100 BC

Meeting C++ 2025-11-06

Marc Mutz <marc.mutz@qt.io>

About Marc Mutz

- Principal Software Engineer at The Qt Company
- For the purposes of this talk:
  - The guy that annoys other Qt devs in API / ABI reviews
  - In particular, not an expert in this topic
    - just more experienced than most devs in these matters

About This Talk

- If you need definite answers, talk to your vendor
- Out of Scope are:
  - C++20 Modules
  - Linker Scripts
  - Static Linking
  - Combining different compiler-vendors, -versions, or -flags

Overview

- Motivating Example
- Definitions
- A look at the KDE BC Page
- Techniques
- Cheating
- Q & A

Example

- Lib v1.0

  ```
  int scan(int fd);
  ```

- Lib v1.1:

  ```
  enum class Options { ~~~ };
  int scan(int fd, Options opt = {});
  ```

Example

- Lib v1.1:

```
enum class Options { ~~~ };
int scan(int fd, Options opt = {}); // BiC
```

- Lib v1.1p1:

```
int scan(int fd);
int scan(int fd, Options opt);
```

Example

- Lib v1.1p2:

```
#if LIB_REMOVED_SINCE(1, 1)
int scan(int fd);
#endif
int scan(int fd, Options opt = {});
```

Definitions

Definitions

- Qt / KDE
- BC / SC
- ABI / API
- forwards / backwards xC
- name mangling / exported symbols

Definitions – Qt and KDE

- Qt: A cross-platform application development framework written in C++
- KDE: A (Unix) Desktop Environment based on Qt
- Both promise BC and SC within major release cycles

Definitions — BC

“A library is binary compatible, if a program linked dynamically to a former version of the library continues running with newer versions of the library without the need to recompile.”

— KDE BC wiki

Definitions — BC

"A library is binary compatible, if a program linked dynamically to a former version of the library continues running with newer versions of the library without the need to recompile."

— KDE BC wiki

Definitions — BC

"A library is binary compatible, if a program linked dynamically to a former version of the library continues running with newer versions of the library without the need to recompile."

— KDE BC wiki

Definitions — BC

"A library is binary compatible, if a program linked dynamically to a former version of the library continues running with newer versions of the library without the need to recompile."

— KDE BC wiki

Definitions — BC

"A library is binary compatible, if a program linked dynamically to a former version of the library continues running with newer versions of the library without the need to recompile."

— KDE BC wiki

Definitions — BC

“A library is binary compatible, if a program linked dynamically to a former version of the library continues running with newer versions of the library without the need to recompile.”

— KDE BC wiki

Observations — BC

- BC is not defined (by KDE) for static linking
  - Doesn’t stop people from trying, though
- “Continues running” is vague
  - “Still links” is just one (necessary) condition

    - Not sufficient. Also need “behaviour compat”.
      - Even a bugfix may break programs

Observations — BC

- You don’t need a library to talk about BC
  - A type can be BC
  - A function can be BC
- A library is just the unit of shipment
  - BC is a property of smaller units

Definition — SC

"If a program needs to be recompiled to run with a new version of a library but doesn't require any further modifications, the library is *source compatible*."

— KDE BC wiki

Observations — SC

- Inline library code can be SiC, but usually not BiC
  - Compiled into the library user
    - Doesn’t change when the library changes
- BC != SC
  - Adding a defaulted argument to a function: SC && !BC
  - Adding a new overload of a function: BC && !SC

Definitions — ABI

- Application Binary Interface
  - “what the linker sees”
  - “below C++”
  - “mangled names”
  - “exported symbols”

Definitions — API

- Application Programming Interface
  - “what the compiler sees”
  - “defined by C++”
  - “overloading” / “default arguments”
  - “typedef”

Definitions — backwards xC

- “Normal” compat:
  - Code compiled against *old* library running against *new* library
- Qt provides backwards BC / SC[1] in major releases
  - 5.0 → 5.15
  - 6.0 → 6.???

[1] see QUIP-6 for acceptable SC breaks

Definitions — forwards xC

- “Reversed” compat:
  - Code compiled against *new* library running against *old* library
- Qt provides forwards BC / SC in minor releases
  - 5.15.0 ← 5.15.19
  - 6.8.0 ← 6.8.???

Definitions — Name Mangling

- Creates unique names for overloaded functions
- Encodes things you can overload on:
  - Name
    - incl. any scopes (“FQN”)
  - signature (= number and type of arguments)
    - Incl. template arguments
    - Incl. cv- and l/rvalue qualifiers of member functions

Definitions — Name Mangling

- Does *not* encode:
  - Argument names and defaults, if any
  - Argument top-level cv-qualifiers (except SunCC in its days)
  - noexcept
  - Access specifier (except MSVC)
  - Return type (except MSVC)

Definitions — Name Mangling

Conclusion:

- Change to name mangling inputs = ABI break
- Change to non-inputs = BC
- Platform differences!

