

Getting the Best From G++

Nathan Sidwell
CodeSourcery LLC
nathan@codesourcery.com

April 30, 2003

Abstract

The 3.0 series of G++ compilers and libraries offers a new multi-vendor ABI and increasing conformance to the C++ standard. The C++ ABI offers increased efficiency for C++ idioms and interoperability with other compilers. Features of the ABI that the G++ user should be aware are described. Both additional and deprecated features in versions 3.2, 3.3 and 3.4 are described. Using various source idioms to aid the G++ optimizers and loading process is shown. The process of tracking the C++ standard as both defect reports and C++0X become available is outlined.

1 The 3.0 ABI

Starting with G++ 3.0 a new C++ ABI is provided. This multi-vendor ABI [2] came from development of an Itanium port of GCC, which included the design of a C++ ABI for the Itanium processor. That ABI was designed by CodeSourcery, EDG, Compaq, HP, Intel, Red Hat and SGI. Although designed for one architecture, the C++ ABI is sufficiently abstracted from Itanium features to allow its use for other processors, and hence the multi-vendor C++ ABI came about.

The 3.0 ABI is a complete redesign of the G++ ABI, which leads to space and speed improvements. The previous G++ ABI had evolved over time as C++ itself stabilized. ABI improvements include,

- Empty structures take zero size when used as a base class.
- Tail padding can be overlaid for non-POD bases and members.
- Derived to base conversions are constant time for both single and multiple inheritance. Conversion to a non-virtual base, requires a fixed adjustment and a single access of the vtable is needed to convert to a virtual base. Having virtual base offsets held in the vtable reduces the object size overhead for virtual inheritance. In most programs virtual inheritance does not increase the size of an object, because nearly all classes with virtual bases have virtual functions too. Previously a virtual base would add a pointer member to each class that derived from it, and base conversion involved following an inheritance path, which could involve several member accesses.
- Pointers to member functions are smaller, and dispatching via them is faster, because the vtable pointer is always at the start of an object.
- Virtual function thunks are all emitted with the thunked to function. This gives better cache coherency, and permits multiple entry point optimizations for thunked functions.¹ These improve the performance of the virtual function calling mechanism. The thunk mechanism is such that even overriding from a virtual base is fast, with a single adjustment using one access into the vtable.
- Covariant return thunks are specified, and implemented in G++. Again, these are emitted with the overriding function that required their emission, and so have the cache coherency improvements and multiple entry point optimization opportunities of the simpler thunks.
- Dynamic cast hints are generated by the compiler, and improve the speed of `dynamic_cast` in common cases. In most cases the speed of

¹G++ does not currently implement multiple entry point optimizations.

`dynamic_cast` is now linear in the number of bases between the dynamic object type and the target type of the cast.

- Runtime type comparison is constant time, which further improves `dynamic_cast` and catch matching. Previously, type comparison involved string comparison.
- Exception handling is a two phase process. The first phase locates a catch handler, and only when one is found is the stack unwound to that handler. If a handler is not found, `std::terminate` can be called in the throwing context, and hence help debugging.
- A new mangling scheme that uses a compression algorithm. This produces shorter names, and so improves link and load times.

Additional improvements in G++ 3.0 were,

- The `std` namespace became a real namespace, rather than an alias for the global namespace.
- A new implementation of the standard template library, which is properly contained in the `std` namespace.
- Type based aliasing is enabled at optimization level `-O2`.

These changes effect user code to varying extents. Other than speeding up code, the new ABI should result in no user visible changes. Of course, all programs and libraries will need to be recompiled. If the user relied on ABI features, then a program might be effected.

1.1 Shared Libraries

The ABI makes use of a link facility that ELF [3] supports called common data. The common data linkage is used for objects that have no well defined object file in which to place them. The C++ ABI relies on common data linkage to implement the constant time comparison of types. This requires the names of type information objects to be globally visible. Libraries are effected because the type information objects must be visible to user programs. Shared libraries that are resolved at load time by the runtime loader, and those opened explicitly with

`dlopen`, as is commonly done for program plugins, are effected in the same way. Static libraries are also effected, but the impact on real programs has not been so great. The link and loading speed of all three kinds of libraries can be improved by the mechanisms described here. A library makes available, or exports, to user programs a set of names. It also has to specify, or import, those names it uses from other libraries. Both importing and exporting use the same mechanisms and the remainder of this paper simply refers to exporting. If the library wishes to `dynamic_cast` or throw exceptions across the library interface, it must export type information names, so that the common linkage is achieved. More complicated export requirements require other types of names to be exported.

