

A Common Vendor ABI for C++ GCC's Why, What and Not

Nathan Sidwell
CodeSourcery LLC
nathan@codesourcery.com

26th February 2003

Abstract

During 2000/2001 a C++ ABI was designed from scratch. GCC implemented that ABI and uses it for all architectures. This paper describes the origin of the ABI, why GCC uses it, what it is, what it is not. The ABI has some optimizations which improve user programs. User programs that have assumed more than the C++ language guarantees may be affected.

1 Why

The C++ language [1] has evolved over the past two decades. As the language changed, compilers had to be altered. One of the major interfaces in implementing a language is the Application Binary Interface or ABI. The ABI specifies everything needed to build modules that can be linked together. Often new language features required changes at the ABI level and prevented compatibility between compiler versions. Naturally, if a single compiler vendor could not maintain compatibility, cross vendor compatibility was impossible.

G++ evolved its ABI throughout its 2.7 series. Attempts were made to maintain backwards compatibility, but compatibility was not guaranteed, and users would have to recompile all source code upon upgrading the compiler in order to prevent ABI incompatibilities.

After C++ was standardized in 1998, it became feasible to specify an unchanging ABI. G++ could either have fixed the ABI that had previously evolved or design a new one from the ground up. Keeping the existing ABI would have the advantage of retaining binary compatibility. Unfortunately there were a number of deficiencies that would be impossible to remedy and still retain compatibility. Also, certain features could be implemented more efficiently. If backwards compatibility was going to be impossible, why not design an ABI with full knowledge of all the required features?

Around the same time that redesigning the ABI became possible, Intel was developing the Itanium architecture. As is normal, they had to define a suitable C ABI for the architecture so that C compilers could interoperate and use a efficient mechanisms optimized for the Itanium architecture. The C ABI had in two parts:

- A generic ABI, which is the System V ABI, and referred to as the gABI [2]. This describes generic methods for structure packing, parameter passing, etc.
- A processor specific description called the psABI [3]. This describes the processor specific details of the ABI such as the size and alignment of scalar types.

Intel also wanted a C++ ABI and a GCC port for the Itanium architecture. The IA64 C++ ABI group was formed from companies involved in Itanium work. These were CodeSourcery, Compaq, HP, Intel, Red Hat and SGI. If the resulting C++ ABI was sufficiently independent of the processor-specific psABI, G++ could use it for all the other targets too.

The group designed the Itanium C++ ABI `citecxxabi`. This ABI has been implemented in several major compilers. Although the ABI contains some Itanium specific features, those features are minor and do not prevent the ABI being used for other architectures.

2 What

As said above, the C++ ABI is built on top of the C ABI. That decision fixes the sizes of scalar types, provides a default packing algorithm, and parameter passing and return schemes. The C ABI also defines the unwind mechanism, which is used for exception handling. The C++ ABI must specify

- Some additional structure packing rules for bitfields.
- Additional types such as pointer to member.
- Class layout including single, multiple and virtual bases.
- A virtual function mechanism.
- Runtime type information and dynamic cast.
- Exception handling.
- Name mangling to distinguish different objects with the same local name.
- Global object construction and destruction.
- Passing reference arguments and empty arguments.

The full gory details are described in the C++ ABI [4]. Here I will cover most of the parts but in less detail.

2.1 Class Layout and Virtual Functions

C++ needs additional packing rules for class inheritance (single, multiple and virtual), virtual functions and type information. Also it requires additional rules for overlong bitfield¹ members and there are also some optimization opportunities. It is important

¹In C++ it is permissible to specify more bits in a bitfield than there are in the underlying type, for example `unsigned field :976`. The meaning of such a declaration is implementation defined.

that plain old data (POD) structures² remain C compatible to allow interoperating with C libraries.

As is commonly done, virtual functions are implemented with a virtual function table. This static table contains function pointers and is shared between all instances of a polymorphic type.³ Each instance of the type contains a data member which points to the virtual function table — a **vtable pointer**. Virtual function calling works by fetching the vtable pointer and then indexing the table appropriately for the particular virtual function being called. A virtual function table for a derived type must be similar to the table for the base it is inherited from.

The virtual function table can be used for additional functionality, namely runtime type information, virtual base location and overriding virtual functions from virtual bases. Thus the ABI has a vtable pointer for every class which declares or inherits virtual functions, or contains virtual bases. The ABI defines these as **dynamic classes**, and they are a superset of the polymorphic classes.