Definitions — Exported Symbols: C++

- extern linkage
  - other TUs can access
    - needs declaration
  - duplicates not allowed
- static linkage
  - only this TU can access
  - duplicates remain
- inline linkage
  - other TUs can access
    - needs definition
  - duplicates are merged

Definitions — Exported Symbols: DSO

- Unix: callable from outside .so/.dylib by default
  - inline (code copied into caller)
  - extern (library code called)
- Windows: callable from outside .dll by default
  - inline (code copied into caller)

Definitions — Exported Symbols

- Exporting is reducing (Unix) / extending (Win) the set of symbols callable from outside the DSO
  - Windows: __declspec(dll{ex,im}port)
  - Unix: -fvisibility=hidden + __attribute__((visibility("default")))

Definitions — Exported Symbols

- These attributes need to differ between building the library and building against the library
- Hidden behind macros
- Qt: QT_(MODULE)_EXPORT
  - generated by CMake

Definitions — Exported Symbols

- Types don’t need exporting (are not symbols)
  - “Exported Class” is just “export all members”
    - class QT_CORE_EXPORT QObject { ~~~ };
- Variables and function do
  - incl. vtable of polymorphic classes (unless inline)

Definitions — Exported Symbols

- “Wholesale” exporting “the class” exports:
  - all member functions
  - all static data members (incl. vtable)
  - inline functions also get exported(!)
    - and, on Windows, the DLL-exported version called
      - or not, if the optimizer inlines it ¯_(ツ)_/¯

Definitions — Exported Symbols

- “Wholesale” exporting “the class” is recursive:
  - nested classes are “exported”, too

Definitions — Exported Symbols

- “Wholesale” exporting “the class” does not export:
  - (hidden) friend functions
  - non-static data members (not variables)
  - member function templates
  - nested class templates

Definitions — Exported Symbols

Conclusion:

- You don’t want to export inline symbols
- You’ll want to *selectively* export extern symbols
  - those that need to be called from outside the DSO
- You don’t want to export an “inline” vtable

“Golden Rules” Of Class Exporting

- Export polymorphic classes “wholesale”
  - and define the dtor out-of-line (even if empty / = default)
- Do not “wholesale” export non-polymorphic classes
  - Export individual functions / variables only
    - e.g. not inlines, not privates (unless called from inline code)

Definitions — Exported Symbols

- Exported symbols are part of the ABI
  - incl. exported inlines (!)
- Non-exported symbols are not
  - the fewer exported symbols, the fewer BC headaches
  - exporting allows “library-private” symbols

Note On Unexported Inline Symbols

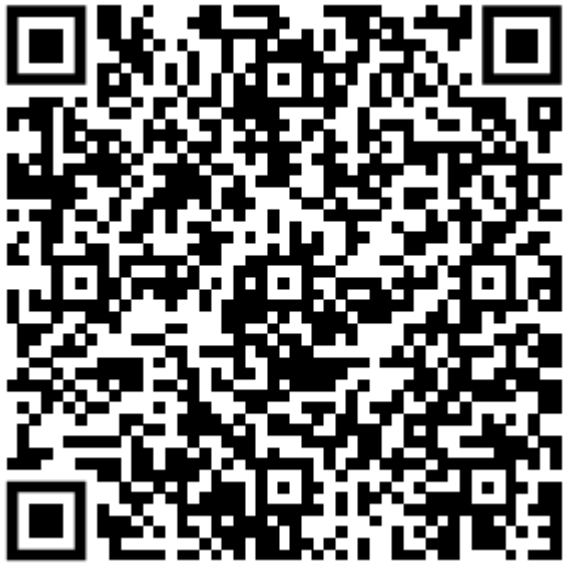
- Linker deduplicates them only *per executable*
  - each DLL, and the app, each get their own copy
  - harmless for functions
    - except their addresses differ
  - surprising for variables

Note On Unexported Inline Symbols

- Solution: Aggressive DRY / SCARY
  - iow: extract such things into separate functions
  - “pull” these functions “behind the ABI boundary”
    - de-inline and export

A Look at the KDE BC Page

- There are many do’s and don’t’s that we have no time for today, let’s peek at the page together:



Techniques

Techniques

- pImpl
- reserved fields
- move semantics
- virtual functions / hooks
- inline namespaces

