Version Guard

A Method to

Prevent C++ Library Users from Linking to the

Wrong Library Version

by Frank Sheeran

[1 Executive Summary 2](#_Toc188323221)

[2 What Problems are Prevented? 2](#_Toc188323222)

[3 What Scenarios can Cause These Problems? 2](#_Toc188323223)

[4 Running the Demo 3](#_Toc188323224)

[5 How It Works 11](#_Toc188323225)

[6 Integrating Into Your Library 13](#_Toc188323226)

[7 FAQ 13](#_Toc188323227)

[8 License 13](#_Toc188323228)

[9 Change Log 14](#_Toc188323229)

[10 Future Directions and TODOs 14](#_Toc188323230)

[11 Alternatives 15](#_Toc188323231)

# Executive Summary

* this paper shows how a C++ library can block its users from compiling with one version of the library's headers, but linking with another version's actual library
* mixing versions like that can cause dangerous malfunctions, mysterious crashes, and time-consuming debugging
* to use this method, add a small header and code file into the library; add a couple lines to all necessary headers; and add a rule to the Makefile
* library users see nothing different, need no changes
* works on Linux, Windows, and other Unix's
* works with static and dynamic libraries

# What Problems are Prevented?

Library users' object files will be based on the headers read at compilation. Various information is extracted from the headers, such as the code in in-line functions, and the offset of virtual functions and of each data member within the object. Between versions of a library, all this can change.

For virtual functions and data members: the order can change, they can be added and deleted, and base classes can do all these things too and have as much effect as if your own class changed. Your call to a virtual function may be virtual function #3 in the classes described by the headers you compile with, and if a different virtual function is #3 in the library you link with, you're calling an unexpected function with, likely, random and unexpected arguments. Data members have a similar risk: the compiler notes the offset of the data member in the object, but if the library you link knew the fields to be in different locations, members can be confused for each other and potentially corrupted.

In addition, data members can also change size and type. In C++, even if a member is protected or private, an object file can contain references to its offset when calling a method in the header that is in-lined.

Finally, any methods that are inlined will be code that probably will fail if it doesn't match the library you link with. Note the compiler may treat the inline keyword with disdain and can inline functions you don't expect.

# What Scenarios can Cause These Problems?

There are many variations but here are a few examples.

* 1. We compile our program with headers in Library A Version X. The library is upgraded but with its old file system name and directory, or an environment variable pointing to the library location is changed. We then link our program with the new Library A Version Y.
  2. We use the library as a dynamic library. We compile and link our program successfully, but then the dynamic library is upgraded but with its old file system name and directory, or an environment variable pointing to the library location is changed, and at runtime the executable attempts to link itself with the new library.
  3. We compile our program with Library A Version X. We link our program to a Library B that was compiled with a Library A Version Y.
  4. Library A is implemented entirely in the headers, so there's no library needed at link time. But some .cpp files in a project are compiled with Library A Version X while others are with Library A Version Y.

One can argue that none of these mistakes "should" happen, but the fact they shouldn't happen may not be of much comfort when they do.

# Running the Demo

## Overview

The distribution has a Hello World program, outputting the traditional greeting. However, the actual output is done by a library.

Both a static and dynamic library are created, and two versions of the program are compiled, one using the static library, and one the dynamic.

All the commands in this demo are in order in Demo.cmd and can be cut and paste in groups. However, it won't work to simply source this file, as the demo requires you to edit some files in a couple place.

We'll go through four builds of the libraries and corresponding binaries:

1. build the baseline distribution as shipped, to be familiar with what it is supposed to look like.
2. upgrade the version number, make an incompatible change, and show how the code can now crash
3. upgrade the version number, change the code again to put the VersionGuard in place
4. upgrade the version number, and show how the library now can no longer link, protecting you from this crash

The version numbers are simply examples and in fact can be freeform text.