Because C++ has no module system, the library programmer cannot indicate at the source level which types, functions and objects are to be exported. The library is forced to export all symbols, to ensure the user can access the exported functionality. There are proposals [5] to add module facilities to the language. It is desirable to indicate a subset of the names as available to users of the library. The currently available mechanism for doing this is symbol versioning [4].

The simplest solution is to export all external names from the shared library. Unfortunately this has two disadvantages. Firstly program load times are increased because the dynamic linker must resolve all these symbols in order to eliminate duplicates with the already loaded program. Secondly, it exposes the internal names of the library implementation that have global scope. Sometimes those names can conflict with the user's names, or those in other libraries used in the program.

The solution to name conflict is to put the internal names into a library specific namespace. For instance, have the exported library functionality in a `'FooLib'` namespace, and the internal names in a `'FooLib::Internal'` namespace. Unfortunately it can be difficult to retro fit such a solution to an existing library that is not namespace aware.

For a simple shared library, where no runtime type information is transferred across its interface, it is simply necessary to export the library interface functions. For a more complicated library, it is necessary to export the type information names, and potentially some of the internal names. This can be done by examining the names in the library object

files using `nm` and using a pattern matcher to extract the important ones. The C++ ABI mangles all names with an initial `‘_Z’`, followed by the mangled name. Certain prefixes are placed between the `‘_Z’` and the mangled name, for particular kinds of names. These are,

- TV Vtables. Pointed to by polymorphic objects and those with virtual bases. These are termed dynamic classes in the ABI.
- TT Vtable table. Used in constructing and destructing polymorphic objects with virtual bases. Not all polymorphic classes will need a vtable table.
- TI Type information. Returned by `typeid` operator, pointed to by the vtable.
- TS Type string. Returned by `type_info::name`, and used for type comparisons.
- GV Guard variable. Used to guard the initialization of function scope static objects that are dynamically initialized. The name of the static object will be the same as the guard variable without the `‘GV’` prefix.
- Th, Virtual function thunks. These are followed by `Tv`, by a mangling of the thunk information, and `Tc` then the mangling of the thunked to function. The second prefix letter indicates whether it is a fixed, virtual or covariant thunk.

The vtable, vtable table, type information and type string are not tightly bound to any particular object file by the language, and so have common data linkage. Potentially any object file that uses them could contain their definition. The C++ ABI has an optimization where the class to which they belong has a non-inline virtual function, the first of which is called a key function. In that case, all these objects are only emitted in the object file that contains the definition of the key function. Other object files will not contain these objects, as it can be determined that their definition will be provided elsewhere. Libraries can be effected by this because, although it might not matter that two instances of a particular object were used in a program, a user program can rely on a definition it knows is in the library.

The type information objects will need exporting, to share type information, as user programs which use the type for `dynamic_cast` or catching, will need

to refer to them. Sometimes these are emitted with internal linkage, in which case they refer directly or indirectly to an incomplete type. Such instances should *not* be exported. Type comparison itself uses the address of the type string. It is necessary for that string to be shared by all instances of the same type. If they are not exported, the type comparison algorithms will consider two types with the same name to be different types. Therefore, external names beginning with `‘_ZTI’` and `‘_ZTS’` should be exported from the library.

If the library exposes inlinable constructors or destructors of dynamic classes to users of the library, it is necessary for the library to export the vtable and vtable table.

If the library exports constructors to the user, all the user callable virtual functions of the class and its ancestors must be exported. Although virtual functions are normally called via the vtable (and therefore their names are not needed, just the index in the vtable), by exposing the constructor it might be possible to determine the dynamic type of an expression at compile time. Should the compiler do that, it may elect to replace a virtual call with a direct call, and hence require the name of the virtual function.

