Binary compatibility for library developers

Thiago Macieira, Qt Core Maintainer

LinuxCon North America,
New Orleans, Sept. 2013

Who am I?

- **Open Source developer for 15 years**
- **C++ developer for 13 years**
- **Software Architect at Intel's Open Source Technology Center (OTC)**
- **Maintainer of two modules in the Qt Project**
  - QtCore and QtDBus
- **MBA and double degree in Engineering**
- **Previously, led the "Qt Open Governance" project**

Definitions

- Binary compatibility
- Source compatibility
- Behaviour compatibility
- Bug compatibility

Binary compatibility

Two libraries are binary compatible with each other if:

- Programs compiled against one will load and run *correctly** against the other

* by some definition of "correct"

Source compatibility

Two libraries are source compatible with each other if:

- Source code written against one will compile without changes against the other

Behaviour and bug compatibility

Two libraries are behaviour-compatible with each other if:

- The program will exhibit the same behaviour with either library

Two libraries are bug-compatible with each other if:

- Expanded version of behaviour compatibility to include buggy behaviour

Forwards and backwards

Depends on the point of view

- Backwards compatibility:
  newer version retains compatibility with older version
  - You can **upgrade** the library
- Forwards compatibility:
  older version “foreshadows” compatibility with newer version
  - You can **downgrade** the library

This presentation focuses on

- **Backwards binary** compatibility

- This depends on the ABI
  - Will focus on the System V ELF ABI for Linux and IA-64 C++ ABI

