Binary Compatibility of Shared Libraries Implemented in C++ on GNU/Linux Systems.

Pavel Shved
Institute for System Programming, RAS
email: shved@ispras.ru

Denis Silakov
Institute for System Programming, RAS
email: silakov@ispras.ru

Abstract

A shared library is a file that contains library code and data in binary form. Application built against the library references the data via symbols and the contents of what's being referenced get known only during the application startup. Library is shipped with header file(s) the program is compiled with.

The problem of the binary compatibility (sometimes called „backward compatibility") arises when the new version of library is installed into system and the program, having not been recompiled, is attempted to run in the environment with the new library. The incompatibility may result in fatal errors during the startup or even during the runtime.

In this article we deduce the rules that must be followed in order to keep the binary compatibility of a shared library.

Unlike most of researches in this area, we also assume that the library may contain its own restrictions upon its usage, more powerful than restrictions of C++ language itself. So the possible restrictions are analyzed as well, and we attempt to weaken the rules of binary compatibility when such restrictions are enforced.

As a conclusion we list the rules a programmer should follow to keep his or her library binary compatible. We also conclude that possible restrictions limiting the use of library allow to weaken these rules in relatively small number of cases.

For the purpose of this study, we create formal notation for the process of building and using a library, introduce formal definitions of source and binary compatibility and of program behavior. We base the assumptions about mapping source code entities to binary level on the Itanium C++ ABI standard, which describes `gcc`'s way of emitting binary code and data.

Index Terms

Languages, Software libraries

1. Introduction

1.1. Premise

C++ nowadays is one of the most popular programming languages [1]. It *is* complex, but the complexity made it become one of the most powerful multiparadigm language, that can combine low-level efficient code with high-level concepts of generic and object-oriented programming.

Like any serious programming language, C++ needs the concept of „library". One of the doctrines of C++ design was compatibility with C language at the source level (with a few exceptions only) and with interfaces of C shared libraries. The decision was necessary to make language more popular by ability to re-use of existing C code [2]. But the questions of C compatibility is still urgent since C is also one of the most popular programming languages ([1]). So, the concepts of C++, that are absent in C, use the same binary interface architecture: the memory the object file occupies is addressed with strings—„symbols". Their semantics is (as in C) not described in binary file, but rather is enforced at compile-time by the header file of the library.

But, compared with C++, libraries have notable peculiarities. They can contain the code of a subroutine provided either at source or at binary level (some libraries, like `boost` or `stdc++`, contain few binary code; most of code is inlined and instantiated in compile time). Further, C++ concepts are much more complex than C's ones, hence the semantics of data being referenced by plain old symbols is also more complex.

Sooner or later, each library faces the question of compatibility with previous versions. For compiled languages the library author should maintain not only source compatibility (i.e. the ability to successfully compile same program code with the new library headers), but also binary compatibility, which lets the program to run with the new binaries of the library without recompilation and to produce the same results as with older version.

Maintaining binary compatibility is a complex task, and it's also important as it takes considerable time to generate effective low-level code out of C++ source. But new versions of popular libraries are released quite often, because every library has numerous bugs that are to be fixed, improvements that are to be implemented and new concepts that allow to create better applications in the future. And the popularity of the library also means that there exist many applications, that use it. They all would have required recompilation if binary compatibility would break with every new release. Furthermore, in the new environment applications may also work differently, but their developers usually promise fixed behavior in advertisements and documentation.

Hence the binary compatibility problem is very important

Table 1. Special symbols used in this paper

$\hookrightarrow$	$A \hookrightarrow B$	A class is an (indirect) base of B
$\mathcal{B}$	$C.\mathcal{B}$	set of all bases of C
$\mathcal{B}^V$	$C.\mathcal{B}^V$	set of all direct virtual bases of C
$\mathcal{P}$	$C.\mathcal{P}$	most derived primary base of C
$\mathcal{V}$	$C.\mathcal{V}$	array of C's virtual tables.
$-$	$C.to - C.from$	offset

for C++ libraries. However, popular articles on this question are not complete. Instead, they introduce mere list of rules one should follow, the rules usually coming out of practice and the completeness of their set never being claimed. No wonder that these lists differ in such researches. No article also takes into account that, apart from the restrictions of C++ language, that are expressed in header files, a library can have additional restrictions, imposed in free form by documentation.[1] It is possible for these additional restrictions to widen the set of compatible library code.

This article aims to fill these gaps and use a formal approach to binary compatibility problem. We first describe the compiling and building process of library and applications that use it, then we study how the new environment affects the application behavior. It is GNU/Linux system and its default instruments: `gcc` compiler(„GNU Compiler Collection") and `ld`, „The GNU Linker". The toolset being used is chosen to be best specified and popular enough ([6], [7]).

We use [4] as a formal ground for binary analysis; `GCC` follows this standard.

We will not study the questions of inter-compiler and inter-platform compatibility. Also we keep aside compatible exception handling (as most of libraries do not use them). What we will concentrate our forces on compatibility of classes and functions, which code is placed into object files.

1.2. Notation

To denote the objects discussed we decided to mix mathematical formalism and the notation used in object-oriented programming. To denote subobjects of compound objects we use C++ notation, where $L.c$ means c subobject in L. The same rule holds for member functions; for example, $x.f(y)$ may be treated as $f(x, y)$.

Finally, the table that shows how special symbols are used and what they mean is shown on figure 1.

1.3. Basic Concepts

1.3.1. GNU/Linux System. In this section the objects involved in program life cycle in GNU/Linux system are described in a formal way. In outline, there are compiler, assembler, linker and underlying semantics of objects they manipulate, regardless of whether they reside in RAM or on disk.

