Binary compatibility for shared libraries

C++ User Group Aachen, 2017-03-09, Daniel Evers

Outline

1. Kinds of compatibility
2. Digression: bits & pieces
3. Safe & unsafe changes to a shared library
4. Best practices for shared library development
5. Tools & links

Kinds of compatibility

Backward compatibility

Backward compatibility is a property of a system, product, or technology that allows for ***interoperability with an older legacy system***, or with input designed for such a system, especially in telecommunications and computing. [...]

Modifying a system in a way that does not allow backward compatibility is sometimes called "breaking" backward compatibility.

Source: Wikipedia

Forward compatibility

Forward compatibility is a design characteristic that allows a system to gracefully ***accept input intended for a later version*** of itself. [...]

A standard supports forward compatibility if a product that complies with earlier versions can "gracefully" process input designed for later versions of the standard; the ability of a system to select known input and ignore unknown input also depends on ***whether the new standard is backward compatible***.

Source: [Wikipedia]

Applied to shared libraries

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

Backward and forward compatibility are mixed here - depending on the point of view:

- Your API is forward compatible - the client program can deal with ***newer*** versions of your library.
- Your ABI is backward compatible - the library can work with programs compiled against an ***older*** version.

Alternatives?

If a program needs to be recompiled in order to work with a new version of library (without any further modifications), the library is **source compatible**.

Or you don’t care… and all hell breaks loose.

*Note: Compatibility generally covers the **structure** and and the **behavior** of your library. This talk is about <u>structure</u>.*

Application Binary Interface (ABI)

According to GNU, in a nutshell:

library API + compiler ABI = library ABI

The ABI specifies:

- object memory layout (including vtables etc.)
- function calling conventions
- exception handling interfaces
- symbol naming / mangling
- other object code conventions
- ...

Compiler ABIs

- There are standards, but no guarantees …
- ABIs vary between compiler releases …
- ABIs vary between compilers …
- ABIs vary between compiler options/flags …

➔ **Mixing compilers is a pain and might require a C library interface :-(**

Digression:
bits & pieces

Demo

Looking at a compiled library and it’s API / ABI …

C/C++ function calling conventions

Architecture	Calling convention name	Operating system, compiler	Parameters in registers	[illegible] on stack	Stack cleanup by
IA-32	cdecl	GCC		[illegible]	Caller
	cdecl	Microsoft		[illegible]	Caller
	stdcall	Microsoft		[illegible] (C)	Callee
		GCC		RTL (C)	Hybrid
	fastcall	Microsoft	ECX, EDX	RTL (C)	Callee
	fastcall	GCC	ECX, [illegible]	RTL (C)	Callee
	register	Delphi and Free Pascal	[illegible]	LTR (Pascal)	Callee
	thiscall	Windows (Microsoft Visual C+[illegible]	[illegible]	RTL (C)	Callee
	vectorcall	Windows (Microsoft Vi[illegible]	[illegible]	RTL (C)	
		Watcom compi[illegible]	EAX, EDX, EBX, ECX	RTL (C)	Callee
x86-64	Microsoft x64 calling convention[12]	Window[illegible] C++, GC[illegible]er, Delphi), [illegible]	RCX/XMM0, RDX/XMM1, R8/XMM2, R9/XMM3	RTL (C) [19]	Caller
	vectorcall	[illegible]ows (Microsoft Visual C++)	RCX/XMM0, RDX/XMM1, R8/XMM2, R9/XMM3 + XMM0-XMM5/YMM0-YMM5	RTL (C)	Caller
	System V [illegible]7]	Solaris, Linux, BSD, OS X (GCC, Intel C++ Compiler)	RDI, RSI, RDX, RCX, R8, R9, XMM0–7	RTL (C)	Caller

Don't mess with calling conventions!

Source: Wikipedia

Symbols (in shared libraries)

- Are the “visible” part of your library / ABI.
- Are resolved by the runtime linker.
  - At load time of the program / shared library
  - Lazily during execution! ← **This can be a blessing or a curse!**
- Failure to do so results in termination of the program!

"Invisible" ABI parts

- Data structures:
  - Their *names* can appear in symbols.
  - Their *properties* won't - size, layout, alignment, ...
- Includes "compiler-generated" structures:
  - vtables
  - type_info (may be required for exception handling and `dynamic_cast`)

Safe & unsafe changes to a shared library

A look at your ABI …

- Think of it the “C” way: data + functions
  - data: all types (classes, enums, …) used by the client *and* the library
  - functions: all symbols exported by the library
- Binary compatibility is broken if:
  - data doesn’t match - memory corruption is coming…
  - symbols are missing - client program can’t run …

Generally safe changes

- Changing data used *only* by the client (e.g. inline, templates)
  - If your library can't see it, it can't break.
- Changing data used *only* inside the library (e.g. internal functions)
  - If your client can't see it, it can't break.
- Adding new exported symbols
  - An older client doesn't know they exist, so it can't break.

Things that are safe - “DO”s

1. Adding new non-virtual member functions
2. Adding new static functions or variables
    a. Please don’t use static variables in your API…
3. Adding a new class / enum / typedef / …
4. Adding a new enum value to an existing enum
    a. If the storage type doesn’t change!
5. Adding / changing inline functions
    a. Includes “un-inlining” a function
    b. Beware of behavioral changes!