Techniques – pImpl

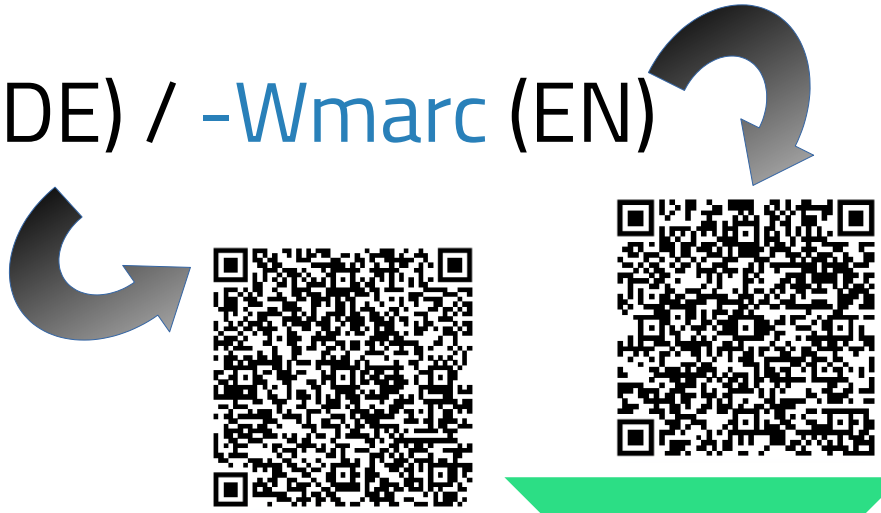
- “Pointer to implementation” / “Compiler Firewall”
  - class Private; Private* pImpl; // only data member
  - externalizes all private state (and functions)
  - private parts can vary independent of public iface

Techniques – pImpl

- Field new'ed up in ctors
  - memory allocation (or pooling)
    - mitigation: implicit sharing, base class reuse, ...
  - Almost all member functions must be out-of-line
  - Exceptions:
    - move SMFs and swap()

Techniques – pImpl

- Lots of gotchas:
  - shallow const; back pointers; Base::pImpl reuse in Derived
- See article series on Heise (DE) / -Wmarc (EN)



Techniques – pImpl

Conclusion:

- Effectiveness: ++
- Runtime overhead: --
  - Moves and swaps are cheap, though

Techniques – reserved fields

- Directly-embedded unused data members
  - [[maybe_unused]] void* reserved = nullptr;
  - unused bits in bit-fields
  - [[maybe_unused]] uint reserved = 0U;
  - explicit padding: [[...]] char padding_n[M] = {};

Techniques – reserved fields

- Fields need to be init'ed, even if unused
  - use NSDMI: = 0U/nullptr
  - if inline code inits, field is already severely restricted
    - e.g. 0 needs to be a valid value (or partially-formed)
  - if inline code destroys, field must be trivially destructible

Techniques – reserved fields

- Construction and destruction must be out-of-line!
  - copy SMFs, dtor, any custom ctor (incl. default ctor)
- Exceptions:
  - move SMFs, swap() → inline
  - delegating constructors → may be inline

Techniques – reserved fields

Conclusion:

- Effectiveness: o
  - Limited capacity (literally) for change
- Runtime overhead: +
  - Pay only for what you use (almost)

BC and Move Semantics

- Move SMFs should be fast → inline
- But they shouldn’t break BC → out-of-line
- Conflict of interest?

BC and Move Assignment

- Natural implementation of move-assignment can be inline:
  - swap(this->m_data, other.m_data)
  - pImpl and reserved field alike
  - “valid, but unspecified”
- BUT: assignment doesn’t destroy LHS
  - OK if type holds only memory resources
  - Alternative: move-and-swap (needs move ctor)

BC and Move Constructor

- Natural implementation of the move constructor can also be inline:
  - m_data{std::exchange(other.m_data, 0U/nullptr)}
  - “Partially-Formed” State (→Meeting C++ 2020)

BC and Move Constructor

- BUT: doesn’t work with smart_ptr
  - Compiler wants to call data member dtor
  - Doesn’t need it, but breaks encapsulation!
  - @ ISO C++: can we fix this?
    - noexcept move ctors oughtn’t call data member dtors

BC and Move Constructor

- smart_ptr pImpl work-around
  - Make the smart_ptr dtor call an out-of-line function
    - Idea: call isn't emitted from move ctor
    - Inside the library, compiler can inline said function
    - No performance overhead
  - BUT: MSVC LTCG doesn't play ball (see references)

BC and Move Semantics

Conclusion:

