ChatGPT

Configuring Clang LibTooling for a Consistent C++ Standard Library (macOS)

Understanding the Conflict

Your compilation errors stem from mixing two different C++ standard library header sets on macOS: **Homebrew's libc++ headers vs. the macOS SDK's libc++ headers.** In your current `args`, you explicitly include Homebrew's libc++ (`/opt/homebrew/Cellar/llvm/21.1.3/include/c++/v1`) *and* set an SDK `-isysroot` (which causes Clang to also find the Apple SDK's `c++/v1` headers). This mix leads to conflicting definitions – for example, macros like `_LIBCPP_INLINE_VISIBILITY` not being recognized because one set of headers expects configuration from the other. In short, **you need to use a single, consistent C++ standard library implementation at a time** to avoid these conflicts.

Use One C++ Standard Library – Not Both

To fix this, pick **one** C++ library and configure Clang accordingly:

- **Option 1: Use Homebrew's libc++ exclusively.** This means *preventing Clang from using the default SDK C++ headers* and pointing it to Homebrew's.
- **Option 2: Use the macOS SDK's (Xcode/CLT) libc++ exclusively.** In this case, remove the Homebrew libc++ include path and ensure Clang uses the SDK's headers.

Either approach can work – the key is **not to mix them** in one invocation.

Option 1 – Homebrew's libc++ (with `-nostdinc++`):

If you want to use the C++20 library that came with your Homebrew LLVM 21.1.3, update your arguments as follows:

- **Disable default C++ include paths:** Add `-nostdinc++`. This flag tells Clang **not** to search any of the standard C++ include directories (like the Mac SDK's `/usr/include/c++/v1`) [1] [2]. This prevents the Apple SDK headers from creeping in. (It does *not* disable C includes, only C++ library includes.)
- **Include Homebrew's libc++ headers:** Since you've disabled auto-includes, you must explicitly add the libc++ include directory that you want. Keep the `-I/opt/homebrew/Cellar/llvm/21.1.3/include/c++/v1` (or better, use `-isystem` for system libs) to point Clang to Homebrew's C++20 headers [1]. This ensures `<vector>`, `<string>`, etc. are taken from the Homebrew LLVM's libc++, not the SDK.
- **Include Clang's builtins:** Continue to include the Clang builtin headers (the ones under `lib/clang/21/include`). You already have `-I/opt/homebrew/Cellar/llvm/21.1.3/lib/clang/21/include`; that's good. (Using `-isystem` here would treat them as system headers to silence

warnings, but functionally it's the same idea.) These provide compiler intrinsics and should match your Homebrew clang version.

- **Retain or adjust `-isysroot` for C headers:** It's fine to keep the `-isysroot /Library/Developer/CommandLineTools/SDKs/MacOSX.sdk` if your code needs standard C headers (e.g. `<stdlib.h>` when you include `<cstdlib>`). The `-nostdinc++` flag *won't* block standard C includes [2], so Clang can still find headers like `<stdlib.h>` or `<stddef.h>` in the SDK. In other words, you'll use the macOS SDK for C library stuff, but all C++ library includes come from Homebrew's directory. This combination is common when using a custom libc++ on macOS.
- **(Optional) Specify the C++ standard library library:** You can add `-stdlib=libc++` explicitly to be clear that you're targeting libc++ (not libstdc++). On macOS, Clang defaults to libc++ anyway, so this is mostly for clarity. Since you're doing analysis (`-fsyntax-only`), linking isn't an issue here; but if you were compiling to run, you'd also want to link against Homebrew's libc++ library (e.g. via appropriate `-L` flags) rather than the system one.

Using the above, your `args` might look like:

```
std::vector<std::string> args = {
    "-std=c++20",
    "-fsyntax-only",
    "-nostdinc++",
    "-isystem", "/opt/homebrew/opt/llvm/include/c++/v1",        // Homebrew
libc++ headers
    "-isystem", "/opt/homebrew/opt/llvm/lib/clang/21/include",  // Clang 21
builtins (same as Cellar path)
    "-isysroot", "/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk",
    // ... any project-specific include paths ...
};
```

This ensures **only Homebrew's C++ library headers are used**, avoiding any mismatches with the SDK. It's effectively what you'd do on the command line when using a custom libc++:

```
clang++ -std=c++20 -stdlib=libc++ -nostdinc++ -I<path-to-libc++>/include/c++/
v1 ...
```

as recommended by the LLVM libc++ docs [1].

> **Note:** `-nostdinc++` specifically disables C++ standard includes but still allows default C includes and system frameworks. There is also `-nostdinc` (without "++"), which would disable *all* standard includes (C and C++). You likely **do not** want to use full `-nostdinc` here, since you'd then have to manually add paths for the C headers too [3]. Stick with `-nostdinc++` to target just the C++ library includes.

Option 2 – macOS SDK's libc++ (Apple default):

Alternatively, you can use the C++ standard library that comes with Xcode's Command Line Tools (CLT) and **avoid using Homebrew's libc++ headers entirely**. In this case:

- **Remove the Homebrew libc++ `-I` path.** You won't need the `-I/opt/homebrew/.../include/c++/v1` if you intend to use the SDK's headers.
- **Keep (or add) the SDK include paths.** Ensure you have the SDK specified so Clang knows where to find the system headers. Your `-isysroot` does this – it tells Clang to treat that SDK as the root for includes. With an SDK specified, Clang on Darwin will look in `$(SDK)/usr/include/c++/v1` for the C++ headers (and in the SDK's `usr/include` for C headers) **if** it's configured to use libc++. Since you're using Homebrew Clang, you should double-check it knows to search the SDK's libc++ directory. Homebrew's clang typically relies on the presence of the CLT to provide headers. You might need to add an explicit include path for the SDK's C++ headers if it isn't picking them up. For example: `-isystem /Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/usr/include/c++/v1`. In many cases, though, specifying the sysroot and using `-stdlib=libc++` is enough for Clang to find the C++ headers in the SDK [4] [5] (Homebrew's Clang will behave similarly to Apple's in this regard, assuming the CLT is installed).
- **Use libc++ (the default).** As above, you can explicitly add `-stdlib=libc++` (though on macOS this is default). This just reiterates to use the C++ library in the SDK.