Static objects in inlinable functions that are exposed in library header files will cause problems. The static objects’ names must be exported, so that only one becomes live in the final executable. Only static objects with a dynamic initialization expression will have a guard variable.

If the library exports types that can be inherited from, then the type information object, all user callable member functions of the class and all virtual functions and thunks must be exported. The class members will be mangled, following any applicable prefix, as a scoped name of the form `‘N<classname><membername>E’`. Both the `classname` and `membername` components are mangled as a numeric length followed by the name, such as `‘6FooLib’`.

Here is an example library header file, showing what needs to be exported, depending on the functionality provided.

```
#include <exception>
#include <new>
```

```

namespace NMS {
  namespace Internal {
    // Helper we do not wish to expose
    // Do not export
    class Helper
    {
    public:
      Helper () {.....};
      virtual int Frob () throw ();
    };
  } // namespace Internal
  // Export type info _ZTIN3NMS5ErrorE
  // Export type string _ZTSN3NMS5ErrorE
  class Error
  {
  // Import std::exception typeinfo
  : public std::exception
  {
  friend class Widget;
  // Do not export, library creates
  Error () throw () {}
  public:
  // Do not export, it is inline
  virtual ~Error () throw () {};;
  // Do not export, called virtually
  virtual char const *what () const
  throw ();
  };
  // Export Widget if it is inheritable
  class Widget
  {
  // Export direct & indirect bases,
  // if Widget is inheritable.
  : Internal::Helper
  {
  private:
  // Do not export, library creates
  Widget () throw ();
  public:
  // Do not export, called virtually
  virtual ~Widget () throw ();
  public:
  // Do no export, called virtually
  virtual int Action () throw (Error);
  public:
  // Export, user can call
  // _ZN3NMS6Widget3NewEv
  static Widget *New ()
  throw (std::bad_alloc);
  };
} // namespace NMS

```

Because the only way of constructing a ‘NMS::

Widget’ object is by calling ‘NMS::Widget::New’, users of the library will always have to use the virtual call mechanism to call ‘NMS::Widget::~~Widget’ and ‘NMS::Widget::Action’, so those two functions do not need to be exported. Both NMS::Error’s type information and type string need exporting so that user programs can successfully catch such an object.

1.2 Library Compatibility

Linking C++ objects from different compilers involves more than just the C++ ABI. If the programs use the standard library, then the library versions must be compatible too. The multi-vendor ABI does not specify the binary compatibility of the library, as that would be too constraining on implementations. The ABI specifies a small runtime support library, necessary to implement the core C++ language. G++ provides that as a separately selectable `libsupc++`. The full library is also provided automatically as `libstdc++`. The G++ 3.0 implementation is a complete redesign of the library. The new library is more standard conformant, and this has lead to some issues with user code,

- The ‘std’ namespace must now be explicitly noted. For example, ‘`vector<int> foo;`’ does not compile. ‘`vector`’, along with everything else, is in the ‘std’ namespace. Previously, G++ also found it in the global namespace, so programs compiled whether ‘`vector<T>`’ or ‘`std::vector<T>`’ was used. Another common instance of this problem is using plain ‘`cout << "Hello World" << endl;`’ The solution is to recognize the failure mode and insert ‘`std::`’ appropriately.
- IO is slower. According to the C++ standard, by default, the standard C++ streams, ‘`std::cin`’, ‘`std::cout`’ and ‘`std::cerr`’, have to be synchronized with the standard C file streams, ‘`stdin`’, ‘`stdout`’ and ‘`stderr`’, so that use of corresponding pairs of streams can be intermixed. A clever trick allowed the previous C++ library to overlay its stream classes on the underlying C library’s file structure, *but* only for one specific C library. With the change in the G++ ABI, and better standard conformance, that trick became impractical to maintain. The standard allows users to explicitly decouple the C and C++ file IO operations by calling, ‘`std::ios::sync_with_stdio`

(false)' before any IO has happened on the standard streams.

