Proceedings of the GCC Developers' Summit

June 22nd–24th, 2005
Ottawa, Ontario
Canada

C/C++ Compatibility on Linux

Requirements, Achievements, and Prospects

Joe Goodman
Intel, Inc.
joe.goodman@intel.com

David L. Moore
Intel, Inc.
david.moore@intel.com

Abstract

This paper discusses the compatibility needs of Linux users, the kinds of compatibility that are needed and means of achieving them. It reviews the successes of recent years in achieving compatibility between C++ compilers on Linux. We show that a still higher level of cross version compatibility is valuable to Independent Software Vendors. We describe an approach that will allow these needs to be met without crippling further development of libstdc++.

Introduction—Why Compatibility is Important to Linux

Linux has become the platform of choice for many proprietary products. Because the products have been used on other platforms for many years, their availability on Linux makes migration to Linux viable for their users. This in turn helps attract other software and hardware vendors to the platform. For this reason, the availability of a wide variety of proprietary products on Linux should be welcomed and encouraged by the community.

Because of the nature of proprietary software development, these products flourish best in a stable environment. This need has the potential to create a tension with those users attracted to Linux as a source of, and vehicle for, innovation. Those innovations are critical to the success of some users. The multi-threading improvements made in the 2.6 kernel are just one example. Many customers need both the stability and the innovation, so choosing the former over the latter is not a good solution for many customers and is not even viable for some.

Through working with both internal and external customers we have gained insight into the compatibility needs of the developers of these products. These needs affect our own product and support plans, and meeting them turns out to be a significant undertaking. This paper describes our view of those requirements and suggests a method by which they can be met in the hardest case while maintaining Linux as an innovative platform.[1]

1 Kinds of Compatibility

It is important to distinguish a number of different levels of compatibility. Roughly, in order of perceived difficulty, we categorize them as follows:

[1]The views expressed in this paper are those of the authors and not necessarily those of Intel.

- Source Compatibility. A program developed on an earlier version of Linux can be compiled and run on a newer version.
- Binary Compatibility. A program (or DSO) compiled on an earlier version of Linux can be run on a newer version.
- Compiler Compatibility. Two compilers are compatible if code compiled with one compiler can be linked with code compiled with the other and the resulting executable runs correctly.
- Library Compatibility. A library linked with one compiler can be linked with code compiled with a newer compiler and set of headers.

1.1 Source Compatibility

We have source compatibility if we can pick up an arbitrary correctly written program that was written against an earlier version of Linux, compile it on our new Linux version, and have it work. This is the level of compatibility required by most Linux distributions as all the packages are recompiled as part of the new distribution.

Problems can arise when programs are not correctly written. For example, some programs broke when `errno` became a macro because they declared this as a variable instead of including the system header. We have also seen programs that used internal structures used by the dynamic loader even though they were documented as private. Such issues represent bugs in the source rather than defects in the level of compatibility.

1.2 Binary Compatibility

We have binary compatibility if we can pick up a program compiled on an earlier version of Linux and run it on a newer version. We deliberately constrain this to not include moving from newer versions of Linux to older versions since this is of insufficient value to justify the constraints it would place on new development.

We find ourselves supporting users who are still on versions of Linux produced five or more years ago, as well as users using the newest releases. So far we have been able to do this, without having to produce multiple versions of our own product, by building our products on the oldest Linux that remains of interest. That we are able to do this represents an important success for Linux over that period.

A good example of an issue during this period that could have broken binary compatibility but which was handled so as to avoid it was the change-over to the use of thread based locales. By gradually phasing in the change, and making it available before it was mandatory, developers were given a means of maintaining binary compatibility over many versions of Linux.

1.3 Compiler Compatibility

We have compiler compatibility if we can mix code compiled with two compilers. The two compilers in question may simply be different versions of the same compiler.

Compiler compatibility, and the stronger library compatibility are of importance to users who cannot or do not wish to recompile the libraries they depend on. The biggest category of such users is proprietary software developers who use third party proprietary libraries since these are often supplied as binaries.