## Version 1.0.1: Baseline

Make a fresh checkout and build it:

> make

if perl -e '$v="ivgdemo.1.0.1"; $v=~s/\W/\_/g;for(@ARGV){$b=$a=`cat $\_`;$a=~s/\w+(\s\*\/\\*Edited by Make\\*\/)/$v$1/g;$m="up-to-date";if($a ne$b){$c=1;$m="REGENERATING";open(C, ">$\_")||die;print C $a;close C||die}print" > $m: $\_\n"}exit !$c' VersionGuard.cpp VersionGuard.h; then \

printf "\n > VERSION UPDATED. Re-starting make.\n\n"; \

make ; \

exit 0;\

fi

> up-to-date: VersionGuard.cpp

> up-to-date: VersionGuard.h

mkdir -p vgdemo.1.0.1/lib

g++ -Wall -Wextra -g -fPIC -I. -c hellolib.cpp VersionGuard.cpp

ar rcs vgdemo.1.0.1/lib/libhellolib.a hellolib.o VersionGuard.o

g++ -Wall -Wextra -g -fPIC -I. -shared -o vgdemo.1.0.1/lib/libhellolib.so hellolib.cpp VersionGuard.cpp

mkdir -p vgdemo.1.0.1/bin

g++ -Wall -Wextra -g -fPIC -I. -o vgdemo.1.0.1/bin/hello\_static main.cpp vgdemo.1.0.1/lib/libhellolib.a

g++ -Wall -Wextra -g -fPIC -I. -o vgdemo.1.0.1/bin/hello\_dynamic main.cpp -Lvgdemo.1.0.1/lib -lhellolib

The statically-linked binary runs:

> vgdemo.1.0.1/bin/hello\_static

Hello World!

The dynamically-linked binary does *not* run because it can't find the library:

> vgdemo.1.0.1/bin/hello\_dynamic

vgdemo.1.0.1/bin/hello\_dynamic: error while loading shared libraries: libhellolib.so: cannot open shared object file: No such file or directory

When we set the LD\_LIBRARY\_PATH, the dynamically-linked binary now runs:

> ( setenv LD\_LIBRARY\_PATH vgdemo.1.0.1/lib ; vgdemo.1.0.1/bin/hello\_dynamic )

Hello World!

## Version 1.0.2: Incompatible API Change

Let's say the original code misinterpreted the meaning of an argument, and we've fixed the library to handle it correctly, which also necessitated a change in the application.

Please make the following edits:

* hellolib.cpp: change if ( bFlag ) to if ( !bFlag )
* main.cpp: change false to true
* Makefile: change VERSION to 1.0.2

Now let's rebuild it:

> make

if perl -e '$v="ivgdemo.1.0.2"; $v=~s/\W/\_/g;for(@ARGV){$b=$a=`cat $\_`;$a=~s/\w+(\s\*\/\\*Edited by Make\\*\/)/$v$1/g;$m="up-to-date";if($a ne$b){$c=1;$m="REGENERATING";open(C, ">$\_")||die;print C $a;close C||die}print" > $m: $\_\n"}exit !$c' VersionGuard.cpp VersionGuard.h; then \

printf "\n > VERSION UPDATED. Re-starting make.\n\n"; \

make ; \

exit 0;\

fi

> REGENERATING: VersionGuard.cpp

> REGENERATING: VersionGuard.h

> VERSION UPDATED. Re-starting make.

make[1]: Entering directory '/t/proj/VersionGuard'

if perl -e '$v="ivgdemo.1.0.2"; $v=~s/\W/\_/g;for(@ARGV){$b=$a=`cat $\_`;$a=~s/\w+(\s\*\/\\*Edited by Make\\*\/)/$v$1/g;$m="up-to-date";if($a ne$b){$c=1;$m="REGENERATING";open(C, ">$\_")||die;print C $a;close C||die}print" > $m: $\_\n"}exit !$c' VersionGuard.cpp VersionGuard.h; then \

