

Compiler version/C++ standard juggling: a review

Authored by Mark Final, with input and contributions from the VFX Reference Platform Committee

[The question](#)

[TLDR](#)

[Platform summary](#)

[Linux](#)

[Mac](#)

[Windows](#)

[What doesn't feel safe/correct?](#)

[Linux](#)

[libstdc++ and RHEL Developer Toolsets](#)

[The DTS linker scripts for libstdc++](#)

[Symbol visibility in libstdc++_nonshared.a](#)

[Test case](#)

[Thought experiments](#)

[Case 1: Use same compiler and C++ std, vendor and client](#)

[Case 2: Client is on older compiler version than vendor](#)

[Case 3: Vendor uses GCC11, client uses GCC13, same C++ standard](#)

[Case 4: Vendor uses GCC11, C++17, client uses GCC13, C++20](#)

[libc_nonshared.a unrelated](#)

[macOS](#)

[libc++](#)

[libc++ feature availability](#)

[Future Xcode usage](#)

[C++20](#)

[Windows](#)

[Additional runtime requirements with new features](#)

[Brief notes on deprecated language/library features & "hidden" compiler changes](#)

[References](#)

The question

If VFX library and product vendors continue to use VFX 25 draft spec compiler versions and C++ standards as they stand today, what support/issue response might there be if clients use a newer compiler and/or new C++ standard in their pipelines, plugins and tools?

TLDR

From this investigation, to get a 'feel' for how much works, there seems to be a fair amount of flexibility across all toolchains and platforms to be flexible on compiler and C++20 versions for end users.

It's likely that updating compiler versions on some OS', while retaining C++17 for the ASWF libraries and vendor apps will allow customer extensions to target C++20.

While VFX24 compilers may not all offer complete C++20 support, the same gist may apply, in that vendors targeting C++17 does not block or preclude C++20 use.

Deprecated C++XX and compiler features may throw up corner cases. The case could be made that vendors could publish the compiler + C++XX combinations they've tested with in addition to what the VFX Platform calls for to forewarn or reduce the need for work-arounds.

Platform summary

Linux

Before any testing was done, this is what 'sounded about right' based upon reading around (see references):

1. the same compiler version everywhere, regardless of C++ standard
2. span across more compiler versions with the caveats of
 - a. only using stable C++ standard features available in all versions
 - b. being wary of ABI changes across GCC versions, or compiler flags that change ABI
3. always use the libstdc++ associated with the most recent compiler version in use

After testing

- When using Developer Toolsets, additional compiler/C++ standard support that doesn't exist on the original host distro version is statically linked into binaries (and made globally available), so there are fewer requirements on target distros to be able to run
 - Assuming the target distro is at least parity in the libstdc++ runtime support of the build hardware (more recent would generally be fine)

Mac

We are bound by what Apple decides goes into the libc++ for each version of macOS we're interested in supporting:

1. The choice of compiler version defines the possibility of using a feature
2. The choice of minimum deployment target (macOS versions) defines the availability of runtime support from libc++ of features used, and will fail at compile time if not available for the selected oldest OS version

Windows

1. Additional C++ support that isn't in baseline runtime support, or has an ABI breaking change in already published DLLs, are introduced as additional DLLs
2. The requirement is that all intended machines have all the required DLLs from all binaries deployed in order to run. This can either be achieved by installing to system locations (redist packages) or by including DLLs in a searchable location beside your software.

What doesn't feel safe/correct?

- Compile and use your own GCC ecosystem - you'll have to distribute all the runtime support, and try not to break a system
- Statically link all runtime libraries - will bloat all of your binaries more than what is happening with the DTS, and would have to deal with all thirdparty dynamic binaries too
- Don't change the ABI compiler flags away from their default values

Linux

libstdc++ and RHEL Developer Toolsets

The following information is key to understanding how the developer toolsets (DTS) can offer newer C++ compilers and newer C++ standard features, across major versions of their distro! 🐧 [So what's the point here? \[LWN.net\]](#)

The magic is that devtoolset (gcc-toolset in RHEL 8) installs a linker script as its "libstdc++.so" that pulls in both the libstdc++.so.6 provided by the base system and a separate libstdc++_nonshared.a that contains those symbols not provided by the base system's libstdc++.so.6.