1. Sometimes these restrictions are considerably significant. For example, destructors of widget classes in `Qt` library, are called during the destruction of parent widget. That forces programmer to create widgets with `new` operator only and prevents user classes from having widget members.

1.3.2. Compilation. Let's mark out the stages involved in compilation and running of a C++ program.

Source code of an application consists of several *source files* (*cpp* files). Some of them have relevant *header files* describing their interfaces. Each source file may include headers both from the application in subject and from other libraries. Then each *cpp*-file $f.cpp$ is preprocessed and stored in RAM. The preprocessed code, being then independent of other program entities or environment, is compiled, resulting in the *object file* $f.o$.

We combine these stages and will write as follows:

$$f.o = f.c.compile(f.h,\, lib.h).$$

As arguments this *compile function* takes header files $f.c$ includes. Indeed, if $f.cpp$ contains **`#include`** `<lib.h>` line, then preprocessed source will change subject to $lib.h$ contents. This change is represented in a natural way—as an argument to function. For example, $f.c.compile(f.h,\, lib'.h)$ denotes that *the contents* of header file included via the particular line with „include" statement are changed, its name acting as reference only and being invariant.

After all sources are compiled, the work flow proceeds to *linking* stage. Its result is an *executable file* $f.exe$, in which the code of all object files and statically linked libraries is comprised. We denote it as

$$f.exe = f.o.link\,(f_1.o,\, f_2.o \ldots f_n.o)\,,$$

where $f.o$ is an arbitrarily chosen object file from the set of all ones being compiled.

However $f.exe$ still doesn't ultimately define the code that will be executed. If the application was compiled with a *shared library*, then, at the start of its execution, dynamic loader's code is run. Its purpose is to link the symbols used in application executable as references to the actual code and data addresses in RAM. The libraries to be loaded dynamically are chosen from the current environment via sonames and may differ from the ones linked to during preparing the application executable (see [5] for details). Therefore, similarly to the compile function notation, the dynamic linking function should also take shared objects as arguments: $f.exe.dynlink(lib_1.so,\, lib_2.so,\, \ldots)$.

The result of running an application and its consequent possible dynamic linking is *application context*. The context then consumes *input data* and produces *output data*. „Input data" comprises very broad range of data: OS state, sequence of random numbers returned by corresponding functions, command line arguments, filesystem state and environment variables. In other words, input data unambiguously define output data.

The application context obtained is therefor a map from set of all input data to set of output data. We denote it like an usual function:

$$f.exe(input) = output \tag{1}$$

$$f.exe.dynlink(lib.so)(input) = output \tag{2}$$

1.3.3. Notes on Specifications. Talking over binary compatibility, not only we require the application to run in new environment, but expect it to yield the correct results. The most common way to define correctness of the behavior is to use concept of *specifications*. Program behavior is described by its context, which is a map from input to output. So, it's natural to define *specification of a program* as map from *correct input* to output, the mapping being *conformant* to library code.

Correct input is an input that doesn't cause precondition violation [2] for shared library functions, as if abstract C++ machine defined in C++ standard [3] is run.

By *conformant mapping* we mean the mapping, that executing the library functions on abstract machine in such way, that their return values and side effects are conformant to the relevant library specifications.

Therefore we can define, that *program conforms to specifications* („works correct") if its context and specification, restricted to the set of correct inputs, are equal. We denote it like this:

$$f.exe.dynlink(lib.so) \stackrel{\cap}{=} spec \tag{3}$$

Contract library specifications may govern not only function calls but virtually every way of using of concepts defined in header files. The consequences were discussed in .

A program or a library may be „badly written", i.e. have empty set of correct inputs. In further analysis we only take „well written" libraries and programs into account, their set is denoted as $\mathbb{P}$.

1.3.4. Shared Libraries. Let's define our subject. A *shared library* Lib is a structure

$$Lib = \langle H = \{h_i\}, CPP = \{cpp_j\}, so, soname, spec \rangle,$$

where

- h_i are header files;
- cpp_j are source code files that contains definitions of (some) functions and variables declares in header files;
- $so = link(..., cpp_j.compile(h_{j_1}, h_{j_2} ...), ...).makeso$ — an object file compiled in a special way, which can be used as an argument for dynamic linking function;
- $soname$ is a library name via which the so-file is found by dynamic linker;
- $spec$ is an encapsulated object with the sense discussed in 1.3.3.

The definition requires *cpp* files to comply with header files declarations—i.e. compile without errors.

2. Binary Compatibility

2.1. Definition

To simplify the further study we will consider program p, that consists only from one source file, and a library L, that also consists from one source and one header file.

2. a library documentation practically doesn't use the word „precondition". But its sense is comprised in phrases like „the value of this parameter *shall be* more than zero" or „the behavior for NULL pointer *is unspecified*".

Table 2. Symbol versioning example

$A'.h$	$B'.h$

`namespace libA{`
`#include <A.h>`
`}`
`using namespace libA;`
	`namespace libA{`
	`#include <A.h>`
	`}`
	`namespace libB{`
	`#include <B.h>`
	`}`
	`using namespace libB;`

Let's assume that a new version is released — L'. Then L' *library is binary compatible with* L, [3] if

$$\begin{aligned} \forall p \in \mathbb{P} \rightarrow & p.cpp.compile(L.h).link(L.so). \\ & dynlink(L.so) \stackrel{\cap}{=} \\ & \stackrel{\cap}{=} p.cpp.compile(L.h). \\ & link(L.so).dynlink(L'.so) \end{aligned} \tag{4}$$

If we would recompile program in the new environment its context would look like that:

$$p.cpp.compile(L'.h).link(L'.so).dynlink(L'.so).$$

The behavior in new environment may hence differ; that's what we are to evade.