Why you should care

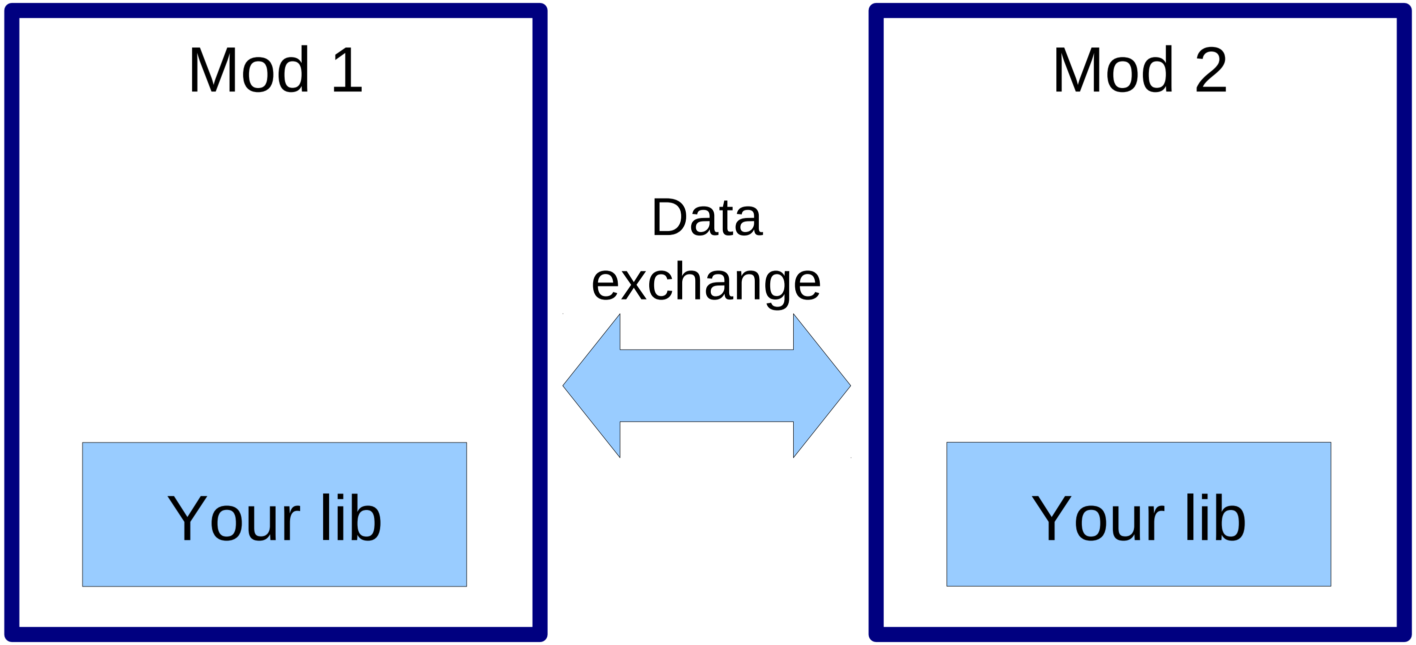
Library used by libraries

- They expose your API in their API
- Their users might want to use a newer version of your library