Compiler compatibility also permits developers to compile their source with different compilers so as to take advantage of the differing strengths of those compilers. For this reason this is also called compiler interoperability.

We shall talk about compiler interoperability in some detail in a later section.

1.4 Library Compatibility

For clarity of exposition we distinguish Library Compatibility from Compiler Compatibility to distinguish the requirements of generating the same code for the same source from those of generating compatible code in the presence of changing headers.

For C, the interfaces defined in the headers are often defined by standards and are therefore fairly stable. Care still needs to be taken when making changes but the rate of change itself is slower.

For C++, the impact of changing headers is greater because a lot of improvements continue to be made and, most of all, because much of the implementation is visible in the headers as templates.

2 The Lifecycle of a Typical Proprietary Product

A typical proprietary product has a rather different lifecycle from the typical open source product. For open source products, the emphasis is on the source code and new development. Users are encouraged to build from source, to modify it, and to contribute their modifications back to the product. The viability of this model for many open source products, and that it has benefits for many users, has been amply demonstrated.

Proprietary products, by contrast, are usually shipped as binary, and are subject to a great deal of validation testing before being released. Customers for these products are generally looking for very low bug counts and low support costs. As a result, lead times for individual versions tend to be longer and new versions less frequent.

It can take a year to validate a new version of a product. Once a product has been validated changing anything that the product depends upon requires a complete new validation cycle. So just upgrading, for example, from one version of gcc to a newer one can be a year's work.

So these products certainly need a high level of binary compatibility across versions of Linux.

If compatibility is broken, because of their product cycle, a user may not be able to upgrade to the newer Linux version for as much as several years. Worse, if the product cycles of the various proprietary products used by a given customer do not mesh, there may be no one version of Linux on which all the user's packages will run.

For the developer of proprietary software, the situation may be still worse.

A product may depend upon third party libraries. In this case, if library compatibility is not maintained, one cannot even start to upgrade to a newer Linux until its third party libraries have been upgraded. This entire process can take years. For example one of our internal customers is still developing using a library supplied in binary built with a 2.96 compiler because that is the latest version of the library available. Hopefully this is an outlier.

So these products need a high level of library compatibility across versions of Linux.

3 Achieving Compiler Interoperability

Interoperability means that object files built with one compiler can be linked with object files built with a second compiler, producing an executable that runs correctly. [Goodman] is a detailed discussion of what is required for interoperability to happen. In this section we will focus on how to achieve interoperability and to verify that interoperability has been achieved.

One might try to achieve interoperability by taking a number of large applications, compiling parts of each with one compiler and parts with the other, linking them together and, when they did not work, debugging the reason why.

In practice this does not work very well for a complex language like C++. Even for a simple language like C it is easy to get unpleasantly surprised. There is one compiler, for example, that passes a 64-bit float, if it is in just the right place in the parameter list, with half on the stack with the other half in a register. Finding such an incompatibility using the above process is largely a matter of luck.

Worse, even when one does discover an incompatibility one has to understand it well enough to duplicate the desired behavior. If the standard for compatibility is "whatever XYZ compiler does" then working out what that is can be quite hard.

Fortunately, for Linux we have both a C++ ABI standard [CodeSourcery] and a community that is committed to maintaining that standard when problems are found and to making sure gcc compilers conform. The community has expended considerable effort to make and keep gcc ABI compliant, starting with gcc 3.4, and has also vigorously resisted an attempt to make the binary standard "whatever gcc 3.3. does."

As a result, we have a well defined standard that can be used as the basis for producing interoperability with any version of gcc as well as maintaining compatibility between versions of gcc itself We can achieve interoperability with the following steps.

- Assess ABI Compliance
- Catalog gcc ABI deviations and emulate
- Validate with mix and match testing

3.1 Assessing ABI Compliance

Given that we have a C++ ABI standard document, we can start interoperability work by ensuring that our C++ compiler conforms to that standard. CodeSourcery developed an ABI conformance suite that does a very good job of determining if a compiler conforms to the ABI standard and in isolating deviations. This can be used both to find problems and to test for regressions.