The vtable pointer is allocated at offset zero from an object's **this** pointer. This differs from the previous G++ ABI, which allocated it at the end of the class that first required it. There are two reasons. Firstly, it simplifies every use of the vtable, as its offset is the same for every class. Secondly, the Itanium architecture has no indexed addressing mode so it would have to use two instructions to access the vtable were it not at offset zero.

The vtable contains

- Virtual function pointers or descriptors.⁴
- An offset to fully derived object.
- A type information pointer.
- Virtual base class offsets.
- Adjustments for functions overriding virtual functions via a virtual base class. These are used for virtual adjusting thunks.

The vtable for a class is derived from the vtable of its primary base. The derived vtable surrounds the vtable of the primary and Figure 1 shows such a vtable layout. One of the novel features of the vtable is that it grows in both directions and the vtable pointer points into the middle of the vtable. Virtual base information and adjustments are held both before the vtable pointer, and virtual function pointers are after it.

Non-virtual base classes are allocated at the start of a class, in declaration order, much as if they are normal data members, except that,

- If one of the bases has a vtable pointer, the first declared base with a vtable pointer is allocated first. This base is called the **primary base**.

²POD structures are the C++ equivalent of C language structures. They have no constructor or destructor, no non-public fields, no virtual functions, no base classes, no copy assignment operator, no reference members and no non-POD members. POD unions are similarly defined.

³A polymorphic type is defined by the C++ standard as a class which defines or inherits a virtual function.

⁴Itanium uses function descriptors for its function pointer objects. The descriptors contain both a code address and a data address.

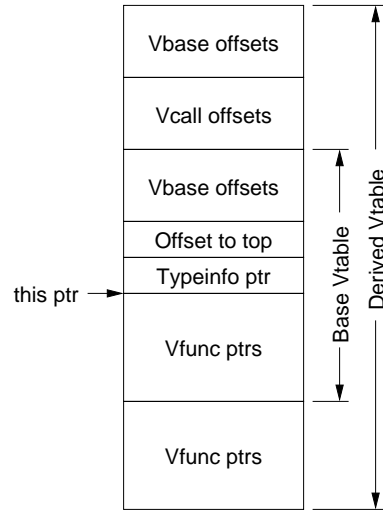


Figure 1: Vtable inheritance

- If no base has a vtable pointer, but the derived class requires one, a vtable pointer is allocated before all the bases.
- Any tail padding in a non-POD base class can be overlaid by another field or base. This is a space optimization, which the C++ standard permits, as that tail padding is not accessible by a user program in any manner. The tail padding of a POD struct is not available for reuse as a user program can legitimately alter it.
- An empty base class can be overlaid with any base, provided that no two bases of the same type are allocated at the same offset within the complete object. This requires a check of the complete base hierarchy to make sure no indirect empty base collides with the direct empty base.

Following the base classes come all the normal data members. Finally, the virtual bases are allocated at the end of the class. Again, these can have their tail padding overlaid, if they are non-POD. These virtual bases are only allocated for complete objects.

There are a number of implications of such packing rules:

- Classes have two distinct sizes. The size of a complete object, which includes the virtual bases and that is reported by `sizeof`. They have a smaller size when used as a base class, which does not include virtual bases, and can also specify an amount of tail padding that can be overlaid.
- Having the vtable pointer always at offset zero simplifies certain other features, but means that a singly inherited base class might *not* be at offset zero because the base had no vtable pointer, but one is needed in the derived class.
- Although C++ permits fields separated by an access specifier to be reordered, this is not used to minimize any internal packing.
- The value of a pointer to a base class is never less than the value of a pointer to the most derived object.

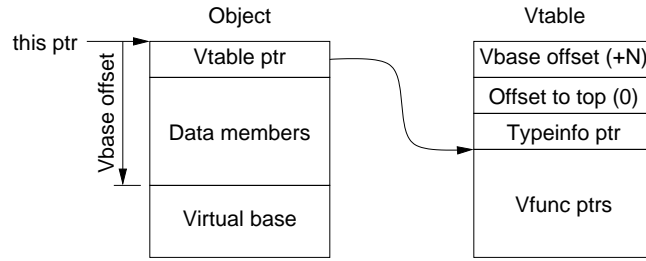


Figure 2: Virtual base

- There can be multiple vtable pointers within a dynamic object: the primary vtable pointer at offset zero and additional ones for each subsequent dynamic base class. These other pointers point to **secondary virtual function tables**.