The DTS linker scripts for libstdc++

As described above, the toolset libstdc++ redirects to both the host system libstdc++.so and a static library.

On a Rocky 8 system, for GCC11

```
Unset
more
/opt/rh/gcc-toolset-11/root/usr/lib/gcc/x86_64-redhat-linux/11
/libstdc++.so
/* GNU ld script
   Use the shared library, but some functions are only in
   the static library, so try that secondarily. */
OUTPUT_FORMAT(elf64-x86-64)
```

```
INPUT ( /usr/lib64/libstdc++.so.6 -lstdc++_nonshared )
```

Similarly for GCC13

```
Unset
more
/opt/rh/gcc-toolset-13/root/usr/lib/gcc/x86_64-redhat-linux/13
/libstdc++.so
/* GNU ld script
   Use the shared library, but some functions are only in
   the static library, so try that secondarily. */
OUTPUT_FORMAT(elf64-x86-64)
INPUT ( /usr/lib64/libstdc++.so.6 -lstdc++_nonshared )
```

The static library contains top-up symbols from the newer compiler features in the DTS that don't exist in the host's system shared libraries. In the cases above, running `nm` on the static libraries shows that symbols such as `std::filesystem` are present. If you check the system `libstdc++` shared library of the host, there are no symbols for `std::filesystem`

```
Unset
more /etc/redhat-release
Rocky Linux release 8.9 (Green Obsidian)

nm -D /usr/lib64/libstdc++.so.6 | c++filt | grep filesystem |
wc -l
0
```

On Rocky 9 though, `std::filesystem` symbols have been published in the shared system library:

```
Unset
more /etc/redhat-release
Rocky Linux release 9.4 (Blue Onyx)

nm -D /usr/lib64/libstdc++.so.6 | c++filt | grep filesystem |
wc -l
```

Diffing the output of `nm` between GCC11 and GCC13 `libstdc++_nonshared.a` shows that it's

- 95% additions in the GCC13 library
- with a few modifications, to what look like implementation details, such as non-public APIs, and in anonymous namespaces

The linker scripts are independent of compiler/linker flags, so apply for all supported C++ standards in the activated DTS, such that the static library for the compiler version must contain all supported features above and beyond that in the base system.

As the additional symbols are linked into binaries, this opens up the wider possibility of running them on other non-RHEL-like distros. The one constraint is to have a `libstdc++.so` in the target distro that supports at least what the build OS (RHEL8) does. Newer symbols used by the binaries should be statically linked in, rather than needing to be sought in the system runtime libraries.

Symbol visibility in `libstdc++_nonshared.a`

Picking on a few functions in the static library, and using `nm` shows

```
Unset
0000000000000000 T
std::filesystem::recursive_directory_iterator::disable_recursion_pending()
0000000000000000 T
std::filesystem::recursive_directory_iterator::pop(std::error_code&)
0000000000000000 T std::filesystem::recursive_directory_iterator::pop()
```

The upper case T prefix means that the visibility is [global and not hidden](#).

Test case

Consider these files:

`lib.cpp`

```
C/C++
#include <filesystem>
#include <iostream>

__attribute__((visibility ("default"))) void current()
{
    std::cout << std::filesystem::current_path() << std::endl;
}
```

`main.cpp`

```
C/C++
extern void current();

int main()
{
    current();
    return 0;
}
```

CMakeLists.txt (note that I explicitly turn off symbol visibility in the shared lib)

```
Unset
cmake_minimum_required(VERSION 3.26 FATAL_ERROR)
project(fstest)

add_library(fstestlib SHARED lib.cpp)
set_target_properties(fstestlib PROPERTIES
CXX_VISIBILITY_PRESET hidden)
target_compile_features(fstestlib PRIVATE cxx_std_17)

add_executable(fstest main.cpp)
target_link_libraries(fstest PRIVATE fstestlib)
set_target_properties(fstest PROPERTIES INSTALL_RPATH
$ORIGIN/../lib)
target_compile_features(fstest PRIVATE cxx_std_17)

install(TARGETS fstestlib fstest)
```

