Code less.
Create more.
Deploy everywhere.

Binary compatibility for C++ library developers

Thiago Macieira, Qt Core Maintainer

Aspen, May 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**

Qt

One major release in 7½ years

- Qt 5.0 was released in December 2012
- Qt 4.0 was released in June 2005
  - Eight feature releases: 4.1, 4.2, ..., 4.8
  - 32 patch releases: 4.0.1, 4.1.1-4.1.5, 4.2.1-4.2.3, 4.3.1-4.3.5, 4.4.1-4.4.3, 4.5.1-4.5.3, 4.6.1-4.6.4, 4.7.1-4.7.4, 4.8.1-4.8.4
  - Latest release (Nov/2012) still is **binary compatible** with Qt 4.0.0

libstdc++.so.6 (libstdc++-v3)

One major release in 9 years

- Last major release:

  GCC 3.4.0 on April 20, 2004

- 3607 symbols as of GCC 4.8.2
  - 1830 non-weak symbols (51%)
  - 44% the count of QtCore 4.8.x
  - 4.5% the count of Qt 4.8.x (excluding QtWebKit)

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
  - Totally outside the C++ Standard rules and guarantees

Why you should care

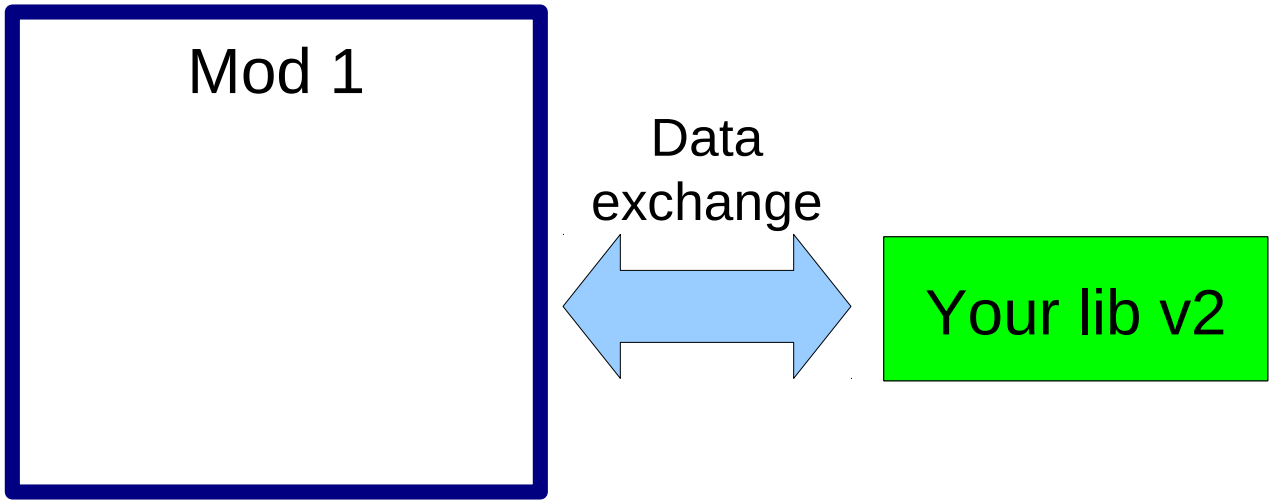
Library used by libraries

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

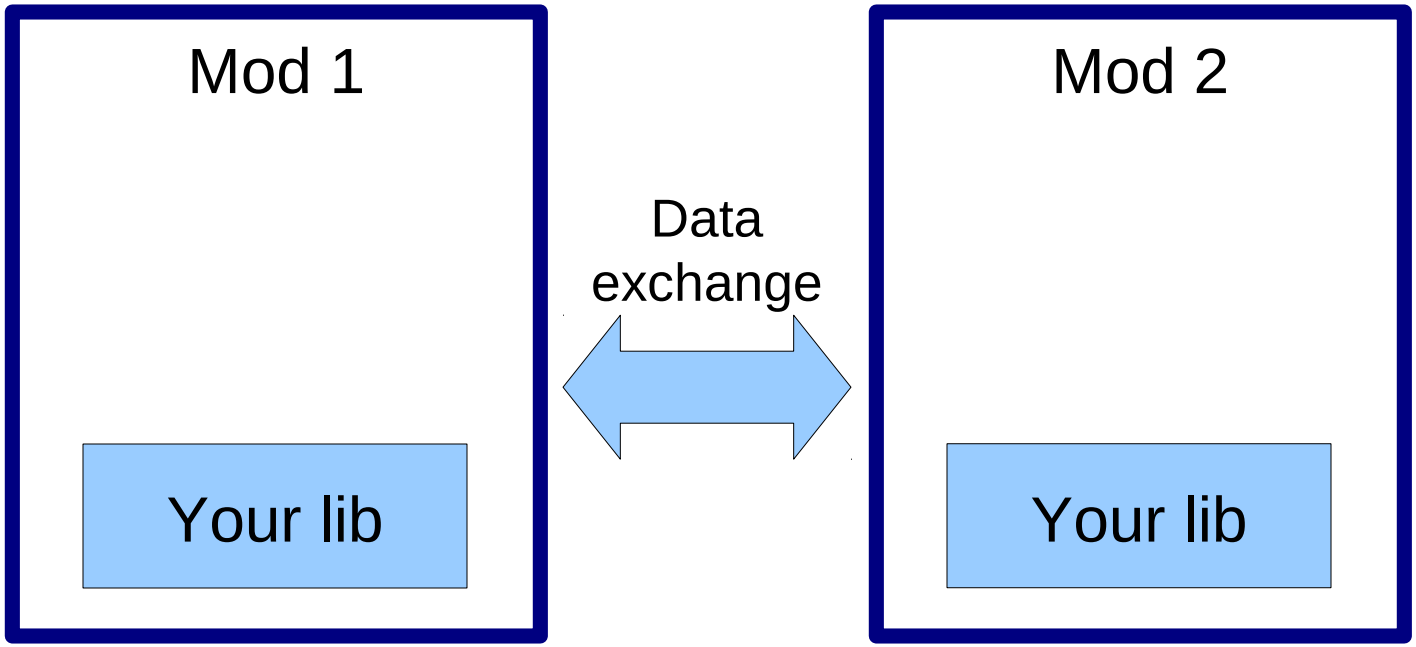
Library used by anything

- Upgrading parts of the system
- Large, complex project

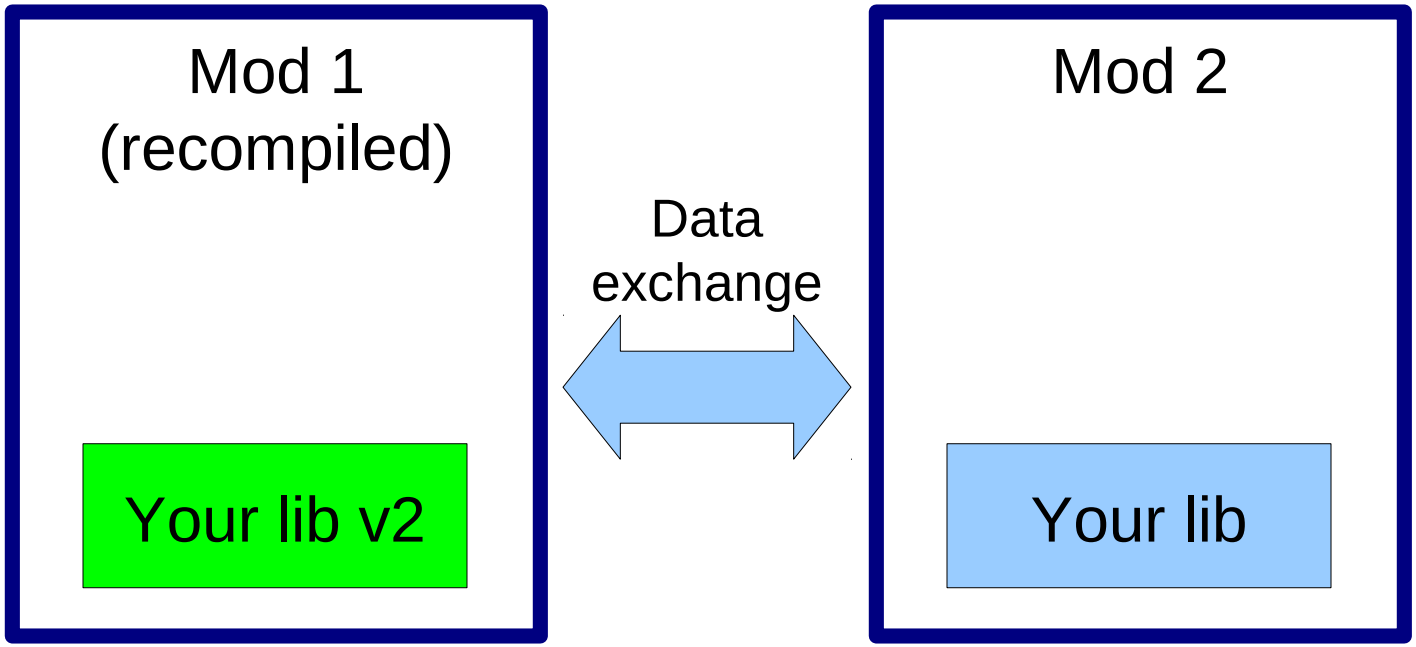
Dynamically linking and system upgrade



