C++ Club UK Meeting 129

Gleb Dolgich

2021-05-13

Contents

The MSVC's infamous Reddit bug

The title of this Reddit post is "Really Microsoft? How on Earth this is valid C++?" The multiple examples of nonsensical code in the thread are happily complied and executed by MSVC. Some of the snippets are:

Godbolt

```
class Foo {
};

int main() {
    Foo* foo;
    Foo bar = *foo.aoehocqaweioh<float>();
    return 0;
}
```

Godbolt

```
class Foo {
};

void test(Foo* foo) {
    foo.aadfasdf<int>();
}
```

Godbolt

```
int main()
{
    return nullptr._<>();
}
```

This comment explains why the compiler is wrong – ill-formed code requires diagnostic, which MSVC doesn't generate, making it non-conforming.

STL replied:

> Our compiler dev Jonathan Emmett created MSVC-PR-321053 titled "Fix the infamous reddit bug". According to his changelog, fixing this accepts-invalid bug led to finding and fixing two more bugs <...>.

The coolest things in C++

A redditor asks, "What's the coolest thing you've created with C++?"

The thread has many great examples, too many to mention:

- Street hole detection for Audi A6
- Indie games
- Software for the Human Brain project for visualizing intracranial plots of hundreds of subjects by cutting a 3D mesh of the brain at realtime and generating inside cut textures from MRI niftii files

Many posts mention hobby projects, which shows that people use C++ for their creative endeavours and not just for work. Go and scroll through the thread to see more examples.

SpaceX

StackOverflow blog interviewed Steven Gerding, Dragon's software development lead:

> The actual work of software development by vehicle engineers such as Gerding is largely done using C++, which has been the mainstay of the company's code since its early days.

C++ compiler fingerprints

A redditor asks:

> Is there any way to tell which compiler was used to create a given C++ object file or executable? If so, how does this process work?

Turns out, in many cases this is indeed possible, which has implications for infosec and computer forensics. Some compilers leave a section in the executable that contains various information about the toolchain used to create it. It is possible to remove that section:

```
strip -s -R .comment -R .gnu.version <binary>
```

On Windows, it is possible to extract this information from the executable file header, which uses Portable Executable (PE) format. The PE header follows the

MZ header, which for Windows programs displays a message telling the unfortunate DOS user that they can't run this program. ('MZ' stands for Mark Zbikowski, a Microsoft engineer who designed the original MS-DOS executable file format.)

There is a program called PEiD which can detect compilers and various executable packers amnd cryptors by inspecting the PE header.

The Reddit thread also mentions IDA disassembler which can detect the compiler using various heuristics.

The most interesting paper mentioned in the thread was Extracting compiler provenance from program binaries, using machine learning:

> We present a novel technique that identifies the source compiler of program binaries, an important element of program provenance. Program provenance answers fundamental questions of malware analysis and software forensics <...> our models identify the source compiler of binary code with over 90% accuracy, even in the presence of interleaved code from multiple compilers.

Abusing `co_await` for error handling

As if all other ways of handling errors in C++ (standard or proposed) weren't enough, here is another way. We have coroutines now, which are amazing. What if we could use them for error reporting instead?

Here is an illustration of the idea:

```
Expected<int, std::string> g(bool success) {
  if (!success)
    return "Error !!";
  return 5;
}

Expected<int, std::string> f(bool success) {
  int result = co_await g(success);
  co_return result * 2;
}
```

Rust's `Result` type is mentioned, too:

```
fn f(success: bool) -> Result<i32, String> {
    let value = g(success)?;
    value * 2
}
```

The author, Antoine Morrier, explains how to implement the coroutine machinery to make this possible. He provides full source code on Wandbox. The Reddit thread offers some interesting feedback, including a link to error handling

benchmarks which has a similar feature in the list or benchmars. It also turns out that Facebook Folly's Expected class by Eric Niebler supports `co_await`.

Designated initializers in C++20

Rainer Grimm writes:

> Designated initialization is an extension of aggregate initialization and empowers you to directly initialize the members of a class type using their names. Designated initialization is a special case of aggregate initialization.

He starts by reminding us what kind of class counts as an aggregate in C++20:

- no private or protected non-static data members
- no user-declared or inherited constructors
- no virtual, private, or protected base classes
- no virtual member functions

An example of an aggregate would be the following class:

```
1 struct Point3D {
2     int x{};
3     int y{};
4     int z{};
5 }
```

Note that Rainer doesn't default-initialize the aggregate class members in his example. I try to make it a habit.

C++11 introduced aggregate initialization using the uniform initialization syntax (curly braces):

```
1 Point3D point3D{1, 2, 3};
```

C++20 added designated initializers based on the C99 syntax (see the proposal P0329R4), with some limitations:

```
1 Point3D point3D{.x:1, .y:2, .z:3};
```

- the designators cannot be reordered
- nested aggregates cannot be initialized in this way
- regular and designated initializers cannot be mixed in the same initialization
- arrays cannot be initialized this way

The proposal mentions that both GCC and Clang have supported designated initialization syntax via extensions for some time.