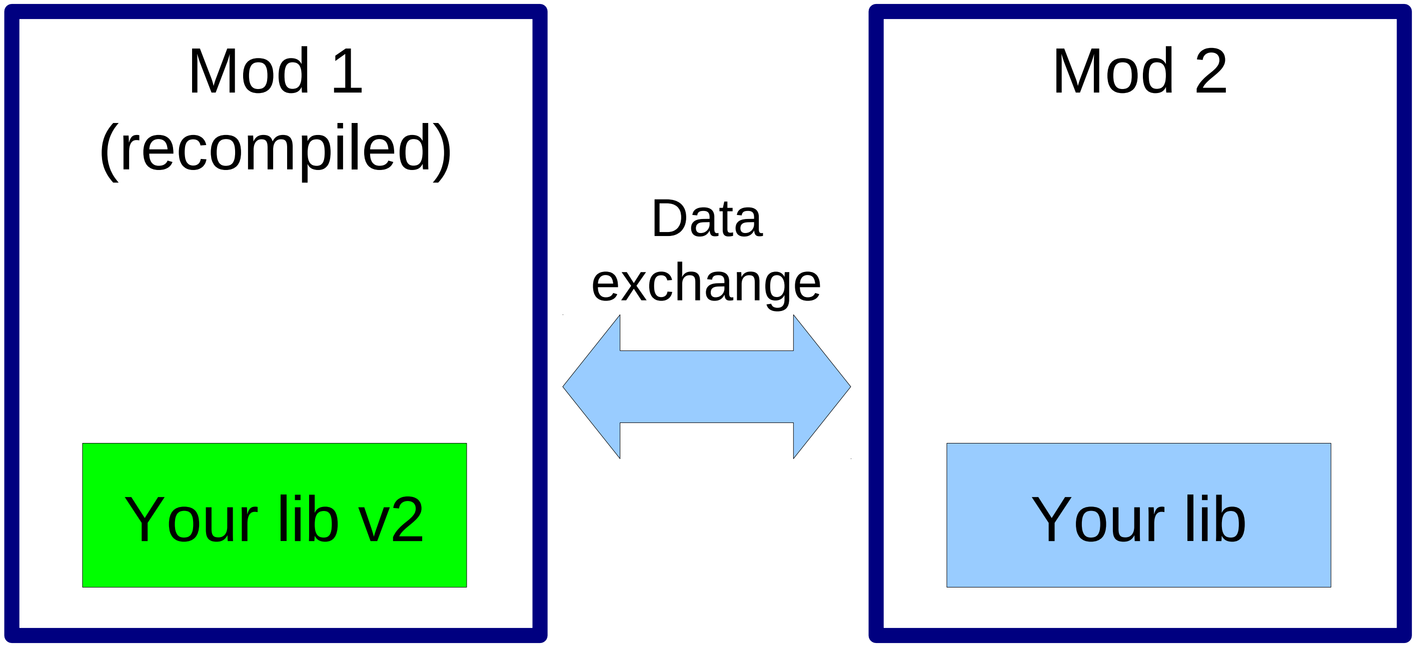
Library used by anything

- Upgrading parts of the system
- Large, complex project

Project with 2 modules: initial state

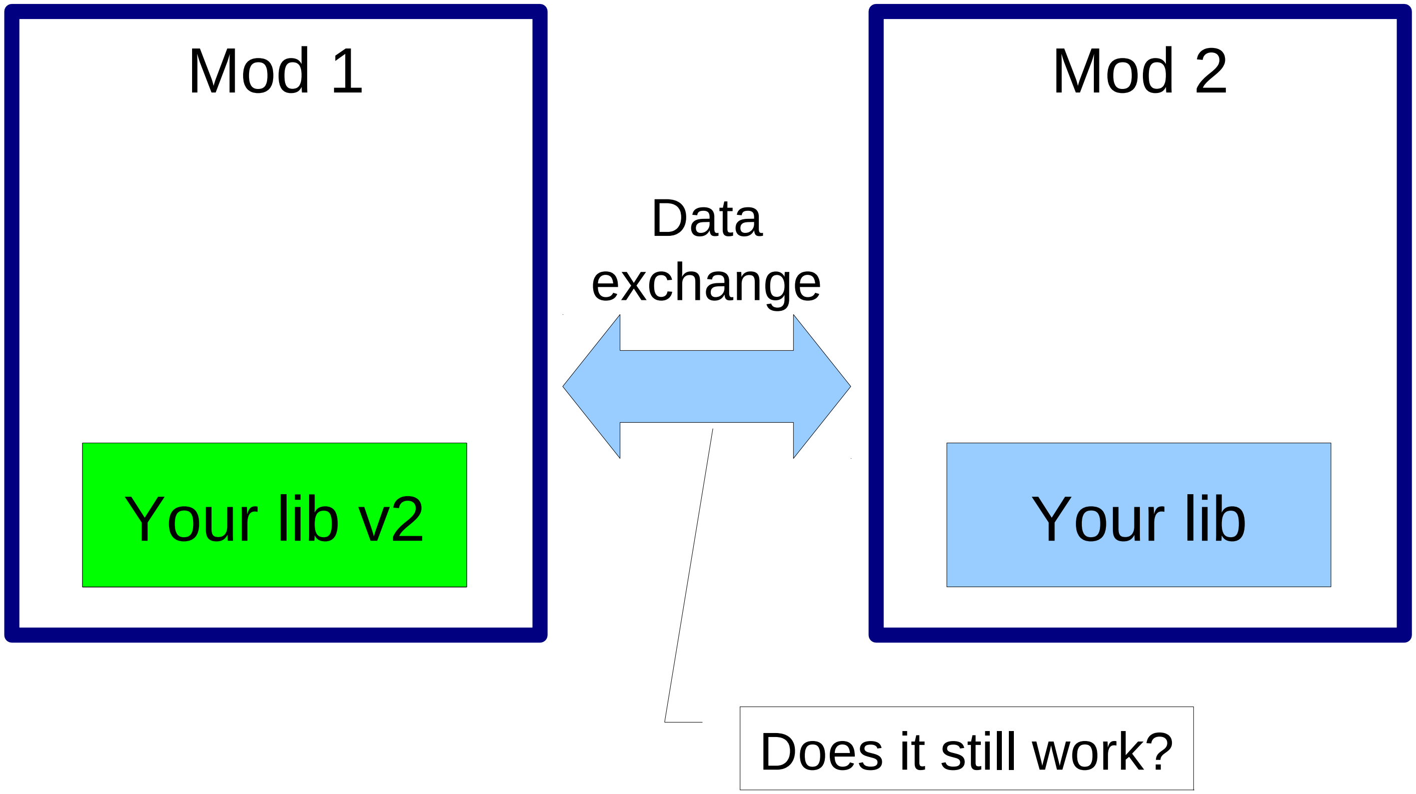


Lib is upgraded in one module



Does this still load?

Co-existing libraries



If you're developing an application...

- This does not apply to you

- Except if the application has plugins
- Or if it has independent modules

The application has libraries

The details

Binary compatibility requires...

