14.17 — Constexpr aggregates and classes

ALEX[1] JANUARY 29, 2025

In lesson F.1 -- Constexpr functions (https://www.learncpp.com/cpp-tutorial/constexpr-functions/)[2], we covered constexpr functions, which are functions that may be evaluated at either compile-time or runtime. For example:

```
#include <iostream>

constexpr int greater(int x, int y)
{
    return (x > y ? x : y);
}

int main()
{
    std::cout << greater(5, 6) << '\n'; // greater(5, 6) may be evaluated at compile-time or runtime

    constexpr int g { greater(5, 6) };  // greater(5, 6) must be evaluated at compile-time
    std::cout << g << '\n';             // prints 6

    return 0;
}
```

In this example, `greater()` is a constexpr function, and `greater(5, 6)` is a constant expression, which may be evaluated at either compile-time or runtime. Because `std::cout << greater(5, 6)` calls `greater(5, 6)` in a non-constexpr context, the compiler is free to choose whether to evaluate `greater(5, 6)` at compile-time or runtime. When `greater(5, 6)` is used to initialize constexpr variable `g`, `greater(5, 6)` is called in a constexpr context, and must be evaluated at compile-time.

Now consider the following similar example:

```
#include <iostream>

struct Pair
{
    int m_x {};
    int m_y {};

    int greater() const
    {
        return (m_x > m_y  ? m_x : m_y);
    }
};

int main()
{
    Pair p { 5, 6 };                    // inputs are constexpr values
    std::cout << p.greater() << '\n'; // p.greater() evaluates at runtime

    constexpr int g { p.greater() };  // compile error: greater() not constexpr
    std::cout << g << '\n';

    return 0;
}
```

In this version, we have an aggregate struct named `Pair`, and `greater()` is now a member function. However, because member function `greater()` is not constexpr, `p.greater()` is not a constant expression. When `std::cout << p.greater()` calls `p.greater()` (in a non-constexpr context), `p.greater()` will be evaluated at runtime. However, when we try to use `p.greater()` to initialize constexpr variable `g`, we get a compile error, as `p.greater()` cannot be evaluated at compile-time.

Since the inputs to `p` are constexpr values (`5` and `6`), it seems like `p.greater()` should be capable of being evaluated at compile-time. But how do we do that?

Constexpr member functions

Just like non-member functions, member functions can be made constexpr via use of the `constexpr` keyword. Constexpr member functions can be evaluated at either compile-time or runtime.

```
#include <iostream>

struct Pair
{
    int m_x {};
    int m_y {};

    constexpr int greater() const // can evaluate at either compile-time or runtime
    {
        return (m_x > m_y  ? m_x : m_y);
    }
};

int main()
{
    Pair p { 5, 6 };
    std::cout << p.greater() << '\n'; // okay: p.greater() evaluates at runtime

    constexpr int g { p.greater() };  // compile error: p not constexpr
    std::cout << g << '\n';

    return 0;
}
```

In this example, we've made `greater()` a constexpr function, so the compiler can evaluate it at either runtime or compile-time.

When we call `p.greater()` in runtime expression `std::cout << p.greater()`, it evaluates at runtime.

However, when `p.greater()` is used to initialize constexpr variable `g`, we get a compiler error. Although `greater()` is now constexpr, `p` is still not constexpr, therefore `p.greater()` is not a constant expression.

Constexpr aggregates

Okay, so if we need `p` to be constexpr, let's just make it constexpr:

```
#include <iostream>

struct Pair // Pair is an aggregate
{
    int m_x {};
    int m_y {};

    constexpr int greater() const
    {
        return (m_x > m_y  ? m_x : m_y);
    }
};

int main()
{
    constexpr Pair p { 5, 6 };        // now constexpr
    std::cout << p.greater() << '\n'; // p.greater() evaluates at runtime or compile-
time

    constexpr int g { p.greater() };  // p.greater() must evaluate at compile-time
    std::cout << g << '\n';

    return 0;
}
```

Since `Pair` is an aggregate, and aggregates implicitly support constexpr, we're done. This works! Since `p` is a constexpr type, and `greater()` is a constexpr member function, `p.greater()` is a constant expression and can be used in places where only constant expressions are allowed.