Rainer says that at the moment of writing the article only MSVC supported C++20 designated initialization syntax.

As we saw before, the new designated initialization syntax can be used to emulate the named arguments feature present in some other languages but not C++:

```
auto result = foo({.x:1, .y:2, .z:3});
```

Ticket Maps

Anthony Williams writes:

> It has been an increasingly common scenario that I've encountered where you have some ID that's monotonically increasing, such as a subscription or connection index, or user ID, and you need your C++ program to hold some data that's associated with that ID value. The program can then pass round the ID, and use that ID to access the associated data at a later point. Over time, IDs can become invalidated, so the data associated with that value is removed, and you end up with a sparse set of currently-active IDs.

For the described use case he implemented an algorithm Sean Parent calls "Russian Coat-Check Algorithm".

> In this algorithm, the map is implemented as a vector of pairs of key/optional data. Because the keys come from a monotonically increasing index, the vector is *always sorted*, and inserts are always at the end. Entries can be removed by clearing the data, and if there are too many empty entries then the vector can be compacted. Lookups are always fast, because the vector is always sorted, so a simple binary search will find the right element.

I'll have to revisit Sean Parent's presentation if only to see what is so Russian about this algorithm.

The implementation is available on GitHub under the Boost Software Licence.

Accessing private members of a friend class from friend function: Which compiler is correct?

A redditor noticed discrepancy between compilers in how they treat friend classes. It seems that Clang allows reciprocal access to field's private members, whereas GCC or MSVC don't. Their question was which compiler is right.

```
class Foo {
    friend class Bar;
    int x_;
};

class Bar {
    friend void foobar(Bar b, Foo f) {
```

```
 8          std::cout << f.x_;
 9          // ^^^ Clang: OK; GCC, MSVC: error
10      }
11  };
```

Someone found a post from 2014 by Richard Smith on the Clang developer mailing list:

> Clang's behavior reflects the current intent of the C++ core working group. This is core issue 1699, and the current resolution to that is to treat everything that is lexically within a befriended entity as being befriended. GCC does not yet implement the resolution to core issue 1699 (which is reasonable, since it's not yet even resolved).

Looks like this issue hasn't been resolved yet. However, the C++ Standard says:

> Friendship is neither inherited nor transitive.

As another redditor explains:

> In the example OP provided, the class `Bar` is a friend of class `Foo`. The function `foobar()` is a friend of class `Bar`.
>
> About transitive: just because `Bar` is friend of `Foo`, it does not mean `foobar()` is automatically also friend of `Foo`. IRL (*in real life*): the friend of your friend is not automatically your friend as well.
>
> Regarding inheritance: the friend of your son or daughter is not automatically your friend.
>
> You could also view `friend` as an explicit statement. Only the thing you explicitly declare as `friend` is actually a friend. No non-declared friends "sneak" into the relationship.

So it looks like MSVC and GCC are correct here and Clang isn't.

Modern C++ class members and initializations the right way

Pranay Kumar reminds us in his article on Medium how to initialize a class properly and in a modern way:

- Use member initializers in the same order as their declaration
- Prefer in-class member initializer over constant initializations OR over default constructor.
- Don't cast away `const`, ever!
- Use delegating constructors to represent common actions for all constructors of a class.

The Sad Truth About C++ Copy Elision

Scott Wolchok writes:

> In this post, I'll walk through an example where an obvious optimization you might expect from your compiler doesn't actually happen in practice.

He comes up with an example that shows copy elision happening in one case but not in the other, when the code is slightly modified (an extra local variable is introduced).

Reddit has not one but two threads discussing this post. Predictably, you can see:

- Rust threads, some of them going into details with code examples
- Bjarne Stroustrup quotes (*citation needed*)
- "Works as expected"
- "We need destructive move"

The best reply award goes to this one at the very bottom of the second discussion:

> You could just replace "The Sad Truth About" by "Amazing optimizations of" in the title, and showcase how neatly optimized is the first example.

Type-safe Pimpl implementation without overhead

Malte Skarupke writes:

> I like the pimpl idiom because I like to keep my headers as clean as possible, and other people's headers are dirty. Unfortunately the pimpl idiom never feels like a good solution because it has runtime overhead that wouldn't be needed if I didn't care about clean headers so much. <…> You always need an extra heap allocation and every method performs an extra pointer dereference.

The author then presents a solution:

> This code fixes that, so that there can be zero runtime overhead. <…> This uses a well-known hack where you put the necessary storage into your class, and then placement-new the forward declared object into the storage.

My impression is that this solution eliminates the main Pimpl advantage, which is removing the need to rebuild client code when implementation changes. Sure, it's more efficient than ordinary Pimpl, but I'm not convinced it's worth it (unless, perhaps, you are an embedded system programmer). If your clients have to rebuild after implementation changes anyway, just keep things simple and use the implementation directly.

Motivational Twitter

Nick Chiappini, a chemistry post-doc, writes:

> Whenever I mess up an experiment I just think of what a pigeon considers a successful nest