- Move SMFs and swap() can (and ought to) be inline
- ISO C++ and / or MSVC may have some bugfixing to do

Techniques – Virtual Hook

- Problem: can’t add virtual function to a class
- Qt/QObject-specific work-around:
  - non-virtual function calls impl slot through meta-object
  - works, because the call is delivered via virtual function
  - idea can be generalized, though

Techniques – Virtual Hook

- Needs to have been thought of in advance:

```
protected:

virtual void virtual_hook(int id, void *args) const;
```

- Must have be overridden in every class

Techniques – Virtual Hook

- New function:

```
int pseudoVirtual(int arg) {
    int argv[2] = {0, arg};
    virtual_hook(ID, argv);
    return argv[0];
}
```

- Impl in virtual_hook():

```
if (id == ID) {
    auto argv =
    static_cast<int*>(args);
    argv[0] = op(argv[1]);
}
```

Techniques — Virtual Hook

Conclusion:

- Good for isolated use
- Only for types that are already polymorphic
- Needs to be added ante hoc
- Hard-to-automate task when extending hierarchy

Techniques — Inline Namespaces

- Type versioning at the C++ level
- Lib v1.0:
  - struct S { int x, y, z; }; int foo(const S &);
- Lib v1.1:
  - struct S { int x, y, z; Flags f; }; int foo(const S &);

Techniques – Inline Namespaces

```
inline namespace S_V1_0 {
    struct  S { int x, y, z; };
}
int foo(const S &s);
```

Techniques – Inline Namespaces

```
~~inline~~ namespace S_V1_0 {
    struct  S { int x, y, z; };
}
int foo(const S_V1_0::S &s); // unchanged!
inline namespace S_V1_1 {
    struct S { int x, y, z; Flags f = {}};
}
int foo(const S &s); // new function!
```

Techniques — Inline Namespaces

- Empowers users:
  - “live at HEAD”: use unqualified S
  - “pin old API”: use S_V1_0::S explicitly
  - can even be mixed in the same TU

Techniques — Inline Namespaces

- Combinatorical explosion in N-ary functions, N > 1:
  - replace(String &, const String &, const String &); // 8×
- Different components extend at different times
  - need different namespaces
    - V1_0 → S_V1_0
  - or lots of boilerplate
    - importing all unchanged V1_0 types into V1_1

Techniques — Inline Namespaces

Conclusion:

- Good for isolated use (e.g. Config struct)
- Quickly gets out of hand when used widely
- Needs to be added ante hoc
- Lots of hard-to-automate tasks on version bumps

Cheating

Cheating

- Separating API and ABI
- Removing functions from one, but not the other
- Qt's user-config'able, rolling BC window
- Bonus: inlining out-of-line functions

Cheating — Observations

- Types don’t exist in the ABI
  - only as a collection of functions acting on them
  - there is no “Point” in the ABI
  - there’s only the agreement of certain functions as to type and location of Point’s fields

Cheating — Observations

- ABI is maintained as long as
  - (existing) memory layout doesn't change
  - (existing) functions are not removed

Cheating — Example API vs. ABI

```
struct LIB_EXPORT Point {
    int m_x, m_y;
    void setX(int x);
    int x() const;

    void setY(int y);

    int y() const;

};
```

- offsetof(Point::m_x) == 0
- offsetof(Point::m_y) == 4
- Point_setX_i();
- Point_x_v();
- Point_setY_i();
- Point_y_v();

Cheating — Example Func Addition

```
struct LIB_EXPORT Point {
    int m_x, m_y;
    void setX(int x);
    int x() const;

    void setY(int y);

    int y() const;
    void setY(long long y);
};
```

- offsetof(Point::m_x) == 0
- offsetof(Point::m_y) == 4
- Point_setX_i();
- Point_x_v();
- Point_setY_i();
- Point_y_v();
- Point_setY_ll();

Cheating — Example Func Addition

```
struct LIB_EXPORT Point {
    int m_x, m_y;
    void setX(int x);
    int x() const;
#if 0
    void setY(int y);
#endif
    int y() const;
    void setY(long long y);
};
```

- offsetof(Point::m_x) == 0
- offsetof(Point::m_y) == 4
- Point_setX_i();
- Point_x_v();

(oops)

- Point_y_v();
- Point_setY_ll();

Cheating — Example Func Addition

```
struct LIB_EXPORT Point {
    int m_x, m_y;
    void setX(int x);
    int x() const;
#if MAGIC_REMOVED_API_ONLY
    void setY(int y);
#endif
    int y() const;
    void setY(long long y);
};
```