Related content

We covered aggregates in lesson [13.8 -- Struct aggregate initialization (https://www.learncpp.com/cpp-tutorial/struct-aggregate-initialization/)](https://www.learncpp.com/cpp-tutorial/struct-aggregate-initialization/)[3].

Constexpr class objects and constexpr constructors

Now let's make our `Pair` a non-aggregate:

```cpp
#include <iostream>

class Pair // Pair is no longer an aggregate
{
private:
    int m_x {};
    int m_y {};

public:
    Pair(int x, int y): m_x { x }, m_y { y } {}

    constexpr int greater() const
    {
        return (m_x > m_y  ? m_x : m_y);
    }
};

int main()
{
    constexpr Pair p { 5, 6 };       // compile error: p is not a literal type
    std::cout << p.greater() << '\n';

    constexpr int g { p.greater() };
    std::cout << g << '\n';

    return 0;
}
```

This example is almost identical to the prior one, except `Pair` is no longer an aggregate (due to having private data members and a constructor).

When we compile this program, we get a compiler error about `Pair` not being a "literal type". Say what?

In C++, a **literal type** is any type for which it might be possible to create an object within a constant expression. Put another way, an object can't be constexpr unless the type qualifies as a literal type. And our non-aggregate `Pair` does not qualify.

Nomenclature

A literal and a literal type are distinct (but related) things. A literal is a constexpr value that is inserted into the source code. A literal type is a type that can be used as the type of a constexpr value. A literal always has a literal type. However, a value or object with a literal type need not be a literal.

The definition of a literal type is complex, and a summary can be found on cppreference (https://en.cppreference.com/w/cpp/named_req/LiteralType)[4]. However, it's worth noting that literal types include:

- Scalar types (those holding a single value, such as fundamental types and pointers)
- Reference types
- Most aggregates
- Classes that have a constexpr constructor

And now we see why our `Pair` isn't a literal type. When a class object is instantiated, the compiler will call the constructor function to initialize the object. And the constructor function in our `Pair` class is not constexpr, so it can't be invoked at compile-time. Therefore, `Pair` objects cannot be constexpr.

The fix for this is simple: we just make our constructor `constexpr` as well:

```cpp
#include <iostream>

class Pair
{
private:
    int m_x {};
    int m_y {};

public:
    constexpr Pair(int x, int y): m_x { x }, m_y { y } {} // now constexpr

    constexpr int greater() const
    {
        return (m_x > m_y  ? m_x : m_y);
    }
};

int main()
{
    constexpr Pair p { 5, 6 };
    std::cout << p.greater() << '\n';

    constexpr int g { p.greater() };
    std::cout << g << '\n';

    return 0;
}
```

This works as expected, just like our aggregate version of `Pair` did.

Best practice

If you want your class to be able to be evaluated at compile-time, make your member functions and constructor constexpr.

Implicitly defined constructors are constexpr if they can be defined as such. Explicitly defaulted constructors must be explicitly defined as constexpr.

Tip

Constexpr is part of the interface of the class, and removing it later will break callers who are calling the function in a constant context.

Constexpr members may be needed with non-constexpr/non-const objects

In the above example, since the initializer of constexpr variable `g` must be a constant expression, it's clear that `p.greater()` must be a constant expression, and therefore `p`, the `Pair` constructor, and `greater()` must all be constexpr.