printf "\n > VERSION UPDATED. Re-starting make.\n\n"; \

make ; \

exit 0;\

fi

> up-to-date: VersionGuard.cpp

> up-to-date: VersionGuard.h

mkdir -p vgdemo.1.0.2/lib

g++ -Wall -Wextra -g -fPIC -I. -c hellolib.cpp VersionGuard.cpp

ar rcs vgdemo.1.0.2/lib/libhellolib.a hellolib.o VersionGuard.o

g++ -Wall -Wextra -g -fPIC -I. -shared -o vgdemo.1.0.2/lib/libhellolib.so hellolib.cpp VersionGuard.cpp

mkdir -p vgdemo.1.0.2/bin

g++ -Wall -Wextra -g -fPIC -I. -o vgdemo.1.0.2/bin/hello\_static main.cpp vgdemo.1.0.2/lib/libhellolib.a

g++ -Wall -Wextra -g -fPIC -I. -o vgdemo.1.0.2/bin/hello\_dynamic main.cpp -Lvgdemo.1.0.2/lib -lhellolib

make[1]: Leaving directory '/t/proj/VersionGuard'

We can run the *new* binary with the *new* library OK:

> ( setenv LD\_LIBRARY\_PATH vgdemo.1.0.2/lib ; vgdemo.1.0.2/bin/hello\_dynamic )

Hello World!

If we try running the new binary with the old library, or old binary with the new library, whether dynamic or static linking, it now aborts. Note the commands here are mixing two versions to illustrate types of errors, and are highlighted in red

> ( setenv LD\_LIBRARY\_PATH vgdemo.1.0.2/lib ; vgdemo.1.0.1/bin/hello\_dynamic )

Abort (core dumped)

> ( setenv LD\_LIBRARY\_PATH vgdemo.1.0.1/lib ; vgdemo.1.0.2/bin/hello\_dynamic )

Abort (core dumped)

> g++ -Wall -Wextra -g -fPIC -I. -o vgdemo.1.0.2/bin/hello\_static main.cpp vgdemo.1.0.1/lib/libhellolib.a

> vgdemo.1.0.2/bin/hello\_static

Abort (core dumped)

This is the point of this entire project so let's write in a big font:

*These aborts illustrate the problem that VersionGuard is meant to eliminate. The aborts aren't just informative messages that you did something wrong. Instead, they are the program running… but dangerously crashing. Our simple program calls abort every time, but in a real, complicated system, problems may be intermittent and path-dependent. Problems may not arise in testing, but rather in production at customers. And, besides crashing, the problem could easily take the form of destroying data—legal records, financial records, and so on.*

## Version 1.0.3: Add VersionGuard

We'll now add the VersionGuard functionality to the hellolib.

Please make the following edits:

* hellolib.h: uncomment the line with HelloLib::VersionGuard
* Makefile: change VERSION to 1.0.3

Let's make it and test it:

> make

if perl -e '$v="ivgdemo.1.0.3"; $v=~s/\W/\_/g;for(@ARGV){$b=$a=`cat $\_`;$a=~s/\w+(\s\*\/\\*Edited by Make\\*\/)/$v$1/g;$m="up-to-date";if($a ne$b){$c=1;$m="REGENERATING";open(C, ">$\_")||die;print C $a;close C||die}print" > $m: $\_\n"}exit !$c' VersionGuard.cpp VersionGuard.h; then \

printf "\n > VERSION UPDATED. Re-starting make.\n\n"; \

make ; \

exit 0;\

fi

> REGENERATING: VersionGuard.cpp

> REGENERATING: VersionGuard.h

> VERSION UPDATED. Re-starting make.