The definition of binary compatibility doesn't really constrain itself to the cases where it's useful. For example, there's no obstacle that prevents us from investigating the issues of compatibility of math library and window manager library. Therefore we should lay down one more condition, namely *source code compatibility*.

Library L' *is source compatible with* L, if

$$\begin{aligned} \forall p \in \mathbb{P} \rightarrow & (\exists p.o = p.cpp.compile(L.h)) \\ & \Rightarrow \left(\exists p'.o = p.cpp.compile(L'.h)\right). \end{aligned} \tag{5}$$

Later on we will discuss only binary compatibility of source compatible libraries.

2.2. Symbol Versioning Approach

Let's examine first the following method referred to as *symbol versioning*. We will now show that any libraries may be made binary compatible even if they're of different kind. Indeed, assume A and B are libraries. Then let's create A' library, that's source compatible with A, and such B' source compatible to B that it is binary compatible with A'.

To achieve this we keep *cpp* files intact, and link the objects files into single *so* file, having rewritten the header files as shown on figure 2.[4]

The libraries produced are binary compatible and same programs may be compiled with them. Source compatibility between B' and A' is equivalent to that between B and A.

However this approach suffers from the code duplication.

3. we say that the *new* version is compatible with the *old* one, and not vice versa

4. `C` language doesn't have namespaces, but `gcc` introduces the other symbol versioning mechanism for C language; see [5]

Note that symbol versioning can not be used with every bugfix release, as each bugfix forces us to create a new function instance. Symbol versioning is used (for example, in `glibc`) with major releases, the functions in bugfix releases sharing the same version and having to be binary compatible.

To prevent duplication we might use aliases for duplicating code and data. But how can we be sure that code is duplicated? The question of whether the code of function is duplicated with version change is essentially equivalent to the question of binary compatibility. Therefore, symbol versioning doesn't govern all binary compatibility problems and further study is necessary.

2.3. Causes of Binary Incompatibility

There are different ways to build an run application:

$$p.cpp.compile(L.h).link(L.so).dynlink(L.so) \quad (6)$$

$$p.cpp.compile(L.h).link(L.so).dynlink(L'.so) \quad (7)$$

$$p.cpp.compile(L'.h).link(L'.so).dynlink(L'.so) \quad (8)$$

Contexts (6) and (7) differ only in dynamic linking function arguments, i.e. in the set and internals of the symbols the library exports. In `C` language it only means that during the calls to the functions with external linkage the other code will be executed. However, C++ libraries contain more symbols than those of functions and static data, and some of auxiliary symbols are used in an unobvious way. Binary compatibility is caused by the difference between symbol sets and contents in different versions of library.

According to the definitions of building the application the object file $p.cpp.compile(L.h)$ already contains information about symbols it might need in the shared library. However, an only source of information about the library so far is header file $L.h$. Therefore, *the set and internal structure of symbols exported are completely defined by header file*. The source file $L.cpp$ only defined internal code and *some* data these symbols point to.

However, during the linking with $L.so$ no code or data defined in $L.cpp$ is inlined into $p.exe$. This code is loaded on dynamic linking stage only. But at this stage the application requires the presence of symbols defined in $L.h$, but the library $L'.so$ the program's being linked to provides symbols, defined by $L'.h$. That's one of the causes of binary incompatibility.

Furthermore, if a code defined in *cpp* file is called via symbol, the callee will consider that arguments are laid out like in callee's native header file, $L'.h$. However, the caller assumes that all library data is laid out like in $L.h$—the file the program was compiled with. That's an important matter, because the semantics may stay unchanged from the user point of view (summarized in C++ „source" standard [3]), it could change from binary point of view (ABI standard [4]).

We can now divide the causes of incompatibility into the following groups:

1) **implementation change**, the case when constant or function whose signature remains intact, changed its definition;
2) **compiled code notions incompatibility**, the case when the notion the function in *cpp* file has of layout of its arguments has changes due to alteration of declarations in header file. These alterations only depend on $L.h \rightarrow L'.h$ change and may be studied separately from 1.
3) **altering or removing of special symbol**, which definition is deduced by the compiler based on header file. The data the symbol refers to could be used in application binary code as in notions given by $L.h$, but handled in $L'.so$ as in $L'.h$.
4) **errors during dynamic linking**, caused by absence of definition of an entity from $L.h$ in $L'.cpp$ hence $L'.so$.

3. Study of Incompatibility Sources

Let's thoroughly study the sources described in 2.3.

3.1. Errors During Dynamic Linking

After the application is built against $L.so$, no static linking errors can arise. However, at the time it's run dynamic linker tries to link it with $L'.so$ instead (because $L.soname = L'.soname$), so it may lack symbols the application references and abnormally terminate before it has any chance to run its code.

Dynamic linker searches all *external dependencies* of $f.exe$ in the libraries loaded. They're fixed at compile-time and marked at link time as external, when the linker finds definition of symbol in a shared library rather than in static one or application's object file. Here the linker ensures that all dependencies are satisfied. The dependency may be global variable's symbol, non-inline function, non-template function of fully-instantiated template function.

We well use term „use symbol" instead of „use declaration the symbol relates to".

We should note that, among external dependencies, there can be the symbols allowed to be used in userspace code. *Userspace code* is a code that compiler takes as an input. It therefore contains $f.cpp$ and $L.h$; and inline functions defined in $L.h$ in particular. The rules for keeping compatibility will be formulated in terms of userspace code, so ***the library developer must follow the rules in his own inline functions code as well***.