- No public[1] symbol be removed
- All public[1] functions retain retain their properties
  - Which arguments are passed in registers, which are passed on the stack, implicit arguments, argument count, etc.
- All public[1] structures retain their layout and properties
  - For both C and C++ aggregates: sizeof, alignof, order & type of publicly-accessible members, etc.
  - For C++ aggregates: *dsize*, *nvsize*, PODness, etc.

1) Symbols intentionally made public as part of the API plus private symbols used in inline functions

Example: simple C library (C++ comes later)

```c
/* lib-header.h. */
struct Data {
    int i;
    int j;
};

extern struct Data global_data;
void function(struct Data *data);

/* lib-source.c */

struct Data global_data = { 1 };
void function(struct Data *data)
{
}
```

```
$ gcc -c /tmp/lib-source.c
$ nm lib-source.o
00000000 T function
00000000 D global_data
```

Could be "B" too

No public symbol is removed

- Easy to do
- Do not remove any variables or functions that exist
- Do not change any variable or function in a way that would cause its external (mangled) name to change

- In our example, we cannot:
  - **Remove** either the function “`function`” or the variable “`global_data`”

All functions retain their properties

- The C++ language helps you
  - This requirement is mostly fulfilled by the previous and next requirements
  - If the data types retain their properties
  - And if the mangled name of a function is retained
  - The function retains its properties
- In C, it's possible to change the arguments without changing the external symbol

- In our example:
  - The function "`function`" must not read more than 1 argument of integral type

All data types retain their properties

- Can be automated with a C or C++ parser and the compiler
- Best avoided:
  - Use opaque types / d-pointers / private implementation
- Examples:
  - Change alignment → user's structure could add or remove padding
  - Change non-padded size → the compiler is allowed to use tail-padding
  - (C++) Make non-POD → user's structure becomes non-POD too
- In our example, we cannot:
  - **Reorder** the members in the struct
  - **Remove** the members
  - Place the members in a union with a *long long* (changes the alignment)

And then there's C++

Life gets more complicated

- External names for all[1] functions are mangled
- External names for all[1] variables in namespaces are mangled
  - In some other ABIs, even those in the base namespace
- Non-POD (Plain Old Data) aggregates have more rules

But we gain too:

- Functions can be overloaded
- Mangled names help in ensuring binary compatibility for functions

1) all except those declared with `extern “C”`

C++ mangled names

IA-64 C++ ABI

- Prefixed by _Z
- Case sensitive
- Doesn’t mangle free variables
- Mangles only what is required for overloads that can co-exist

Microsoft Visual Studio

- Prefixed by question mark (?)
- Case insensitive
- Mangles free variables
- Mangles **everything**, including:
  - Return type
  - Struct vs class
  - Public, protected, private
  - Near, far, 64-bit pointers
  - cv-qualifiers

Example: our library in C++

```
/* lib-header.h. */
struct Data {
    int i;
    int j;
};

extern struct Data global_data;
void function(struct Data *data);

/* lib-source.cpp */

struct Data global_data = { 1 };
void function(struct Data *data)
{
}
```

```
$ g++ -c /tmp/lib-source.cpp
$ nm lib-source.o
00000000 T _Z8functionP4Data
00000000 D global_data
```

Note the presence of "Data"

What works

Guidelines

- Don’t expose what you don’t need
- Be conservative in what you change
  - Follow the “Binary Compatibility with C++”[1] guidebook
- Use automated test tools

[1] http://techbase.kde.org/Policies/Binary_Compatibility_Issues_With_C++

Minimal exported API

- Design a minimal API
  - If you're unsure about something, don't include it (yet)
  - Limit exports by using ELF symbol visibility:

```
-fvisibility=hidden -fvisibility-inlines-hidden
__attribute__((visibility(“default”)))
```

- Use opaque or simple types
  - Private implementation, d-pointers
- Use an API based on functions
  - Avoid exported variables
  - Avoid returning pointers or C++ references to internal variables

Why functions and private implementations?

In C

- Use opaque types

```
typedef struct _GDir GDir;
```

Forbidden!