The CodeSourcery suite has been used by us and others to find ABI deviations in gcc 3.2 and gcc 3.3 that, when reported as bugs, were fixed by the gcc developers so as to produce a high level of ABI compliance in gcc 3.4. They have also been used to find any deviations that crept into gcc 4.0 and these have also been fixed by the gcc developers.

We found that the CodeSourcery suite does a very good job of finding deviations from the ABI standard. It has been the basis of our own work on the Intel® C++ Compiler to make it compatible with a range of gcc compilers from 3.2 through 3.4.

The most important benefit of having interoperable compilers is that it allows third party libraries to be distributed as a single version. As discussed elsewhere in this paper, we believe this is a very important goal on Linux, and to support it the compiler writer has no alternative but to produce the highest level of interoperability achievable. In turn this means that an accurate and complete ABI standard is a key requirement for Linux to meet the goals described in this paper.

3.2 Emulating Deviations in a Released Compiler

As well as identifying deviations in our compiler, the CodeSourcery suite can be used to identify deviations we will need to emulate in order to interoperate with a released compiler.

There are two possible kinds of ABI deviation: those that result in a correctly running program, and those that cause the problem to fail. Only the discepencies that do not cause program failure should be emulated.

3.3 Validating with Mix and Match Testing

Not all interfaces are currently defined in the ABI document. For example, stack unwinding table formats are not defined. These tables need to be compatible so that two compilers can share the same language support run-time. If the run-time cannot be shared then the binaries cannot be linked together at all.

In order to test the interoperability of these items and also to validate the success of the other stages, the next step is to do extensive mix and match testing.

There are two approaches to performing mix and match testing. The first approach is to create a random test generator that generates C++ source files. Each generated source file is compiled with both compilers and the contents of the object code are examined. The object files are checked to see that names are mangled compatibly and data structures are allocated compatibly. Any difference in name mangling could be exposed as a link time error. Any difference in data structure allocation could be exposed as a run-time error.

The second approach is to take a number of large applications, build each with both compilers, and link intermixed objects from the two builds. If there are no interoperability issues, each of the combinations of object files will produce a running program. If there are issues, these will usually show up as a link-time or run-time error. The larger the application and the more thorough the tests available for the application, the greater the chance of this process finding any remaining problems.

If a problem is found, a compilation unit that evinces the problem can be isolated by starting with binaries entirely from one compiler and substituting in the binaries of compilation units from the other compiler one by one until a failure ensues. When doing this it is important to start with the binaries from the compiler used to create the run time libraries.

4 Cross Version C++ Standard Library Compatibility

Maintaining compatibility across versions of the C++ libraries is much more difficult than maintaining compatibility across C library versions. Now that we have converged on the C++ ABI this is the next compatibility issue that needs to be addressed.

C library compatibility is greatly helped by the fact that there is a well defined binary interface. A few macros, such as that for errno, certainly exist, but these are simple and give one some additional flexibility in preserving compatibility rather than creating extra constraints. In C++, on the other hand, the library contains a large number of templates and these templates contain large portions of the implementation of the library.

Because the implementation is exposed, this greatly increases the coupling between the library and the source program. Large portions of the implementation of the standard C++ library get compiled into both the third party library and the user program. As a result, the standard technique of creating an interface that the library keeps fixed, or at least compatible, does not work.

While it might be possible to create a standard library in which the exposed templates were just proxies for hidden implementations, such an implementation would pay a very substantial performance penalty for the additional layer of abstraction.

The only simple way to maintain compatibility across versions of the library is to keep the implementation unchanged. This is a very high barrier to innovation.

If any innovation is going to be allowed then the affected parts of the library are going to be incompatible from one version of the library to the next.

5 A Simple Approach

As the C++ headers and libraries are so tightly bound together, a simple approach is to maintain multiple versions of both the headers and the binary libraries so that a user can choose to use an older set of headers and libraries for compatibility purposes.

The only requirement placed on the maintainers by this approach is to make sure that new versions of gcc can continue to build and run the older library sources.

In the past we would also have had to maintain the ability to generate non-ABI compliant code and this was hard to do, but with gcc 3.4 and beyond this problem should not arise again. At present we believe the level of ABI compatibility between gcc 3.4 and gcc 4.0 is very high and we anticipate that will remain high in the future.

