A.4 — C++ FAQ

There are certain questions that tend to get asked over and over. This FAQ will attempt to answer the most common ones.

Q1: Why shouldn't we use "using namespace std"?

The statement `using namespace std;` is a **using directive**. Using directives import all of the identifiers from a namespace into the scope of the using directive.

You may have seen something like this:

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!";

    return 0;
}
```

This allows us to use names from the `std` namespace without having to explicitly type `std::` over and over. In the above program, we can just type `cout` instead of `std::cout`. Sounds great, right?

However, when the compiler encounters `using namespace std`, it will import every identifier it can find in `namespace std` into the global scope (since that's where the using directive has been placed). This introduces 3 key challenges:

- The chance for a naming collision between a name you've picked and something that already exists in the `std` namespace is massively increased.
- New versions of the standard library may break your currently working program. These future versions could introduce names that cause new naming collisions, or in the worst case, the behavior of your program might change silently and unexpectedly!
- The lack of std:: prefixes makes it harder for readers to understand what is a std library name and what is a user-defined name.

For this reason, we recommend avoiding `using namespace std` (or any other using directive) entirely. The small savings in typing isn't worth the additional headaches and future risks.

Related content

See lesson 7.12 -- Using declarations and using directives for more detail and examples.

Q2: Why can I use type T without including the header that defines T?

For example, many readers ask why their program that uses `std::string_view` needs to `#include <string_view>` when it seems to work fine without the #include.

Headers can #include other headers. When you #include a header, you also get all of the additional headers that it includes (and all of the headers that those headers include too). All of the additional headers that come along for the ride that you didn't explicitly include are called "transitive includes".

In your *main.cpp* file, you probably `#include <iostream>`. On your compiler, if your *iostream* header includes the *string_view* header (for its own use), then when you `#include <iostream>` you will get the contents of the *string_view* header (and any other headers that *iostream* includes). This means your *main.cpp* will be able to use the `std::string_view` type without explicitly including *string_view* header.

Even though this may compile on your compiler, you should not rely on this. What compiles for you now may not compile on another compiler, or even on a future version of your compiler.

There is no way to warn when this happens, or prevent it from happening. The best you can do is take care to explicitly include the proper headers for all of the things you use. Compiling your program on several different compilers may help identify headers that are being transitively included on other compilers.

Related content

Covered in lesson 2.11 -- Header files.

Q3: My code that produces undefined behavior appears to be working fine. Is this okay?

aka: "I did that thing you told me not to do, and it worked. So what's the issue?"

Undefined behavior occurs when you perform an operation whose behavior is not defined by the C++ language. Code implementing undefined behavior may exhibit *any* of the following symptoms:

- Your program produces a different result every time it is run.
- Your program behaves inconsistently (sometimes produces the desired result, sometimes not).
- Your program consistently produces the same incorrect result.

- Your program initially seems like its working but produces some incorrect result later in the program.
- Your program crashes, either immediately or later.
- Your program works on some compilers or platforms but not others.
- Your program works until you change some other seemingly unrelated code.
- Your program appears to produce the desired result anyway.

One of the biggest problems with undefined behavior is that the behavior exhibited by the program may change at any point, for any reason. So while such code may appear to work now, there is no guarantee it will do so when run again later.

Related content

Undefined behavior is covered in lesson 1.6 -- Uninitialized variables and undefined behavior.

Q4: Why does my code that produces undefined behavior generate a certain result?

Readers often ask what is happening to produce a specific result on their system. In most cases, it's difficult to say, as the result produced may be dependent upon the current program state, your compiler settings, how the compiler implements a feature, the computer's architecture, and/or the operating system. For example, if you print the value of an uninitialized variable, you might get garbage, or you might always get a particular value. It depends on what type of variable it is, how the compiler lays out the variable in memory, and what's in that memory beforehand (which might be impacted by the OS or the state of the program prior to that point).

And while such an answer may be interesting mechanically, it's rarely useful overall (and likely to change if and when anything else changes). It's like asking, "When I put my seat belt through the steering wheel and connect it to the accelerator, why does the car pull left when I turn my head on a rainy day?" The best answer isn't a physical explanation of what's occurring, it's "don't do that".

Q5: Why am I getting a compile error when I try to compile an example that seems like it should work?

The most common reason for this is that your project is being compiled using the wrong language standard.

C++ introduces many new features with each new language standard. If one of our examples uses a feature that was introduced in C++17, but your program is compiling using the C++14 language standard, then it probably won't compile because the feature we're using isn't supported by the language standard we've selected.

Try setting your language standard to the latest version your compiler supports and see if that resolves the issue. You can also check that your compiler is properly configured to use the language standard you're expecting by running the program here: 0.13 -- What language standard is my compiler using?.

Related content

Covered in lesson 0.12 -- Configuring your compiler: Choosing a language standard.

It is also possible that your compiler either doesn't support a specific feature yet, or has a bug preventing use in some cases. In this case, try updating your compiler to the latest version available.

The CPPReference website tracks compiler support for each feature per language standard. You can find those support tables linked from their home page, top right, under "Compiler Support" (by language standard). For example, you can see which C++23 features are supported here.

Q6: Why should I #include "foo.h" from foo.cpp?

It is a best practice for a source file (e.g. foo.cpp) to include its paired header (e.g. foo.h). In many cases, foo.h will contain definitions that foo.cpp will need to compile correctly.

However, even if foo.cpp compiles file without foo.h, including the paired header allows the compiler to discover certain types of inconsistencies between the two files (e.g. when the return type of a function doesn't match the return type of its forward declaration). Without the inclusion, this might cause undefined behavior.

The cost of the #include is negligible, so there is little downside to including the header.

Related content

Covered in lesson 2.11 -- Header files.

Q7: Why does my project only compile when I #include "foo.cpp" from "main.cpp"?

This is almost always due to forgetting to add foo.cpp to your project and/or compilation command line. Update your project and/or command line to include each source (.cpp) file. When you compile your project, you should see each source file being compiled.

Normally in a project that has multiple files, the compiler will compile each source (.cpp) file individually. After all the source files have been compiled, the linker links them together and creates the resulting output file (e.g. an executable). However, if you split your code between

two or more files (e.g. main.cpp and foo.cpp) and then only compiles main.cpp, you will probably get a compilation error or linker error, since part of the code required for your project is not being compiled.

New programmers sometimes discover that they can make their program work by adding `#include "foo.cpp"` to main.cpp instead of adding foo.cpp to the project or compilation command line. After doing so, when main.cpp is compiled, the will preprocessor will create a translation unit consisting of the code from both foo.cpp and main.cpp, which will then be compiled and linked. In smaller project, this may work. So why not do this?

There are a few reasons:

1. It can result in naming collisions between files.
2. It can be hard to avoid ODR violations.
3. Any change to any .cpp file will result in your entire project being recompiled. This can take a long time.

Related content

Covered in lesson 2.11 -- Header files.

Q8: When I compile an example from this website, I get an error similar to “argument list for class template XXX is missing”. Why?

The most likely reason is that the example makes use of a feature called Class Template Argument Deduction (CTAD), which is a C++17 feature. Many compilers default to C++14, which doesn’t support this feature.

If the following program doesn’t compile for you, that’s the reason:

```
#include <utility> // for std::pair

int main()
{
    std::pair p2{ 1, 2 }; // CTAD used to deduce std::pair<int, int> from the
initializers (C++17)

    return 0;
}
```

You can check which language standard your compiler is configured for using the program in lesson 0.13 -- What language standard is my compiler using?.

We cover CTAD in lesson 13.14 -- Class template argument deduction (CTAD) and deduction guides.