We should also note that for a member function call not only its own symbol may be required, but also *virtual table* (vtable) may be involved for virtual functions and for non-virtual member functions of virtual bases calls. Among the other info (see 3.4.2 for details) it contains pointers to virtual functions definitions as symbols, that are resolved in runtime.

$L'.cpp$ may lack symbol described in $L.h$ only because that symbol is not deduced from $L'.h$ (by the property of shared library definition — see 1.3.4). Among explicit non-auxiliary symbols only non-inline functions and global data present in header file. If one of such symbols is withdrawn, and it can be used in $p.exe$, L' would then be incompatible with L'. As it is obligatory to explicitly use such symbol

Table 3. Similar functions but different external names

$L.h$	$L'.h$
`typedef int Type;`	`typedef float Type;`
`void function(Type);`	`void function(Type);`

for dependency to appear, the incompatibility of library is equivalent to the plausibility of this symbol usage in userspace.

Therefore, ***it's impossible to withdraw from header file a symbol that is allowed to use in userspace code***.

A symbol name is constructed out of its declaration through use of „mangling". The mangling procedure is fully described in section 5.1 of [4]; here we only outline some key conclusions.

A symbol (external) name for a declaration for `GCC` compiler is uniquely and unambiguously defined by the conjunction of the following properties:

1) **filly-qualified name of declaration**, the name of declaration and fully-qualified name of enclosing scope (class or namespace)[5];
2) **vector of argument types and vector of template parameter types (for instantiations)**. The types with all typedefs substituted are considered for this purpose, however, structures, classes and unions are not expanded—their name is taken only. For example the functions shown on the figure 3 have different external names.
   All fully-substituted types are encoded in unique way, the types and member functions being also distinguished by cv-qualifiers.
3) **return type of a template function**. If template function is instantiated, its return type is also encoded into external name.
4) **set of function's thunks**. For functions that require adjustment to **`this`** pointer (overloaded functions of non-primary base, separately for virtual bases, virtual base subobjects[6] and all other bases) or to return value (for covariant return types) special entry points are created and then used in the function call algorithm. Special functions called „thunks" for different ways to call the functions are emitted into object file[7].
   The application, that doesn't derive the defining class, can't have them as direct dependencies, because the calls to them can only be encountered through use of vtable. But if application is allowed to derive a class with such functions and it actually does, then the use of them in derived class' members definitions will require adjustment and henceforth the relevant dependency.
   The set of thunks define the rules of confronting the function call operator and the symbol that references the actual code. Therefore, if the number of entry points or the causes of their appearance is changed, it should be encountered as symbol name change. [8]
   Let's formulate the rule: ***let the virtual function `C::f`, be such that $C \in D.\mathcal{B}$. Then the alteration of any of the following properties will cause binary incompatibility:***
   - whether it has covariant return type or, if it has, the return type itself
   - whether $C = D.\mathcal{P}$;
   - if $C \neq D.\mathcal{P}$, whether $C \in D.\mathcal{B}^V$;
   - class W, such that for W holds

$$W \in D.\mathcal{B}^V,\ C \hookrightarrow W \hookrightarrow D \quad (9)$$

$$W \notin D.\mathcal{P} \quad (10)$$

$$\forall W' \in D.\mathcal{B} \rightarrow \left(C \hookrightarrow W' \hookrightarrow W \Rightarrow W' \notin D.\mathcal{B}^V\right) \quad (11)$$

   Compatibility issues that arise from thunks' contents are discussed in 3.4.1.

So, if a property of the definition alternates, the library loses binary compatibility. One of the most stunning examples of it, described in [8], is when one adds default argument, the mechanism initially designed to keep compatibility (unluckily, source one). Indeed, when you add a new default argument to the function, its vector of arguments changes and hence the external name changes as well.

In this list the property „whether function is inline" is absent, because there's no external names for inline functions at all. However, many developers do treat it as function property, so here's the rule for that: *to keep binary compatibility the **`inline`** qualifier must not be added to the function allowed to be called from the userspace code*.

3.2. Implementation Change

New versions of libraries are released to add the new or to remove the obsolete functionality, fix bugs or improve the underlying implementation algorithms. From a newbie's point of view bugfixes do not cause any harm. because it hardly changes anything, especially in the set of external names. From a formal point of view only the improvement of existing algorithm is insignificant as long as it doesn't change the contract of function. What we call „a bugfix" is actually a change of specification and a confession that L's specification is not $L.spec$ defined by help files, but something else, and only $L'.spec$ made true specifications coincide with the alleged one.

When developer changes symbol specifications, the application calls the new implementation, namely the one in $L'.so$ and may change behavior, what leads to losing binary compatibility. It also may not.

Consider the following example. L contains a `streq` function, that compares strings. L' introduces a new feature to compare them case-insensitively, if the global trigger variable **`bool`** `case_insen` was set to true, its initial value being false. The specifications and behavior of `streq()`

5. treat this as recursive definition

6. the term „morally virtual" is used to name such classes

7. they don't have to be separate functions, sometimes they're merely different entry points into single piece of code

8. The set of thunks may be treated as a single „multisymbol" for the current one.

are changed, but no program compiled with L is capable to use the new L''s functionality and actually yield improper behavior.

The change of function implementation doesn't lead to binary incompatibility iff the new functionality is unreachable from any correct executable linked to prior library version.

From this point of view bugfixing *is* a binary noncompliance. The program will work better, but in the other way. Sometimes bugfixes in L cause errors in p, if p implemented a workaround for the bug fixed and it became broken with the new version[9]. However bugfixes are considered useful rather than harmful, because the abstraction of the code into third-party libraries intends to separate the workflow of the application and of the code it uses as backend.

