (a pointer to a function descriptor on Itanium).
- A `void*` operand to be passed to the function.
- A `void*` handle for the *home DSO* of the entry (below).

The representation of this structure is implementation defined. All references are via the API described below.

3.3.5.3 Runtime API

A. Object construction:

After constructing a global (or local static) object, that will require destruction on exit, a termination function is *registered* as follows:

```
extern "C" int __cxa_atexit ( void (*f)(void *), void *p, void *d );
```

This registration, e.g. `__cxa_atexit(f,p,d)`, is intended to cause the call `f(p)` when DSO `d` is unloaded, before all such termination calls registered before this one. It returns zero if registration is successful, nonzero on failure.

The registration function is not called from within the constructor.

B. User `atexit` calls:

When the user registers exit functions with `atexit`, they should be registered with NULL parameters and DSO handles, i.e.

```
__cxa_atexit ( f, NULL, NULL );
```

It is expected that implementations supporting both C and C++ will integrate this capability into the `libc` `atexit` implementation so that C-only DSOs will nevertheless interact with C++ programs in a C++-standard-conforming manner. No user interface to `__cxa_atexit` is supported, so the user is not able to register an `atexit` function with a parameter or a home DSO.

C. Termination:

When linking any DSO containing a call to `__cxa_atexit`, the linker should define a hidden symbol `__dso_handle`, with a value which is an address in one of the object's segments. (It does not matter what address, as long as they are different in different DSOs.) It should also include a call to the following function in the FINI list (to be executed first):

```
extern "C" void __cxa_finalize ( void *d );
```

The parameter passed should be `&__dso_handle`.

Note that the above can be accomplished either by explicitly providing the symbol and call in the linker, or by implicitly including a relocatable object in the link with the necessary definitions, using a `.fini_array` section for the FINI call. Also, note that these can be omitted for an object with no calls to `__cxa_atexit`, but they can be safely included in all objects.

When `__cxa_finalize(d)` is called, it should walk the termination function list, calling each in turn if `d` matches `__dso_handle` for the termination function entry. If `d == NULL`, it should call all of them. Multiple calls to `__cxa_finalize` shall not result in calling termination function entries multiple times; the implementation may either remove entries or mark them finished.

When the main program calls `exit`, it must call any remaining `__cxa_atexit`-registered functions, either by calling `__cxa_finalize(NULL)`, or by walking the registration list itself.

Note that the destructors must be called by `__cxa_finalize()` in the opposite of the order in which they were enqueued by `__cxa_atexit`.

Since `__cxa_atexit` and `__cxa_finalize` must both manipulate the same termination function list, they must be defined in the implementation's runtime library, rather than in the individual linked objects.

3.4 Demangler API

Synopsis:

```
namespace abi {
    extern "C" char* __cxa_demangle (const char* mangled_name,
                                     char* buf,
                                     size_t* n,
                                     int* status);
}
```

- `mangled-name` is a pointer to a null-terminated array of characters. It may be either an external name, i.e. with a `"_Z"` prefix, or an internal NTBS mangling, e.g. of a type for `type_info`.
- `buf` may be null. If it is non-null, then `n` must also be nonnull, and `buf` is a pointer to an array, of at least `*n` characters, that was allocated using `malloc`.
- `status` points to an `int` that is used as an error indicator. It is permitted to be null, in which case the user just doesn't get any detailed error information.

Behavior: The return value is a pointer to a null-terminated array of characters, the demangled name.

Ambiguities are possible between `extern "C"` object names and type manglings, e.g. `"i"` may be either an object named `"i"` or the built-in `"int"` type. Such ambiguous arguments are assumed to be type manglings. If the user has a set of external names to demangle, they should check that the names are in fact mangled (that is, begin with `"_Z"`) before passing them to `__cxa_demangle`.

If there is an error in demangling, the return value is a null pointer. The user can examine `*status` to find out what kind of error occurred. Meaning of error indications:

- 0: success
- -1: memory allocation failure
- -2: invalid mangled name
- -3: invalid arguments (e.g. `buf` nonnull and `n` null)

Memory management:

- If `buf` is a null pointer, `__cxa_demangle` allocates a new buffer with `malloc`. It stores the size of the buffer in `*n`, if `n` is not NULL.
- If `buf` is not a null pointer, it must have been allocated with `malloc`. If `buf` is not big enough to store the resulting demangled name, `__cxa_demangle` must either a) call `free` to deallocate `buf` and then allocate a new buffer with `malloc`, or b) call `realloc` to increase the size of the buffer. In either case, the new buffer size will be stored in `*n`.

Chapter 4: Exception Handling

See [Exception Handling](#) document, currently just the base psABI-level material, and the HP [exception handling](#) working paper, 8 December 1999.

Chapter 5: Linkage and Object Files

5.1 External Names (a.k.a. Mangling)

5.1.1 General

This section specifies the *mangling*, i.e. encoding, of external names (external in the sense of being visible outside the object file where they occur). The encoding is formalized as a derivation grammar along with the explanatory text, in a modified BNF with the following conventions:

- Non-terminals are delimited by diamond braces: "<>".
- Italics in non-terminals are modifiers to be ignored, e.g. <function name> is the same as <name>.
- Spaces are to be ignored.
- Text beginning with '#' is comments, to be ignored.
- Tokens in square brackets "[]" are optional.
- Tokens are placed in parentheses "()" for grouping purposes.
- '*' repeats the preceding item 0 or more times.
- '+' repeats the preceding item 1 or more times.
- All other characters are terminals, representing themselves.