Another issue with `'std::cin'` and `'std::cout'` is that they are synchronized with each other. C++ requires that, by default, intermixed input and output will display in the correct order. This synchronization can be removed by calling `'std::cin.tie (0)'`. C does not have such fine grained synchronization on `'stdin'` and `'stdout'`, these are normally only synchronized at newline characters.

C++ IO is more expressive than that provided in C, and because the C++ library is implemented on top of the C library, C++ IO will never be faster than C IO. Work is ongoing in improving IO performance.

- Iterators do not have pointer types. Some code presumes that iterators are implemented as pointer types, and contain code such as `'&myIterator'`, expecting to get a `'T **'`. Because the previous library implemented them as such, that code 'worked', even though that implementation is neither required nor guaranteed by the standard. Now iterators are implemented as templated classes, which gives better type safety, but breaks such erroneous code. Code which assumes the underlying representation of an iterator can be forced to work simply by `&myIterator`, as the `*` operator will provide a reference to the iterated object, whose address can be taken.

2 What is in G++ 3.3

The multi-vendor ABI is very complicated and its first G++ implementation in G++ 3.0 turned out to have some bugs. Several of the defects were discovered in time for G++ 3.2. More issues have been discovered since then, by testing interoperation with other compilers and by using CodeSourcery's test-suite. It is very inconvenient to change the ABI, as that means that all object files and libraries need to be recompiled with the new compiler. Some ABI bugs merely effect inter-operation with other compilers, and are unimportant to a significant user base. Rather than force an ABI change on all users, G++ implements two flags to warn about ABI discrepancies and to select ABI version. The `-Wabi` flag warns when G++ is emitting code or data that is known to be at variance with the multi-vendor

ABI. The `-fabi-version=<n>` flag allows the user to specify which set of known ABI fixes to include. The current default version is 1. Whenever an ABI bug is discovered, code for both options is added to the compiler, and the warning code is backported to the previous stable release branch, for a subsequent minor release. Of course, because time machines are nonexistent, it is not possible to backport it to the previously released version. All known ABI fixes can be selected with `-fabi-version=0`. Which fixes that includes depends on the version of G++, so using this value implies that the same version of G++ must be used to compile all the object files and libraries of a program. When a sufficiently stable set of fixes has been implemented, another ABI version number will be added, and `-fabi-version=2` will be selectable. It is likely that G++ 3.4 will implement such an ABI version number, but it is undecided whether that will be made the default value. Version 0 will still be selectable, to obtain all the subsequent fixes added after version 2 has been stabilized.

G++'s implementation of the standard template library has not yet stabilized. Because the library exposes much of its implementation in header files containing class, inline function and template definitions, it is very difficult to improve the library without changing something that effects binary compatibility. There are no planned library ABI changes between the 3.2 and 3.3 releases. However, the 3.4 release will not be binary compatible, and the shared object version number has been incremented. Because it is provided as a shared library, and the version number has changed, users will get a link error, rather than mysterious runtime failures, if they attempt to mix versions.

One of the more significant changes in G++ 3.3 is the removal of the implicit typename extension. The extension was deprecated in G++ 3.2, and elicited a warning at every use. In a template class, names from dependent bases are not visible when the template is defined — they are only looked up at instantiation time. G++ had an extension that made names visible before instantiation, so G++ knew which were types and which were not. The standard requires that those that name a type be referred to using the `typename` keyword and qualified name.

```

template <typename T>
class Base
{
    typedef int Type;
    typedef int Other;
};
typedef unsigned Other;
template <typename T>
class Derived : public Base<T>
{
    Type a; // Implicit typename use.
    // Standard conforming way.
    typename Base<T>::Type b;
    Other c; // Which Other?
};

```

The implicit `typename` extension became impossible to keep when updating G++'s parser to be more conformant. The extension is also problematic in itself. In the example, when instantiating `Derived` for some particular type `U`, `Base<U>` might have a specialization for which `Base<U>::Type` is not an `int`, or even a type. Another confusion is shown in the example by the use of `Other` in `Derived`. If the implicit `typename` extension is in operation, it will be `Base<T>::Other`, whereas without it, it should find `::Other`. Having a program's meaning change between two valid interpretations by changing a command line flag (`-pedantic`), is really bad — better to remove the extension.

