API & ABI versioning
How to handle compatibility with your C++ libraries
When I change my code, what are the impacts?
About this talk

- Changes and impacts on API & ABI
- Categorizing changes
- Mitigating impacts
- Handling change through versioning
Hello!

I am Mathieu Ropert

I’m a C++ developer at Murex and the co-organizer and host of the Paris C++ User Group.

You can reach me at:

✉️ mro@puchiko.net
@MatRopert
🌐 https://mropert.github.io
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
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>`
Defined in header `<utility>`

```cpp
template< class T >
void swap( T& a, T& b );
```

(1) (until C++11)

```cpp
template< class T >
void swap( T& a, T& b ) noexcept(/* see below */);
```

(2) (since C++11)

```cpp
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

   \[ \text{std::is_move_constructible_v<T> } \&\& \text{std::is_move_assignable_v<T> is true} \] (since C++17)

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

Parameters

a, b - the values to be swapped

Type requirements

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

Return value

(none)

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 are 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
  - Performance tuning
  - Refactoring
  - Bugfixes
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
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
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
Before

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

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

After

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

inline void foo() {
    details::bar(0);
}
```
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;
};
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
## The two schools of versioning

<table>
<thead>
<tr>
<th>Semantic versioning</th>
<th>Live at Head</th>
</tr>
</thead>
<tbody>
<tr>
<td>◦ Parallel releases</td>
<td>◦ Serial releases</td>
</tr>
<tr>
<td>◦ Complex scheme</td>
<td>◦ Source based</td>
</tr>
<tr>
<td>◦ Asynchronous upgrade</td>
<td>◦ No diamond conflicts</td>
</tr>
<tr>
<td>◦ Binary compatibility</td>
<td>◦ Automated client code migration</td>
</tr>
</tbody>
</table>
Semantic Versioning

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

- Major releases break API
- Minor releases change API without breaking
- Patch release have no impact on API
- Says nothing about ABI!
  - But we can fix that
Semver reloaded

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

- Offers a degree of binary compatibility
  - Both upgrades and rollbacks

- Can work around the diamond inheritance problem as long as the major is the same

- Most common scheme today
Coined by Titus Winters at CppCon 2017
Each change simply increments a serial number
Clients must use the same version across a binary
Each breaking change comes with an automatic refactoring script
Live at **Head**

- Guarantees no diamond problem can occur
- Facilitates upgrade and ensure clients stay at “head”
- No binary compatibility
- No support on older releases
How to include API in versioning?

◉ Pick a versioning convention
◉ Tell clients which guarantees you offer
◉ 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, also don’t bother
  - 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 go for revised semver
Inline namespaces

- Available since C++11
- Make names available through the parent namespace in the API, but not in the ABI
- Can be used to version symbols
Inline namespaces

namespace mylib {
    namespace v1 {
        void foo(int);
    }
    inline namespace v2 {
        void foo(int);
    }
}

- References to `mylib::foo()` will alias to `mylib::v2::foo()` in ABI
- Older clients will still be able to use `mylib::v1::foo()`
What about dependencies?

- Breaking changes on public dependencies also break your API
  - ... and possibly your ABI too

- Breaking changes on private dependencies break your ABI
What the future may hold

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

- Modules TS should help you enforce which parts of your library are public API
Quizz
Did you follow everything?
Quizz #1

**Before**

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

**After**

```c
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

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

After

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

Breaking API change (*) & breaking ABI change
Before

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

After

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

No change
Quizz #5

Before

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

After

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

Invisible breaking API change
Before

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

After

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

Breaking ABI change
Quizz #7

Before

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

After

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

Breaking API change(*) & ABI change(*)
Before

void foo(int);

After

void foo(int, bool = false);

API change & breaking ABI change
Quizz #9

Before

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

After

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

Breaking API & breaking ABI change
Quizz #10

**Before**

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

**After**

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

**Breaking API change**
Quizz #10 and half

Before

```cpp
namespace details {
    int bar(int);
}

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

After

```cpp
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 or automatic migration
Versioning

Communication between maintainers and users about the changes in a software
Thanks!

Any questions?

You can reach me at

✉️ mro@puchiko.net
🐦 @MatRopert
้ว @mropert
🌐 https://mropert.github.io