Things that are safe - “DO”s (2)

6. Removing private non-virtual functions
   a. If they aren’t and have never been called by inline functions!
7. Removing private **static** members
   a. If they aren’t and have never been called by inline functions!
8. Adding / removing friend declarations
9. Extending reserved bit fields / memory areas
10. Exporting symbols that were previously not exported

(Note: Treat templates as inline functions or classes with only inline methods …)

Generally unsafe changes

- Changing data used by both the client and your library.
  - Leads to “misunderstandings”, killed kittens and segmentation faults...
- Removing/changing exported symbols
  - The library’s client won’t start or will terminate ...

Things that are unsafe - “DON’T”s

1. Unexporting / removing a previously exported function/class/variable/…
2. Changing the class hierarchy
3. Changing template arguments
4. Inlining a previously non-inline function
5. Changing function signatures
    a. Exception: changing default arguments

Things that are unsafe - “DON’T”s (2)

6. Adding/changing/removing/reordering/… virtual functions
   a. Just don’t touch them, ok? Changing the vtable is easy…
7. Changing non-static data members
   a. Don’t add/change/reorder them
   b. Exception: changing signedness (or similar)

Best practices for shared library development

Best practice 1: Control your API

- Things that aren't part of the libraries interface can't cause problems!
- "When in doubt, leave it out"
  - YouTube: Joshua Bloch: How To Design A Good API and Why it Matters
  - Once something is *public*, you can **never** get rid of it.
- Don't let the library's dependencies leak through
  - If you rely on library X in the public interface, changes to library X can break your API/ABI.
  - Best examples: Boost, STL

Best practice 2: Good API design

- Avoid variables - use getters and setters
  - You can never change variables in an API …
- Use version namespaces (“inline” if C++ 11 or later)
  - `inline namespace v1 { … }`
  - Allows different versions of your API to co-exist (even in the same header)
- Avoid macros. Always. Forever. Seriously. No really.
  - Worse than inline functions: cannot be versioned properly.
  - Well, maybe except for compiler directives (later…)

Best practice 3: Information hiding

- Avoid leaking internal details:
  - Separate “public” and “private” headers
  - Use a different namespace for internals (“detail”, “internal”, …)
  - Forward declare types that only “pass through” (best example: PIMPL)
- Avoid inline functions
  - including auto-generated constructors, destructors, …
- Use the PIMPL idiom (Private IMPLementation)

PIMP(L) my library!

```
// PIMPL example class
class Foo
{
public:
   Foo(/* constructor args */);
   ~Foo();

   bool bar(const char* param);
   /* other methods ... */

private:
   // private implementation - hidden
   class Implementation;
   // pointer to the private instance
   std::unique_ptr<Implementation> impl;
};
```

```
// sample implementation

Foo::Foo(/*...*/)
: impl(new Implementation(/*...*/))
{}

Foo::~Foo() {}


bool Foo::bar(const char* param)
{
   // method calls are usually
   // forwarded to the private
   // implementation
   return impl->bar(param);
}
```

PIMPL alternatives

- C-style interface :-(

```
struct Foo;
Foo* createFoo();
void destroyFoo(Foo* foo);
bool bar(Foo* foo, const char* param);
// more functions ...
```

- OOP factory

```
class IFoo {
public:
    virtual ~IFoo() = default;
    virtual bool bar(const char* param) = 0;
    // more functions ...
};
std::unique_ptr<IFoo> createFoo();
```

Best practice 4: Export control

- Internal functions/classes/... shouldn't be visible in the "binary" library.
  - With GCC:
    - compile with `-fvisibility=hidden`
    - mark "public" functions/classes/namespaces with `__attribute__((visibility("default")))`
  - With MSVC
    - Use `__declspec(dllexport)` / `__declspec(dllimport)` selectively

Export macros - example

```
#ifdef _WIN32
    #ifdef BUILDING_MYLIB
        #define MYLIB_PUBLIC __declspec(dllexport)
    #else
        #define MYLIB_PUBLIC __declspec(dllimport)
    #endif
    #define MYLIB_PRIVATE
#else
    #define MYLIB_PUBLIC  __attribute__((visibility("default")))
    #define MYLIB_PRIVATE __attribute__((visibility("hidden")))
#endif

MYLIB_PUBLIC void function(int a);
class MYLIB_PUBLIC Foo
{
    // …
};
```

Best practice 4: Export control (2)

- Use “static” declarations
- Use anonymous namespaces
- Be very minimalistic.
  - Adding is easy.
  - Removing is impossible.

Best practice 5: Be your own client

- Use your own API.
  - e.g. with unit / integration tests
- Have (automated) regression tests
  - Keep “old” binaries around for this - simple but very effective.
- Consider versioning the library file
  - `libfoo.so.`*`VERSION`*
  - But beware: Can two versions coexist in the same program?

Tools & links

Tools

- You shouldn’t need to manually check ABI compatibility …
- Have automated regression tests!
- Try the ABI Compliance Checker (disclaimer: haven’t used it myself)

Links

- This talk: https://github.com/dermojo/presentations
- Policies/Binary Compatibility Issues With C++ (KDE Community Wiki)
- Itanium C++ ABI *(this is hard-core)*
- Papers by Ulrich Drepper:
  - How To Write Shared Libraries
  - Good Practices in Library Design, Implementation, and Maintenance
- Visibility (GCC Wiki)