2.1 Optimization

Previously G++ had a named return value extension to help functions that returned a class by value. Because returning a class value requires a copy of the return value into the area provided by the caller, such functions would invoke a copy constructor just before returning. The idea of the named return value extension was to allow the programmer to use that area directly and avoid the copy. This extension did not work with template functions, and has been removed. In its place is the return value optimization, which notices when a function is returning a temporary by value, and will directly construct the temporary in the return area.

2.2 Exception Specifications

G++ 3.2 had poorer inlining performance than desired. It would not make sensible choices about

what to inline, and the inlining process could lead to long compile times and large compiler memory size. This has been fixed by taking advantage of `'throw ()'` exception specifications. If none of the functions called by a particular function can throw exceptions, the inliner can do a better job.

Exceptions specifications can also be used to reduce the size of a program. In the following program, `CLASS1`, `CLASS2`, `FOO` and `BAZ` can be defined to be empty, or `'throw ()'`. The code and exception data sizes for various combinations using G++ 3.2 for `i686-pc-linux-gnu` producing optimized code is shown in Table 1. The `'Check'` column indicates whether the `-fno-enforce-eh-specs` option was used.

```

struct Class1
{
    int m;

    Class1 () CLASS1;
    ~Class1 () CLASS1;
};
struct Class2
{
    int m;

    Class2 () CLASS2;
    ~Class2 () CLASS2;
};
void Foo () FOO;
void Baz () BAZ
{
    Class1 c1;
    Class2 c2;

    Foo ();
}

```

The worst case is a factor of 4 in program size, however the more common case is probably the penultimate line of the table where none of the functions have an exception specification. The `-fno-enforce-eh-specs` option tells G++ not to add code to a function to check that it is throwing only the exceptions listed in its exception specification. A correct program will only throw such exceptions, so such checking code is behaving as `assert` macros. However it is notoriously difficult to exercise exceptional paths in program flow. The author has used a custom allocation library to rigorously test allocation failures in a command line application, to good effect.

Exception specification					Code	Data	Total	Overhead
CLASS1	CLASS2	FOO	BAR	Check				
throw ()	throw ()	throw ()	Either	Either	63	0	63	-
throw ()	throw ()	None	Either	No	83	92	175	178%
None	None	throw ()	Either	No	95	88	183	190%
None	None	None	None	Either	103	104	207	226%
None	None	None	throw ()	Yes	137	113	250	296%

Table 1: Exception Overhead Example

Exceptions		Utility Library				3D Library			
Specs	Checked	Code	Data	Total	Overhead	Code	Data	Total	Overhead
Yes	No	147871	23037	170908	-	219124	49422	268546	-
Yes	Yes	158000	35070	193070	13%	238691	77538	316229	18%
No	Either	150760	39685	190445	11%	224646	83306	307952	15%

Table 2: Library Exception Overhead

Such a small example might be skewed to give large overheads — it has no real code in it, and nothing can be inlined. Two C++ libraries of about 30,000 lines of code each were examined. One was a low level utility library, and the other a higher level 3D toolkit. Both libraries have been written with exception specifications on every function, most of which were no-throw, but many were allocation failure exceptions. Each library was compiled in three different ways:

- With exception specifications, but with `-fno-enforce-eh-specs` enabled to remove the exception checking code.
- With exception specifications and with exception checking enabled.
- With `throw` defined as a variadic `throw(...)` macro, so that the exception specifications were removed. With no exception specifications, checking exception specifications would have no effect on code size, so `-fno-enforce-eh-specs` would make no difference.

Table 2 shows that the overhead is between 11% and 18%. The code size of a checked exception specified library is larger than that of the library without exception specifications, because of the number of non-empty exception specifications. G++ is not clever enough to notice whether functions that have a non-empty exception specification only call functions that can throw the listed exception types — it still emits code to verify. A suitable optimization

will be able to remove that extra checking code. The same is not true of the extra code added when there are no exception specifications. That code has been added to destroy local variables that will go out of scope, should an exception be thrown. The compiler cannot determine only from a function declaration with no exception specification that the function will not actually throw an exception, so it must presume the worst and emit appropriate destruction code.