- Can't be constructed by the user – always passed by pointer
- Free to be changed at will

In C++

- Use private implementation
- Your public types won't change much or at all
  - Lowers the risk of changing the type's properties
- You can freely change the private implementation
- Adding new functions is easier than modifying the public type

C99 7.1.3p1: "All identifiers that begin with an underscore and either an uppercase letter or another underscore are always reserved for any use"

Changing non-virtual functions[1] (C and C++)

You can

- Add a new function
- De-inline an existing function
  - If it's acceptable that the old copy be run
- Remove a private function
  - If it has never been called in an inline function, ever
- (C++) Change default parameters

You cannot

- Unexport or remove public functions
- Inline an existing function
- (C) Change the parameter so it would be passed differently
- (C++) Change its signature:
  - Change or add parameters
  - Change cv-qualifier
  - Change access rights
  - Change return type

1) includes all C functions as well as C++ functions with `extern "C"`

Changing virtual functions (C++ only)

You can

- Override an existing virtual
  - Only from primary, non-virtual base
- Add a new virtual to a leaf (final) class

You cannot

- Add or remove a virtual to a non-final class
- Change the order of the declarations
- Add a virtual to a class that had none

“Anchoring” the virtual table (C++ only)

- Make sure there’s one **non-inline** virtual
  - Preferably the destructor
- Avoid virtuals in template classes

Changing non-static members in aggregates (C and C++)

You can

- Rename private members[1]
- Repurpose private members[2]
- Add new members to the end, provided the struct is:
  - POD (C++98 and all C structs)
  - Standard-layout (C++11)
- and:
  - (C++) The constructor is private; OR
  - The struct has a member containing its size

You cannot

- Reorder public members in any way
- Remove members

You should not

- (C++) Change member access privileges
- (C++) Add a reference or const or non-POD member to a struct without one

1) provided the class has no friends; 2) provided old inline functions still work

Testing compliance

- Run automated tests frequently
- Run full tests at least once before the release
- On Windows: use the exports file
- On Unix: use nm, otool (Mac), readelf (ELF systems)
- GCC: use -fdump-class-hierarchy
- Everywhere: use the Linux Foundation's ABI Compliance Checker[1]
  - Confirmed to run on Mac, Windows and FreeBSD

[1] http://ispras.linuxbase.org/index.php/ABI_compliance_checker

Manual checking before release

- Do a “header diff”

- `git diff --diff-filter=M` *`oldtag`* `-- \*.h`
  - Manually exclude headers that aren’t installed
  - Or obtain the list of installed headers from your buildsystem

Be careful with false positives

- You probably want a white and black list
- White-list your library’s own API
- Black-list “leaked” symbols from other libraries
  - Inlines and “unanchored” virtual tables

Further: experimental API

- Don’t do it like ICU
- Place it in a separate library
- In fact, place it in a separate source release
  - Keeps Linux distributions happy

Further: breaking binary compatibility

- Announce in well in advance
- Keep previous version maintained for longer than usual
- Try to keep **source** compatibility
- Change your library names (ELF soname)

Resources

- Binary compatibility guide in KDE Techbase:
  - http://techbase.kde.org/Policies/Binary_Compatibility_Issues_With_C++
  - Examples: http://techbase.kde.org/Policies/Binary_Compatibility_Examples
- Calling convention article (includes MSVC, Sun CC):
  - http://www.agner.org/optimize/calling_conventions.pdf
- IA-64 / Cross-platform C++ ABI:
  - http://mentorembedded.github.io/cxx-abi/abi.html
  - http://refspecs.linux-foundation.org/cxxabi-1.86.html
- “How to Write Shared Libraries”, by Ulrich Drepper
  - http://www.akkadia.org/drepper/dsohowto.pdf
- libabc
  - https://git.kernel.org/cgit/linux/kernel/git/kay/libabc.git

Thiago Macieira

thiago.macieira@intel.com

INTEL OPEN SOURCE
TECHNOLOGY CENTER