See the separate [table](#) summarizing the encoding characters used as terminals. See the separate [table](#) in the separate ABI examples document.

In the various explanatory examples, we use Ret? for an unknown function return type (i.e. that is not given by the mangling), or Type? for an unknown data type.

5.1.2 General Structure

Entities with C linkage and global namespace variables are not mangled. Mangled names have the general structure:

```
<mangled-name> ::= _Z <encoding>
<encoding> ::= <function name> <bare-function-type>
             ::= <data name>
             ::= <special-name>
```

Thus, a name is mangled by prefixing "_Z" to an encoding of its name, and in the case of functions its type (to support overloading). At this top level, function types do not have the special delimiter characters required when nested (see below). The type is omitted for variables and static data members.

For the purposes of mangling, the name of an anonymous union is considered to be the name of the first named data member found by a pre-order, depth-first, declaration-order walk of the data members of the anonymous union. If there is no such data member (i.e., if all of the data members in the union are unnamed), then there is no way for a program to refer to the anonymous union, and there is therefore no need to mangle its name.

All of these examples:

```
union { int i; int j; };
union { union { int : 7 }; union { int i; }; };
union { union { int j; } i; };
```

are considered to have the name `i` for the purposes of mangling.

```
<name> ::= <nested-name>
        ::= <unscoped-name>
        ::= <unscoped-template-name> <template-args>
        ::= <local-name>           # See Scope Encoding below

<unscoped-name> ::= <unqualified-name>
                  ::= St <unqualified-name>   # ::std::
```

```

<unscoped-template-name> ::= <unscoped-name>
                           ::= <substitution>

```

Names of objects nested in namespaces or classes are identified as a delimited sequence of names identifying the enclosing scopes. In addition, when naming a class member function, CV-qualifiers may be prefixed to the compound name, encoding the `this` attributes. Note that if member function CV-qualifiers are required, the delimited form must be used even if the remainder of the name is a single substitution.

```

<nested-name> ::= N [<CV-qualifiers>] <prefix> <unqualified-name> E
               ::= N [<CV-qualifiers>] <template-prefix> <template-args> E

<prefix> ::= <prefix> <unqualified-name>
           ::= <template-prefix> <template-args>
           ::= <template-param>
           ::= # empty
           ::= <substitution>

<template-prefix> ::= <prefix> <template unqualified-name>
                   ::= <template-param>
                   ::= <substitution>

<unqualified-name> ::= <operator-name>
                    ::= <ctor-dtor-name>
                    ::= <source-name>

<source-name> ::= <positive length number> <identifier>
<number> ::= [n] <non-negative decimal integer>
<identifier> ::= <unqualified source code identifier>

```

<number> is a pseudo-terminal representing a decimal integer, with a leading 'n' for negative integers. It is used in <source-name> to provide the byte length of the following identifier. <number>s appearing in mangled names never have leading zeroes, except for the value zero, represented as '0'. <identifier> is a pseudo-terminal representing the unqualified identifier for the entity in the source code.

Note that <source-name> in the productions for <unqualified-name> may be either a function or data object name when derived from <name>, or a class or enum name when derived from <type>.

5.1.3 Operator Encodings

Operators appear as function names, and in nontype template argument expressions. Unlike Cfront, unary and binary operators using the same symbol have different encodings. All operators are encoded using exactly two letters, the first of which is lowercase.

```

<operator-name> ::= nw      # new
                 ::= na      # new[]
                 ::= dl      # delete
                 ::= da      # delete[]
                 ::= ps      # + (unary)
                 ::= ng      # - (unary)
                 ::= ad      # & (unary)
                 ::= de      # * (unary)
                 ::= co      # ~
                 ::= pl      # +
                 ::= mi      # -
                 ::= ml      # *
                 ::= dv      # /
                 ::= rm      # %
                 ::= an      # &
                 ::= or      # |

```



```

::= eo      # ^
::= aS      # =
::= pL      # +=
::= mI      # -=
::= mL      # *=
::= dV      # /=
::= rM      # %=
::= aN      # &=
::= oR      # |=
::= eO      # ^=
::= lS      # <<
::= rS      # >>
::= lS      # <<=
::= rS      # >>=
::= eq      # ==
::= ne      # !=
::= lt      # <
::= gt      # >
::= le      # <=
::= ge      # >=
::= nt      # !
::= aa      # &&
::= oo      # ||
::= pp      # ++
::= mm      # --
::= cm      # ,
::= pm      # ->*
::= pt      # ->
::= cl      # ( )
::= ix      # [ ]
::= qu      # ?
::= st      # sizeof (a type)
::= sz      # sizeof (an expression)
::= cv <type> # (cast)
::= v <digit> <source-name> # vendor extended operator

```

Vendors who define builtin extended operators (e.g. `__alignof`) shall encode them as a 'v' prefix followed by the operand count as a single decimal digit, and the name in `<length,ID>` form.

i *For a user-defined conversion operator the result type (i.e., the type to which the operator converts) is part of the mangled name of the function. If the conversion operator is a member template, the result type will appear before the template parameters. There may be forward references in the result type to the template parameters.*

5.1.4 Other Special Functions and Entities

Associated with a virtual table are several entities with mangled external names: the virtual table itself, the VTT for construction, the typeinfo structure, and the name it references. Each has a `<special-name>` encoding that is a simple two-character code, prefixed to the type encoding for the class to which it applies.

```

<special-name> ::= TV <type> # virtual table
                ::= TT <type> # VTT structure (construction vtable index)
                ::= TI <type> # typeinfo structure
                ::= TS <type> # typeinfo name (null-terminated byte string)