- offsetof(Point::m_x) == 0
- offsetof(Point::m_y) == 4
- Point_setX_i();
- Point_x_v();
- Point_setY_i();
- Point_y_v();
- Point_setY_ll();

Cheating — removed_api.cpp

```
// point.cpp

void Point::setY(long long i) {
    ~~~~
}
void Point::setY(int i) {
    ~~~~
}
```

```
// removed_api.cpp

#define MAGIC_REMOVED_API_ONLY 1
#include “point.h”
```

Cheating — removed_api.cpp

```
// point.cpp

void Point::setY(long long i) {
    ~~~~
}
void Point::setY(int i) {
    ~~~~~~
}
```

```
// removed_api.cpp

#define MAGIC_REMOVED_API_ONLY 1
#include "point.h"

void Point::setY(int i) {
    setY(static_cast<long long>(i)); // new overload
}
```

Cheating — REMOVED_SINCE

- That's all!
- You can get fancy and version the thing:
  - API_REMOVED_SINCE(1, 1)
- You need this macro per-library
  - so let your build system generate these
  - but only one removed_api.cpp per library

Cheating — REMOVED_SINCE

- more sophisticated symbol versioning exists
- REMOVED_SINCE's advantage is
  - purely C++-based
    - every C++ developer can understand how to apply it
    - cross-platform
  - works post-hoc

Cheating — Homework

- Use REMOVED_SINCE() to remove noexcept from an exported function again
  - Think about syntax in header and removed_api.cpp
  - Also think about semantics of removing noexcept

REMOVED_SINCE — Cut-Off Version

- What happens if I introduce a cut-off version X.Y?
  - i.e. REMOVED_SINCE() = 0 for all versions before X.Y
- I have just introduced a (user-config'able) cut-off for the ABI zombie functions in my DSO
  - IOW: a config'able, rolling BC window

A Rolling BC Window

- Upper bound:
  - The library version I'm building against
- Lower bound:
  - Said cut-off version
- Qt's called -disable-deprecated-up-to

Cheating — REMOVED_SINCE

- Biggest drawback is: doesn't work for virtuals
  - necessary precondition not met:
    - need to be able to add something to remove something else
  - if it weren't for virtuals, Qt could stay two decades more on 6.x
    - alternatively, Qt 7 could be BC with Qt 6

Key Take-Aways

1. ABI ≠ API; both can vary independently
2. Qt has a user-config'able, rolling BC window
3. "Wholesale" exported classes and virtual functions are big impediments to ABI evolution

Q & A

References

- KDE BC page
- Marc Mutz - On removing functions from the API, but not the ABI – Qt development ML 2022-07-13
- Marc Mutz - Partially-formed Objects for fun and profit - Meeting C++ 2020
- Marc Mutz - Pimp My Pimpl – Heise Developer 2010
  - English Translation thereof
- Discussion about MSVC LTCG bug in Qt bug tracker
  - Stack Overflow discussion: “Where does the destructor hide in this code?”
- QUIP-6: Acceptable Source-Incompatible Changes
- Marc Mutz – Fun With Exceptions - -Wmarc 2010-08-04
- Qt documentation: QT_DISABLE_DEPRECATED_UP_TO

Cheating — Bonus Slides

Cheating — Inlining Functions

- Next Level: inlining formerly out-of-line functions
- Idea:
  - REMOVE_SINCE the out-of-line function
  - add the inline function
- Two problems…

Cheating — Inlining Functions

- Obvious problem: the two don’t overload
  - just use #else
- Not-so-obvious problem: ODR violation
  - REMOVED_SINCE is, too, but non-actionable
  - inline vs. extern linkage, or even vs. exported *is* actionable

Cheating — Inlining Functions

- We have three domains now:
  - removed_api.cpp
  - callers external to the library
  - callers internal to the library

Cheating — Inlining Functions

- external users should see the function as inline
- removed_api.cpp must see it as out-of-line
  - to maintain ABI
- internal users must see it as out-of-line, too
  - else: ODR violation when linked with removed_api.cpp

Cheating — INLINE_SINCE

```
// point.h
struct Point { ~~~
   API_INLINE_SINCE(1, 1)
   int x() const;
~~~ };
#if API_INLINE_IMPL_SINCE(1, 1)
int Point::x() const { ~~~ }
#endif
```

```
// removed_api.cpp
~~~
#include “point.h” // inlined API
```

Cheating — INLINE_SINCE

- INLINE_SINCE
  - inline
    - external callers
  - /* not inline */
    - removed_api.cpp
    - internal callers
- INLINE_IMPL_SINCE
  - 1
    - external callers
    - removed_api.cpp
  - 0
    - internal callers

The End