Library versions with libtool and ABI compatibility

Tuomo Turunen
2019-11-07

What is API? What is ABI?

- API – Application Programming Interface
  - In the scope of this slide set we consider API as **public C and C++ headers** provided by a SW component (i.e. headers available in devel package)
- ABI – Application Binary Interface
  - In the scope of this slide set we consider ABI as the binary interface between a shared library and an application

What is API and ABI compatibility

- A change to API is *backward compatible*, if source code of applications using the API do not have to be changed
  - The rest of this slide set will not talk about API compatibility, but concentrates only on ABI compatibility
- A change to ABI is *backward compatible*, if binaries using the ABI do not have to be recompiled
  - The rest of this slide set will use these terms:
    - ABI compatible: binaries do not have to be recompiled
    - ABI **incompatible**: binaries must be recompiled

Shared library versioning with libtool

- Most RCP subsystems use libtool for building shared libraries
- Libtool adds a version number at the end of a library name based on three values: **current**, **revision** and **age**
- The rules for updating **current**, **revision** and **age** are explained in https://autotools.io/libtool/version.html. Copy-pasted also here:
  - Always increase the **revision** value
  - Increase the **current** value whenever an interface has been added, removed or changed
  - Increase the **age** value only if the changes made to the ABI are backward compatible
  - Note #1: if there are no changes made to the ABI (i.e. **current** wasn't increased), then **age** must not be touched
  - Note #2: **revision**, **current** or **age** are never decreased or reset to zero
- There are also other (more complex) guidelines for updating **current**, **revision** and **age**, but in RCP we use this one
- **Warning**: A common mistake is to assume that **current**, **revision** and **age** values map directly into the three numbers at the end of the library name. This is not the case, and indeed, **current**, **revision** and **age** are applied differently depending on the operating system that one is using.
  - Linux: `libxxx.so.{current – age}.{age}.{revision}`
  - FreeBSD: `libxxx.so.{current}`
  - `OpenBSD: libxxx.so.{current}.{revision}`
  - Android: `libxxx.so`

Examples

- Example 1: No interface changes
  - Increase **revision**
  - Do not touch **current** or **age**
- Example 2: ABI compatible change
  - Increase **current**, **revision** and **age**
- Example 3: ABI **incompatible** change
  - Increase **current** and **revision**
  - Do not touch **age**

Shared libraries in Linux

- In Linux a shared library consists of two symbolic links and the actual library:
  ```
  libxxx.so -> libxxx.so.{current – age}.{age}.{revision}
  libxxx.so.{current – age} -> libxxx.so.{current – age}.{age}.{revision}
  libxxx.so.{current – age}.{age}.{revision}
  ```
- The symbolic link without version number (i.e. `libxxx.so`) is used in development environment for linking applications using the shared library
  - This file is needed only in the development environment and thus belongs to the devel package
- The symbolic link with one version number and the actual library are used by loader when executing binaries
  - These files are needed in both development environment and runtime environment
  - Typically these files belong to libs package, but can be also in the base package is there is no need for separate libs package
- "*Library soname*" or "*SONAME*" is `libxxx.so.{current – age}`
  - soname is used for both backward-compatibility information and dependency information

Example

- libgenapi: current = 11, revision = 26, age = 7

```
[tuomo@linux:/usr/lib64]
$ ls -l libgenapi.so*
lrwxrwxrwx 1 root root     19 Oct 23 10:35 libgenapi.so -> libgenapi.so.4.7.26
lrwxrwxrwx 1 root root     19 Oct 23 10:35 libgenapi.so.4 -> libgenapi.so.4.7.26
-rwxr-xr-x 1 root root 463160 Oct 23 10:35 libgenapi.so.4.7.26
[tuomo@linux:/usr/lib64]
$ readelf -a libgenapi.so.4.7.26 | grep soname
 0x000000000000000e (SONAME)             Library soname: [libgenapi.so.4]
[tuomo@linux:/usr/lib64]
$ rpm -qf libgenapi.so
genapi-devel-1.4.0-1.wf30.x86_64
[tuomo@linux:/usr/lib64]
$ rpm -qf libgenapi.so.4
genapi-1.4.0-1.wf30.x86_64
[tuomo@linux:/usr/lib64]
$ rpm -qf libgenapi.so.4.7.26
genapi-1.4.0-1.wf30.x86_64
```

Why library versions are important?