We should note that sometimes binary compatibility is understood as the ability of a program to behave in the new way in the environment with the new library version. Of course, from this point of view, bugfixes do not affect binary compatibility.

However we think that this approach is incorrect. First of all, the part of L and the notion about it is anyway inlined into the application's executable (as a side effect it could even reduce the number of compatible libraries in the other cases, if the alternative definition of compatibility is used). Secondly, as it was said in the intro, an application developer should fix the behavior of the program in a help file of sorts, so the library changes would reduce the separation of application and underlying library.

3.3. Compiled Code Notions Incompatibility

In this section we will look for the such ways of altering the header file, that cause $L'.cpp$'s code, that implements data access accordingly to $L'.h$'s notion, to access the same data that are laid out as in $L.h$. In the other words, we will study the raw memory layout semantics and how to prevent incorrect access to it.

We can separate two different directions:

1) **access to the library's memory from the userspace** through global variables and specifications-compliant operations with them;
2) **the access to the application-allocated memory from the library**; the memory's having been allocated for the library's types declared in $L.h$ through functions in $L'.so$.

In the point 1 we mean direct access to global variables. Indirect access to the memory through interface functions is under the library's control and doesn't cause incompatibility directly (i.e. is studied somewhere else). Just as well, calls to class static members or global variable's members don't cause incompatibility immediately.

All operations considered are equivalent to reading the variable of integral type and to direct writing to it or to complex structure as a whole. When recording the structure, if its copy implementation is deduced by compiler (i.e. the copy constructor is not overloaded) only semantic violation errors my arise. Therefore, ***binary compatible access to variable, that may be accessed from the userspace for reading and writing, is possible only if the alignment of T' coincides with alignment of T on the first $sizeof(T)$ bytes.*** In the other words, you only can add fields to T class, but you can't alter the ones defined in $L.h$.

The analysis of 2 should be more elaborative.

It differs from the point 1 by the capability of library to call its own types with arbitrary class of operations that's broader (at least not more narrow) than what is possible from outside the library. The ultimate principle can be formulated like this: *$L'.cpp$ should be created in the way for it to be able to distinguish the origin of the data being handled, whether it's $L.h$ or $L'.h$.* [10] This approach is equivalent to symbol versioning, which, as shown in section 1.3.4, still requires further study of what can be done without it.

Let's assume that a function takes one value as an argument (member functions are implemented as simple `C`-like functions that take pointer to **`this`** as its first argument), the type (possibly, indirectly, via pointers and references) depending on T type, declared in $L.h$ and $L'.h$. Let's call T' what T became in L'.

If the argument is passed by value, then in binary compatible application $T = T'$, because these types should have equivalent sizes (as the memory in stack for them is allocated via caller) and semantics on first $sizeof(T')$ bytes. Therefore, *only the types that are passed via pointer/reference to the library functions may be altered.*

The rules from 1 are applicable also to the types, that are returned by value from library routines, because caller is responsible for copying values back from stack.

If object of type T is passed by reference into the function, that expects reference to T', it can control the access to memory access within the object.

However, if it's possible for T to be allocated into automatic storage in userspace (that includes being base class or class member), the pointer to it may address less memory than T' denotes; the behavior being undefined upon access to it. Moreover, if a class has an explicit constructor, its call may lead to memory access violation. Therefore, the following rule holds: ***the increase of size of a class that can be allocates in the automatic storage in the userspace causes binary incompatibility if the functionality, that accesses the new memory, is reachable keeping the conformance to „old" library specifications***. Note that new members may not extend class size; that's especially notable for bitfields (see [8] for bitfields as a technique to maintain compatibility).

Note also, that class size may be increased not only with new members, but with new bases as well. If such new class requires more memory (it may not for a nonempty, but relatively small class; however it may require for an empty class as well; refer to [4]), then new memory is laid out

9. The known example nowadays is a bug in `Qt` 4.4 fixed in version 4.5, but many `KDE` 4 applications contained a workaround and become incorrect

10. as an example, one may require to explicitly specify the version of library used in the application by the special function call or global variable or something else.

before any of the members, all data members shifting. For a class with at least one accessible data field that causes incompatibility.

Therefore, practically, the paragraph above means that *adding a new base that increases the size of the class leads to binary incompatibility*.

There are several techniques that allow adding functionality to the class maintaining binary compatibility: „d-pointer", described in [8], techniques that emulate interpreted languages elements (see [8], „Adding new data members to classes without d-pointer"). One can avoid problems by disallowing to allocate memory in automatic storage in userspace, hence forbidding to derive the class, but that undermines the basis of OOP and has limited use.

The conclusion follows: ***The change of class hierarchy (except cases when it involves the change of size of no bases), change of members' order, size and increase (change, in case the class can be passed to or returned from library function by value) of their amount leads to binary incompatibility***.

To apply this rule into practice you might want to experiment with size of your structures, but that's hardly will be useful. Empty bases (and these are nearly all bases that don't make class change) are best served virtual and the restrictions on virtual bases are more strong, what you will see in section 3.3.

3.4. Auxiliary Symbols Change

Along with symbols described in 3.1, `GNU` C++ compiler adds auxiliary symbols to binary level. They are used in virtual functions call algorithm (vtables), expose support for low-level inheritance-related code generation (VTT, several variants of constructors and destructors) and several thunks.

