8.11 — Static member variables

BY ALEX ON SEPTEMBER 14TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

Review of static keyword uses

In the lesson on **file scope and the static keyword**, you learned that static variables keep their values and are not destroyed even after they go out of scope. For example:

```
#include <iostream>

int generateID()
{
    static int s_id = 0;
    return ++s_id;
}

int main()
{
    std::cout << generateID() << '\n';
    std::cout << generateID() << '\n';
    std::cout << generateID() << '\n';

    return 0;
}
```

This program prints:

```
1
2
3
```

Note that s_id has kept its value across multiple function calls.

The static keyword has another meaning when applied to global variables -- it gives them internal linkage (which restricts them from being seen/used outside of the file they are defined in). Because global variables are typically avoided, the static keyword is not often used in this capacity.

Static member variables

C++ introduces two more uses for the static keyword when applied to classes: static member variables, and static member functions. Fortunately, these uses are fairly straightforward. We'll talk about static member variables in this lesson, and static member functions in the next.

Before we go into the static keyword as applied to member variables, first consider the following class:

```
class Something
{
public:
    int m_value = 1;
};

int main()
{
    Something first;
    Something second;

    first.m_value = 2;
```

```
13
14      std::cout << first.m_value << '\n';
15      std::cout << second.m_value << '\n';
16
17      return 0;
18  }
```

When we instantiate a class object, each object gets its own copy of all normal member variables. In this case, because we have declared two Something class objects, we end up with two copies of m_value: first.m_value, and second.m_value. first.m_value is distinct from second.m_value. Consequently, the program above prints:

```
2
1
```

Member variables of a class can be made static by using the static keyword. Unlike normal member variables, static member variables are shared by all objects of the class. Consider the following program, similar to the above:

```
1   class Something
2   {
3   public:
4       static int s_value;
5   };
6
7   int Something::s_value = 1;
8
9   int main()
10  {
11      Something first;
12      Something second;
13
14      first.s_value = 2;
15
16      std::cout << first.s_value << '\n';
17      std::cout << second.s_value << '\n';
18      return 0;
19  }
```

This program produces the following output:

```
2
2
```

Because s_value is a static member variable, s_value is shared between all objects of the class. Consequently, first.s_value is the same variable as second.s_value. The above program shows that the value we set using first can be accessed using second!

Static members are not associated with class objects

Although you can access static members through objects of the class (as shown with first.s_value and second.s_value in the example above), it turns out that static members exist even if no objects of the class have been instantiated! Much like global variables, they are created when the program starts, and destroyed when the program ends.

Consequently, it is better to think of static members as belonging to the class itself, not to the objects of the class. Because s_value exists independently of any class objects, it can be accessed directly using the class name and the scope resolution operator (in this case, Something::s_value):

```
1   class Something
```

```
{
public:
    static int s_value; // declares the static member variable
};

int Something::s_value = 1; // defines the static member variable (we'll discuss this section

int main()
{
    // note: we're not instantiating any objects of type Something

    Something::s_value = 2;
    std::cout << Something::s_value << '\n';
    return 0;
}
```

In the above snippet, s_value is referenced by class name rather than through an object. Note that we have not even instantiated an object of type Something, but we are still able to access and use Something::s_value. This is the preferred method for accessing static members.

Defining and initializing static member variables

When we declare a static member variable inside a class, we're telling the compiler about the existence of a static member variable, but not actually defining it (much like a forward declaration). Because static member variables are not part of the individual class objects (they are treated similarly to global variables, and get initialized when the program starts), you must explicitly define the static member outside of the class, in the global scope.

In the example above, we do so via this line:

```
int Something::s_value = 1; // defines the static member variable
```

This line serves two purposes: it instantiates the static member variable (just like a global variable), and optionally initializes it. In this case, we're providing the initialization value 1. If no initializer is provided, C++ initializes the value to 0.

Note that this static member definition is not subject to access controls: you can define and initialize the value even if it's declared as private (or protected) in the class.

If the class is defined in a .h file, the static member definition is usually placed in the associated code file for the class (e.g. Something.cpp). If the class is defined in a .cpp file, the static member definition is usually placed directly underneath the class. Do not put the static member definition in a header file (much like a global variable, if that header file gets included more than once, you'll end up with multiple definitions, which will cause a compile error).

Inline initialization of static member variables

There are a few shortcuts to the above. First, when the static member is a const integral type (which includes char and bool) or a const enum, the static member can be initialized inside the class definition:

```
class Whatever
{
public:
    static const int s_value = 4; // a static const int can be declared and initialized directl
};
```

In the above example, because the static member variable is a const int, no explicit definition line is needed.