In order to save space in the final executable, the vtable is only emitted in certain object files. If there is a non-inline virtual function declared in a class, the vtable is emitted only in the object file that the function is emitted in. The special virtual function is called a **key function**. If there is no key function, the vtable is emitted in every object file that might require its definition. However, it is given a special **vague linkage type**. This is often called common data, or **comdat**, so that the linker knows to eliminate duplicates.

2.1.1 Where is a Virtual Base?

Because the vtable contains the adjustments for virtual bases, converting to a virtual base is simply a question of adding the relevant offset, called a **vbbase offset** from the vtable to an object pointer. This is an improvement over the previous C++ ABI, which simply had pointers within the object to immediate virtual bases. In that case converting to a virtual base could involve several memory accesses to follow a path to the desired virtual base. Now any virtual base can be found with a single lookup. Also objects are normally smaller, because most classes with virtual bases are also polymorphic anyway, thus the vtable pointer is already present. Figure 2 shows such a virtual base, and the arrangement of the vtable to support it.

2.1.2 Overridden Virtual Functions

The use of the vtable to implement virtual functions has already been described. There are two additional cases to consider. The first is a virtual function which is overridden in a derived class, and the second is a virtual function overrider which returns a pointer to a class derived from that returned by the overridden function — a covariant return type. These are implemented by small pieces of code called **thunks**, which perform some adjustment before or after the overriding function executes. Instead of putting a pointer to the overriding virtual function at the vtable index allocated by the overridden function, it places a pointer to a thunk, and then emits code to implement the thunk.

When a virtual function overrides a virtual function from a non-primary base, the passed **this** pointer will point to the base subobject and not the more derived

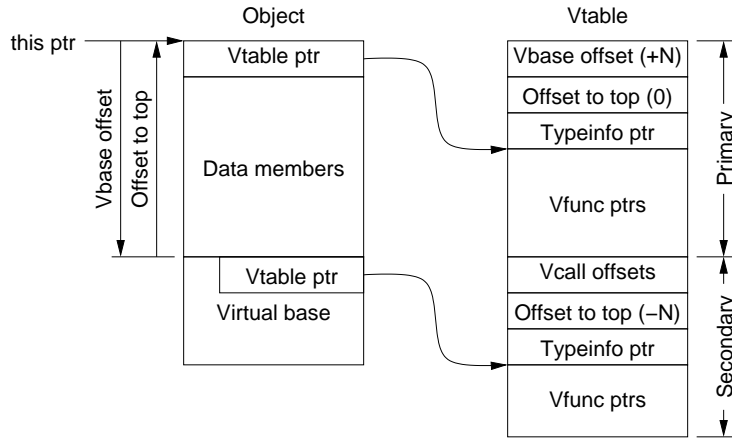


Figure 3: Virtual base

object of the overriding function. The emitted thunk must adjust the `this` pointer from the overridden base to the more derived object. When a virtual base is not involved, the adjustment is a negative constant. When a virtual base is involved the adjustment can vary, as the overriding class might itself be embedded in a more derived object with a different virtual base layout. The vbase offsets in the vtable are not suitable for the adjustment, as they are used to convert from a derived object to a virtual base. The solution is to allocate additional slots in the vtable to hold the offset from the overridden virtual base to the `this` pointer of the overriding function. These slots are called **vcall offsets**. The thunk function knows the position of the vcall offset within the vtable, and uses the value found there to adjust the `this` pointer. Figure 3 shows such a vtable, the vcall offsets are allocated in the secondary vtable belonging to the virtual base so that they are accessible from the passed `this` pointer. It is possible for both fixed and variable adjustments to be needed in a single thunk. This occurs when the overridden virtual function is in a non-primary base of the virtual base. The ABI refers to such bases as **morally virtual**.

The case of covariant return types is handled in a similar manner, except that it is the return type which has to be adjusted, and the adjustment is converting from a more derived type to a base type. Again there can be fixed and variable offsets, but this time the variable offset is simply a vbase offset. Of course it is possible to have covariant thunks to `this` adjusting thunks.