```

Initialization of function-scope static objects requires a guard variable to prevent multiple initialization. These are mangled as a 'GV' prefix with the mangled name of the object being guarded.

```

<special-name> ::= GV <object name> # Guard variable for one-time initialization

```

No <type>

Virtual function override thunks come in two forms. Those overriding from a non-virtual base, with fixed *this* adjustments, use a "Th" prefix and encode the required adjustment offset, probably negative, indicated by a 'n' prefix, and the encoding of the target function. Those overriding from a virtual base must encode two offsets after a "Tv" prefix. The first is the constant adjustment to the nearest virtual base (of the full object), of which the defining object is a non-virtual base. It is coded like the non-virtual case, with a 'n' prefix if negative. The second offset identifies the vcall offset in the nearest virtual base, which will be used to finish adjusting *this* to the full object. After these two offsets comes the encoding of the target function. The target function encodings of both thunks incorporate the function type; no additional type is encoded for the thunk itself.

```
<special-name> ::= T <call-offset> <base encoding>
                  # base is the nominal target function of thunk
<call-offset> ::= h <nv-offset> _
                  ::= v <v-offset> _
<nv-offset> ::= <offset number>
              # non-virtual base override
<v-offset> ::= <offset number> _ <virtual offset number>
              # virtual base override, with vcall offset
```

Virtual function override thunks with covariant returns are twice as complex. Just as normal virtual function override thunks must adjust the *this* pointer before calling the base function, those with covariant returns must adjust the return pointer after they return from the base function. So the mangling must also encode a fixed offset to a non-virtual base, and possibly an offset to a vbase offset in the vtable to get to the virtual base containing the result subobject. We achieve this by encoding two <call-offset> components, either of which may be either virtual or non-virtual.

```
<special-name> ::= Tc <call-offset> <call-offset> <base encoding>
                  # base is the nominal target function of thunk
                  # first call-offset is 'this' adjustment
                  # second call-offset is result adjustment
```

Constructors and destructors are simply special cases of <unqualified-name>, where the final <unqualified-name> of a nested name is replaced by one of the following:

```
<ctor-dtor-name> ::= C1      # complete object constructor
                  ::= C2      # base object constructor
                  ::= C3      # complete object allocating constructor
                  ::= D0      # deleting destructor
                  ::= D1      # complete object destructor
                  ::= D2      # base object destructor
```

5.1.5 Type encodings

Types are encoded as follows:

```
<type> ::= <builtin-type>
        ::= <function-type>
        ::= <class-enum-type>
        ::= <array-type>
        ::= <pointer-to-member-type>
        ::= <template-param>
```

```

::= <template-template-param> <template-args>
::= <substitution> # See Compression below

```

Types are qualified (optionally) by single-character prefixes encoding cv-qualifiers and/or pointer, reference, complex, or imaginary types:

```

<type> ::= <CV-qualifiers> <type>
        ::= P <type>      # pointer-to
        ::= R <type>      # reference-to
        ::= C <type>      # complex pair (C 2000)
        ::= G <type>      # imaginary (C 2000)
        ::= U <source-name> <type>      # vendor extended type qualifier

<CV-qualifiers> ::= [r] [V] [K]      # restrict (C99), volatile, const

```

Vendors who define extended type qualifiers (e.g. `_near`, `_far` for pointers) shall encode them as a 'U' prefix followed by the name in `<length,ID>` form.

In cases where multiple order-insensitive qualifiers are present, they should be ordered 'K' (closest to the base type), 'V', 'r', and 'U' (farthest from the base type), with the 'U' qualifiers in alphabetical order by the vendor name (with alphabetically earlier names closer to the base type). For example, `int* volatile const restrict _far p` has mangled type name `U4_farrVKPi`.

Vendors must therefore specify which of their extended qualifiers are considered order-insensitive, not necessarily on the basis of whether their language translators impose an order in source code. They are encouraged to resolve questionable cases as being order-insensitive to maximize consistency in mangling.

For purposes of substitution, given a CV-qualified type, the base type is substitutable, and the type with all the C, V, and r qualifiers plus any vendor extended types in the same order-insensitive set is substitutable; any type with a subset of those qualifiers is not. That is, given a type `const volatile foo`, the fully qualified type or `foo` may be substituted, but not `volatile foo` nor `const foo`. Also, note that the grammar above is written with the assumption that vendor extended type qualifiers will be in the order-sensitive (not CV) set. An appropriate grammar modification would be necessitated by an order-insensitive vendor extended type qualifier like `const` or `volatile`.

i *The restrict qualifier is part of the C99 standard, but is strictly an extension to C++ at this time. There is no standard specification of whether the restrict attribute is part of the type for overloading purposes. An implementation should include its encoding in the mangled name if and only if it also treats it as a distinguishing attribute for overloading purposes. This ABI does not specify that choice.*

Builtin types are represented by single-letter codes:

```

<builtin-type> ::= v # void
                ::= w # wchar_t
                ::= b # bool
                ::= c # char
                ::= a # signed char
                ::= h # unsigned char
                ::= s # short
                ::= t # unsigned short
                ::= i # int
                ::= j # unsigned int
                ::= l # long
                ::= m # unsigned long
                ::= x # long long, __int64

```

```

::= y # unsigned long long, __int64
::= n # __int128
::= o # unsigned __int128
::= f # float
::= d # double
::= e # long double, __float80
::= g # __float128
::= z # ellipsis
::= u <source-name> # vendor extended type

```

Vendors who define builtin extended types shall encode them as a 'u' prefix followed by the name in <length,ID> form.

