API & ABI versioning

How to handle compatibility with your C++ libraries

When I change my code,
what are the impacts?

About this talk

- Semver!
- Binary compatibility!

About this talk (for real)

- Changes and impacts on API & ABI
- Categorizing changes
- Avoiding impacts
- Advertising change through versioning

Hello!

*I am **Mathieu Ropert***

I’m a C++ developer at Murex where I work on internal frameworks and Open Source initiatives.

You can reach me at:

- ✉ mro@puchiko.net
- @MatRopert
- @mropert

1

Library lifecycle

Asking yourself the right questions

So you want to publish a library

- Will all users’ code belong to the same repo as your library?
- If yes, versioning is not mandatory
- But even then, it will not hurt to think about the impacts

So you want to publish a library

- Will you ever break backward compatibility?
- Remember that removing old / deprecated features is still breaking compatibility
- If you do it, even rarely, you need a way to distinguish changes

So you want to publish a library

- Do you want your users to be able to hotswap your library in production?

- Not an option for header-only libraries

- If the answer is “yes”, you will have to monitor ABI changes

Things to keep in mind

- It’s important to be careful when changing API
  - Even if you can patch all your clients at once
- If binary compatibility is a concern, you also need to keep an eye on ABI impacts
- You’ll need to inform your users about changes and their impacts

Versioning

Communication between maintainers and users
about the changes in a software

Reasonable use

- Some users will expect unreasonable guarantees from your code
  - Line numbers
  - Symbol addresses (and being able to get them)
  - Real type of `auto` types
  - Layout of private members

- This talk is not about how to handle that

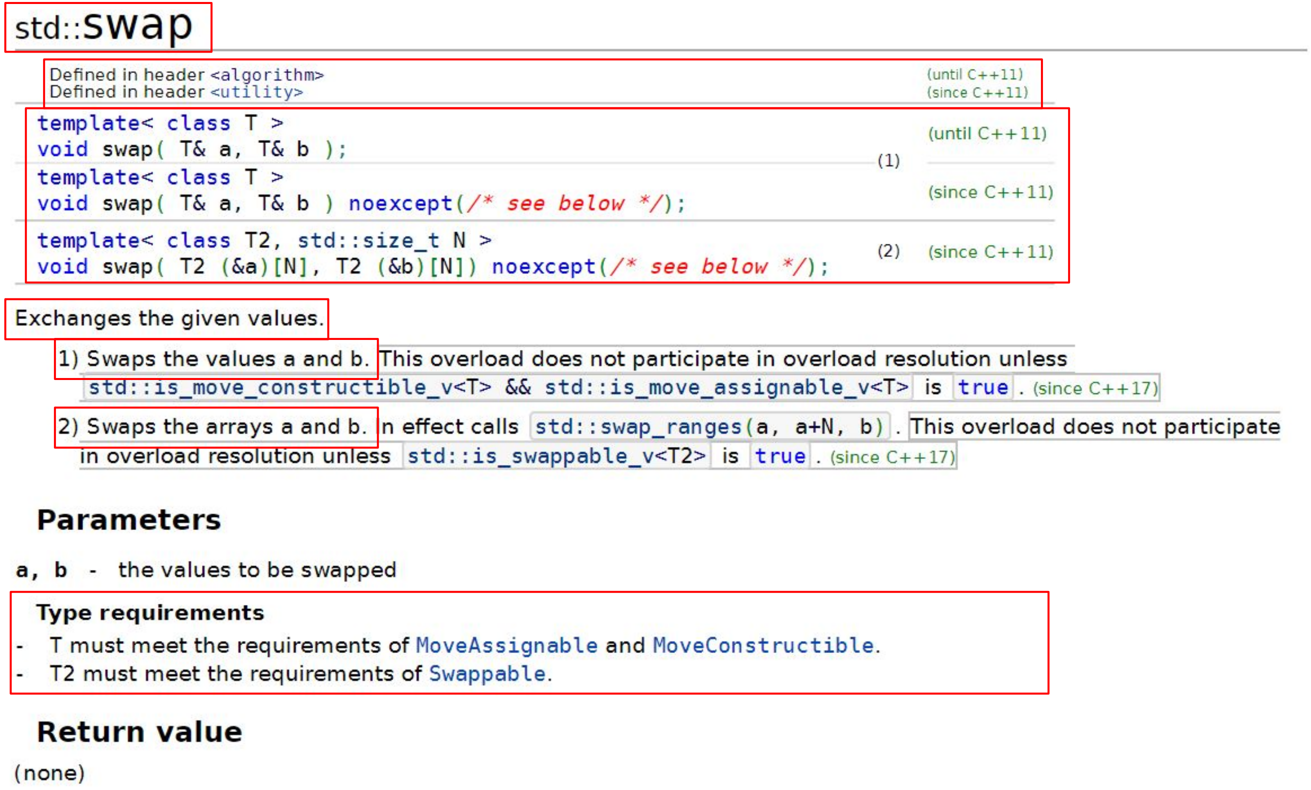
2

Changes in API

Contracts and how not to breach them

What’s an API?