All these thunks are very small (`this` adjusting thunks can be between one and four instructions) followed by a jump to the thunked-to function. Because Itanium, in common with many modern architectures, has a high branch penalty, it is desirable to have a thunk simply fall into the thunked function, or implemented via some other similar multiple entry point mechanism. Also, if the thunks are kept close to the thunked-to function, cache coherency is improved. For this to be achievable, it must be possible to determine what all the necessary thunks are at the time the thunked function is emitted. The ABI does this, and permits such multiple entry point optimizations. C++ does emit the thunks with the overriding function, but does not take advantage of a multiple entry point scheme. The previous ABI did not have vcall offsets, and the compiler would have to emit thunks at every point a final

overrider of a virtual base virtual function was inherited in a more derived class with a different virtual base layout.

2.1.3 Constructors and Destructors

Object construction and destruction occurs in two contexts,

- Processing a complete object. The function has complete knowledge of the class layout and must deal with the virtual bases.
- Processing a base subobject of a more derived object. The function has incomplete knowledge of the class layout and must not process virtual bases (but must still deal with direct non-virtual bases).

During base construction and destruction, the type of the object is considered to be the type of the base being processed. Thus virtual functions and `typeid` must behave as if the base type is the full object. For functions without virtual bases, base object construction and destruction is the same as for a full object. They simply fill the vtable slots of the object with pointers to the base type's vttables. The fully derived object constructor will overwrite those with the proper vttables once base construction has completed. That approach does not work when virtual bases are involved. The problem is that the base's vttables will indicate a virtual base layout for when the base is the most derived object. When the type is embedded in a more derived object, those virtual base allocations will be incorrect. The virtual base allocations are only known by the most derived object and indicated in its vttables. The most derived object must generate special construction vttables which indicate a virtual base layout for the complete object and virtual function pointers for the base under construction or destruction. There is a whole set of construction vttables, one for each base having virtual bases, and this table of vttables is passed as an additional argument to base constructors and destructors. The vtable table is arranged in a hierarchical manner so that bases can use an appropriate entry in it when constructing their direct non-virtual bases that also require construction vttables.

The ABI requires two versions of each constructor and destructor, one for the complete object and one for base subobjects. These are always required even if their implementations are the same. An implementation is free to refer to the same function with both names though.

2.2 Pointer to Member

C++ has pointer to member types. These are usually separated into pointer to member data and pointer to member function. The latter is more complicated as it has to deal with both virtual and non-virtual functions.

A pointer to member data is simply the offset from the start of the object to the member. Because of the structure layout, this offset is always non-negative.⁵ To access the member, the offset is added to the `this` pointer. As data members can

⁵It is ill-formed to form a pointer to member of a class where the class type of the pointer is a virtual base class of the member pointed to.

occur at offset zero, there is a difficulty with a null pointer to member. The ABI uses a value of -1 to indicate a null pointer to member data. It is important that the compiler initializes pointer to member data objects correctly. Of course, this representation is not visible to the user programmer.

A pointer to member function must indicate whether the function is virtual or non-virtual, a pointer to the function or a vtable index, and an adjustment to be applied to the `this` pointer. The adjustment is necessary for pointers to base member functions, because the base might not be at offset zero. All this information can be held in three fields, as the vtable index and function pointer can be overlaid. To differentiate virtual functions from non-virtual ones, bit zero of the pointer or vtable index is used. A null pointer to member function is represented by a null function pointer value.

To call using a pointer to member function requires examining bit zero and branching to separate code at the call site for the two types of pointer. An alternative scheme of always having non-virtual dispatch for pointers to member functions, and fixing up virtual dispatch with a thunk to call via the vtable was not selected, as that would have two indirect branches in quick succession. The previous ABI required the pointer to member function to also indicate the offset of the vtable pointer from the `this` pointer, as that was not fixed.

2.3 Runtime Type Information

Information in the vtable is also used for type identification and dynamic casting. The primary and secondary vtables for all objects of the same type all contain pointers to the same type information object, which is at index -1 of the vtable. The type information is stored using instances of various type information classes. There are such classes for

- Fundamental built-in scalar types.
- Array types.
- Function types.
- Enumerated types.
- Classes with no base classes.
- Classes with a single public non-virtual base class at offset zero.
- Classes with virtual and/or multiple base classes.
- Pointer types.
- Pointer to member types.