Function types are composed from their parameter types and possibly the result type. Except at the outer level type of an <encoding>, or in the <encoding> of an otherwise delimited external name in a <template-parameter> or <local-name> function encoding, these types are delimited by an "F..E" pair. For purposes of substitution (see [Compression](#) below), delimited and undelimited function types are considered the same.

Whether the mangling of a function type includes the return type depends on the context and the nature of the function. The rules for deciding whether the return type is included are:

1. Template functions (names or types) have return types encoded, with the exceptions listed below.
2. Function types not appearing as part of a function name mangling, e.g. parameters, pointer types, etc., have return type encoded, with the exceptions listed below.
3. Non-template function names do not have return types encoded.

The exceptions mentioned in (1) and (2) above, for which the return type is never included, are

- Constructors.
- Destructors.
- Conversion operator functions, e.g. `operator int`.

Empty parameter lists, whether declared as `()` or conventionally as `(void)`, are encoded with a void parameter specifier (`v`). Therefore function types always encode at least one parameter type, and function manglings can always be distinguished from data manglings by the presence of the type. Member functions do not encode the types of implicit parameters, either `this` or the VTT parameter.

A "Y" prefix for the bare function type encodes extern "C". If there are any cv-qualifiers of `this`, they are encoded at the beginning of the <qualified-name> as described above. This affects only type mangling, since extern "C" function objects have unmangled names.

```

<function-type> ::= F [Y] <bare-function-type> E
<bare-function-type> ::= <signature type>+
# types are possible return type, then parameter types

```

A class, union, or enum type is simply a name. It may be a simple <unqualified-name>, with or without a template argument list, or a more complex <nested-name>. Thus, it is encoded like a function name, except that no CV-qualifiers are present in a nested name specification.

```

<class-enum-type> ::= <name>

```

Array types encode the dimension (number of elements) and the element type. Note that "array" parameters to functions are encoded as pointer types. For variable length arrays (C99 VLAs), the dimension (but not the '_' separator) is omitted.

```

<array-type> ::= A <positive dimension number> _ <element type>
              ::= A [<dimension expression>] _ <element type>

```

When the dimension is an expression involving template parameters, the second production is used. Thus, the declarations:

```

template<int I> void foo (int (&)[I + 1]) { }
template void foo<2> (int (&)[3]);

```

produce the mangled name "_Z3fooILi2EEvRAp1T_Li1E_i".

Pointer-to-member types encode the class and member types.

```

<pointer-to-member-type> ::= M <class type> <member type>

```

Note that for a pointer to cv-qualified member function, the qualifiers are attached to the function type, so

```

struct A;
void f (void (A::*)() const) {}

```

produces the mangled name "_Z1fM1AKFvvE".

When function and member function template instantiations reference the template parameters in their parameter/result types, the template parameter number is encoded, with the sequence T_, T0_, ... Class template parameter references are mangled using the standard mangling for the actual parameter type, typically a substitution. Note that a template parameter reference is a substitution candidate, distinct from the type (or other substitutable entity) that is the actual parameter.

```

<template-param> ::= T_          # first template parameter
                  ::= T <parameter-2 non-negative number> _
<template-template-param> ::= <template-param>
                           ::= <substitution>

```

Template argument lists appear after the unqualified template name, and are bracketed by I/E. This is used in names for specializations in particular, but also in types and scope identification.

```

<template-args> ::= I <template-arg>+ E
<template-arg> ::= <type>                # type or template
                 ::= X <expression> E    # expression
                 ::= <expr-primary>      # simple expressions

<expression> ::= <unary operator-name> <expression>
               ::= <binary operator-name> <expression> <expression>
               ::= <trinary operator-name> <expression> <expression> <expression>
               ::= st <type>
               ::= <template-param>
               ::= sr <type> <unqualified-name>                # dependent name
               ::= sr <type> <unqualified-name> <template-args> # dependent template-id
               ::= <expr-primary>

<expr-primary> ::= L <type> <value number> E                # integer literal
                ::= L <type> <value float> E                # floating literal
                ::= L <mangled-name> E                       # external name

```

Type arguments appear using their regular encoding. For example, the template class "A<char, float>" is encoded as "1A1cfE". A slightly more involved example is a dependent function parameter type "A<T2>::X" (T2 is the second template parameter) which is encoded as "N1AIT0_E1XE", where the "N...E" construct is used to describe a qualified name.

Literal arguments, e.g. "A<42L>", are encoded with their type and value. Negative integer values are preceded with "n"; for example, "A<-42L>" becomes "1AILln42EE". The bool value false is encoded as 0, true as 1.

Floating-point literals are encoded using a fixed-length lowercase hexadecimal string corresponding to the internal representation (IEEE on Itanium), high-order bytes first, without leading zeroes. For example: "Lf bf800000 E" is -1.0f on Itanium.

The encoding for a literal of an enumerated type is the encoding of the type name followed by the encoding of the numeric value of the literal in its base integral type (which deals with values that don't have names declared in the type).