However, if we replace `p.greater()` with a constexpr function, things get a little less obvious:

```cpp
#include <iostream>

class Pair
{
private:
    int m_x {};
    int m_y {};

public:
    constexpr Pair(int x, int y): m_x { x }, m_y { y } {}

    constexpr int greater() const
    {
        return (m_x > m_y  ? m_x : m_y);
    }
};

constexpr int init()
{
    Pair p { 5, 6 };    // requires constructor to be constexpr when evaluated at
compile-time
    return p.greater(); // requires greater() to be constexpr when evaluated at
compile-time
}

int main()
{
    constexpr int g { init() }; // init() evaluated in compile-time context
    std::cout << g << '\n';

    return 0;
}
```

Remember that a constexpr function can evaluate at either runtime or compile-time. And when a constexpr function evaluates at compile-time, it can only call functions capable of evaluating at compile-time. In the case of a class type, that means constexpr member functions.

Since `g` is constexpr, `init()` must be evaluated at compile-time. Within the `init()` function, we define `p` as non-constexpr/non-const (because we can, not because we should). Even though `p` is not defined as constexpr, `p` still needs to be created at compile-time, and therefore requires a constexpr `Pair` constructor. Similarly, in order for `p.greater()` to evaluate at compile-time, `greater()` must be a constexpr member function. If either the `Pair` constructor or `greater()` were not constexpr, the compiler would error.

Key insight

When a constexpr function is evaluating in a compile-time context, only constexpr functions can be called.

Constexpr member functions may be const or non-const C++14

In C++11, non-static constexpr member functions are implicitly const (except constructors).

However, as of C++14, constexpr member functions are no longer implicitly const. This means that if you want a constexpr function to be a const function, you must explicitly mark it as such.

Constexpr non-const member functions can change data members Optional

A constexpr non-const member function can change data members of the class, so long as the implicit object isn't const. This is true even if the function is evaluating at compile-time.

Here's a contrived example of this:

```
#include <iostream>

class Pair
{
private:
    int m_x {};
    int m_y {};

public:
    constexpr Pair(int x, int y): m_x { x }, m_y { y } {}

    constexpr int greater() const // constexpr and const
    {
        return (m_x > m_y  ? m_x : m_y);
    }

    constexpr void reset() // constexpr but non-const
    {
        m_x = m_y = 0; // non-const member function can change members
    }

    constexpr const int& getX() const { return m_x; }
};

// This function is constexpr
constexpr Pair zero()
{
    Pair p { 1, 2 }; // p is non-const
    p.reset();       // okay to call non-const member function on non-const object
    return p;
}

int main()
{
    Pair p1 { 3, 4 };
    p1.reset();                     // okay to call non-const member function on non-
const object
    std::cout << p1.getX() << '\n'; // prints 0

    Pair p2 { zero() };             // zero() will be evaluated at runtime
    p2.reset();                     // okay to call non-const member function on non-
const object
    std::cout << p2.getX() << '\n'; // prints 0

    constexpr Pair p3 { zero() };   // zero() will be evaluated at compile-time
//    p3.reset();                   // Compile error: can't call non-const member
function on const object
    std::cout << p3.getX() << '\n'; // prints 0

    return 0;
}
```

As we work through this example, remember:

- A non-const member function can modify members of non-const objects.
- A constexpr member function can be called in either runtime contexts or compile-time contexts.

These two things work independently.

In the case of `p1`, `p1` is non-const. Therefore, we are allowed to call non-const member function `p1.reset()` to modify `p1`. The fact that `reset()` is constexpr doesn't matter here because nothing we're doing requires compile-time evaluation.

The `p2` case is similar. In this case, the initializer to `p2` is a function call to `zero()`. Even though `zero()` is a constexpr function, in this case it is invoked in a runtime context, and acts just like a normal function. Within `zero()`, we instantiate non-const `p`, call non-const member function `p.reset()` on it, and then return `p`. The returned `Pair` is used as the initializer for `p2`. The fact that `zero()` and `reset()` are constexpr don't matter in this case, because nothing we're doing requires compile-time evaluation.