All these types are derived from the `std::type_info` type. The `std::type_info` type has a vtable pointer and a pointer to a unique string encoding the name of the type in a mangled form. The inherited class type information objects contain information about the base classes, and the pointer type objects contain information about

the pointed-to type. This information is needed for dynamic casting and exception handling.⁶

The `typeid` operator simply returns a pointer to the `std::type_info` object. Type comparison used to involve comparing the unique mangled string of the type with `std::strcmp`. The ABI takes advantage of `comdat` linkage to force the linker to combine objects with the same name. Thus type comparison can simply use address comparison, requiring only constant time. Not only does this improve the `std::type_info::operator==` operator, but it also improves dynamic cast and exception handling, as both use type comparisons. The optimization has an impact on shared libraries, as they must ensure that objects of the same type have their type information correctly merged.

Prior to the type information pointer is the offset to the most derived object. Dynamic casting to a `void` pointer simply adds that value to the object pointer to generate a pointer to the most derived object.

Dynamic casting to class type involves a search of a class's inheritance graph, starting at the most derived object and searching for the desired target base. There are a number of other conditions that must be checked for if the target base is multiply inherited within the fully derived object. Dynamic casting could be sped up with a hashing technique listing all the direct and indirect bases of a class in its type information. This was discounted as it would require more space to store the information with little gain in common class hierarchies. The most common use of dynamic cast is within a singly inherited non-virtual hierarchy, and that case is optimized for by using two different kinds of hints,

- The first hint tells about how the target base is derived from the static starting base known at compile time. It can be a publicly singly derived at a particular non-virtual offset, not publicly derived, multiply non-virtually derived, or derived otherwise (privately and or virtually).
- The second hint is held in each class's type object. A class lists all its direct base class type objects. For multiply inherited classes, it indicates whether it contains distinct indirect base instances and whether a single virtual base instance is reachable via multiple direct bases (a **diamond-shaped** graph).

Using these two hints, the dynamic cast algorithm can prune the inheritance walk and avoid excess walking for common cases. In particular the single non-virtual inheritance case simply has to walk the inheritance graph from the most derived type until an instance of the target type is located. (During this walk, it does not matter whether multiple inheritance is involved.) Once the target type is located, a quick pointer comparison can determine whether the starting base is indeed held within the particular target base discovered. If it is, the dynamic cast has succeeded. If it is not, the dynamic cast may have succeeded, depending on whether repeated inheritance

⁶Implementations of the ABI are permitted to add unspecified virtual functions to `std::type_info` and the ABI defined type information objects. As these routines must be consistent across all modules in a program, care must be taken to ensure that no object file from a different compiler preempts the vtable of these objects over that provided by the runtime environment. There is a heuristic in G++ to ensure the vtables are emitted at the appropriate place.

has been discovered during the initial walk — if it has, walking must continue to rule out a better target base instance.

2.4 Name Mangling

Name mangling is used to ensure that objects with the same local name are distinguishable. This is done by appending some context, type and or template parameter information. The ABI mangles all names with the prefix `_Z`.

The mangling is specified by a BNF grammar, and uses a compression scheme known as **squangling** (squashed mangling). When complete type information of a function's parameters is used, the name can get very long and overflow some system's limitation on external names. Often the same type will have occurred in several places in the name, and squangling allows the repeated use to simply refer to the original type using an abbreviation. The mechanism used is similar to LZ77 compression.

2.5 Exception Handling

Exception handling is specified in an auxiliary document [5]. Handling an exception consist of

- Construction of the thrown object.
- Unwinding the call frame stack. This is language independent, and described by a supplement to the psABI [6].
- Destruction of objects discarded from the unwound call frame.
- Matching the thrown object to an exception handler.

The unwinding is separated into two phases,

- A **search** phase. This searches the call stack and repeatedly calls language dependent catching routines to determine if the exception can be caught. If this is unsuccessful, a terminate function is called rather than proceeding to the next phase.
- A **cleanup** phase. After a suitable catcher is located, the stack is properly unwound by calling language dependent cleanup routines, and restoring registers in a language independent manner.

The two phase process is not necessary to implement C++ exception semantics but does aid debugging, as the full stack frame is available for inspection in the case of failure.⁷

The unwinding process relies on discovering an exception table for every instruction pointer found during the unwinding process. These exception tables are registered during program and library loading. The exception table consists of instruction pointer ranges, pointers to language dependent handling functions, and register use encodings.