- An API is a contract between the maintainer and the user
- It’s divided in two parts
  - Pre-conditions: what the caller must provide
  - Post-conditions: what the callee will ensure if the pre-conditions are met



std::swap

Defined in header `<algorithm>` (until C++11)
Defined in header `<utility>` (since C++11)

```
template< class T >
void swap( T& a, T& b );                                        (1) (until C++11)
template< class T >
void swap( T& a, T& b ) noexcept(/* see below */);                  (since C++11)
template< class T2, std::size_t N >
void swap( T2 (&a)[N], T2 (&b)[N]) noexcept(/* see below */);   (2) (since C++11)
```

Exchanges the given values.

1) Swaps the values a and b. This overload does not participate in overload resolution unless `std::is_move_constructible_v<T> && std::is_move_assignable_v<T>` is `true`. (since C++17)

2) Swaps the arrays a and b. In effect calls `std::swap_ranges(a, a+N, b)`. This overload does not participate in overload resolution unless `std::is_swappable_v<T2>` is `true`. (since C++17)

Parameters

a, b - the values to be swapped

Type requirements

- T must meet the requirements of MoveAssignable and MoveConstructible.
- T2 must meet the requirements of Swappable.

Return value

(none)

source: http://en.cppreference.com/w/cpp/algorithm/swap

API in C++ terms

Internal

- Names
- Signatures
- Declarations locations

External

- Pre-conditions
- Post-conditions
- Misc guarantees

API in C++ terms

- Not all parts of an API are part of the language or seen by the compiler
- You must rely on some form of documentation to express the rest
- Caution is advised when changing parts not covered by the language itself

API changes by impacts

- API breaking change
  - Clients must adapt their code
- API non-breaking change
  - Guaranteed to be backward compatible, but not always forward compatible
- No change to API
  - Guaranteed to be both backward and forward compatible

Changes with no impact

- Any change that does not add, remove or change a contract

- Changes to implementation
  - Bugfixes
  - Performance tuning
  - Refactoring

Changes with no impact

- No name or signature has changed or moved
- Defined behaviour is still the same…
- …including specific guarantees
  - Complexity
  - Iterator validity

API non-breaking changes

- Adding a new contract
  - New function
  - New overload(*)
  - New type
  - New namespace

API non-breaking changes

- Relaxing an existing contract
  - New default argument to a function(*) or template
  - New struct member
  - Relaxing pre-conditions
  - Narrowing post-conditions
  - Narrowing guarantees
  - Defining undefined behaviour

API breaking changes

- Changing a signature
  - Argument types or order
  - Return type
- Renaming
- Moving declaration to another header file

API breaking changes

- Narrowing a contract
  - Narrowing pre-conditions
  - Relaxing post-conditions
  - Relaxing existing guarantees

API breaking changes

- Narrowing a contract
  - Narrowing pre-conditions
  - Relaxing post-conditions
  - Relaxing existing guarantees
- Evil!

API breaking changes

- Narrowing a contract
  - Narrowing pre-conditions
  - Relaxing post-conditions
  - Relaxing existing guarantees
- Evil!
- Seriously, don’t do that

Invisible breaking change

Before

```
// Sorts a vector of integers
// Complexity: O (n * log n)
void foo(std::vector<int>& v) {
   std::sort(begin(v), end(v));
}
```

After

```
// Sorts a vector of integers
// Complexity: O(n!)
void foo(std::vector<int>& v) {
   while (!std::is_sorted(begin(v), end(v)))
      std::random_shuffle(begin(v), end(v));
}
```

3

Changes in ABI

Compatibility between binaries

What is ABI?