An impact of this approach is that distributions must supply the backwards compatible headers as well as the backwards compatible binaries, but this does not seem like a large burden.

While this will work for many, it leaves the developer discussed earlier who is using a third party library stuck with the C++ Standard Library used by that library. So they will not be able to take advantage of new work and, given that quite a bit of this work is likely to be directed towards better performance and not just new functionality, this leaves the developer at a competitive disadvantage.

Worse, a developer relying on several such libraries may not be able to select any one version of the library that works for all the third party libraries he or she is using.

6 Obtaining Compatibility with Attribute Strong

Fortunately there is a way in which even these worst case user requirements can be met, with reasonably straight-forward use of attribute strong. The application developer with mutually incompatible third party libraries will have to do some extra work, as we shall show, but would not be totally blocked.

According to the documentation the semantics of attribute strong may change over time. However we believe that the need is sufficiently clear that this will not happen.

What attribute strong does, in general terms, is make any symbol declared in the namespace being used behave as if it was declared in the namespace doing the using. As we shall see, an important point is that the name mangling of the symbol remains unchanged. It continues to be mangled with the name of the namespace in which it was actually declared.

The gcc 3.4 manual (section 6.9) states:

> A using-directive with __attribute ((strong)) is stronger than a normal using-directive in two ways:
>
> - Templates from the used namespace can be specialized as though they were members of the using namespace.
> - The using namespace is considered an associated namespace of all templates in the used namespace for purposes of argument-dependent name lookup.
>
> This is useful for composing a namespace transparently from implementation namespaces.

The intent here is that there be an as-if rule in force: name resolution in the using name space behaves as if the templates declared in the used namespace were in fact declared in the using namespace. The simple example in Figure 1 demonstrates the use of this attribute.

```
namespace ns1 {
  class T1 { int x;};
};
namespace ns2 {
  class T2 { int x;};
};
namespace ns3 {
  using namespace ns1;
  using namespace ns2
     __attribute((__strong__));
  int f2(T2 t2);
  int f1(T1 t1);
};
int main() {
  ns3::T1 n1;
  ns3::T2 n2;
  f1(n1);  //error
  f2(n2); // ok
}
```

Figure 1: The difference attribute strong makes

The argument dependent lookup for function `f1` is done just in the namespace `ns1` and therefore fails to find `f1`. The lookup for `f2` checks namespace `ns2` and, because of the use of attribute strong, namespace `ns3`.

Clearly the use of attribute strong has subtle and non-local effects to name resolution and therefore needs to be used carefully; however, this does not seem to present a major problem with its use in the context of the C++ Standard Library.

The existing use of this attribute is to separate the implementation of the C++ Standard Library into debug and release versions. Therefore we have evidence that attribute strong can be used without a detrimental impact on the application developer.

6.1 The Product Lifecycle

To demonstrate how attribute strong would be used to version libraries in practice, here is a very simplified example of a header that we might get with a third party library. We assume that the library vendor shipped us the source for a number of headers but that the implementation is only available to us in binary. Figure 2 shows an example header.

```
#include <list>
void acme_libcall(
  const std::list<int> &x);
```

Figure 2: An example header from a third party library

Let's say that the implementation of `acme_libcall` was compiled with an older compiler being used by the supplier. If we, as application developers, compile using the latest compiler, and if the implementation of list has changed, we are in trouble. Internally the library will use the implementation of List from the headers of the old compiler while we will use the newer implementation from the current headers. The compiler will not detect a problem since the supplied third party headers, when we compile with them, will also use the new implementation, but our program will not run.

But suppose the C++ library developers had declared the list type as shown in figure 3

There are only a couple of points of interest here. Most importantly the list class has been couched inside a versioned namespace. Second, this is then imported into std unless we inhibit this by defining `__DONT_USE_STD`. Third, we have versioned the guards around the header file.

Now suppose a C++ Standard Library developer decides an improved list class would be in order for the next release. In order to maintain backwards compatibility, they would write the new definition as shown in Figure 4.