Second, as of C++11, static constexpr members of any type that supports constexpr initialization can be initialized inside the class definition:

```
#include <array>
```

```
2
3  class Whatever
4  {
5  public:
6      static constexpr double s_value = 2.2; // ok
7      static constexpr std::array<int, 3> s_array = { 1, 2, 3 }; // this even works for classes t
8  };
```

An example of static member variables

Why use static variables inside classes? One great example is to assign a unique ID to every instance of the class. Here's an example of that:

```
1  class Something
2  {
3  private:
4      static int s_idGenerator;
5      int m_id;
6
7  public:
8      Something() { m_id = s_idGenerator++; } // grab the next value from the id generator
9
10     int getID() const { return m_id; }
11 };
12
13 // Note that we're defining and initializing s_idGenerator even though it is declared as priva
14 // This is okay since the definition isn't subject to access controls.
15 int Something::s_idGenerator = 1; // start our ID generator with value 1
16
17 int main()
18 {
19     Something first;
20     Something second;
21     Something third;
22
23     std::cout << first.getID() << '\n';
24     std::cout << second.getID() << '\n';
25     std::cout << third.getID() << '\n';
26     return 0;
27 }
```

This program prints:

1
2
3

Because s_idGenerator is shared by all Something objects, when a new Something object is created, the constructor grabs the current value out of s_idGenerator and then increments the value for the next object. This guarantees that each instantiated Something object receives a unique id (incremented in the order of creation). This can really help when debugging multiple items in an array, as it provides a way to tell multiple objects of the same class type apart!

Static member variables can also be useful when the class needs to utilize an internal lookup table (e.g. an array used to store a set of pre-calculated values). By making the lookup table static, only one copy exists for all objects, rather than making a copy for each object instantiated. This can save substantial amounts of memory.

8.12 -- Static member functions

Index

8.10 -- Const class objects and member functions

C++ TUTORIAL | PRINT THIS POST

141 comments to 8.11 — Static member variables

« Older Comments 1 2

Mingrui Zhang
January 17, 2020 at 7:11 am · Reply

I think maybe we can add a small section to talk about const static members as a warning to the people:
Since they cannot be initialized within the class definition.

nascardriver
January 17, 2020 at 7:17 am · Reply

`static const` members can be initialized in the class, as is shown in the lesson. Do you mean something else?

Fan
January 11, 2020 at 8:53 am · Reply

In "inline initialization of static member variables", does the inline mean that the "one definition rule" is overriden? Otherwise I guess if the definition of a class with inline initialization of static

member variables is included in multiple files it will violate the "one definition rule".

nascardriver
January 12, 2020 at 4:20 am · Reply

Yes, `inline` variables can be defined multiple times. This is covered in the recently updated **lesson 6.8**.

Anthony
January 5, 2020 at 1:59 pm · Reply

Hi again,

I'm currently browsing a project where they've made a whole bunch of classes where each class has a static pointer named 'singleton' pointed at the class itself, like this:

```
class A {
public:
    static A *singleton;

    // ctor
    A() {
        singleton = this;
        // ...
    }

    // ...
};
```

Could you tell me why they're doing this?

Anthony
January 6, 2020 at 10:39 am · Reply

Just to add a little detail, it's more like this:

```
class A {
private:
    static A *singleton;

public:
    // ctor
    A() {
        singleton = this;  // this is weird!
        // ...
    }

    static A* get_singleton() {
        return singleton;
    }

    // ...
};
```

Why have a static pointer that would point to every instance of the class, and assign it a value of `this`, which surely applies to a single instance? If you instantiate A, `singleton = this;` causes the linker to error with 'undefined reference'.

nascardriver
January 7, 2020 at 2:29 am · Reply

That's a singleton class, ie. a class of which only 1 instance can exist. The way the code looks, I'd say they're creating an `A` at some other point, because it depends on some other code to have run first. They only create an `A` one time! After that, whenever they want to use an `A`, they do

```
1 A::get_singleton()->something();
```

Singletons are like global objects, but more complex.

> If you instantiate A, `singleton = this;` causes the linker to error with 'undefined reference'.
The `singleton` in line 3 is only a declaration. The variable needs to be defined outside of the class, as shown in this lesson.

Anthony
January 9, 2020 at 6:20 pm · Reply

I see now. So we write the following definition outside the body of the class (typically in the .cpp):

```
1 A *A::singleton;
```

Excellent. Thank you.

Samira Ferdi
November 28, 2019 at 6:13 pm · Reply

Hi, Alex and Nascardriver!

So, correct me if I'm wrong, the meaning of static member variable are:
1. That member variable is shared between all objects of the class. So, any objects of the class refer the same original member variable of the class (not the copy). It's analogues with many references that refer to the same normal variable. It means all changes from one identifier affect all.