- Application Binary Interface
- Defines how binary components talk to each others
- Not covered by the C++ Standard(*)

ABI in C++ terms

Infrastructure

- Calling convention
- Exception handling
- Mangling
- C++ runtime

Code

- Symbol names
- Binary representation of API types
- vtable layout

Symbol names

- Each exported symbol has an id:

Name + Signature => id

```
void foo(int)    =>  _Z3fooi
void foo(double) =>  _Z3food
```

Symbol names

- Changing the id of any public symbol will break ABI

- Public symbols are all API symbols *and* all symbols used by inline functions in public headers

Implementation changes

Before

```
namespace details {
    MY_EXPORT void bar();
};

inline void foo() {
    details::bar();
}
```

After

```
namespace details {
    MY_EXPORT void bar(int);
};

inline void foo() {
    details::bar(0);
}
```

vtable Layout

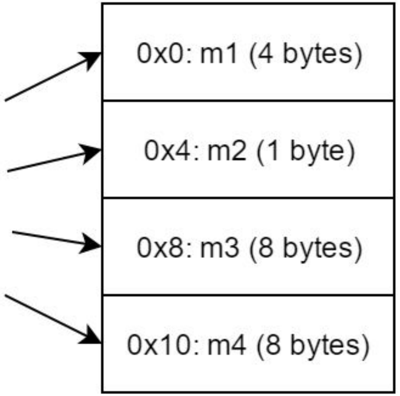
- How pointers to virtual methods are stored
- Depends on the compiler
    - Usually one standard per OS
- Breaks when you reorder virtual methods
- Or when you add a new one

Binary representation

- Each public structure has a particular layout in the ABI
  - Structure size
  - Size of each member
  - Starting offset of each member

- Actual layout depends on various platform rules

```
struct A {
    int     m1;
    bool    m2;
    char*   m3;
    double  m4;
};
```

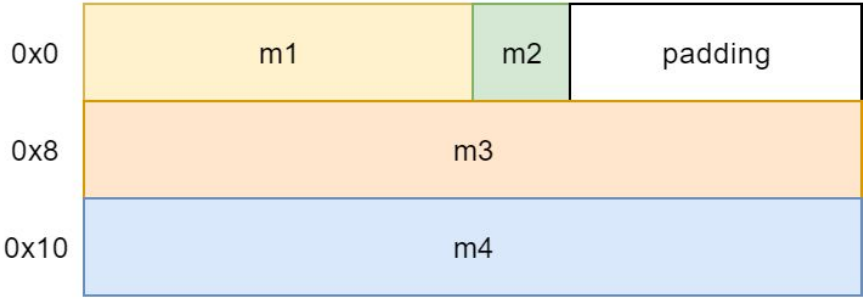


0x0: m1 (4 bytes)

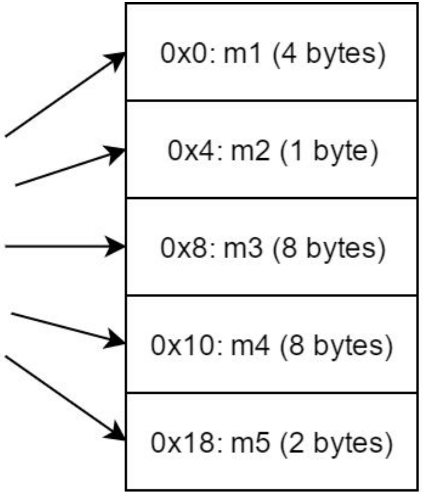
0x4: m2 (1 byte)

0x8: m3 (8 bytes)

0x10: m4 (8 bytes)



```
struct A {
    int    m1;
    bool   m2;
    char*  m3;
    double m4;
    short  m5;
};
```



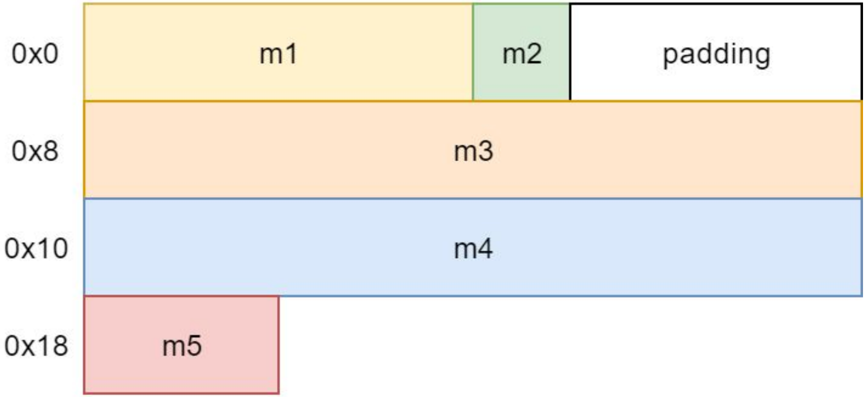
0x0: m1 (4 bytes)

0x4: m2 (1 byte)

0x8: m3 (8 bytes)

0x10: m4 (8 bytes)

0x18: m5 (2 bytes)