- There are multiple reasons to have correctly versioned libraries
- In RCP the most important reason is RPM dependency tracking
  - When a binary rpm is built, all shared libraries used by the rpm and provided by the rpm are (automatically) listed as dependencies
  - For example:

    ```
    shareddatalayer-libs-3.7.8-1.wf30.x86_64.rpm Provides libsdl.so.8()(64bit)
    trafficmanager-1.0.4-4.wf30.x86_64.rpm Requires libsdl.so.8()(64bit)
    ```
  - The above dependency means that installing trafficmanager 1.0.4-4 requires also compatible shareddatalayer-libs
- If shared library ABI is changed in incompatible manner, the library version changes and RPM notices the conflict – to fix the conflict, rpms depending on the old library version must be rebuilt
- If library versions are missing or are updated incorrectly, RCP build will not work!
  - This wasn’t a problem with the legacy RCP build, because everything was always rebuilt
  - This is a major problem with Koji build, where rpms are rebuilt only when needed

Example

- genapi 1.2.2-1 provides libgenapi.so.3 (libgenapi.so.3.6.24)
- ministarter 0.1.9-1 depends on libgenapi.so.3
- genapi 1.3.0-1 has interface change, which is ABI **incompatible** (but API compatible)
  - Library name is changed to libgenapi.so.4 (libgenapi.so.4.6.25)
- ministarter has to be rebuilt
  - ministarter 0.1.9-2 depends on libgenapi.so.4
- genapi 1.4.0-1 has interface change, which was ABI compatible
  - Library name doesn’t change; it is still libgenapi.so.4 (libgenapi.so.4.7.26)
- There is no need to rebuild ministarter; it is still ministarter 0.1.9-2

How to determine if interface change is ABI compatible?

- Sometimes it is not easy to determine whether an interface change is ABI compatible
  - C++ standard doesn’t say much about binary interface
  - ABI compatibility cannot be proven by testing (but ABI **incompatibility** can)
- ABI documents are long and complex:
  - http://static.coldattic.info/restricted/science/syrcose09/cppbincomp.pdf
  - http://itanium-cxx-abi.github.io/cxx-abi/abi.html
- Rules of thumb:
  - If public headers are not touched, then the change is ABI compatible
  - Adding new interfaces is ABI compatible
  - Removing existing interfaces is ABI **incompatible**
  - Changing existing interfaces depends on the change – see the examples in the following slides

Adding new interfaces

- Adding new macros, constants, structs, classes, enums and functions is ABI compatible

```
+ #define NEW_MACRO ... ✓
+ constexpr int NEW_CONSTANT = ...; ✓
+ struct NewStruct { ... }; ✓
+ class NewClass { ... }; ✓
+ enum class NewEnum { ... }; ✓
+ void newFunction(int a); ✓
```

Removing existing interfaces

- Removing existing macros, constants, structs, classes, enums and functions is ABI **incompatible**

```
- #define EXISTING_MACRO ...  ✖
- constexpr EXISTING_CONSTANT = ...;✖
- struct ExistingStruct { ... };✖
- class ExistingClass { ... };✖
- enum class ExistingEnum { ... }; ✖
- void existingFunction(int a);✖
```

Changing existing macros

- Changing existing macros is ABI **incompatible**, because macros are evaluated at compile time

```
- #define MY_BIT_MASK 0x7
+ #define MY_BIT_MASK 0xF
```

✖

Changing existing constants

- Changing existing constants is ABI **incompatible**, because `constexpr` statements are evaluated at compile time

```
- constexpr size_t MAX_SIZE = 32;
+ constexpr size_t MAX_SIZE = 64;
```

✖

Changing existing inline functions

- Changing existing inline functions, changing existing non-inline functions to inline or changing existing inline functions to non-inline is ABI **incompatible**, because inline functions are evaluated at compile time

- This applies also to the default constructors, destructors, copy constructors, copy operators, move constructors and move operators either implicitly or explicitly created by compiler

```
  class ExistingClass
  {
  public:
-     ExistingClass(): size(0) { } ✖
+     ExistingClass();
-     ~ExistingClass(); ✖
+     ~ExistingClass() = default;
-     size_t getSize() const { return size; } ✖
+     size_t getSize() const { return size * 2; }
  };
```

Changing existing functions

- Changing existing function return value type is ABI **incompatible**
- Changing existing function parameter type is ABI **incompatible**
- Adding parameter with default value to existing function is ABI **incompatible**

```
- uint32_t getBits();
+ uint64_t getBits();  ✖
- void print(Object object);
+ void print(const Object& object);  ✖
- Configuration getConfiguration();
+ Configuration getConfiguration(bool validate = false);  ✖
```

Changing existing member functions

- Adding or removing `const` keyword is ABI **incompatible**
- Adding or removing `virtual` keyword is ABI **incompatible**
- Adding or removing `noexcept` keyword is ABI **incompatible**

```
  class ExistingClass
  {
  public:
-     void draw(Canvas& canvas) const;
+     virtual void draw(Canvas& canvas) const;   ✖
-     size_t getSize();
+     size_t getSize() const;   ✖
-     void clear();
+     void clear() noexcept;   ✖
  };
```

Adding or removing member variables

- Adding or removing (private, protected or public) member variable from a struct or a class is ABI **incompatible**, because this changes the size of the struct or class

```
  class ExistingClass
  {
      ...
  private:
      int x;
+     int y; ✖
  };
```

Reordering public member variables

- Reordering public or protected member variables in a struct or a class is ABI **incompatible**, because this changes the variable offset in the struct of class
- This applies also to private member variables used by inline functions

```
  struct ExistingStruct
  {
-     int y; ✖
      int x;
+     int y;
  };
```