With this configuration, you're effectively treating your tool like Apple's clang would treat a normal compile on macOS (using the platform's standard library). **Do not** add the Homebrew include path in this mode. The errors about `_LIBCPP_INLINE_VISIBILITY` should disappear, because now all the C++ headers (e.g. `<string> and its dependencies`) will consistently come from the SDK's libc++ implementation, which is internally self-consistent.

LibTooling vs. Normal Compilation Flags

There isn't a special set of "LibTooling-only" flags – you use the same compiler options you would with `clang++` on the command line. The key difference is that when you use the LibTooling API (like `runToolOnCodeWithArgs`), **Clang won't automatically infer system include paths** unless you tell it to. In a normal compile, the driver knows where to find libc++ headers, but `runToolOnCodeWithArgs` bypasses the driver logic and requires you to provide needed include paths manually [6] [7]. This is why you encountered the issue – the tool was picking up both sets of includes due to the flags given.

So, the flags we discussed (like `-nostdinc++`, `-isysroot`, `-isystem` etc.) are exactly what you'd use with `clang++` to control include paths. For example, passing `-- -nostdinc++ -isystem <path>` to a LibTooling-based tool is a common way to override standard includes [3]. In short, **LibTooling requires you to explicitly set up the include environment**, just as if you were invoking the compiler manually with those options.

Additional Tips

- **Preventing SDK headers when using Homebrew's libc++:** As noted, `-nostdinc++` is the critical flag to stop Clang from automatically adding *any* default C++ include directories (including the Mac

SDK paths) (1). With that in place, Clang will only use the include directories you specify for C++ headers – ensuring the SDK's `c++/v1` doesn't sneak in.
- **Using a Compilation Database:** If you have a `compile_commands.json` for your project or can generate one, you could use Clang's `CompilationDatabase` (via `CommonOptionsParser` and `ClangTool`) instead of manually constructing the `args`. That way, the include paths and flags from a real build are used. In your case, where you want to run analysis without a full build, providing the flags manually is fine – just be sure they mirror a consistent build configuration.
- **Querying Clang's default include paths:** You asked if there's a way to query LLVM for its default include paths. There's no direct LibTooling API call to fetch those, but you can invoke the compiler in verbose mode to see them. For example, running `clang++ -std=c++20 -v -x c++ -E /dev/null` in your environment will print the default include search paths (this is a quick way to see what **directories Clang would normally search** for headers). This can help you verify where your Homebrew Clang expects to find libc++ by default. Usually, Homebrew's Clang will look in its install prefix for `include/c++/v1` and also check the SDK paths – the verbose output will make it clear. You can then mimic those in your `args` if needed.
- **Alternative APIs:** Using `clang::tooling::ClangTool` with a custom `FixedCompilationDatabase` (or `CommonOptionsParser`) is an alternative to `runToolOnCodeWithArgs`. Under the hood, though, it ends up doing similar things – you still need to specify the correct include paths/flags. Switching APIs likely won't solve the fundamental issue of include configuration. It's more about convenience (for example, `ClangTool` will automatically handle multiple files and integrate with a compile_commands database). For a single-file analysis, `runToolOnCodeWithArgs` is fine – just feed it the right flags as discussed.

In summary, the **success criteria** is to have all standard C++ headers coming from one source. To achieve that on macOS with Homebrew LLVM: **use `-nostdinc++` and supply only the include directory for the C++ library you want to use** (1). Whether you choose Homebrew's libc++ or the macOS SDK's, ensure the other one is completely excluded. By applying these changes, your tool should be able to parse Boost.JSON (and other C++20 code) without the avalanche of conflicts, all while maintaining correct C++20 semantics and access to the necessary library features. The Clang AST will then be built against a consistent set of headers, matching a real compilation environment.

Sources:

- Clang/LibTooling usage of `-nostdinc++` to override standard library headers (1) (2).
- Stack Overflow – *"How to use standard library with Clang and LibTooling"* (passing `-nostdinc` and custom include paths to LibTooling tools) (3).
- Discussion of Clang's default include search on macOS (Homebrew vs Xcode paths) (4) (5).
- LLVM issue – LibTooling not automatically adding C++ include dirs, requiring manual configuration (6) (7).

---

(1) (2) (4) (5) c++ - When running clang built from source, how to specify location of libc++, or, someone explain to me what -stdlib=libc++ does - Stack Overflow
https://stackoverflow.com/questions/25920130/when-running-clang-built-from-source-how-to-specify-location-of-libc-or-som

(3) c++ - How to use standard library with Clang and LibTooling - Stack Overflow
https://stackoverflow.com/questions/27092593/how-to-use-standard-library-with-clang-and-libtooling

⑥ ⑦ Driver/Libtooling: Clang can’t seem to find C++ system include directories · Issue #75014 · llvm/llvm-project · GitHub
https://github.com/llvm/llvm-project/issues/75014