Build with

```
Unset
cmake -GNinja -Bbuild -S. -DCMAKE_INSTALL_PREFIX=install
ninja -C build install
```

Running gives

```
Unset
./install/bin/fstest
```

```
"/opt/stapp/src/fs"
```

and the shared library has plenty of global `std::filesystem` symbols

```
Unset
nm -D ./install/lib/libfstestlib.so | c++filt | grep filesystem
000000000001a480 T std::filesystem::__cxx11::filesystem_error::what() const
000000000001be00 T std::filesystem::__cxx11::filesystem_error::path1() const
000000000001be10 T std::filesystem::__cxx11::filesystem_error::path2() const
0000000000023ac0 T std::filesystem::__cxx11::directory_iterator::operator*() const
0000000000023bd0 T
std::filesystem::__cxx11::recursive_directory_iterator::recursion_pending() const
0000000000023b70 T std::filesystem::__cxx11::recursive_directory_iterator::depth()
const
0000000000023b60 T std::filesystem::__cxx11::recursive_directory_iterator::options()
const
0000000000023be0 T std::filesystem::__cxx11::recursive_directory_iterator::operator*()
const
...
```

Thought experiments

Assume an application ships an SDK including some shared libraries for plugin authors to use. Use of a DTS is assumed.

Case 1: Use same compiler and C++ std, vendor and client

This is what we do today for active VFX reference platforms. There aren't known issues with this.

If we move to GCC13 and C++20 as vendors, then clients can use that too.

Case 2: Client is on older compiler version than vendor

This is not supported. An example of this is the hard break from VFX23 between CentOS7 and RHEL8/9.

Case 3: Vendor uses GCC11, client uses GCC13, same C++ standard

The application shared libraries will contain any features from GCC11 beyond the baseline system support. The plugin will link against the application shared libraries, and the GCC13 static lib, so can use all the features exposed in GCC13, and can deploy to any compatible distro.

Case 4: Vendor uses GCC11, C++17, client uses GCC13, C++20

The application shared libraries contain all the features from GCC11 and using C++17 interfaces (because that's what the vendors tested against internally). Plugins can use GCC13 and C++20 features that their compiler supports. The C++17 interfaces will be consumable, as the assumption is that stable features in GCC11 will be at least stable in GCC13. The GCC13 static library will be used to provide the symbols for any features used in the plugin that weren't available in the vendor shared libraries. Plugins can be deployed to any compatible distro.

libc_nonshared.a unrelated

In RHEL systems there is a `/usr/lib64/libc_unshared.a` as well. However, this appears to be unrelated to DTS', and in fact used by the system `/usr/lib64/libc.so` as that is a linker script

Unset

```
more /usr/lib64/libc.so
/* GNU ld script
   Use the shared library, but some functions are only in
   the static library, so try that secondarily. */
OUTPUT_FORMAT(elf64-x86-64)
GROUP ( /lib64/libc.so.6 /usr/lib64/libc_nonshared.a
AS_NEEDED ( /lib64/ld-linux-x86-64.so.2 ) )
```

The symbols within do look stat related, that is also suggested in this Stack Overflow question, [What is the purpose of libc_nonshared.a?](#)

macOS

libc++

The libc++ dylib is not even visible on modern macOS versions, as it's in the dynamic shared library cache. This is baked into the system for extra security, and not discoverable as a normal file on the filesystem.

Mac builds require a minimum deployment target, the oldest macOS version intended to run your software.

Xcode SDKs have 'availability' macros, for both C++ and ObjectiveC/C++, which compares against the minimum deployment target, and will error when support is not available in the designated target for the code you write. See example below with `std::filesystem`.

The Xcode SDK headers know which is the oldest macOS version that runtime support exists for certain features.