When the body of a function is visible, G++ can determine if it does not throw by noting whether it calls a function that could throw, or contains a `throw` expression. If it cannot throw, G++ will optimize appropriately. In the small example above, such analysis could only be done on `Baz`, and the specification checking code can be deleted as unreachable, if all the other functions have a `throw ()` exception specification. Both 3.2 and 3.3 will remove this unreachable code, but 3.3's compile time performance will be better, as it notices much earlier in the translation process that the checking code is unreachable. G++ cannot currently tell whether an exception will be caught inside the function, so appropriate `throw (...)` exception specifications should be added to function declarations and function definitions that contain `try ... catch` clauses.

One final note about code size. A static image of 'Hello world' is surprisingly large. For instance, the program `int main () {return 0;}` has a static code size of 289,281 bytes on the author's gnu-linux system. Both C and C++ sources gave the same

size. The size is a glibc [8] issue, not a GCC problem. Glibc is not designed to be used as a static library, and embedded systems should use an alternative library.

3 What Will be in G++ 3.4

G++ 3.4² will feature a much better parser, which correctly deals with more ambiguous parsing situations than G++ 3.3 does. C++ has an ambiguous grammar where a construct can look like both a declaration and an expression, it is not until deep within the statement that the parser can tell which one it is. The previous Bison [6] based parser could not deal with several cases that were reasonably common. Bison parsers deal with LALR(1) [7] grammars, but C++ is not such a grammar. The Bison based parser has some C++ specific hacks to deal with some of the ambiguities. The new parser is a handwritten recursive descent design, with arbitrary back tracking. Here is an example, where as G++ 3.3 fails on every line of 'Foo', G++ 3.4 will parse them all.

```

struct A
{
    A (int = 0);
};

struct C
{
    C (A, A = A ());
    void oneFish () const;
};

A Foo (int thing1, int thing2)
{
    C redFish (A (), A (1));
    C blueFish (A (thing1), thing2);
    C (A (2)).oneFish ();
    return (A ());
}

```

Having a better parser is good, but it is also more picky about name lookup in template definitions. Names can be looked up during template definition and during template instantiation. Depending on

²This section describes the development version of G++ as at 28th April 2003. When 3.4 is released, it might differ from what is described here.

context, a name might be looked up only during definition, or only during instantiation, or both. This is called two-stage name lookup. C++ programs developed only with G++ are more than likely to have template name lookup problems — switching to the new parser will produce compilation errors. There are two cases of interest, one involving dependent bases and the other to do with argument dependent lookup (Koenig lookup).

The dependent base problem is similar to the implicit typename issue that was removed in G++ 3.3. Here is an example,

```

template<typename T>
struct Base
{
    int count;
    int total;
};

int count;

template<typename T>
class Derived : public Base<T>
{
    // Wrong Thing
    void Flangify ()
    {
        // 3.3 defers both to instantiation
        // time and finds those in Base<T>.
        // Should bind to ::count.
        int ix = count;
        // Error, should not be found.
        int jx = total;
    }
    // Right Thing
    void Flangicate ()
    {
        // Deferred to instantiation.
        int ix = this->count;
    }
};

```

As 'Derived::Flangify' shows, the compiler will give an error at template definition time, if a non-dependent name is not found. Unfortunately, it could bind to an unintended object, which happens for 'count'. 'Derived::Flangicate' shows the correct way of forcing name lookup of members to be deferred until instantiation time. The idiom has the advantage of making explicit to the programmer that the name refers to a member. That members in dependent bases are not searched for, unless

preceded by ‘this->’, is very surprising to programmers unfamiliar with the rule. The intent is to allow more checking and precompilation of template definitions, before instantiation, and only defer to instantiation time those lookups that are demonstrably dependent on a template parameter.