The `p3` case is the interesting one. Because `p3` is constexpr, it must have a constant expression initializer. Therefore, this call to `zero()` must evaluate at compile-time. And because we're evaluating in a compile-time context, we can only call constexpr functions. Inside `zero()`, `p` is non-const (which is allowed, even though we're evaluating at compile-time). However, because we're in a compile-time context, the constructor used to create `p` must be constexpr. And just like the `p2` case, we're allowed to call non-const member function `p.reset()` on non-const object `p`. But because we're in a compile-time context, the `reset()` member function must be constexpr. The function then returns `p`, which is used to initialize `p3`.

Author's note

Yes, we used a non-const object to initialize a constexpr object. If this breaks your brain, it's probably because you haven't fully separated const from constexpr.

There is no requirement that a constexpr variable be initialized with a const value. It may seem that way because most of the time we initialize constexpr variable using literals (which are const) or other constexpr variables (which are implicitly const), and because the terms `const` and `constexpr` have similar names.

The requirement is actually that a constexpr variable be initialized with a constant expression. For functions (and operators), constexpr does not imply const, and constexpr functions (and operators) can make use of non-const objects and even return them.

The important thing isn't the const, its that the compiler can determine the value of the object at compile-time. And in the case of constexpr functions, that's possible even when they return a non-const object!

Constexpr functions that return const references (or pointers) Optional

Normally you won't see `constexpr` and `const` used right next to each other, but one case where this does happen is when you have a constexpr member function that returns a const reference (or (const) pointer-to-const).

In our `Pair` class above, `getX()` is a constexpr member function that returns a const reference:

```
1 | constexpr const int& getX() const { return m_x; }
```

That's a lot of const-ing!

The `constexpr` indicates that the member function can be evaluated at compile-time. The `const int&` is the return type of the function. The rightmost `const` means the member-function itself is const so it can

be called on const objects.

As an aside...

A member function that returned a const pointer to const instead might look something like this:

```
1 | constexpr const int* const getXPtr() const { return &m_x; }
```

Isn't it beautiful? No? Okay, fine.

Next lesson
14.x Chapter 14 summary and quiz

5

Back to table of contents

6

Previous lesson
14.16 Converting constructors and the explicit keyword

7

8

Ashok Sahu
May 9, 2025 8:20 am PDT

Question regarding
// p3.reset(); // Compile error: can't call non-const member function on const object

Which is the const object?? I didn't get this can this be explained better

0 Reply

Dietskittles
Reply to Ashok Sahu [12] May 21, 2025 7:08 pm PDT

I was also very confused on what made p3 const there. In the following authors note, it mentions that constexpr variables are implicitly const, which I completely forgot about since initially learning about constexpr. Hence constexpr p3 is const without needing to be explicitly labeled as such.

For functions, constexpr does not imply const. Just one more thing to keep in mind, lol.

Hopefully that helps and hopefully I'm understanding that correctly.

Last edited 1 month ago by Dietskittles

1 Reply

Dinny
March 9, 2025 3:50 am PDT

Cool, I think I get const vs constexpr now.

const = doesn't change
constexpr = can be known at compile time

nice

11 Reply

XenophonSoulis
January 29, 2025 8:43 am PST

I think one more const-ing is possible in the last example. If we use const pointers-to-const, so if we replace

```
1 | constexpr const int& getX() const { return m_x; }
```

with

```
1 | constexpr const int* const getX() const { return &m_x; }
```

(and make the necessary adjustments to `main()` if we want to print the values and not the addresses), there is one more const.

2 Reply

Alex Author

Reply to XenophonSoulis [13] January 29, 2025 11:46 pm PST

That's fun. Added to lesson. Thanks!

4 Reply

X_Taste

Reply to Alex [14] May 15, 2025 10:59 am PDT

is that perversion.

0 Reply

tmp

Reply to Alex [14] March 20, 2025 6:00 am PDT

You can even add more const if you have a const function parameter :)

Really pretty

0 Reply

here4cpp

December 27, 2024 9:49 am PST

you said that a constexpr basically means its required to be evaluated at compile time. But clearly, if a function contains stuff that are non-constexpr (like a member variable, say), then that should cause a compile time error ryt?