make[1]: Entering directory '/t/proj/VersionGuard'

if perl -e '$v="ivgdemo.1.0.3"; $v=~s/\W/\_/g;for(@ARGV){$b=$a=`cat $\_`;$a=~s/\w+(\s\*\/\\*Edited by Make\\*\/)/$v$1/g;$m="up-to-date";if($a ne$b){$c=1;$m="REGENERATING";open(C, ">$\_")||die;print C $a;close C||die}print" > $m: $\_\n"}exit !$c' VersionGuard.cpp VersionGuard.h; then \

printf "\n > VERSION UPDATED. Re-starting make.\n\n"; \

make ; \

exit 0;\

fi

> up-to-date: VersionGuard.cpp

> up-to-date: VersionGuard.h

mkdir -p vgdemo.1.0.3/lib

g++ -Wall -Wextra -g -fPIC -I. -c hellolib.cpp VersionGuard.cpp

ar rcs vgdemo.1.0.3/lib/libhellolib.a hellolib.o VersionGuard.o

g++ -Wall -Wextra -g -fPIC -I. -shared -o vgdemo.1.0.3/lib/libhellolib.so hellolib.cpp VersionGuard.cpp

mkdir -p vgdemo.1.0.3/bin

g++ -Wall -Wextra -g -fPIC -I. -o vgdemo.1.0.3/bin/hello\_static main.cpp vgdemo.1.0.3/lib/libhellolib.a

g++ -Wall -Wextra -g -fPIC -I. -o vgdemo.1.0.3/bin/hello\_dynamic main.cpp -Lvgdemo.1.0.3/lib -lhellolib

make[1]: Leaving directory '/t/proj/VersionGuard'

> vgdemo.1.0.3/bin/hello\_static

Hello World!

> ( setenv LD\_LIBRARY\_PATH vgdemo.1.0.3/lib ; vgdemo.1.0.3/bin/hello\_dynamic )

Hello World!

We can use the nm utility to see the symbols in the library that have been added. Here we see the binary has the symbol U, meaning undefined in the binary and must be supplied by the library, while the symbol is present in the library. (A similar U entry would be found in main.o for the static version.)

> nm -C vgdemo.1.0.3/bin/hello\_dynamic | grep vgdemo

U HelloLib::ivgdemo\_1\_0\_3

> nm -C vgdemo.1.0.3/lib/libhellolib.so | grep vgdemo

000000000000402c B HelloLib::ivgdemo\_1\_0\_3

Exercise for the reader: do you think you can link the 1.0.3 binaries with the 1.0.2 libraries? What about 1.0.2 binaries with 1.0.3 libraries? Why or why not?

Answer: the 1.0.3 binaries have references to the undefined VersionCheck symbol HelloLib::ivgdemo\_1\_0\_3, so they cannot link with older libraries. Sadly, the older binaries have no such undefined symbol, and allow the linker to link to potentially incompatible newer libraries.

## Version 1.0.4: Update Version with VersionGuard

We'll now update the library version number. We can pretend we might be making an incompatible change, but don't need to actually make one. Please make the following edit:

* Makefile: change VERSION to 1.0.4

Let's make it and test it:

> make

g++ -Wall -Wextra -g -fPIC -I. -o vgdemo.1.0.4/bin/hello\_static main.cpp vgdemo.1.0.3/lib/libhellolib.a

if perl -e '$v="ivgdemo.1.0.4"; $v=~s/\W/\_/g;for(@ARGV){$b=$a=`cat $\_`;$a=~s/\w+(\s\*\/\\*Edited by Make\\*\/)/$v$1/g;$m="up-to-date";if($a ne$b){$c=1;$m="REGENERATING";open(C, ">$\_")||die;print C $a;close C||die}print" > $m: $\_\n"}exit !$c' VersionGuard.cpp VersionGuard.h; then \

printf "\n > VERSION UPDATED. Re-starting make.\n\n"; \

make ; \

exit 0;\

fi

> REGENERATING: VersionGuard.cpp

> REGENERATING: VersionGuard.h

> VERSION UPDATED. Re-starting make.