As with dynamic cast, the catch matching algorithm uses the type information objects to locate the correct catch clause.

⁷It also allows restartable exception handling, for languages that require it.

2.6 Allocation and Initialization

The array delete operator requires knowledge of the number of objects in the array in order to run the destructors for them. This information is stored by the array new operator in an **array cookie** just before the array object itself. The ABI specifies that the cookie is only stored when both the destructor is non-trivial and the usual array deallocation function does not take two arguments. The array cookie must not break any alignment restriction on the array type, additional padding is inserted before the cookie to retain alignment.

The ABI defines some helper functions which might be used by a compiler for certain common tasks. These are

- A one-time construction. This is used to acquire and release guard variables, which are needed when initializing static variables in a threading aware compiler.
- Array construction and destruction. These functions take a base pointer, element count and size, and constructor and destructor pointers. There are allocating and non-allocating variants.
- Pure virtual function stub. This function is used in vtables wherever a pure virtual function would be placed. These can occur within the constructor vtables of abstract classes.
- A demangler. This can be used to produce human readable forms of mangled names, including that returned by `typeid(expr).name()`.
- A mechanism to correctly order global destructors. Global destructors must be run in the reverse order of the constructors completing, which is not necessarily the same as the reverse order of the constructors starting.

The ABI interfaces are defined in a `<cxxabi.h>` header file. The interface is specified inside a `__cxxabiv1` namespace. The header file aliases the namespace as `namespace abi = __cxxabiv1;`, to provide a more convenient name.

3 Not

The ABI does not define the internal Standard Template Library (STL) interface so STL implementations are not necessarily binary compatible. To obtain binary compatibility at this level would require defining all data members for STL classes, all virtual functions, their declaration ordering, and all internal helper base classes. This is not a practical option at this time. The G++ library is provided in two forms. A small ABI runtime support library, `libsupc++`, which contains merely the runtime routines and data structures mandated by the ABI. There is a larger standard library `libstdc++` which contains the ABI support and the GNU implementation of the Standard Template Library. The GNU STL is also at version 3.0, and it is known that there will be breakage in the internal STL ABI in a future version.

The ABI does not define an implementation of the `export` keyword since there were no implementations in existence at the time the ABI was defined.

The ABI does not specify an ordering for initializing template static data members beyond that defined by the standard.

4 Benefits to Users

Desirable user impacts of the ABI include

- Faster, more efficient code.
- Full implementability of the C++ standard.
- Interoperability between code generated by different compilers.

All those have been achieved within G++ and several other compilers to various degrees. There are still known interoperability issues, which I will discuss later. Undesirable user impacts are where user code has assumed some implementation details that are not guaranteed by the standard, but were defined by the previous ABI. If the new ABI behaves differently, the code will no longer function as desired.

- The order of functions in the vtable is their declaration order. Thus reordering virtual functions will break binary compatibility.
- A singly inherited base class might not be at offset zero.
- Type information, dynamic cast and exception handling rely on comdat symbol merging. This can have an impact on shared libraries and dynamically loaded modules.

4.1 Shared Objects

In order for a shared library to operate correctly with a main program and shared libraries, the vague linkage symbols that the ABI uses must be correctly merged. Without this merging there will be duplicates in the executable. Such duplicates will break

- Type comparison, dynamic casting and exception handling. Objects created in the library will reference the type information generated with the library, and the main program will consider them different.
- Static objects within inline functions. Two instances of an inline function, included in both the library and the main program that contains static data, should only reference one single instance of that static data.
- Address comparison of duplicate objects with vague linkage such as instantiations of template functions. If duplicates are present, the addresses will compare unequal even though they are the same function.

Without a doubt, it is the first of these issues which affects most users who have problems with shared libraries. There are options in how a shared library is built and loaded:

- During building, a partial link can be done so references to global objects within the library (or to specified static libraries) are fixed, and cannot be overridden by any user of the shared library. The intent of this option is to reduce the library load time, and presumes that it is equivalent to refer to an object in the library or to one from the main program. Doing this will break the object merging that the ABI presumes.