But here,

```
constexpr void reset() // constexpr but non-const
{
    m_x = m_y = 0; // non-const member function can change members
}
```

But here, you have done this. Now if I define a class object like you did:

```
Pair p1 { 3, 4 };
p1.reset();
```

Then, here, p1 not being a const/constexpr type means its member vars arent constexpr either anymore. So the constexpr function reset() should basically not be able to evaluate itself at compile time as it cant actually check up m_x and m_y since they are part of runtime and arent constexpr themselves. So reset should not work ryt?

This was just an example that I think is appropriate for my question, but my question is based on the Author's Note you wrote by the end. If u feel like giving a better example then I'm all for it. Can you explain the mechanism behind whats going on here?

0 Reply

Alex Author

Reply to here4cpp [15] January 3, 2025 5:09 pm PST

Constexpr means it can be evaluated at compile-time or runtime.

`reset()` is constexpr but non-const. The constexpr which means it can be evaluated at compile-time or runtime, and the non-const means it can modify members. `p1` is non-const, so we can call a non-const member function on it. The fact that `reset()` is constexpr is irrelevant here.

The `p3` case in the example shows when `reset()` is actually evaluated at compile-time. Because `zero()` is evaluated at compile-time in this case, `reset()` must be evaluated at compile-time. But because `p` and `reset()` are non-const, the member of `p` can be modified even though this modification is happening as part of compile-time evaluation.

1 Reply

here4cpp

Reply to Alex [16] January 3, 2025 6:11 pm PST

so the idea is that:

- constexpr for variables makes them constant as well but it is forced to be assigned at compile time e.g. I can do `int x = 10;` but I can't do `int x = a+b+c;`.
- constexpr for functions means it should be 'capable' of being evaluated at compile time. That means whether it will be evaluated at runtime or compile time depends on how its called. But it as long as the input can be evaluated at compile time, the function will also be in compile time. And that's why a constexpr function cannot use things like I/O or dynamic allocation etc.

As a simple example I made this:

```
1  constexpr int f(int x){
2      x++;
3      int y = 10;
4      // map<int, int> m = new map<int, int>({1, 2}); // will cause error
5  as this is dynamic allocation
6      return x+y;
7  }
8
9  int main(){
10     int x = 10;
11     x = f(x);
12     cout<<x<<endl;
13     map<int, int> *m = new map<int, int>(); // no error here
14     (*m)[1] = 2;
   }
```

Now, this shows that simple arithmetic operations are capable of being done during compile time itself. Clearly, I/O and dynamic memory operations are inherently runtime. Can you tell me like a list of operations which are done at runtime ONLY?

Also, the example above, is it a good example to demonstrate what u said in the author's note at the end "And in the case of constexpr functions, that’s possible even when they return a non-const object!" ?

0 Reply

Alex Author

Reply to here4cpp [17] January 16, 2025 10:12 pm PST

The initializer of a constexpr variable only needs to be a constant expression, not a literal. So `constexpr int x = a+b+c;` is fine so long as `a`, `b`, and `c` are constexpr.

> But it as long as the input can be evaluated at compile time, the function will also be in compile time.

No. The function must be called at compile-time if it is called in a context where compile-time evaluation is required. Otherwise, if it is capable of being called at compile-time, it may or may not. And clearly if it is not capable of being called at compile-time, it can't be.

> Can you tell me like a list of operations which are done at runtime ONLY?

I don't think I've ever seen one. If you're not sure, you can always try it (just call the function in a context that requires a constant expression, or make it consteval temporarily to ensure it's getting called at compile-time. The compiler will tell you if you do something invalid.

> Also, the example above, is it a good example to demonstrate what u said in the author's note at the end

Well, you don't use a constexpr function that returns a non-const object to initialize a constexpr variable, so no. If you did something like this: `constexpr int x = f(1);` then sure.