make[1]: Entering directory '/t/proj/VersionGuard'

if perl -e '$v="ivgdemo.1.0.4"; $v=~s/\W/\_/g;for(@ARGV){$b=$a=`cat $\_`;$a=~s/\w+(\s\*\/\\*Edited by Make\\*\/)/$v$1/g;$m="up-to-date";if($a ne$b){$c=1;$m="REGENERATING";open(C, ">$\_")||die;print C $a;close C||die}print" > $m: $\_\n"}exit !$c' VersionGuard.cpp VersionGuard.h; then \

printf "\n > VERSION UPDATED. Re-starting make.\n\n"; \

make ; \

exit 0;\

fi

> up-to-date: VersionGuard.cpp

> up-to-date: VersionGuard.h

mkdir -p vgdemo.1.0.4/lib

g++ -Wall -Wextra -g -fPIC -I. -c hellolib.cpp VersionGuard.cpp

ar rcs vgdemo.1.0.4/lib/libhellolib.a hellolib.o VersionGuard.o

g++ -Wall -Wextra -g -fPIC -I. -shared -o vgdemo.1.0.4/lib/libhellolib.so hellolib.cpp VersionGuard.cpp

mkdir -p vgdemo.1.0.4/bin

g++ -Wall -Wextra -g -fPIC -I. -o vgdemo.1.0.4/bin/hello\_static main.cpp vgdemo.1.0.4/lib/libhellolib.a

g++ -Wall -Wextra -g -fPIC -I. -o vgdemo.1.0.4/bin/hello\_dynamic main.cpp -Lvgdemo.1.0.4/lib -lhellolib

make[1]: Leaving directory '/t/proj/VersionGuard'

Now we see that the newest release of both the statically- and dynamically-linked binaries works correctly with the newest release of the library, as required and expected:

> vgdemo.1.0.4/bin/hello\_static

Hello World!

> ( setenv LD\_LIBRARY\_PATH vgdemo.1.0.4/lib ; vgdemo.1.0.4/bin/hello\_dynamic )

Hello World!

However, we now see, thanks to the VersionGuard symbol, linking across versions is no longer allowed. The first two cases show the dynamic linking will fail at run time, old to new or new to old. The third case fails at link time. Note again, the commands here are mixing two versions to illustrate types of errors, and are highlighted in red. This time, the links all fail and we're protected from malfunctions.

LIBRARY

> ( setenv LD\_\_PATH vgdemo.1.0.4/lib ; vgdemo.1.0.3/bin/hello\_dynamic )

vgdemo.1.0.3/bin/hello\_dynamic: symbol lookup error: vgdemo.1.0.3/bin/hello\_dynamic: undefined symbol: \_ZN8HelloLib13ivgdemo\_1\_0\_3E

> ( setenv LD\_LIBRARY\_PATH vgdemo.1.0.3/lib ; vgdemo.1.0.4/bin/hello\_dynamic )

vgdemo.1.0.4/bin/hello\_dynamic: symbol lookup error: vgdemo.1.0.4/bin/hello\_dynamic: undefined symbol: \_ZN8HelloLib13ivgdemo\_1\_0\_4E

> g++ -Wall -Wextra -g -fPIC -I. -o vgdemo.1.0.4/bin/hello\_static main.cpp vgdemo.1.0.3/lib/libhellolib.a

/usr/bin/ld: /tmp/ccbuN8w3.o: in function `HelloLib::VersionGuard::VersionGuard()':

/t/proj/VersionGuard/main.cpp:27:(.text.\_ZN8HelloLib12VersionGuardC2Ev[\_ZN8HelloLib12VersionGuardC5Ev]+0xb): undefined reference to `HelloLib::ivgdemo\_1\_0\_4'

/usr/bin/ld: /t/proj/VersionGuard/main.cpp:27:(.text.\_ZN8HelloLib12VersionGuardC2Ev[\_ZN8HelloLib12VersionGuardC5Ev]+0x17): undefined reference to `HelloLib::ivgdemo\_1\_0\_4'

collect2: error: ld returned 1 exit status

Again, this is the point of the paper, so let's get the big font out one more time.