- During building, you can specify which symbols are exported from the shared library, or you can export all the global symbols. If the shared library is to interoperate with other shared libraries, you must make all symbols available, so that duplicates can be merged between shared libraries.
- During explicit loading of a shared library, you can specify whether global symbols in the dynamic object are available to other (later loaded) dynamic objects or not. If the libraries are to interoperate, you must make the symbols available.
- During loading a shared object, its exception table must be registered with the runtime system. This is normally done automatically via a global constructor inserted by the compiler.
- Shared objects must be unloaded in the reverse order of loading. This should be correctly handled by the system library, although some versions of the GNU/Linux library had problems with this. Before unloading, the exception table should be deregistered, and this is usually done by a global destructor inserted by the compiler.

Sample code to build various configurations of shared libraries is available at [7].

5 Implementation

The C++ ABI is a complicated specification having many interactions between the interoperating parts. Its design proceeded in tandem with implementation in several compilers. Many bright people were involved in the design, and this may have lead to an overly complicated specification. There is one particular optimization, which I was involved in implementing, to do with polymorphic virtual bases that have no data members. These simply contain a vtable pointer and correspond to Java interfaces, in that they merely declare a set of virtual functions and are intended to be overridden. It was noted that such classes are suitable for use as a primary virtual base. That gives a space advantage because no separate virtual base needs to be allocated. Because only one instance of a virtual base exists in a complete object, it is possible for a primary virtual base to be stolen by another base within a more derived hierarchy. This optimization caused an astounding number of headaches and corner cases during implementation. So many, that had we known in advance of the difficulty, we would probably not have chosen to do it.

CodeSourcery has implemented the ABI in a number of compilers, including G++ 3.0, for various companies. During this work we saw the need for a separate testsuite to validate the implementations. CodeSourcery has built such a testsuite and doing so has been worthwhile. We found and fixed tens of bugs in G++ using it and have licensed the testsuite to several major compiler vendors.

5.1 Going Forward

Originally, it was planned to maintain G++'s old ABI alongside the new ABI. Indeed implementation of the new ABI proceeded in public on GCC's CVS mainline. After

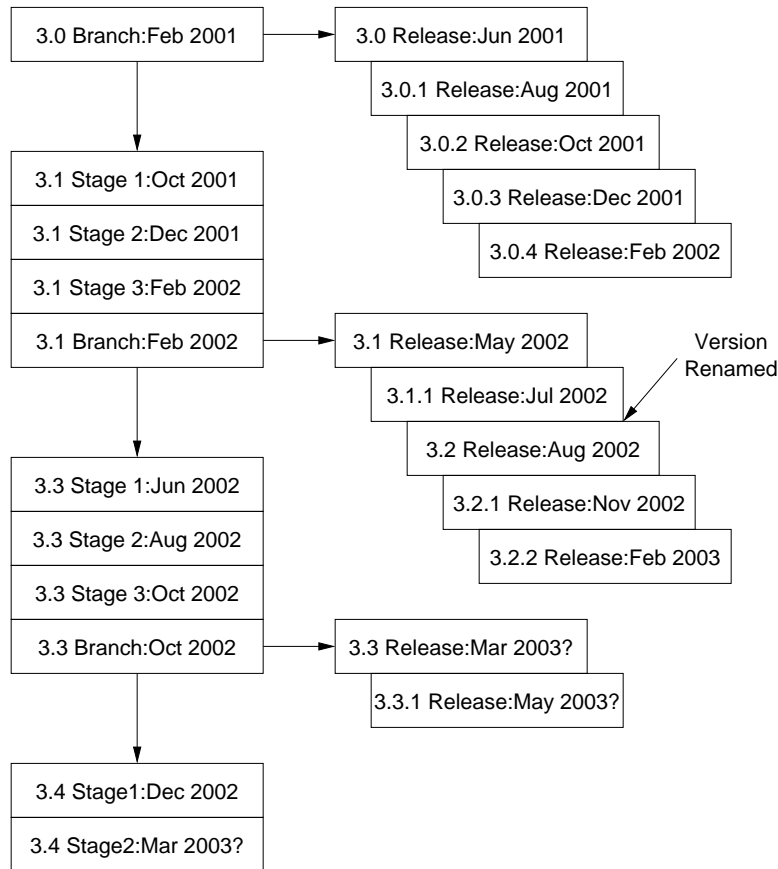


Figure 4: GCC Development Tree

the ABI was implemented, the GCC steering committee voted to remove the old ABI, thus eliminating a significant maintenance burden.