The other place effected by name lookup is in function calling and argument dependent name lookup. When a function is called using unqualified name lookup (something like ‘foo (arg)’), but it also happens on overloaded operators), the function is looked up in the current scope as normal and in the classes and namespaces of the arguments’ types. If the arguments’ types are template dependent, that part of the lookup is deferred until instantiation time. The non-dependent part of the lookup is done at definition time, and not repeated at instantiation time. Here is an example,

```
namespace NMS {
    class MyClass {};
    void Foo (MyClass);
} // namespace NMS

void Foo (int);

template <typename T>
void Bar (T thing)
{
    Foo (1); // #1
    Foo (thing); // #2
}
```

The first call, ‘Foo (1)’, will find ‘::Foo (int)’ at definition time. The second call, ‘Foo (thing)’, will find the global ‘Foo’ at definition time, but it will also find ‘NMS::Foo (MyClass)’ during the instantiation of ‘Bar<NMS::MyClass>’. The two declarations of ‘Foo’ are added to the overload set, upon which overload resolution is performed. Overload resolution could be done at definition time for the first call, as that does not contain any template dependent expressions. At the current time, the development version of G++ still defers lookup for function calls until instantiation time, and therefore does not have the correct two-stage lookup behavior here.

Another impact of this, is that G++ will not mangle some templated names correctly. In some cases the mangling depends on knowing what is a dependent expression and what is not. Without that knowledge, although the manglings are unique, they do not adhere to that specified by the ABI.

A lookup case that G++ still gets wrong is where a name refers ambiguously to a member of a dependent base and of a non-dependent base. At definition time the dependent base will be ignored and the name found unambiguously in the non-dependent base. The ambiguity *should not* be discovered at instantiation time, as the lookup is not repeated. G++ will repeat the lookup at instantiation time and discover an ambiguity.

```
template <typename T>
struct Wump
{
    int zed;
};

struct Gump
{
    int zed;
};

template <typename T>
struct Sneetch : Wump<T>, Gump
{
    Sneetch ()
    {
        // Both 3.3 and 3.4 find an
        // ambiguity at instantiation.
        // Should bind to Gump::zed.
        zed = 5;
    }
};
```

Because G++ still does not do the correct two-stage lookup for function call, some cases of the first described name lookup issue can still remain undetected. When the intent is to call a member function of a dependent base, the name lookup is incorrectly deferred until instantiation time. Even if the function parameters are template dependent, a non-friend member from a dependent base should not be found — only those names found by argument dependent lookup should be added at instantiation time.

```
template<typename T>
struct Base
{
    void Deflange ();
    void Flange ();
};

void Deflange ();
```

```

template<typename T>
class Derived : public Base<T>
{
    // Wrong Thing
    void Flangify ()
    {
        // 3.4 binds both of these to
        // members of Base<T>
        // Should bind to ::Deflange.
        Deflange ();
        // Error, should not be found.
        Flange ();
    }
    // Right Thing
    void Flangicate ()
    {
        // Deferred to instantiation.
        this->Deflange ();
    }
};

```

To get two stage lookup correct requires better tracking of the symbol table so that, at instantiation time, it is known what declarations are visible at both the definition context and the instantiation context. Whether this work will be completed by the time 3.4 is released is unknown.

4 Tracking the Standard

The C++ standard is an evolving document. Since the 1998 C++ standard was released, various changes have been made. A Technical Corrigendum 1 (TC1) is in the process of being released. That bundles all of the accumulated changes into single document. Issues can be raised by anyone, and are collated via an email list. Every six months, a global meeting of the ANSI J16 and ISO WG21 [9] committees takes place. These meetings are open to all interested parties, and membership of J16 is not required. Affiliation does effect voting rights. There are three subgroups within those meetings,

- Core Working Group. This group deals with issues in the core language (that documented in clauses 2 to 16). The core defect reports are available [10].
- Library Working Group. This group deals with issues in the libraries (clauses 17 to 27). The library defect reports are available [11].

- Evolution Working Group. This group deals with extensions and other changes to the language and library. The group is currently considering what significant changes should be made for the next version of the standard, code named 'C++0X'.

The output of the core and library working groups are lists of defect resolutions. A report may be deemed to be not a defect (the standard requires no change). Alternatively the standard may require clarification, or require change. The wording of the changes is discussed and goes through a process of drafting until it is ready to be accepted.