This section investigates the class of $L.h$ alterations that don't cause binary incompatibility through auxiliary symbols alterations. The key problem is that part of C++ low-level code support mechanisms are generated in compile-time and the symbols are assumed to be laid out as in $L.h$. During the dynamic linking the application references these symbols in the way described in $L'.h$, what causes incorrect program behavior. The bodies of these symbols also can change, but part of these changes, namely $L.cpp \rightarrow L'.cpp$, has already been studied in 3.1 and 3.2; we will elaborate the other part here.

Let's study how each symbol type influences the compatibility.

3.4.1. Symbols Introduced via Functions.

- **Constructors and destructors** cause compiler to emit symbols for compete object constructor with and without memory allocation and for base object constructor (for construction of non-static members and non-virtual bases); the same symbols are created for destructor (but with freeing the memory instead of its allocation). These symbols should have played an important role in binary compatibility, but unfortunately it's not always possible to encapsulate memory allocation procedure in the shared library. Therefore the possibility of compatibility violation is restricted to matters discussed in 3.3, and to that, upon withdrawing of all virtual bases (which causes incompatibility, as we will see in 3.4.2), relevant symbols also disappear from library, causing link-time error.
- **thunks**. Thunks are described in section 3.1. Here we will assume that set of thunks didn't change; only actual offsets could.

  Part of these values, that concern **`this`** adjustment, are described in 3.2.3 section of [4]. It clearly shows that adjustment is done with different offset values, that, as described in section 5.1.4 of that standard, are encoded into external names of thunks, examples shown on figure 4.

  According to comments in `gcc` code[11], entry point body only depends on these values, on whether covariant type presents and on external name of the function the thunk is associated to.

  In the other words for thunks with same names equivalent bodies are emitted (limited to the possible discrepancy in the actual function's body).

  As any shift of classes through the hierarchy is incompatible (see 3.3), *thunk bodies will coincide in $L.so$ and $L'.so$ if the other binary compatibility conditions are held.*

Table 4. Thunk names

`_ZThn8_N7Derived3virEv`	non-virtual
`_ZTv0_n48_N7Derived3virEv`	virtual
`_ZTvn8_n48_N7Derived3virEv`	virtual+vbase offset

3.4.2. Vtables. In this section we will use concept of *class hierarchy*, the tree, that depicts the class' direct bases, then their bases and so on, the edged representing direct derivation. Some rules will also be formulated in terms of class hierarchy. The developer should remember that ***when the hierarchy of C is alternated, hierarchies of some classes that derive C may also change*** (and most of them will, unless the change is adding a new virtual base that already presents in all classes derived from C before it in preorder). Practically that means that without additional internal requirements to inheritance (we've just outlined one possible rule in parentheses) these rules are useless and it's best to prototype and check them, or apply to classes which derivations developer can control.

Vtables is a structure that supports virtual function call algorithm, virtual base access and RTTI for **`dynamic_cast`**. In C++ all static types are known in compile time, therefore virtual tables are referenced through pointers, each for every primary base group. Every pointer references some data placed in the translation unit, where first virtual function body is emitted.

Vtable structure is fully described in [4], section 2.5. We will only give some general information.

11. check the description in files `gcc/cp/cp-tree.h`, line 3317, and `gcc/cp/method.c`, `make_thunk(); function.`

Table 5. Vtable group layout example

Classes				Entry	Offset
C				vcall offset for $B_2 :: f_3$	-72
C				vbase offset for D_1	-64
C				vbase offset for B_1	-56
C	B_3			vcall offset for $B_2 :: f_3$	-40
C	B_3			vbase offset for B_1	-32
C	B_3	B_2		vbase offset for B_1	-24
C	B_3	B_2	B_1	offset-to-top (zero)	-16
C	B_3	B_2	B_1	RTTI (of C)	-8
C	B_3	B_2	B_1	$B_1 :: f_1$	0
C	B_3	B_2	B_1	$B_1 :: f_2$	8
C	B_3	B_2	B_1	$B_2 :: f_3$	16
C	B_3	B_2		$B_2 :: g_1$	24
C	B_3	B_2		$B_2 :: g_2$	32
C	B_3	B_2		$B_2 :: f_3$	40
C	B_3			$B_3 :: h_1$	48
C	B_3			$B_3 :: h_2$	56
C	B_3			$B_3 :: h_3$	64
C		D_2		vbase offset for D_1	-24
C		D_2	D_1	offset-to-top (nonzero)	-16
C		D_2	D_1	RTTI (of C)	-8
C		D_2	D_1	$D_1 :: g_1$	0
C		D_2	D_1	$D_1 :: g_2$	8
C		D_2		$D_2 :: h_1$	16
C		D_2		$D_2 :: h_2$	24

Let's consider vtable group $C.\mathcal{V}$ of C class. They're laid out consequently, in the same order the base classes are placed in C's body. Every vtable $V \in C.\mathcal{V}$ is aligned around the point of origin referenced by $C.ptrto(V)$; it relates to the primary base group $B_1 \hookrightarrow B_2 \hookrightarrow \dots B_n$. Immediately before the zero, with negative offset, RTTI pointer and $C - C.ptrto(V)$ offset (*offset-to-top*) are places. Then, for $i := 1..n$, are appended the pointers to final virtual function overriders of the functions first introduces into B_i. To the beginning, at negative offset, if $B_i.\mathcal{B}^V \neq \emptyset$, offsets $B_i.\mathcal{B}^V_j - C.ptrto(V)$ (vbase offsets) are appended; the less i is, the closer to point of origin offset's placed. Then, for each virtual function, declared in base S of $B_i.\mathcal{B}_j$, such that $\forall D : K \hookrightarrow D \hookrightarrow B_i.\mathcal{B}_j \Rightarrow D \notin \mathcal{V}$, and that it's finally overridden in K_k, the offsets $B_i.\mathcal{B}^V_j - K_k$ (vcall offsets) are appended in the same way as vbase offsets, but after them in the „negative" direction.