Adding new non-virtual member functions

- Adding new non-virtual (private, protected or public) member function is ABI compatible
- The order of non-virtual functions is irrelevant

```
  class ExistingClass
  {
  public:
      void oldFunction();
+     void newFunction(); ✓
  };
```

Adding new virtual member functions

- Adding new virtual (private, protected or public) member function is ABI compatible, if the function is added as the last virtual function in the class
- Adding new virtual member function between or before existing virtual functions is ABI **incompatible**
- The order of virtual functions is very important!

```
  class ExistingClass
  {
  public:
      virtual void oldFunction();
+     virtual void newFunction(); ✓
  };
```

```
  class ExistingClass
  {
  public:
+     virtual void newFunction(); ✗
      virtual void oldFunction();
  };
```

Adding new static member variables or functions

- Adding new static (private, protected or public) member variable or function is ABI compatible

```
  class ExistingClass
  {
+     static int variable; ✓
+     static void function(); ✓
  };
```

Changing class hierarchy

- In general, changing class hierarchy in anyway is ABI **incompatible**

```
- class ExistingClass
+ class ExistingClass: public NewBaseClass
  {
      ...;
  };
```

✖

Adding new enum values

- Adding a new enum value without changing any of the existing enum values is ABI compatible (as long as the new value fits into the underlying type, which is by default `int`)
- Changing an existing enum value is ABI **incompatible**

```
  enum class ExistingEnum
  {
      RED,
      GREEN,
+     BLUE,  ✓
  };
```

```
  enum class ExistingEnum
  {
      RED,
+     BLUE,  ✗
      GREEN,
  };
```

Techniques to avoid incompatible ABI changes

- API (i.e. the public headers) must be designed so that it reveals as little as possible about the actual implementation – this must be the design from the very beginning
- Avoid macros in public headers
- Avoid inline functions in public headers
  - Don’t inline functions for optimization, unless you have measured performance and measurement proves that inlining functions is the solution
- Always add new enums at the end of the list
- Avoid functions with default value parameters in public headers
  - Use function overloading instead
- Do not expose class implementation in public headers
  - Forward declaration pattern
  - PImpl pattern
  - Interface pattern with abstract classes

Forward declaration pattern

- Forward declaration pattern is typically used in C APIs
- The idea is to expose only struct name in public header, but nothing about its implementation – application can use the struct only via the functions declared in the public header

public_header.h

```
struct my_struct; // forward declaration

struct my_struct *create_my_struct(...);
void use_my_struct(struct my struct *);
void destroy_my_struct(struct my_struct *);
```

implementation.c

```
struct my_struct
{
    int x;
    int y;
};
```

PImpl pattern

- *Private Implementation* pattern is basically C++ version of the forward declaration – the real implementation is not in the public header

public_header.hpp

```
class MyClass
{
public:
    MyClass();
    ~MyClass();
    void function();
private:
    class Impl; // forward declaration
    std::unique_ptr<Impl> impl;
};
```

implementation.cpp

```
class MyClass::Impl
{
public:
    void function();
private:
    ...
};

MyClass::MyClass(): impl(new Impl()) { }

MyClass::~MyClass() { }

void MyClass::function() { impl->function(); }
```

Interface pattern

- Exposing only abstract classes in public headers not only completely hides the implementation from application but also allows *decoupling* as defined by Dependency Inversion Principle (DIP)

public_header.hpp

```
class MyClass
{
public:
    MyClass() = default;
    virtual ~MyClass() = default;
    virtual void function() = 0;
    static std::unique_ptr<MyClass> create();
};
```

implementation.cpp

```
class MyClassImpl: public MyClass
{
public:
    void function() override;
private:
    ...
};

std::unique_ptr<MyClass> MyClass::create()
{
    return std::make_unique(MyClassImpl);
}
```

Incompatible ABI changes via FCI

- Doing ABI **incompatible** changes via FCI is explained in https://confluence.int.net.nokia.com/display/RCP/FCI+support+for+koji
- Briefly:
  - If a subsystem has ABI **incompatible** change, then add the source code repository name to ”`Incompatible ABI Change`” box
  - Determining whether ABI has no changes, has ABI compatible changes or has ABI **incompatible** changes is up to the **developer** – this is not done automatically by FCI
  - Updating library version numbers is up to the **developer** – this is not done automatically by FCI

Links

- Autotools Mythbuster – Library Versioning:
  https://autotools.io/libtool/version.html
- soname
  https://en.wikipedia.org/wiki/Soname
- Binary Compatibility of Shared Libraries Implemented in C++ on GNU/Linux Systems
  http://static.coldattic.info/restricted/science/syrcose09/cppbincomp.pdf
- Itanium C++ ABI
  http://itanium-cxx-abi.github.io/cxx-abi/abi.html
- FCI support for koji
  https://confluence.int.net.nokia.com/display/RCP/FCI+support+for+koji