*These error messages illustrate the Ve­­­­rsionGuard solution. You now have a technical guarantee that if you are mixing different versions of headers and libraries, a static-linked application cannot even be created, and a dynamic-linked application cannot even start. It is not intermittent. It is not path-dependent. It doesn’t depend on how you tested. It is a technical certainty. Not only is the user protected from a software developer error, but also from a user error of the form of setting the wrong library path.*

# How It Works

The solution, in broad strokes, is to make every object file compiled with any header from Library A, Version X, have a reference to an external int called iA\_X. We'll call iA\_X the "Guard Variable."

Library A Version X defines iA\_X. So, when linking such object files to the correct library, the external reference is satisfied. Linking succeeds and an executable program is created. Or, for applications using shared libraries, the execution can start.

Another version of Library A, Version Y, will not have this symbol, so any dangerous attempt to link will fail.

The author of an application or other library dependent on Library A need do nothing to avail themselves of this check.

This will protect against:

* building any part of your application with Version X headers, then linking to Version Y
* linking to any Library B that was in turn compiled against Library A Version Y

## Version Guard Header

We add a header VersionGuard.h to the library with an external declaration of the guard variable.

Further, to assure that C++ compilers cannot optimize this reference away, we define a tiny class with a constructor that increments the variable.

The functional code in its entirety is:

#pragma once

namespace MyLib {

extern int iMyLib\_MyVersion /\*Edited by Make\*/;

class VersionGuard {

public:

inline VersionGuard() {

iMyLib\_MyVersion /\*Edited by Make\*/ ++;

}

};

}

## Version Guard Source

The functional code in VersionGuard.cpp in its entirety is:

#include <MyLib/VersionCheck.h>

int MyLib::iMyLib\_MyVersion /\*Edited by Make\*/ = 0;

## Other Library Headers

Every other header in the library (example: MyHeader.h) will need the following added:

#include <MyLib/VersionGuard.h>

static HelloLib::VersionGuard versionguardMyHeader;

There is almost never a compelling reason to define, not just declare, storage in a header, and it's even odder to do this on behalf of code the header (and library) author isn't in charge of, familiar with, or likely even aware of. However, other ways to accomplish the goal do not come to mind. This step isn't taken lightly and feedback as potential and actual problems arising, and possible alternatives, are quite welcome.

## Makefile

The sample implementation will re-generate the Version Guard Header and Source *if and only if* the version number defined in the header also changes. This is accomplished via the following mechanism.

We define a DISTRIBUTION and VERSION, then compose a DISTRIBUTION\_NAME from them. (Only the latter is actually referred to; this just illustrative usage.)

We make a fake target called checkversion. We inform gmake that this is not a real file but rather a recipe to run every time, by making it a dependency of the meta-target .PHONY.

Checking and *possibly* updating the VersionGuard files is done with a perl command (in effect, a small script on its command line) that performs the following. For each file named as an argument, it reads the file into a before and after variable. The after variable then has the alphanumeric-and-underscore token before the sentinel comment /\*Edited by Make\*/ changed to the new version number. *If and only if* this actually changes the variable, is the file written back out. At the end, it outputs a courtesy message that files have or have not been regenerated, and returns an status code. Under normal usage, the Header and Source is not regenerated, minimizing the work gmake must do.

Should the files be changed, however, there's a problem. gmake has already determined the nest of dependencies, and *will not* note the change of the newly-written files and recalculate its work. So, we have the perl command give an exit code indicating a change was made. This is checked by the sh interpreter. If a change was made, gmake is rerun by sh. Once this finishes, the initial gmake would normally continue and potentially duplicate many make steps. Therefore, we have sh tell the gmake running it to exit, by with the exit 0; command.

# Integrating Into Your Library

Add VersionGuard.h and VersionGuard.cpp to your library project. Rename and edit them as allowed by the license.

Add the special commands to the Makefile to run the given script. Again, rename, and edit.

Update the library headers to define the VersionGuard variable. Be careful, as there's no mechanism to detect if you skip this for some or all files, and if you skip this, any object files compiled by library users that do not include a header with this variable will not have protection. Note that you may not need this variable in every header: if your library is such that a given basic header is always included by all users of the library, for instance, then you may be fine with putting the VersionGuard only in that file. Still, it shouldn’t hurt to have more, and it will future-proof you against changes.