Binary representation

- Changing the type or the order of members in a struct will break ABI
- Adding a new member will break it too
- Changing a member visibility may also break ABI

4

C++ Versioning

Semver reloaded

Semantic Versioning

- A formal convention to express compatibility between versions
- Created in 2011 by Tom Preston-Werner
- 3 numbers sequence: X.Y.Z
- X is major release
- Y is minor release
- Z is patch release

Major release

- 1st component of *semver* convention
- Indicates an important and *non-backward-compatible* change
- Users will have to change their code to upgrade or downgrade
- Some features may not be available anymore

Minor release

- 2nd component of *semver* convention
- Indicates the addition of new features that do not impact the existing ones
- Existing users can safely upgrade without changing their code
- Downgrading is also possible if the new features are not used

Patch release

- 3rd component of *semver* convention
- Indicates the release contains only bugfixes
- Existing users can safely upgrade without changing their code
- Downgrading is also possible (although usually not recommended)

How to include API in versioning?

- Follow semver convention
- Maintain a changelog
- Document your contracts
- Avoid invisible breaking changes

How to include ABI in versioning?

- Don’t!
  - If your clients always recompile, don’t bother
  - If your library is header only
  - But make it clear in your documentation

How to include ABI in versioning?

- Don’t!
  - If your clients always recompile, don’t bother
  - If your library is header only
  - But make it clear in your documentation

- Or adapt semver convention to include ABI

Semver reloaded

- API or ABI breaking change: major revision
- API or ABI non-breaking change: minor revision
- No change: patch revision

What about dependencies?

- Changing the major version of a public dependency will break API
  - … and possibly ABI too

- Changing the major version of a private dependency will break ABI

Can I do more?

- Advertising change is important, but you can go the extra mile by providing migration scripts

- For example, Clang based refactoring tools

- This will encourage clients to upgrade quickly

What the future may hold

- Contracts TS should help you detect changes to your API more easily

- Modules TS may change the way you distribute binaries (possibly without headers)

Quizz

Did you follow everything?

Quizz #1

Before

```
void foo(int);
```

After

```
void foo(int, bool);
```

Breaking API change
& breaking ABI change

Quizz #2

Before

```
int foo(int);
```

After

```
int foo(long);
```

API change
& breaking ABI change

Quizz #3

Before

```
struct A {
    int i;
    char *s;
};
```

After

```
struct A {
    char *s;
    int i;
};
```

Breaking API change
& breaking ABI change

Quizz #4

Before

```
struct A {
    void foo();
    void bar();
};
```

After

```
struct A {
    void bar();
    void foo();
};
```

No change

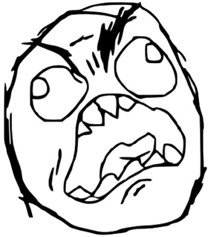
Quizz #5

Before

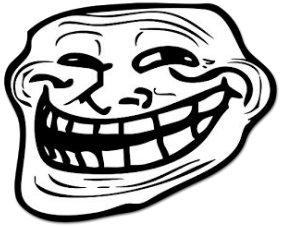
```
int foo(int a, int b) {
    return a + b;
}
```

After

```
int foo(int a, int b) {
    return a > b ? a : b;
}
```



Invisible breaking API change



Quizz #6

Before

```
struct A {
    virtual void foo();
    virtual void bar();
};
```

After

```
struct A {
    virtual void bar();
    virtual void foo();
};
```

Breaking ABI change

Quizz #7

Before

```
struct A {
    int i;
    bool b;
    char *s;
};
```

After

```
struct A {
    int i;
    bool b;
    char t[2];
    char *s;
};
```

ABI change(*)

Quizz #8

Before

```
void foo(int);
```

After

```
void foo(int, bool = false);
```

API change
& breaking ABI change ☹

Quizz #9

Before

```
void foo(int);
```

After

```
void bar(int);
```

Breaking API
& breaking ABI change 🙁

Quizz #10

Before

```
struct A {
   int i;
   char *s;
};
```

After

```
struct A {
   int i;
   char *str;
};
```

Breaking API change ☹

Quizz #10 and half

Before

```
namespace details {
int bar(int);
}
inline int foo(int x) {
   return details::bar(x);
}
```

After

```
namespace details {
int bazz(int);
}
inline int foo(int x) {
   return details::bazz(x);
}
```

Breaking ABI change

No system became successful by breaking backward compatibility…

… especially without warning its users beforehand

“

Versioning

Communication between maintainers and users
about the changes in a software

Thanks!

Any questions ?

You can reach me at

✉ mro@puchiko.net

@MatRopert

@mropert