A reference to an entity with external linkage is encoded with "L<mangled name>E". For example:

```
void foo(char); // mangled as _Z3fooc
template<void (&)(char)> struct CB;
// CB<foo> is mangled as "2CBIL_Z3foocEE"
```

The <encoding> of an extern "C" function is treated like global-scope data, i.e. as its <source-name> without a type. For example:

```
extern "C" bool IsEmpty(char *); // (un)mangled as IsEmpty
template<void (&)(char *)> struct CB;
// CB<IsEmpty> is mangled as "2CBIL_Z7IsEmptyEE"
```

An expression, e.g., "B<(J+1)/2>", is encoded with a prefix traversal of the operators involved, delimited by "X...E". The operators are encoded using their two letter mangled names. For example, "B<(J+1)/2>", if J is the third template parameter, becomes "1BI Xdv pl T1_Li1E Li2E E E" (the blanks are present only to visualize the decomposition). Note that the expression is mangled without constant folding or other simplification, and without parentheses, which are implicit in the prefix representation. Except for the parentheses, therefore, it represents the source token stream. (C++ Standard reference 14.5.5.1 p. 5.)

If an expression is a qualified-name, and the qualifying scope is a dependent type, one of the *sr* productions is used, rather than the <mangled-name> production. If the qualified name refers to an operator for which both unary and binary manglings are available, the mangling chosen is the mangling for the binary version.

5.1.6 Scope Encoding

A nonlocal scope is encoded as the qualifier of a qualified name: it can be the top-level name qualification or it can appear inside <type> to denote dependent types or bind specific names as arguments. Qualified names are encoded as:

```
N <qual 1> ... <qual N> <unqual name> E
```

where each <qual K> is the encoding of a namespace name or a class name (with the latter possibly including a template argument list).

Occasionally entities in local scopes must be mangled too (e.g. because inlining or template compilation causes multiple translation units to require access to that entity). The encoding for such entities is as follows:

```

<local-name> := Z <function encoding> E <entity name> [<discriminator>]
              := Z <function encoding> E s [<discriminator>]
<discriminator> := _ <non-negative number>

```

The first production is used for named local static objects and classes, which are identified by their declared names. The *<entity name>* may itself be a compound name, but it is relative to the closest enclosing function, i.e. none of the components of the function encoding appear in the entity name. It is possible to have nested function scopes, e.g. when dealing with a member function in a local class. In such cases, the function encoding will itself have *<local-name>* structure. The discriminator is used only for the second and later occurrences of the same name within a single function. In this case *<number>* is $n - 2$, if this is the n th occurrence, in lexical order, of the given name.

The second production is used for string literals. The discriminator is used only if there is more than one, for the second and subsequent ones. In this case *<number>* is $n - 2$, if this is the n th distinct string literal, in lexical order, appearing in the function. Multiple references to the same string literal produce one string object with one name in the sequence. *Note that this assumes that the same string literal occurring twice in a given function in fact represents a single entity, i.e. has a unique address.*

For both named objects and string literals, the numbering order is strictly lexical order based on the original token sequence. All objects occurring in that sequence are to be numbered, even if subsequent optimization makes some of them unnecessary. The ordering of literals appearing in a mem-initializer-list shall be the order that the literals appear in the source, which may be different from the order in which the initializers will be executed when the program runs. It is expected that this will be the 'natural' order in most compilers. In any case, conflicts would arise only if different compilation units including the same code were compiled by different compilers, and multiple entities requiring mangling had the same name.

For static objects in constructors and destructors, the mangling of the complete object constructor or destructor is used as the base function name, i.e. the C1 or D1 version. This yields mangled names that are consistent across the versions.

Example:

```

namespace N {
    inline char* f(int i) {
        static char *p = "Itanium C++ ABI"; // p = 1, "... " = 2
        { struct X { // X = 3
            void g() {}
        }; }
        return p[i];
    }
}

```

- `"_ZZN1N1fEiE1p"`: encoding of `N::f::p` (first local mangled entity)
- `"_ZZN1N1fEiEs"`: encoding of `N::f::"Itanium C++ ABI"` (no discriminator)
- `"_ZNZN1N1fEiE1X1gE"`: encoding of `N::f::X::g()` (third local mangled entity used as a class-qualifier)

See additional examples in the [ABI examples](#) document.

5.1.7 Compression

To minimize the length of external names, we use two mechanisms, a substitution encoding to eliminate repetition of name components, and abbreviations for certain common names. Each non-terminal in the grammar above for which *<substitution>* appears on the right-hand side is both a source of future substitutions and a candidate for being substituted. There are two exceptions that appear to be substitution candidates from the grammar, but are explicitly excluded:

- <builtin-type> other than vendor extended types, and
- function and operator names other than extern "C" functions.

All substitutions are for entities that would appear in a symbol table. In particular, we make substitutions for prefixes of qualified names, but not for arbitrary components of them. Thus, the components `::n1::foo()` and `::n2::foo()` appearing in the same name would not result in substituting for the second "foo." Similarly, we do not substitute for expressions, though names appearing in them might be substituted. The reason for this is to facilitate implementations that use the symbol table to keep track of components that might be substitutable.

Note that the above exclusion of function and operator names from consideration for substitution does not exclude the full function entity, i.e. its name plus its signature encoding.

Logically, the substitutable components of a mangled name are considered left-to-right, components before the composite structure of which they are a part. If a component has been encountered before, it is substituted as described below. This decision is independent of whether its components have been substituted, so an implementation may optimize by considering large structures for substitution before their components. If a component has not been encountered before, its mangling is identified, and it is added to a dictionary of substitution candidates. No entity is added to the dictionary twice.

The type of a non-static member function is considered to be different, for the purposes of substitution, from the type of a namespace-scope or static member function whose type appears similar. The types of two non-static member functions are considered to be different, for the purposes of substitution, if the functions are members of different classes. In other words, for the purposes of substitution, the class of which the function is a member is considered part of the type of function.