2. Who can access that static member variable. Static member variable can be accessed through it's class itself because it belongs to the class or through the instantiated class (the object of the class) with the effect that explain in point 1 above. Non-static member variables can only be accessed through the object of the class and cannot be accessed through it's own class.

3. Static keyword means when apply to a class is, to access the members of the class, we don't have to instantiated the class, although it's fine if we access the static members through the object of the class (of course with consideration of the exceptional things).

Alex, if you feel my explanation makes the meaning of static member variable clearer, I hope you can update this lesson for clearer explanation at the first place.

But, I'm not sure that I understand why static non-const member variable should initialized (or define) outside the class. Is this initialization outside the class is a statement that "these (static) member variables are not belong to the objects of the class" as static member variables belong to the class itself and because all non-static member variables that define inside a class belong to the object of that class? So, the static keyword in this term is like the separation of what belong to who. But, this understanding is not safe (apply in general) because the object of the class can have (access) the static member variables of the class. Any thought?

nascardriver
November 29, 2019 at 3:23 am · Reply

1. Correct.

2. Correct. `static` member variables are affected by access specifiers (private, protected, public), which can restrict access from the outside.

3. Correct.

> why static non-const member variable should initialized (or define) outside the class
As most variables, a static member variable must only be defined once. Since the class declaration occurs many times (Everywhere where its header is included), it wouldn't be clear who is responsible for the initialization if the definition was in the header. .cpp files only get compiled once, so we can define and initialize the variable there.

Nguyen
November 13, 2019 at 1:52 pm · Reply

Hi,

In the last example, I assume that s_idGenerator @ line 8 keeps its value and is not destroyed even after it goes out of scope and therefore when a new Something object is created, the constructor grabs the current value out of s_idGenerator and then increments the value for the next object. Please let me know if my assumptions are correct.

Thanks

nascardriver
November 14, 2019 at 1:45 am · Reply

You're right. `s_idGenerator` is unique and only dies when the program ends.

hellmet
October 29, 2019 at 7:54 am · Reply

Hello Nascardriver and Alex!
Hellmet again with doubts!
1) "Note that this static member definition is not subject to access controls: you can define and initialize the value even if it’s declared as private (or protected) in the class."

What does that mean? If that means to say that "private is of no use, anyone initialize that variable!", then okays! No issues.

2) Having been taught Python before C++; I tried to use Something.s_idGenerator, which was clearly incorrect. Any reasoning as to why the namespace resolution operator was used here? (It is the standard, true, but a technical and/or simple explanation would help me understand the language better!)

3) "Do not put the static member definition in a header file (much like a global variable, if that header file gets included more than once, you’ll end up with multiple definitions, which will cause a compile error)". So, this means that I can safely initialize my static class variable to the value I want and am rest assured that no one can use that same 'definition' to 'override' my definition, right?
Say a client put his own 'int Something::s_idGenerator = 1;' before his main while using my library (via source header files and precompiled library). The imports happen fine, but the linker fails complaining that there was a redefinition, right?

Thank you!

nascardriver

October 29, 2019 at 9:06 am · Reply

1)
Yes, anyone can initialize the variable. Just like anyone can define private member functions.

2)
A class with static members is more like a namespace than like an instance. I don't know if that's the reasoning, but it seems good to me.

3)
I never even thought of trying that. But yes, the linker will complain about multiple definitions. Like member functions, static member variables should be initialized in a corresponding source file.

hellmet
October 29, 2019 at 9:09 am · Reply

Perfect!
The consequence of 1, 3 is that I must make sure I initialize all the static variables of my class in the .cpp file else someone can change them under my nose, correct?

nascardriver
October 29, 2019 at 9:13 am · Reply

If you don't initialize them you'll get a linker error because they're never defined :)

hellmet
October 29, 2019 at 9:15 am · Reply

Whoops my bad! Should've tried that, apologies!
Thank you!

Phuc
October 17, 2019 at 7:53 pm · Reply

Hello !
Why method (int getID() const { return m_id; }) is const But the object created is not const but it still calls the method const

```
1    Something first;
2     Something second;
3     Something third;
4
5     std::cout << first.getID() << '\n';
6     std::cout << second.getID() << '\n';
7     std::cout << third.getID() << '\n';
```

nascardriver
October 18, 2019 at 1:31 am · Reply

There is no non-const `getID`, so the const `getID` is called.
Non-const objects can use const member functions, just the other way around doesn't work.

Samira Ferdi
October 1, 2019 at 4:45 pm · Reply

Hi, Alex and Nascardriver!