# FAQ

Q: But many new library releases simply add features or fix bugs and thus will be compatible, no? Yet you're removing the possibility of running in such cases.

A: The version number discussed here should be a version number that changes only when incompatible changes are made. So, the scenarios you describe wouldn't involve changing this library version number.

Technically, this number needn't be the public version of your software. However, it will be more confusing if it isn't, as the library user will see the version number used in an error message if they manage to incorrectly link the library.

Note that even when we are sure you are not introducing incompatible changes… we may be wrong. The safest strategy would be to lock every version's libraries to its headers, though this obviously can come at some cost of convenience.

Q: I write code based on a library that uses Version Guard, but I need to disable the check somehow. How?

A: If you absolutely have to, you can add a definition of MyLib::iMyLib\_MyVersion to the code that is using MyLib. This seems like a recipe for disaster, but in an emergency or for certain testing it is good to have a way to do this.

# License

In case of disagreement, the file LICENSE contains the definitive version, but its contents at this moment are reproduced here for your convenience.

Copyright 2025 Frank Sheeran.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

# Change Log

## Version 1.0

Initial Release.

# Future Directions and TODOs

The Makefile is a rush job and should be cleaned and tested, but seems to work in the situations covered in the demo.

The VersionGuard should work fine on Windows as well. An example should be provided of that.

The alternatives section should be more detailed.

The symbol iMyLib\_MyVersion shouldn't need to have the library name version since it's namespaced.

# Alternatives

### Suffixing All Symbols

The GNU C Library (glibc) uses versioned symbols to differentiate between different ABI versions. Symbols include a version suffix, such as \_\_libc\_start\_main@GLIBC\_2.2.5.

This seems to require non-standard functionality that is beyond the reach of regular software authors.

### Compile-Time Checks

Libraries like Boost and Qt include compile-time checks in headers to ensure that the library version matches the one used during compilation.

#if BOOST\_VERSION != EXPECTED\_BOOST\_VERSION

#error "Boost version mismatch!"

#endif

#if OPENSSL\_VERSION\_NUMBER < 0x10100000L

#error "OpenSSL version too old!"

#endif

These rely on the programmer doing something specific to check versions, however. The VersionGuard system is more automated. These also may only catch compilation errors, not linking errors.

### Shared Library SONAMEs

Shared libraries on Linux use the SONAME (Shared Object Name) mechanism to enforce ABI compatibility at runtime:

When a shared library (e.g., libMyLib.so.3) is created, its SONAME is embedded into the binary during linking.

At runtime, the dynamic linker ensures the correct SONAME is available, failing if it's missing or incompatible.

gcc -shared -Wl,-soname,libMyLib.so.3 -o libMyLib.so.3.0.17 versioncheck.o

When the binary is linked, the SONAME (libMyLib.so.3) is recorded, ensuring the correct library version is loaded.

However, this fails if the SONAME isn't manually updated correctly. With VersionGuard, by design, the version number that controls the output directory is embedded in all object files. If you try to create a new distribution directory, you automatically update the version number checked.

### CMake Configurations

Modern C++ libraries often use CMake's find\_package and target\_link\_libraries mechanisms to enforce version compatibility:

A library can define a CMake configuration file (LibraryConfig.cmake) specifying the required version.

Users specify the version they expect:

find\_package(Library 3.0.17 REQUIRED)

If the library version doesn't match, CMake generates an error during the configuration step.

The drawbacks include1) the user has to remember to do this and 2) it only works with Cmake.

### Microsoft-Specific Techniques

On Windows, some libraries embed version strings or GUIDs into object files to enforce compatibility. Libraries embed a GUID or version string in headers, and any mismatches produce a linker error. For instance, Windows Runtime (WinRT) uses metadata to describe interfaces, ensuring compatibility between components.

The drawback is that it only works on Windows.