1 Reply

Torto

December 5, 2024 9:52 am PST

Hey, I don't really understand why making the construction `constexpr` here lead to a linker error ? `Monster.h` :

```
#ifndef MONSTER_H
#define MONSTER_H

#include <string>
#include <string_view>


class Monster
{
public:
    enum Type
    {
        dragon,
        goblin,
        ogre,
        orc,
        skeleton,
        troll,
        vampire,
        zombie,
        maxMonsterTypes,
    };

private:
    Type m_type{};
    std::string m_name{"???"};
    std::string m_roar{"???"};
    int m_hp{};

public:
    //Linker error LNK2019 if constructor made constexpr
    Monster(const Type type, const std::string_view name, const std::string_view roar,
const int hp);

    constexpr std::string_view getTypeString() const;
};

#endif
```

`Monster.cpp`:

```cpp
#include "Monster.h"
#include <string_view>

//Linker error LNK2019 if constructor made constexpr
Monster::Monster(const Type type, const std::string_view name, const std::string_view
roar,const int hp)
    : m_type {type}, m_name{name}, m_roar {roar}, m_hp{hp}
{
}


constexpr std::string_view Monster::getTypeString() const
{
    switch (m_type)
    {
    case dragon:
        return "dragon";
    case goblin:
        return "goblin";
    case ogre:
        return "ogre";
    case orc:
        return "orc";
    case skeleton:
        return "skeleton";
    case troll:
        return "troll";
    case vampire:
        return "vampire";
    case zombie:
        return "zombie";
    default:
        return "???";
    }
}
```

`main.cpp` :

```cpp
#include "Monster.h"

int main()
{
    Monster skeleton{ Monster::skeleton, "Bones", "*rattle*", 4 };

    return 0;
}
```

Last edited 6 months ago by Torto

0 Reply

gpowo

Reply to Torto [18] March 16, 2025 3:45 am PDT

gpt said :

The error occurs because marking the constructor (or any member function) as constexpr requires its definition to be available in every translation unit that uses it. In your code, if you change the constructor to constexpr, its definition in Monster.cpp won't be visible to translation units that need it (for inline expansion or compile-time evaluation), leading to the unresolved external symbol error (LNK2019).

In short, constexpr functions must be defined in the header (or marked inline) so that every file including the header sees the definition. Since your constructor is defined in Monster.cpp, making it constexpr violates this rule and causes the linker error.

1 Reply

Alex Author

Reply to Torto [18] December 6, 2024 2:12 am PST

I think it's because you have `std::string` members in the class, and `std::string` isn't constexpr compatible.

1 Reply

Albert

November 28, 2024 1:50 pm PST

Man hopefully in the future the compiler will be smart enough to add constexpr whenever needed by themself so we won't have to worry about it. Lol

0 Reply

Alex Author

Reply to Albert [19] December 1, 2024 9:53 pm PST

I doubt we'll see that happen. Constexpr is part of the interface of a function or class. If a function or class that is constexpr later has constexpr removed (because it can no longer be so for some reason), then that will breaking code that uses that function or class in places where a constant expression is required.

In other words, making a function or class constexpr also implies that "this will always be constexpr in the future", and that's not something that we want to entrust to the compiler to determine on purely technical grounds.

2 Reply

David Pinheiro

November 6, 2024 6:29 am PST

Is there a reason why constructors can be made constexpr / consteval for classes whose objects cannot be constexpr? Why is this allowed to compile? It seems impossible to instanciate an object of this class with a consteval constructor

```cpp
#include <string>

class Foo
{
public:
    consteval Foo () {}        // why is constexpr / consteval use allowed

private:
    std::string m_string {};
};


int main()
{
    constexpr Foo f {};        // when this then fails to compile

    return 0;
}
```

0 Reply

Alex Author

Reply to David Pinheiro [20] November 9, 2024 9:41 pm PST

A constexpr/consteval function or class may be conditionally valid depending on the arguments passed in. For this reason, such functions and classes are evaluated for validity at the point of invocation/instantiation, not when defined.