G++ aims to track the standard with its collection of defect reports. We do not make a distinction between the 1998 standard and the standard plus defect reports. Active participation in the C++ standards meetings allows the G++ maintainers to both know how defects are likely to be resolved, and to influence that process. When C++0X is released, G++ will probably have a command line switch to select which version of C++ is to be accepted (just as either C89 and C99 can be selected between in GCC).

Many people have suggested extensions that G++ should accept. Often these proposals are of the form 'It would be neat if I could write ...', rather than a complete specification. Such vague descriptions can prove problematical with a language as complicated as C++ — all the implications of an extension are not apparent, even after some thought. Without care, extensions can either silently change the meaning of a C++ program, or fail in obscure ways under some circumstances. Several GCC extensions have caused such problems when ported to G++. Some of note are,

- It used to be possible for `__PRETTY_FUNCTION__` to participate in string concatenation. Unfortunately this does not fit well with templates, where the expansion of `__PRETTY_FUNCTION__` depends on the instantiation, much later than string concatenation occurs. GCC has been changed throughout, so that `__FUNCTION__`, `__PRETTY_FUNCTION__` and the C99 defined `__function__` all behave the same way as constant arrays of characters.
- Variable length arrays have a type which is not fixed at compile time. This causes a problem

with `typeid`, because there is no fixed `std::type_info` object that can be returned. `typeid` was changed to return the type info for the array member type. Also template deduction suffers, because the type has a size that is not fixed. Template deduction will not deduce variable length arrays by reference. They can still be deduced as pointer types via the normal array to pointer decay rule.

- GCC allows empty structures as a C extension, and gives them a size of zero. C++ allows empty structures, but specifies that their size is not zero. They have a non-zero size to preserve the invariant that no two objects of the same type have the same address. GCC does not keep that invariant for such empty types. Structures that contain empty members will be laid out differently in C and C++.
- The implicit typename extension described above has now removed from G++.

Because C++ extensions can have so many unseen consequences, the G++ maintainers require a very strong argument and implementation in favor of an extension, before accepting it. Incompletely documented extensions lead to problems in maintaining G++ [12].

5 Closing Remarks

C++ support in the 3.x versions of G++ has improved considerably over that in the previous 2.x versions. Improving C++ conformance is not without pain to users who have unknowingly been writing ill-formed C++. G++ aims to smooth the transition by deprecating inappropriate features and giving a warning in one version and then removing the feature in the next version. When a new error message is added, because of better standard conformance, explanatory text might be added to help the user correct their code.

Various improvements and ways that user programs can effect the quality and speed of compilation have been described. Library writers are particularly inhibited by the lack of a module system, and workarounds are shown so that library link time can be reduced.

There are still new optimization opportunities in G++, for instance a multiple entry point mechanism for thunks, so that multiple inheritance is even cheaper in both speed and code size. Better exception tracking can be added to remove unnecessary runtime checks.

References

- [1] Programming Languages — C++, ISO/IEC 14882:1998.
- [2] Itanium C++ ABI, <http://www.codesourcery.com/cxx-abi/abi.html>.
- [3] System V Application Binary Interface, <http://www.caldera.com/developers/gabi/>.
- [4] Using GNU ld, <http://sources.redhat.com/binutils/docs-2.12/ld.info/index.html>.
- [5] Pete Becker, Draft Proposal for Dynamic Libraries in C++, <http://std.dkuug.dk/jtc1/sc22/wg21/docs/papers/2003/n1428.html>.
- [6] Bison Parser Generator, <http://www.gnu.org/software/bison/bison.html>.
- [7] Aho, Sethi, Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [8] Gnu C Library, <http://www.gnu.org/software/libc/libc.html>.
- [9] ISO/IEC Working Group 21 <http://std.dkuug.dk/jtc1/sc22/wg21/>.
- [10] C++ Core Issues List, http://std.dkuug.dk/jtc1/sc22/wg21/docs/cwg_active.html.
- [11] C++ Library Issues List, http://std.dkuug.dk/jtc1/sc22/wg21/docs/lwg_active.html.
- [12] Zachary Weinberg, A Maintenance Programmer's View of GCC, GCC Developer's Summit, 2003.