So any missing support for the intended target OS is known at compile time.

libc++ feature availability

This is controlled by the minimum deployment target setting, the oldest version of macOS that is intended to run the software.

An example:

main.cpp

```
C/C++
#include <filesystem>
#include <iostream>

int main()
{
    std::cout << std::filesystem::current_path() << std::endl;
    return 0;
}
```

CMakeLists.txt, note the old deployment target

```
Unset
cmake_minimum_required(VERSION 3.26 FATAL_ERROR)
project(fs_libcpp)

set(CMAKE_OSX_DEPLOYMENT_TARGET 10.13)

add_executable(fslibcpp main.cpp)
target_compile_features(fslibcpp PRIVATE cxx_std_17)
```

To build

```
Unset
cmake -GNinja -Bbuild -S.
ninja -C build -v
```

which fails

```
Unset
ninja: Entering directory `build'
[1/2]
/Applications/Xcode13.2.1.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/c++
-std=gnu++17 -isysroot
/Applications/Xcode13.2.1.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX12
.1.sdk -mmacosx-version-min=10.13 -MD -MT CMakeFiles/fslibcpp.dir/main.cpp.o -MF
CMakeFiles/fslibcpp.dir/main.cpp.o.d -o CMakeFiles/fslibcpp.dir/main.cpp.o -c
/Users/mark.final/dev/Tests/older_libc++/main.cpp
```

```

FAILED: CMakeFiles/fslibcpp.dir/main.cpp.o
/Applications/Xcode13.2.1.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/c++
-std=gnu++17 -isysroot
/Applications/Xcode13.2.1.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX12
.1.sdk -mmacosx-version-min=10.13 -MD -MT CMakeFiles/fslibcpp.dir/main.cpp.o -MF
CMakeFiles/fslibcpp.dir/main.cpp.o.d -o CMakeFiles/fslibcpp.dir/main.cpp.o -c
/Users/mark.final/dev/Tests/older_libc++/main.cpp
/Users/mark.final/dev/Tests/older_libc++/main.cpp:6:35: error: 'current_path' is unavailable:
introduced in macOS 10.15
    std::cout << std::filesystem::current_path() << std::endl;
                                ^
/Applications/Xcode13.2.1.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX12
.1.sdk/usr/include/c++/v1/filesystem:1745:39: note: 'current_path' has been explicitly marked
unavailable here
inline _LIBCPP_INLINE_VISIBILITY path current_path() {

```

This error comes from clang itself, because of this in the Xcode SDK

```

C/C++
#   define _LIBCPP_AVAILABILITY_FILESYSTEM_PUSH \
    _Pragma("clang attribute
push(__attribute__((availability(macosx,strict,introduced=10.15))),
apply_to=any(function,record))") \
    _Pragma("clang attribute
push(__attribute__((availability(ios,strict,introduced=13.0))),
apply_to=any(function,record))") \
    _Pragma("clang attribute
push(__attribute__((availability(tvos,strict,introduced=13.0))),
apply_to=any(function,record))") \
    _Pragma("clang attribute
push(__attribute__((availability(watchos,strict,introduced=6.0))),
apply_to=any(function,record))")

```

Future Xcode usage

The Xcode SDKs will be aware when libc++ support became available in macOS versions, so it depends what features you want to use, and when they were made available at runtime. Plugin developers using new features than the vendor supplied libraries may then have a more restrictive minimum deployment target than the vendors, but that's at their discretion. The choice of toolchain opens up the possibility of new features, but it is the minimum deployment target that sets the final call on whether you can use them, and that is an OS version choice, rather than compiler.

What the VFX reference platform reports as the minimum deployment target doesn't particularly track what the compilation toolchain supports, but instead loosely tracks that Apple do release a new OS version each year, and eventually older versions will become obsolete. Additionally, new hardware - Apple silicon - has set a new minimum bar for the oldest version supported.

C++20

I've tested a number of features from C++20, both core features and library, in Xcode 15.2 on Ventura, and have so far not encountered any need for a significantly more recent macOS version.