Therefore, in the following example:

```
typedef void T();
struct S {};
void f(T*, T (S::*)) {}
```

the function `f` is mangled as `_Z1fPFvvEM1SFvvE`; the type of the member function pointed to by the second parameter is not considered the same as the type of the function pointed to by the first parameter. Both function types are, however, entered the substitution table; subsequent references to either variant of the function type will result in the use of substitutions.

Substitution is according to the production:

```
<substitution> ::= S <seq-id> _
                ::= S_
```

The <seq-id> is a sequence number in base 36, using digits and upper case letters, and identifies the <seq-id>-th encoded component, in left-to-right order, starting at "0". As a special case, the first substitutable entity is encoded as "S_", i.e. with no number, so the numbered entities are the second one as "S0_", the third as "S1_", the twelfth as "SA_", the thirty-eighth as "S10_", etc. All substitutable components are so numbered, except those that have already been numbered for substitution. A component is earlier in the substitution dictionary than the structure of which it is a part. For example:

```
"_ZN1N1TiiiE2mfES0_IddE": Ret? N::T<int, int>::mf(N::T<double, double>)
```

since the substitutions generated for this name are:

```
"S_" == N (qualifier is less recent than qualified entity)
"S0_" == N::T (template-id comes before template)
(int is builtin, and isn't considered)
```



```
"S1_" == N::T<int, int>
"S2_" == N::T<double, double>
```

Note that substitutable components are the represented symbolic constructs, not their associated mangling character strings. Thus, a substituted object matches its unsubstituted form, and a delimited <function-type> matches its <bare-function-type>.

In addition, the following catalog of abbreviations of the form "Sx" are used:

```
<substitution> ::= St # ::std::
<substitution> ::= Sa # ::std::allocator
<substitution> ::= Sb # ::std::basic_string
<substitution> ::= Ss # ::std::basic_string < char,
                                ::std::char_traits<char>,
                                ::std::allocator<char> >
<substitution> ::= Si # ::std::basic_istream<char, std::char_traits<char> >
<substitution> ::= So # ::std::basic_ostream<char, std::char_traits<char> >
<substitution> ::= Sd # ::std::basic_iostream<char, std::char_traits<char> >
```

The abbreviation St is always an initial qualifier, i.e. appearing as the first element of a compound name. It does not require N...E delimiters unless either followed by more than one additional composite name component, or preceded by CV-qualifiers for a member function. This adds the case:

```
<name> ::= St <unqualified-name> # ::std::
```

For example:

```
"_ZSt5state": ::std::state
"_ZNSt3_In4wardE": ::std::_In::ward
```

5.2 Vague Linkage

Many objects in C++ are not clearly part of a single object file, but are required by the ODR to have a single definition. This section identifies, for such objects, where (i.e. in which objects) they should be emitted, and what special treatment might be required if duplicates are possible.

In many cases, we will deal with duplicates by putting possibly duplicated objects in distinct ELF sections or groups of sections, and using the COMDAT feature of SHT_GROUP sections in the gABI to remove duplicates. We will refer to this simply as using a COMDAT group, and specify the symbol to be used to identify duplicates in the SHT_GROUP section. *COMDAT groups are a new gABI feature specified during the Itanium ABI definition, and may not be implemented everywhere immediately. See the separate [ABI examples](#) document for a discussion of alternatives pending COMDAT implementation.*

Note that nothing in this section should be construed to require COMDAT usage for objects with internal linkage unless they may in fact be referenced outside the translation unit where they appear, for instance due to inlining.

5.2.1 Out-of-line Functions

It may sometimes be necessary or desirable to reference an out-of-line copy of a function declared inline, i.e. to reference a global symbol naming the function. This may occur because the implementation cannot, or chooses not to, inline the function, or because it needs an address rather than a call. In such a case, the function is to be emitted in each object where its name is referenced. A COMDAT group is used to eliminate duplicates, with the mangled name of the function as the identifying symbol.

5.2.2 Static Data

Inline functions, whether or not declared as such, and whether they are inline or out-of-line copies, may reference static data or character string literals, that must be kept in common among all copies by using the local symbol mangling defined above. These objects are named according to the rules for local names in the [Scope Encoding](#) section above, and the definition of each is emitted in a COMDAT group, identified by the symbol name described in the [Scope Encoding](#) section above. Each COMDAT group must be emitted in any object with references to the symbol for the object it contains, whether inline or out-of-line.

Local static data objects generally have associated guard variables used to ensure that they are initialized only once (see [3.3.2](#)). If the object is emitted using a COMDAT group, the guard variable must be too. It is suggested that it be emitted in the same COMDAT group as the associated data object, but it may be emitted in its own COMDAT group, identified by its name. In either case, it must be weak.

5.2.3 Virtual Tables

The virtual table for a class is emitted in the same object containing the definition of its *key function*, i.e. the first non-pure virtual function that is not inline at the point of class definition. If there is no key function, it is emitted everywhere used. The emitted virtual table includes the full virtual table group for the class, any new construction virtual tables required for subobjects, and the VTT for the class. They are emitted in a COMDAT group, with the virtual table mangled name as the identifying symbol. *Note that if the key function is not declared inline in the class definition, but its definition later is always declared inline, it will be emitted in every object containing the definition.*



In the abstract, a pure virtual destructor could be used as the key function, as it must be defined even though it is pure. However, the ABI committee did not realize this fact until after the specification of key function was complete; therefore a pure virtual destructor cannot be the key function.