As with all new designs, a number of bugs came to light after the GCC 3.0 release. There were faults in the ABI specification and bugs in GCC's implementation. Some of these were fixed in GCC 3.1. At the time that 3.1.2 was to be released, it was already known that there were additional ABI bugs which were to be fixed in the next major release (3.2). There was also a push by GNU/Linux system vendors to synchronize their releases of GCC, because their system compiler was still either 2.95 or 2.96RH⁸. We knew that the next major GCC release would not be ABI compatible with the 3.1 series, we just did not know how incompatible it would be — there could be remaining undiscovered bugs. There was great pressure to backport the 3.2 ABI fixes to the 3.1 branch, in the hope that there were no further undiscovered ABI bugs. The probability of further bugs was non-zero, but the fixes were ported anyway, and, to indicate the ABI change, what would have been called 3.1.2, was renamed 3.2, and the development version bumped to 3.3. Figure 4 shows the organization of recent GCC releases.

Inevitably further ABI bugs came to light after the 3.2 release. Going forward, GCC is keeping the 3.2 ABI implementation. By default, only bugs that cause implementation failure will be fixed. Bugs which only break interoperability with other

⁸2.96RH is a release of the development branch made by Red Hat

compilers will not be corrected by default. To enable interoperability, we have implemented two new features.

- Wabi A warning, which is enabled by default, indicating where G++ is generating code that is not compatible with the ABI specification. Currently these mainly involve some structure layout issues. In those cases the user can circumvent the ABI deviation by adding or altering data members, thus obtaining a structure which is laid out in the same way in both versions of the ABI. This option is in the current 3.2.2 release.
- fabi-version=*n* A new compiler switch which allows the ABI conformance to be changed. Currently the version is 1 by default, which corresponds to the 3.2 series ABI. When the version is set to 2 all current known fixes will be enabled. The `-fabi-version` switch will appear in the next major GCC release (3.3).

No decision has been made on when or whether `-fabi-version=2` will be made a default. As interoperability bugs are discovered, a `-Wabi` test will be added and backported to the most recent release branch. Also a fix will be added to the current development branch, enabled by the current `-fabi-version=n` value.

Some template manglings require knowledge of what is a dependent type. G++ does not currently have sufficient information to get this right in all cases. In particular it can give two distinct functions the same mangled name in some cases (rather than mangling them distinctly, but incorrectly). Now that G++ has a new recursive descent parser in the development sources, these bugs are fixable. The new parser will be released in GCC 3.4, by which time the template mangling bugs should be fixed too.

In summary, the C++ ABI provides a standard that compilers can implement and thereby produce modules that can interoperate. Differences between the new and the old ABI are:

- The vtable position within an object is fixed, resulting in fewer parameters in various places.
- Locating a virtual base is a constant time operation, rather than traversing a path within the hierarchy.
- Class layout is more efficient, allowing the overlaying of tail padding of some base classes.
- Runtime type comparisons are constant time.
- The name mangling scheme results in shorter names, and can correctly distinguish certain cases that the previous ABI could not.
- Covariant thunks are supported. G++ did not support them with the previous ABI.
- Thunks are more efficient, and may be implemented by a multiple entry point mechanism.
- Construction vtables are supported.

- Dynamic cast and exception matching are more efficient.
- Global destructors are executed in the correct order, in all cases.

There are still opportunities to implement various optimizations of the ABI in C++ in an interoperable manner. For instance, constructor and destructor combining and multiple entry points for thunks.

References

- [1] Programming Languages — C++, ISO/IEC 14882:1998.
- [2] The System V Application Binary Interface, <http://www.sco.com/developer/devspecs>.
- [3] The Intel Unix System V Application Binary Interface, Itanium Processor Supplement, <http://www.intel.com/design/itanium/downloads/245370.htm>.
- [4] C++ ABI for Itanium, <http://www.codesourcery.com/cxx-abi/abi.html>.
- [5] C++ ABI for Itanium: Exception Handling, <http://www.codesourcery.com/cxx-abi/abi-eh.html>.
- [6] The Intel Itanium Software Conventions and Runtime Architecture Guide, <http://www.intel.com/design/itanium/downloads/245358.htm>.
- [7] Sample Shared Library Source, http://www.codesourcery.com/publications_folder/accu2003.html.