Windows

From VisualStudio 2015, deployment of runtime support libraries was changed to

- a universal C runtime library part of the system (*ucrt.dll)
- VisualStudio specific versions of library components (vcruntime*.dll)
- concurrency runtime (concr*.dll)

The C++ runtime library differs in version across VisualStudio, e.g. version 12 of the C++ runtime library is msvcp120.dll, and version 14 is msvcp140.dll.

Expansions to a library may be spread across multiple extra DLLs, known as dot libraries.

For example, some functionality in the standard library released in Visual Studio 2017 version 15.6 was added into msvcp140_1.dll, to preserve the ABI compatibility of msvcp140.dll. If you use Visual Studio 2017 version 15.6 (toolset 14.13) or later, you may need to locally deploy both these dot libraries and the main library. These separate dot libraries will eventually be added to the base library, when the ABI changes.

Documentation discusses the problems associated with supporting multiple CRT versions, and how to mitigate them. [C runtime \(CRT\) and C++ standard library \(STL\) lib files](#)

Additional runtime requirements with new features

Consider this example as a baseline reference. Microsoft (R) C/C++ Optimizing Compiler Version 19.36.32537 for x64 is used.

main.cpp

```
C/C++
#include <iostream>
#include <filesystem>
int main()
{
    std::cout << std::filesystem::current_path() << std::endl;
    return 0;
}
```

CMakeLists.txt

```
Unset
cmake_minimum_required(VERSION 3.26 FATAL_ERROR)
project(cppstd)
add_executable(test main.cpp)
```

```
target_compile_features(test PRIVATE cxx_std_27)
install(TARGETS test)
```

And build with

```
Unset
cmake -GNinja -Bbuild -S. -DCMAKE_INSTALL_PREFIX=install
-DCMAKE_BUILD_TYPE=Release
ninja -Cbuild -v install
```

The runtime dependencies can be analysed with

```
Unset
dumpbin /DEPENDENTS install\bin\test.exe
Dump of file install\bin\test.exe
File Type: EXECUTABLE IMAGE
Image has the following dependencies:
  MSVCP140.dll
  VCRUNTIME140.dll
  VCRUNTIME140_1.dll
  api-ms-win-crt-runtime-l1-1-0.dll
  api-ms-win-crt-heap-l1-1-0.dll
  api-ms-win-crt-locale-l1-1-0.dll
  api-ms-win-crt-math-l1-1-0.dll
  api-ms-win-crt-stdio-l1-1-0.dll
  KERNEL32.dll
```

This shows that the executable depends on the original vcruntime140 but also an extended version vcruntime140_1.

If the source is then extended to use C++20 co-routines, e.g. with code borrowed from [here](#),

```
C/C++
#include <iostream>
#include <filesystem>

#include <coroutine>
#include <iostream>
```

```

#include <stdexcept>
#include <thread>

auto switch_to_new_thread(std::jthread& out)
{
    struct awaitable
    {
        std::jthread* p_out;
        bool await_ready() { return false; }
        void await_suspend(std::coroutine_handle<> h)
        {
            std::jthread& out = *p_out;
            if (out.joinable())
                throw std::runtime_error("Output jthread parameter not
empty");
            out = std::jthread([h] { h.resume(); });
            // Potential undefined behavior: accessing potentially destroyed
*this
            // std::cout << "New thread ID: " << p_out->get_id() << '\n';
            std::cout << "New thread ID: " << out.get_id() << '\n'; // this
is OK
        }
        void await_resume() {}
    };
    return awaitable{&out};
}

struct task
{
    struct promise_type
    {
        task get_return_object() { return {}; }
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }
        void return_void() {}
        void unhandled_exception() {}
    };
};

task resuming_on_new_thread(std::jthread& out)
{
    std::cout << "Coroutine started on thread: " <<
std::this_thread::get_id() << '\n';
    co_await switch_to_new_thread(out);
    // awaiter destroyed here
    std::cout << "Coroutine resumed on thread: " <<
std::this_thread::get_id() << '\n';
}