5.2.4 Typeinfo

The RTTI `std::type_info` structure for a complete class type is emitted in the same object as its virtual table if dynamic, or everywhere referenced if not. The RTTI `std::type_info` structure for an incomplete class type is emitted wherever referenced. The RTTI `std::type_info` structures for various basic types as specified by the [Run-Time Type Information](#) section are provided by the runtime library. The RTTI name NTBS objects are emitted with each referencing `std::type_info` object.

The RTTI `std::type_info` structures for complete class types and basic types are emitted in COMDAT groups identified by their mangled names. The RTTI `std::type_info` structures for incomplete class types are emitted with other than the ABI-defined complete type mangled names; an implementation may choose to emit them as local static objects, or in COMDAT groups with implementation-defined names and COMDAT identifiers. The RTTI name NTBS objects are emitted in separate COMDAT groups identified by the NTBS mangled names as weak symbols.

5.2.5 Constructors and Destructors

Constructors and destructors for a class, whether implicitly-defined or user-defined, are emitted under the same rules as other functions. That is, user-defined constructors or destructors, unless the function is declared inline, or has internal linkage, are emitted where defined, with their complete, and base object variants. For destructors, in classes with a virtual destructor, the deleting variant is emitted as well. A user-defined constructor or destructor with non-inline, internal linkage is emitted where defined, with only the variants actually referenced. Implicitly-defined or inline user-defined constructors and destructors are emitted where referenced, each in its own COMDAT group identified by the constructor or destructor name.

This ABI does not require the generation or use of allocating constructors or deleting destructors for classes without a virtual destructor. However, if an implementation emits such functions, it must use the external names specified in this ABI. If such a function has external linkage, it must be emitted wherever referenced, in a COMDAT group whose name is the external name of the function.

5.2.6 Instantiated Templates

An instantiation of a class template requires:

- In the object where instantiated, the virtual table, any subobject construction virtual tables, and the VTT, are emitted in a COMDAT identified by the virtual table mangled name.
- Any static member data object is emitted in a COMDAT identified by its mangled name, in any object file with a reference to its name symbol.
- In the object where instantiated, virtual member functions are emitted in COMDAT groups identified by the function name.
- A non-inline, non-virtual member function is emitted in any object where its symbol is referenced (i.e. if the function is called without inlining, or its name is referenced without calling it), in a COMDAT group identified by the function name.

An instantiation of a function template or member function template is emitted in any object where its symbol is referenced (non-inline), in a COMDAT group identified by the function name.

5.3 Unwind Table Location

As described in the Itanium psABI, Itanium implementations shall produce unwind table entries in a `SHT_IA_64_UNWIND` section, and unwind information descriptors in a section that will be linked with the associated code. Itanium linkers shall put the unwind table, the unwind information table, and the associated code in a single text segment, with a `PT_IA_64_UNWIND` program table entry identifying the unwind table location.

Appendix R: Revision History

This version of this document is \$Revision: 1.3 \$. No special significance should be attached to the form of the revision number; it is simply a identifying number.

[050504] Remove use of `out0` for by-value return types on Itanium.

[050211] Reverse treatment of ambiguous arguments to `__cxa_demangle` (3.4).

[041118] Clarify the layout of bitfields.

[041025] Indicate that the TC1 definition of POD is intended in the section defining a "POD for the purpose of layout". Clearly indicate that an array whose elements are not PODs for the purpose of layout is itself not a POD for the purpose of layout.

[040923] Clarify behavior of `__cxa_vec_delete`.

[040219] Clarify substitution of member function types.

- [031128] Fix alphabetization of company names.
- [031123] Add note about forward references to template parameters in member template conversion operators.
- [031102] Specify the behavior of `__cxa_vec_delete` when the `array_address` is `NULL`.
- [030905] Specify the behavior of `__cxa_vec_new`, `__cxa_vec_new2`, and `__cxa_vec_new3` in the event that the allocation function returns `NULL`.
- [030609] Use `void*` instead of `dso_handle`.
- [030518] Specify behavior of `__cxa_vec_new2` and `__cxa_vec_new3` when the deallocation function throws an exception.
- [030518] Define "POD for the purpose of layout."
- [030316] Add acknowledgements section.
- [030313] Correct broken links and incorrect formatting.
- [030103] Clarify definition of substantively different types.
- [021222] Document mangling for anonymous unions.
- [021204] Remove note about 32-bit RTTI variation.
- [021125] Clarify guard functions.
- [021110] Clarify definition of nearly empty class.
- [021110] Clarify ordering of string literals in `mem-initializer-list`.
- [021110] Remove unnecessary V-adjusting thunks.
- [021110] Clarify VTT contents.
- [021021] Specify place and manner of emission for deleting destructors.
- [021021] Clarify mangling of pointer-to-member functions.
- [021016] Clarify mangling of floating-point literals.
- [021014] Clarify use of `sr` in mangling.
- [021011] Add mangling for unary plus.
- [021008] Make the names used for constructors and destructor entry points consistent throughout.
- [021008] Define manglings for `typename` types.
- [020916] Clarify ordering of functions in virtual function table. Correct mangling substitution example.
- [020906] Add trinary expression variant. Remove use of "low-order" to describe bytes in guard variables.
- [020827] Clarify definition of nearly empty class, `dsize`, `nvsized`, `nvaligned`.
- [020827] Clarify handling of tail-padding.

[020326] Clarify wording in `__cxa_demangle` memory management specification.

[020220] Clarify pointer to member function mangling (5.1.5).

[010407] Don't assume that virtual functions can be called through intermediate bases. Add notes about missed opportunities. The VTT parm isn't mangled, either.