So now we have a header that will include the legacy header (which is going to be moved to a version2 subdirectory) whenever we define `__BACKWARDS_COMPATIBILITY`.

```
#ifndef __LIST2__
#define __LIST2__
namespace version2 {
 template<class T> class list {
  ...
  };
 }
#ifndef __DONT_USE_STD
namespace std {
  using namespace version2
    __attribute((__strong__));
  };
#endif
#endif
```

Figure 3: A list implementation (fragment)

```
#ifndef __LIST3__
#define __LIST3__

#ifdef __BACKWARDS_COMPATIBILITY
#ifndef __DONT_USE_STD
#define __DONT_USE_STD
#include <version2/list>
#undef __DONT_USE_STD
#else
#include <version2/list>
#endif
#endif
namespace version3 {
 template<class T> class list {
  ...
  };
 }
#ifndef __DONT_USE_STD
namespace std {
 using namespace version3
    __attribute((__strong__));
 };
#endif
#endif
```

Figure 4: A new implementation of list

Notice that the version of `list` that appears in `std` will always be the new one.

We can now see how this helps our application developer interface to the legacy library. Recall that the original header included the system list class and then declared a function that took a list as a parameter.

If the original library implementor understood the versioning system and was diligent in supporting it, that header would have been written as shown in figure 5.

```
#include <list>
void acme_libcall(
  const version2::list<int> &x);
```

Figure 5: A versioned library header

Or, more likely, as they want to be able to upgrade at some point, as shown in figure 6.

```
#include <list>
#include "acme_version_info.h"
void acme_libcall(
  const ACME_VERSION::list<int> &x);
```

Figure 6: Just keeping the version in one place

Now, as application developers, when we use this header in our environment, the declaration will still work as it did for the library developer. We will get the version 2 list. Our code will link to the supplied binary libraries correctly.

In our code we need to be careful to type our own lists that we want to pass to `acme_libcall` with type `ACME_VERSION::list`. If we fail to do this, and instead use the `std::list` template class, we will get compilation errors which, as they are in our own source, are amenable to being fixed.

Now you are going to complain that it is a lot to expect the library developer to follow this convention so as to make it possible for us to use the newer compiler and library. The nice thing about this approach is that, even if the original library developer did not code the headers in the required form, we can do it ourselves and the library will still work.

This is so because even though the library developer declared the parameter to be of type `std::list` its name mangled using the type `version2::list`. So after we change the header to follow the convention, linkage will still work.

6.2 Linking the Application

With this new scheme the use of versioned C++ libraries is no longer desirable. Rather than creating versions of symbols, we create versioned namespaces so that the symbol names themselves change.

A simple approach is to retain all of the old symbols in an updated library. So, in the examples given above, the version 3 library will contain everything that was in the version 2 library. This may be an unnecessary burden on users who do not need the backwards compatibility.

However we can arrange things so that users who do not need backwards compatibility do not pay anything for it. The library with the old compatibility interfaces, but the new implementation as well, can be shipped using the old version number, while a smaller new library without the legacy binaries in it will be shipped with a newer version number.

In this paper we have only discussed the impact of changes in templates on C++ Library compatibility. There are, of course, parts of the C++ Library, especially those involved with input-output that are embodied in the library binaries that would need to remain compatible from version to version. The versioning system described here can be used to give the maintainer of those parts of the library flexibility and in addition these portions of the library do not present the same level of difficulty as do the template based parts of the library. They are much more equivalent to the C library situation where it has been possible to maintain compatibility for many versions.

So we believe that the approach we have outlined is sufficient to meet the backwards compatibility needs of C++.

Conclusion

In this paper we have shown how compatibility is just as important an attribute for Linux as some attributes of longer standing because they allow an important class of users to embrace the platform. We have shown that even the most difficult problems with cross-version compatibility can be addressed without impacting the continued evolution of Linux.

References

[Goodman] Joe Goodman *Interoperability & C++ Compilers*, C/C++ Users Journal, Volume 22, Number 3, 2004.

[CodeSourcery] CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, SGI *Itanium C++ ABI* `http://www.codesourcery.com/cxx-abi/abi.html`