```

```
int main()
{
    std::cout << std::filesystem::current_path() << std::endl;
    std::jthread out;
    resuming_on_new_thread(out);
    return 0;
}
```

and update the CMakeLists.txt to use C++20 (since otherwise you get warning STL4038: The contents of <coroutine> are available only with C++20 or later or /await:strict).

```
Unset
cmake_minimum_required(VERSION 3.26 FATAL_ERROR)
project(cppstd)

add_executable(test main.cpp)
target_compile_features(test PRIVATE cxx_std_20)

install(TARGETS test)
```

now builds and the dependencies are updated to

```
Unset
dumpbin /DEPENDENTS install\bin\test.exe
Microsoft (R) COFF/PE Dumper Version 14.36.32537.0
Copyright (C) Microsoft Corporation. All rights reserved.

Dump of file install\bin\test.exe

File Type: EXECUTABLE IMAGE

Image has the following dependencies:

    MSVCP140.dll
    MSVCP140_ATOMIC_WAIT.dll
```

```
VCRUNTIME140.dll
VCRUNTIME140_1.dll
api-ms-win-crt-runtime-l1-1-0.dll
api-ms-win-crt-heap-l1-1-0.dll
api-ms-win-crt-locale-l1-1-0.dll
api-ms-win-crt-math-l1-1-0.dll
api-ms-win-crt-stdio-l1-1-0.dll
KERNEL32.dll
```

which now includes MSVCP140_ATOMIC_WAIT.dll. It would be a requirement to have this DLL available on target machines, either locally beside the executable or installed on the system.

Brief notes on deprecated language/library features & "hidden" compiler changes

Corner cases by way of deprecated language and library features, particularly when exposed in plugin APIs, should be considered. For example:

- An obvious earlier example would be the hypothetical case where a plugin API made use of `std::auto_ptr`, which was deprecated in C++11, and finally removed in C++17.
- Such deprecations and removals can sometimes be targeted more aggressively by compilers. For example, XCode 14/clang marked `sprintf` as deprecated (due to security reasons) outside of any particular C++XX standard.
- Deprecations might not be immediately obvious from straight-forward compilation tests. Another example from XCode 14 was that it continued to quietly support `std::unary` for C++17 use for a brief period of time, even though it was meant to be deprecated in C++11 and fully removed in C++17.
- Compilers may introduce new flags to support newer C++ standards. For example, C++17 updates may have required adding `-faligned-new` to build scripts on Linux and macOS.
- There's also the risk that new compilers may introduce bugs, and that like the above, they need fixes or work-arounds to be applied.

In general, there's always a risk in compiling a plugin with a newer compiler version and C++ standard than the software was built against. In practice, the uglier corners of the language tend not to be used in APIs, which reduces the risk. The risk can be further minimised via app vendors adhere to stable API/ABI features and follow deprecation warnings as closely as they can.

A proposal to consider further is for vendors to test their development kits against newer compiler versions and C++ standards, and to publish those notes in addition to what the VFX Platform calls for. Members from the wider community could also consider providing input to specific vendors when they become aware of issues in forward-looking compatibility.

References

[Is it safe to link C++17, C++14, and C++11 objects](#)

For runtime

The simple rule is to ensure the shared library the program uses at run-time is at least as new as the version used to compile any of the objects.

For compile time

For GCC it is safe to link together any combination of objects A, B, and C. If they are all built with the same version then they are ABI compatible, the standard version (i.e. the `-std` option) doesn't make any difference.

Why? Because that's an important property of our implementation which we work hard to ensure.

Where you have problems is if you link together objects compiled with different versions of GCC and you have used unstable features from a new C++ standard before GCC's support for that standard is complete.

[C++ Dialect Options \(Using the GNU Compiler Collection \(GCC\)\)](#)

GCC ABI changes over time

[Jonathan Wakely - Re: GCC compatibility of shared libraries with STL objects in their inte](#)

In general C++11 doesn't change anything, the library ABI doesn't depend on the `-std` option.