[010315] Many outstanding updates. Empty classes passed as ordinary classes (3.1.3). Secondary virtual pointers for subobjects reachable via a virtual path (text of 2.6.1, text and example in 2.6.2). Note about locating virtual bases statically during construction (2.6.1). Rename IA-64 to Itanium throughout. Add `__cxa_vec_cleanup` (3.3.3).

[000817] Updates from 17 August meeting, email.

[000807] Added base document section (1.5). Further RTTI field name cleanup (2.9.5). Update proposed one-time construction API (3.3.2). Update proposed object construction priority API (3.3.4). Removed `<name>` substitution (5.1.2). COMDAT not generally necessary for internal linkage (5.2). COMDAT for local static guard variables (5.2.2).

[000727] Updates from 20 July meeting. Added section on controlling object construction order (3.3.4).

[000707] Introduce consistent `type_info` field names (2.9.5). Removed vmi flags for publicly/non-publicly inherited bases (2.9.5). Collect all construction/destruction APIs in one section (3.3). Added one-time initialization API (3.3.2). Vector construction/destruction routines are extern "C" (3.3.3). Added routines for vector construction/destruction (3.3.3). Added copy construction runtime API (3.3.3). Make Alex's changes in mangling grammar (5.1). Add `<special-name>` cases for covariant override thunks (5.1.4). Allow expressions as array type dimensions (5.1.5). Discuss vague linkage for virtual function override thunks (5.2.6).

[000621] Add scope section 1.4. Specify guard variables and vague linkage of static data (5.2.2) and instantiated templates (5.2.4). Clarify vcall offsets (2.5.3), VTT (2.6.2), mangling compression rules (5.1.7), and mangling examples.

[000511] Specify 32-bit form of `vmi_offset_flags`. Add export template note.

[000505] Updates from 4 May meeting. VTT is preorder, like everything else. Add issue C-3 destructor API. Added demangler API. Yet another try at the nested-name mangling grammar. Don't mangle builtin types (except vendor extended ones). Reverse mangling substitution order, and fix mangling substitution examples. Add vague linkage information for instantiated templates. Specify location of unwind tables.

[000502] Fixed mangling of template parameters again.

[000427] Reorganization and section numbering. Added [non-virtual function calling conventions](#).

[000417] Updates from 17 April meeting. Clarify order of vcall offsets. More elaboration of construction virtual table. Specification of COMDAT RTTI name. Reorganization of pointer RTTI. Modify mangling grammar to clarify substitution in compound names. Clarify Vague Linkage section.

[000407] Updates from 6 April meeting, email. More elaboration of construction vtable. Updates/issues in RTTI. Minor mangling changes. Added Vague Linkage section.

[000327] Updates from 30 March meeting. Define base classes to include self, proper base classes. Modify local function mangling per JFW proposal.

[000327] Updates from 23 March meeting. Adopt construction vtable Proposal B, and rewrite. Further work on mangling, especially substitution.

[000320] Clarify class size limit. Editorial changes in vtable components description. Add alternate to construction vtable proposal. Clarification in array cookie specification. Removed COMMON proxy from class RTTI. Extensive changes to mangling writeup.

[000314] Construction vtable modifications. RTTI modifications for incomplete class types. Mangling rework: grammar, new constructs, function return types.

[000309] Add limits section. Specify NULL member pointer values. Combine vtable content and order sections; clarify ordering. Specify when distinct virtual function entries are needed for overrides. Define (and modify) vector constructor/destructor runtime APIs. Virtual base offsets are promoted from non-virtual bases.

[000228] Add thunk definition. Revise inheritance graph order definition. Fix member function pointer description (no division by two). Move bitfield allocation description (much modified) to the non-virtual-base allocation description. Replace virtual function calling convention description.

[000228] Add thunk definition. Revise inheritance graph order definition. Fix member function pointer description (no division by two). Move bitfield allocation description (much modified) to the non-virtual-base allocation description. Replace virtual function calling convention description.

[000217] Add excess-size bitfield specification. Add namespace/header section. Touch up array new cookies. Remove construction vtable example to new file. Add mangling proposal.

[000214] Complete array new cookie specification. Remove unnecessary RTTI flags. Correct repeated inheritance flag description. Move all type_info subclasses in namespace abi, not namespace std. Note requirements for an implementation to prevent users from emitting invalid vtables for RTTI classes. Include construction vtable proposal.

[000203] Incorporate discussion of 3 February. Remove __reference_type_info (issue A-22). Restructure struct RTTI and flags (issue A-23). Clarify __base_class_info layout.

[000125] Incorporate discussion of 20 January, generally clarifications. Resolved A-19 (choice of a primary virtual base). Answered Nathan's questions about RTTI. Included RTTI "Deliberations" as rationale notes in the specification, or removed redundant ones. Added array operator new section.

[000119] Clarify when virtual base offsets are required. Note that a vtable has offset-to-top and RTTI entries for classes with virtual bases even if there are no virtual functions. Resolve allocation of a virtual base class that is a primary base for another base (A-17). Resolve choice of a primary virtual base class that is a primary base for another base (A-19). Describe the (non-)effect of virtual bases on the alignment of the non-virtual part of a class as the base of another class (A-18).

[991230] Integrate proposed resolution of A-16, A-17 in base class layout. Add outstanding questions list, and clean up questions in text.

[991229] Clarify definition of nearly empty class, layout of virtual bases.

[991203] Added description of vfunc calling convention from Jason.

[991104] Noted pair of vtable entries for virtual destructors.

[991019] Modified RTTI proposal for 14 October decisions.

[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.