Project with 2 modules, statically linked: initial state

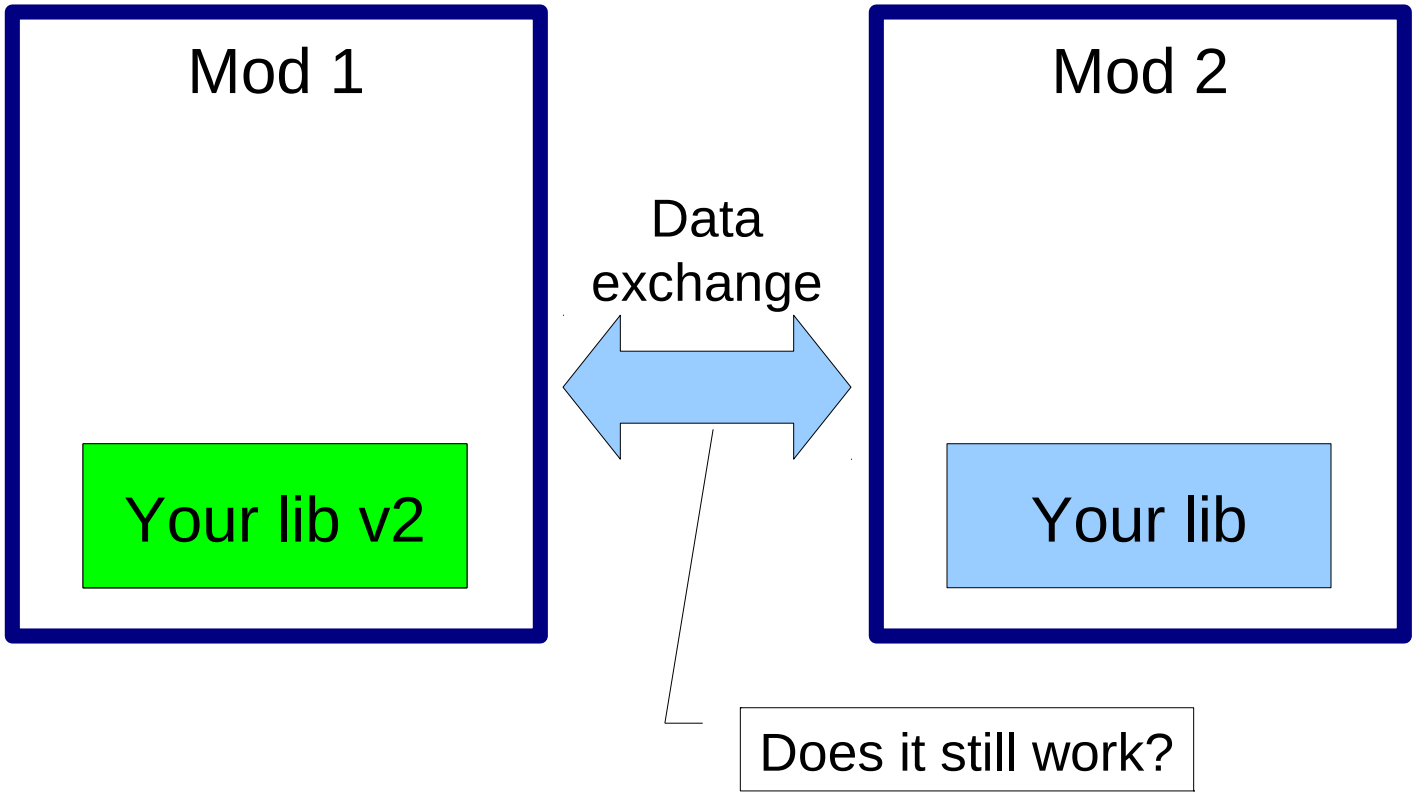


Lib is upgraded in one module



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
  - sizeof, alignof, *dsize*, *nvsize*, PODness, order & type of publicly-accessible members, etc.

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

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

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

All data types retain their properties

- Can be automated with a 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
  - Make non-POD → user's structure becomes non-POD too

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

What doesn’t work

Declaring that no BC guarantees are provided

- It’s ignoring the problem...
- Prevents your library be used in the contexts I’ve mentioned

Don’t change anything, only add new classes

- It’s a solution...
- But it means never providing new features
  - Or even bug fixes

Rename all symbols with macros

- Example: ICU
- Requires renaming the library too (ELF soname)
  - An OS solution
- Potentially loads more than one library into memory
  - Increased memory usage
- Does not solve the data exchange problem
- Does not solve the user's ABI problem