0 Reply

David Pinheiro

November 5, 2024 1:50 am PST

Are implicit/defaulted constructors constexpr? If so, a note in this lesson would be great! It seems so from this example:

```cpp
class Pair
{
private:
    int m_x {};
    int m_y {};

public:
    Pair() {};                      // does NOT compile
    constexpr Pair() {};            // compiles
    Pair() = default;               // compiles
    constexpr Pair() = default;     // compiles
    // implicit default constructor //compiles

};

int main()
{
    constexpr Pair f { };

    return 0;
}
```

Last edited 7 months ago by David Pinheiro

0 Reply

Alex Author

Reply to David Pinheiro [21] November 7, 2024 6:39 pm PST

Added to lesson: "Implicitly defined constructors are constexpr if they can be defined as such. Explicitly defaulted constructors must be explicitly defined as constexpr."

Thanks for the great question!

0 Reply

jyc

October 29, 2024 7:56 am PDT

Is the comment in the main function a typo (under "Constexpr members may be needed with non-constexpr/non-const objects")? You meant compile-time instead of runtime? Thanks for the time!

```
1  int main()
2  {
3      constexpr int g { init() }; // init() evaluated in runtime context
4      std::cout << g << '\n';
5
6      return 0;
7  }
```

0 Reply

Alex Author

Reply to jyc [22] November 2, 2024 2:26 pm PDT

Yup, the lesson text was correct. The comment has been fixed. Thanks for pointing this out.

2 Reply

Ali

September 28, 2024 11:03 am PDT

in the `constexpr void reset()` function, if you make the function `consteval`, the compiler will complain. I guess it's totally an bad idea to make such a function work at compile time... isn't it?

1 Reply

Alex Author

Reply to Ali [23] September 30, 2024 5:12 pm PDT

No, it's not a bad idea in general, it's just a bad idea here. We want `reset()` to be usable at runtime too, and making it `consteval` disables this possibility.

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/constexpr-functions/
3. https://www.learncpp.com/cpp-tutorial/struct-aggregate-initialization/
4. https://en.cppreference.com/w/cpp/named_req/LiteralType
5. https://www.learncpp.com/cpp-tutorial/chapter-14-summary-and-quiz/
6. https://www.learncpp.com/
7. https://www.learncpp.com/cpp-tutorial/converting-constructors-and-the-explicit-keyword/
8. https://www.learncpp.com/constexpr-aggregates-and-classes/
9. https://www.learncpp.com/cpp-tutorial/what-language-standard-is-my-compiler-using/
10. https://www.learncpp.com/cpp-tutorial/using-function-templates-in-multiple-files/
11. https://gravatar.com/
12. https://www.learncpp.com/cpp-tutorial/constexpr-aggregates-and-classes/#comment-609909
13. https://www.learncpp.com/cpp-tutorial/constexpr-aggregates-and-classes/#comment-607122
14. https://www.learncpp.com/cpp-tutorial/constexpr-aggregates-and-classes/#comment-607177
15. https://www.learncpp.com/cpp-tutorial/constexpr-aggregates-and-classes/#comment-605781
16. https://www.learncpp.com/cpp-tutorial/constexpr-aggregates-and-classes/#comment-606176
17. https://www.learncpp.com/cpp-tutorial/constexpr-aggregates-and-classes/#comment-606190
18. https://www.learncpp.com/cpp-tutorial/constexpr-aggregates-and-classes/#comment-604915
19. https://www.learncpp.com/cpp-tutorial/constexpr-aggregates-and-classes/#comment-604614
20. https://www.learncpp.com/cpp-tutorial/constexpr-aggregates-and-classes/#comment-603886
21. https://www.learncpp.com/cpp-tutorial/constexpr-aggregates-and-classes/#comment-603856
22. https://www.learncpp.com/cpp-tutorial/constexpr-aggregates-and-classes/#comment-603644
23. https://www.learncpp.com/cpp-tutorial/constexpr-aggregates-and-classes/#comment-602452
24. https://g.ezoic.net/privacy/learncpp.com