[ABI Policy and Guidelines](#)

It says these versions

- GCC 12.1.0: `libstdc++.so.6.0.30`
- GCC 13.1.0: `libstdc++.so.6.0.31`
- GCC 13.2.0: `libstdc++.so.6.0.32`

[\[committed\] libstdc++: Implement C++20 <format> \[PR104166\]](#)

This is the first drop of it, and it looks to be all in `format.h`, so no library requirements.

[\[committed\] libstdc++: Implement C++23 <print> header \[PR107760\]](#)

Has a symbol in `libstdc++` but seems to be only used on Windows.

Rocky 8 `libstdc++` versions

```
Unset
ls -al /usr/lib64/libstdc++.so.6
lrwxrwxrwx 1 root root 19 Jun 15 2023 /usr/lib64/libstdc++.so.6 ->
libstdc++.so.6.0.25

strings /usr/lib64/libstdc++.so.6 | grep LIBCXX
GLIBCXX_3.4
GLIBCXX_3.4.1
```


```
GLIBCXX_3.4.2
GLIBCXX_3.4.3
GLIBCXX_3.4.4
GLIBCXX_3.4.5
GLIBCXX_3.4.6
GLIBCXX_3.4.7
GLIBCXX_3.4.8
GLIBCXX_3.4.9
GLIBCXX_3.4.10
GLIBCXX_3.4.11
GLIBCXX_3.4.12
GLIBCXX_3.4.13
GLIBCXX_3.4.14
GLIBCXX_3.4.15
GLIBCXX_3.4.16
GLIBCXX_3.4.17
GLIBCXX_3.4.18
GLIBCXX_3.4.19
GLIBCXX_3.4.20
GLIBCXX_3.4.21
GLIBCXX_3.4.22
GLIBCXX_3.4.23
GLIBCXX_3.4.24
GLIBCXX_3.4.25
GLIBCXX_DEBUG_MESSAGE_LENGTH
GA+GLIBCXX_ASSERTIONS
```

Rocky 9 libstdc++ versions

```
Unset
ls -al /usr/lib64/libstdc++.so.6
lrwxrwxrwx 1 root root 19 Apr  2 04:09 /usr/lib64/libstdc++.so.6 ->
libstdc++.so.6.0.29

strings /usr/lib64/libstdc++.so.6 | grep GLIBCXX
GLIBCXX_3.4
GLIBCXX_3.4.1
GLIBCXX_3.4.2
GLIBCXX_3.4.3
GLIBCXX_3.4.4
GLIBCXX_3.4.5
GLIBCXX_3.4.6
GLIBCXX_3.4.7
GLIBCXX_3.4.8
GLIBCXX_3.4.9
GLIBCXX_3.4.10
```

```
GLIBCXX_3.4.11
GLIBCXX_3.4.12
GLIBCXX_3.4.13
GLIBCXX_3.4.14
GLIBCXX_3.4.15
GLIBCXX_3.4.16
GLIBCXX_3.4.17
GLIBCXX_3.4.18
GLIBCXX_3.4.19
GLIBCXX_3.4.20
GLIBCXX_3.4.21
GLIBCXX_3.4.22
GLIBCXX_3.4.23
GLIBCXX_3.4.24
GLIBCXX_3.4.25
GLIBCXX_3.4.26
GLIBCXX_3.4.27
GLIBCXX_3.4.28
GLIBCXX_3.4.29
GLIBCXX_DEBUG_MESSAGE_LENGTH
```

- macOS now has system runtime libraries in a cache and not visible on disk
- [Extract the system libraries on macOS Big Sur.](#) [Where is /usr/lib/libc++.1.dylib a... | Apple Developer Forums](#) min target, not working on older, dynamic linker shared cache
- VisualStudio runtime library deployment [Deployment in Visual C++](#)
- Windows C++ runtime libraries [C runtime \(CRT\) and C++ standard library \(STL\) lib files](#)
- VisualStudio STL  [GitHub - microsoft/STL: MSVC's implementation of the C++ Standard Library.](#)