  - Your mangled names are incorporated into their mangled names

Rename symbols with inline namespaces

- Examples: Boost.Filesystem, LLVM's libc++
- Might not require new library
  - If both old and new symbols are present (Boost.Filesystem)
- Does not solve the data exchange problem
- Does not solve the user's ABI problem

Rename the library but not the symbols

- Example: Boost.Regex (regex vs regex-mt)
- Potentially loads more than one library into memory
  - Might crashes at load- or run-time

Experimental, public symbols in the same library

- Seems nice to provide your users with experimental features
- Linux distributions will not like you
- It means experimental symbols are **off-limits** (private)

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 ELF or Mach-O visibility or __declspec(dllexport)
- Use opaque or simple types
  - Private implementation, d-pointers
- Use an API based on functions
  - Avoid exported variables
  - Avoid returning pointers or references to internal variables

Why private implementations and functions?

- 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 member variables

Changing non-virtual functions (static or not)

You can

- Add a new function
- De-inline an existing function
  - If it's acceptable that the old copy be run
- Change default parameters
- Remove a private function
  - It cannot have been called in an inline function, ever

You cannot

- Unexport or remove public functions
- Inline an existing function
- Change its signature:
  - Change or add parameters
  - Change cv-qualifier
  - Change access rights
  - Change return type

Changing virtual functions

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

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

Changing non-static member data

You can

- Rename private members[1]
- Repurpose private members[2]
- Add new members to the end, provided the struct is std-layout and:
  - The constructor is private; OR
  - The struct has a member containing its size

You cannot

- Reorder members in any way
- Remove members

You should not

- Change member access privileges
- 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

- Place it in a separate library
- In fact, place it in a separate source release

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 (for Qt and KDE):
  - 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


TURE INTEL LINUX WIRELESS GUPNP KVM POKY
OP OFONO LINUX KERNEL
CS YOCTO CONNMAN XEN
SYNCEVOLUTION SIMPLE FIRMWARE INTERFACE (SFI) ENTERPRISE SECURITY INFRASTRUCTURE
INTEL OPEN SOURCE
TECHNOLOGY CENTER