So the information about primary bases is „sliced" so they share vtable and vtable for base class is, at the memory such vtable is allowed to access, coincides with vtable as if it was allocated separately. See the example of vtable layout on figure 5 (the hierarchy is B_1 is a primary virtual base of B_2, which is primary base of B_3, which is primary virtual base of C, which also derives D_2, which has a primary virtual base D_1; virtual function f_3 is defined in B_1 and overloaded in B_2).

We can see, that pointers to functions, offsets and data pointers present in the vtable. It's irrelevant if they all have different sizes because they can't intermix due to carefully elaborated vtable layout. However as vtable group is referenced by only one symbols, the size of each can't change, as offsets from the first vtable to any other$C.\mathcal{V}_i$ are precompiled in $f.exe$.

Hence $C.\mathcal{V}$ depends on mutual location of virtual base groups, on the order of classes within these groups and on their virtual functions.

Therefore an only $C.\mathcal{V}$ change possible is to add new vtables or extend the last one in the group ($C.\mathcal{V}_{|C.\mathcal{V}|}$). However, if that's the table of V such that $V \in C.\mathcal{B}^V$, developer can't add virtual functions overridden in derived classes as it would ass vcall offset shifting table the shift to which is precompiled. But unfortunately the function will most likely be added to virtual base as their vtables are places at the end of $C.\mathcal{V}$.

Furthermore, when you derive the class in the userspace, the derived vtable is constructed like in $L.h$, but the library functions will implement virtual functions call algorithm according to $L'.h$ notion. So, ***extension of vtable is impossible without compatibility loss, if extending the table functions or classes are used in userspace code.*** We will assume that this rule holds. Let's study the ways of table extending.

1) **New base class**
   The extension may be achieved by increasing the amount of dynamic bases; whenever a class made virtual or a first virtual function is added to one of them. But the new class can't be virtual as it would add vbase offset to the beginning of vtable. As change of class mutual interposition is forbidden as well, only two options remain.
   a) Add class N that will share virtual table $C.\mathcal{V}_{|C.\mathcal{V}|}$ with other classes. This would just add a new „slice" to the vtable. Such a change is only compatible when nothing would be added to the „negative" side of vtable. That's in turn possible only when non-virtual dynamic class is added, the class having no virtual derivant (otherwise virtual functions of N would cause new vbase offsets). Non-virtual class, that doesn't define virtual functions is not dynamic, therefore ***an only compatible way to add a class sharing the last vtable in the group, is adding a nonvirtual base in case when*** $C.\mathcal{B}^V = \emptyset$
   b) Add class N, that yields new vtable in the group. As it's non-virtual class, $C.\mathcal{B}^V = \emptyset$; and N will be added after all dynamic classes in preorder. The overload of virtual functions by other classes will be studied in point 4, and the conclusion will be that it won't cause incompatibility. Therefore, ***non-virtual dynamic class can be added to the end (in preorder) of hierarchy of class without virtual bases***.
2) **Adding a completely new virtual function**
   Let a virtual function be added to subclass B, the virtual function being added not overriding and being overridden by any other function. It can only be added to the very end of vtable group, i.e. to the most derived class of this group. Such an addition can't cause vcall offset only if B is virtual and B doesn't have virtual bases. However, if B does have virtual derivants, $C.\mathcal{B}^V \neq \emptyset$, i.e. B, as most derived class of

$C.\mathcal{V}_{|C.\mathcal{V}|}$ group, is virtual base itself, what proves that new vcall offsets would never be added.
Therefore, ***adding not overloaded and not overloading virtual function to the end of the most derived class of the last group of virtual base doesn't break binary compatibility***.

3) **Withdrawing a virtual function**
The withdrawing of function may retain compatibility if adding of it to the resultant classes causes appending to the very end of virtual table. Of course, the same rules as in 3.1 apply to the function being withdrawn, if the function can be called from the userspace. But in some cases virtual function's symbol isn't referenced directly, so there's no external dependency on it.
Okay, let's assume that the function is called through virtual table. If the class can be derived in userspace, then the new vtable is created for it at compile time and, upon the call of the virtual function in subject through that vtable, it will fail. But then the call will fail in runtime due to absence of the proper symbol.
Therefore an only conclusion possible is that ***withdrawing a virtual function (and making nonvirtual a virtual function[12]) breaks binary compatibility***.
4) **Adding an overloading virtual function**
Let the new function $K :: f$ be such that it overloads a (probably pure) virtual function of some base class. In case of covariant overloading it requires a new vtable entry; point 2 applies in this case. Otherwise it's required to overwrite all pointers to function by replacing them with the relevant entry points; this doesn't cause binary incompatibility. An only condition remaining is for a function not to add new vcall offsets. Therefore, ***adding a virtual function to K class doesn't cause binary incompatibility iff the function doesn't covariantly overload and $\forall V \in C.\mathcal{B}^V \Rightarrow K \notin V.\mathcal{B}$.***
5) **Adding a function becoming overloaded**
Assume a function is added that doesn't overload any other. As this function is added relatively close to vtable's point of origin it can only keep compatibility if its derivants the class is a primary base for do not add more functions to the vtable and if this function doesn't add more vcall offsets (see 4).

3.4.3. VTT. VTT[13] is a structure that keeps pointers to virtual tables during construction. These tables do not possess own symbols; it is only VTT they can be referenced through. Each VTT entry refers to a vtable keeping information about the state of object during construction within larger object. The virtual function pointers will only point to the routines of class constructed so far (in the process of complex hierarchy initialization), its RTTI will be proper and vbase offsets will point to virtual bases allocated as in bigger object. Only the latter is the reason of introducing the new entities compared to the usual vtables; therefor it's only classes with more than one (indirect) virtual base who have VTT assigned.

Where VTT is used is construction code for derived classes. Let's consider a class that can be derived in the userspace code. Taking the conclusions of section 3.4.2 into account, we reduce the analysis to adding non-virtual dynamic classes to the hierarchy of class that doesn't contain any virtual bases. However, no VTT is created for such classes. Therefore *study of VTT doesn't yield any results*.

12. C++ rules state that if function f is declared as virtual in B class, then $\forall C : B \in C.\mathcal{B}$, $C :: f$ is also virtual

13. most likely, it's an abbreviation of „virtual tables table"

4. Conclusion

4.1. How to Keep Compatibility

Let's sum our study up. We have studied enough to formulate the rules the developer is to follow to retain binary compatibility with the old version of shared C++ library in GNU system, assuming that additional constraints in addition to C++ rules apply. We called the code compiled into application *userspace code*; it includes inline functions of library headers and application code itself. The classes that are allowed to be instantiated in userspace code are called *userspace classes*, other classes are *internal*. The functions that can be called from the userspace code are referred to as *userspace functions*.

Let us have L library that we're going to alternate and get L' library, the new version. Then, L' will be binary compatible with L, if all following rules apply:

1) **any userspace function with „external linkage" shall retain its external name** (1.3.4). Therefore, you should keep true arguments type as they appear after all `typedef` substitutions and their number. To learn what changes external name, refer to 3.1.
2) **no userspace function may be removed or made inline, either member or global, virtual or non-virtual; no virtual function may be removed even for internal class**. Refer to sections 3.1 and, for virtual function discussions, to point 3 of section 3.4.2;
3) **no function implementation defined in *cpp* file may be changed in incompatible way**, i.e. if user calls new functions in an old way, that must be plausible and behavior must be the same (section 3.2). A special exception holds for bugfixes, but note, that they may break workarounds;
4) **layout of the first $sizeof(T)$ bytes of types of directly accessible userspace global data must be the same**; this holds for both static class variables and for internal classes (3.3, point 1);
5) **the size of userspace class must be the same** if it has non-inline constructors; if all constructors are inline, you should use symbol versioning of sorts to prevent access to new part of the type layout from the new function;
6) **classes in hierarchy of *all* userspace classes must be the same and in the same order** unless the classes being moved through hierarchy are empty bases of non-dynamic class (but you still need an experiment to ensure that sizes are the same). See 3.3 and 3.4.2;

7) **dynamicity of classes in hierarchy of userspace class must be the same except for userspace class without virtual bases, where you can make non-dynamic class after all dynamic classes in preorder** see (3.3 and point 1 of section 3.4.2);
8) **you can introduce new virtual functions overloading the old ones, except for the case of covariant overloading and overloading of function of a virtual base** (point 4 of section 3.4.2). You should be assured that the call to this function will yield the same results as if it were called in a way allowed by L specifications;
9) **a completely new virtual function may be added to the end of the most derived class if its hierarchy doesn't contain any virtual base** (point 2 of section 3.4.2).

4.2. Conclusion

We may compare the rules deduced by us and summarized in 3.4.3 with the compatibility guide [8] suggested by the KDE developers and known to be most complete.

We see that our formal approach didn't yield more results than the developers deduced from the practical experience. It appears that requirements for binary compatibility can't be relaxed due to additional restrictions on the library use except for a limited number of cases. Namely, there's more freedom for internal classes, that can't be instantiated in automatic memory or derived by user. However, such classes can't be considered as exhaustively using C++'s OOP flavors, although implement quite a popular „singleton" concept (see [9]).

Therefore we conclude that the current C++ ABI is incapable to provide more compatibility even with additional restrictions upon the use of C++ constructs provided by library's headers.

We should also note that the current `gcc` ABI is influenced by the desire to keep away from inserting elements of interpreted languages into ABI and by „incremental" way of binary representation (the architecture when the most common cases—single inheritance and simple virtual functions—induce more simple and fast binary representation). As a result, complex concepts are both considered unsafe and their uncareful use causes incompatibility.

Perhaps, the other ABI model would better fit compatibility aims, but this question needs further research to discover the best balance between maintainibility and performance, and it also needs careful cost estimation, as the benifits of its change should be greater than the expenses required to adopt it.

References

[1] *TIOBE Programming Community Index for January 2009*. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html;

[2] Bjarne Stroustrup. *A History of C++: 1979-1991*. History of Programming Languages conference, 1993;

[3] ISO/IEC 14882:2003. *Programming languages — C++*

[4] "informal industry coalition consisting of (in alphabetical order) CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI". *Itanium C++ ABI (Revision: 1.86)*

[5] Urlich Drepper. *How To Write Shared Libraries*. 2006.

[6] B. Guptill, B. McNee. *Booming Support for Mission-Critical Application Workloads on Linux*. http://research.saugatech.com/fr/researchalerts/304RA.pdf

[7] IDC. *Open Source in Global Software: Market Impact, Disruption, and Business Models*. http://www.idc.com/getdoc.jsp?containerId=202511

[8] *Binary Compatibility Issues With C++*. http://techbase.kde.org/Policies/Binary_Compatibility_Issues_With_C%2B%2B

[9] E. Gamma et. all *Design Patterns: Elements of Reusable Object-Oriented Software*

[10] *Using the GNU Compiler Collection (GCC)*, chapter 8. http://gcc.gnu.org/onlinedocs/gcc/Compatibility.html