What do you mean 'copy'? I often very confuse about this term.

nascardriver
October 2, 2019 at 4:44 am · Reply

When multiple objects have the same value and will always have the same value, they're copies of each other.

```
for (int i{ 0 }; i < 3; ++i)
{
  int iIndex{ i }; // @iIndex is a copy of @i

  // unless we modify @iIndex here.

  // If we don't modify @iIndex, there's no reason
  // not to use @i and remove @iIndex.
}
```

mmp52
August 6, 2019 at 11:47 pm · Reply

Why do we use "int" declaration here? Don't we already have the definition of the static member variable when we define the class itself?

```
int Something::s_value = 1; // defines the static member variable
```

I was expecting a direct assignment:

```
Something::s_value = 1; // defines the static member variable
```

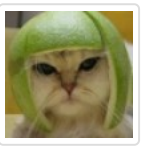
Alex
August 8, 2019 at 7:41 pm · Reply

No -- what's in the class (static int s_value) is a declaration.

In order to be instantiated, the variable needs a definition. The top one is a definition. The bottom one is just a statement.

McDidda
July 5, 2019 at 10:09 pm · Reply

why definitions of static member function/variable (defined outside of class) doesn't have "static" keyword used to define the function/variable?



Alex
July 8, 2019 at 10:27 am · Reply

I'm actually not sure. Anybody else have any insights on why the definition/initializer doesn't require the static keyword?

nascardriver
July 9, 2019 at 4:26 am · Reply

The function definition cannot be ambiguous, ie. it can be uniquely resolved to its declaration, even without `static` at the definition. Just like `virtual` or default arguments, adding `static` to the definition would be redundant.

McDidda
July 4, 2019 at 9:53 pm · Reply

Hi,
Could you please explain the reason for not defining/initializing static member s_value inside class Something, and instead defining it outside, but in case of const static member variable we define it inside class?

```
1  class Something
2  {
3  public:
4      static int s_value; // declares the static member variable
5  };
6
7  int Something::s_value = 1; // defines the static member variable (we'll discuss this secti
8
9  int main()
10 {
11     // note: we're not instantiating any objects of type Something
12
13     Something::s_value = 2;
14     std::cout << Something::s_value << '\n';
15     return 0;
16 }
```

Alex
July 7, 2019 at 12:31 pm · Reply

If the static is a const integral value, the compiler can usually optimize it away. Otherwise, it needs to be defined outside the class because it's essentially a global variable, not tied to objects of the class.

Michael Djamoos
August 21, 2019 at 10:29 am · Reply

Also, since the variable is essentially a global variable, you would not want each new object instantiation to initialize it again.

Alex
August 21, 2019 at 8:42 pm · Reply

While true, C++ handles this fine for local static variables, which can have an initializer, but are only initialized once despite the initializer being inside the function.

sekhar
May 5, 2019 at 1:50 am · Reply

Hi Alex, Tutorials are amazing and leaned a lot, Thanks for these awesome tutorials.

Have a question for below code snippet :

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  class Sample
6  {
7  private:
8      static const int x;
9  public:
10
11     Sample() { cout << "Static object constructor invoked " << endl; }
12
13     static const Sample s;
14
15     static int getstatic() const // applying const to static function gives error
16     {
17         return x;
18     }
19 };
20
21 const int Sample::x = 10;
22 const Sample Sample::s;
23
24 int main()
25 {
26     cout << Sample::s.getstatic() << endl; //error: static member function ‘static int Samp
27     return 0;
28 }
```

Just need clarity if really makes sense to apply const to static member functions/objects in a class and why const can't be applied to static function even though it gets invoked with static const object ?

nascardriver
May 5, 2019 at 2:05 am · Reply

* Line 13, 3, 8: Initialize your variables with brace initializers.
* Line 15, 26: Limit your lines to 80 characters in length for better readability on small displays.
* Don't use "using namespace".

static members aren't linked to an object. s.getstatic() is the same as Sample::getstatic().

sekhar
May 6, 2019 at 8:17 am · Reply

Thanks nascardriver. Could you please let me know the reason why const can't be applied to static member functions ? Is there any particular reason that you can think of ?

nascardriver

May 6, 2019 at 8:26 am · Reply

There's no reason why this should be possible. They aren't members of an object, so the pseudo-object they're accessing can't be const, so they can always modify it and you can always call them.
Marking those functions const would only help you to not accidentally modify any static members in a function where you didn't intend to. Just like marking pass-by-value arguments as const, this has no practical effect.

Fan
January 12, 2020 at 2:25 pm · Reply

Maybe another reason is that the "const" in member functions really means that the this pointer passed to the function is pointing a const object. Static member functions do not operate on objects, so they do not have a this pointer as an argement, so there is no way to make the this pointer constant.

Matty J
February 7, 2019 at 5:47 pm · Reply

So let's say you build a class Monster, and nest a class Keese within it.
Let's say you wanted Link's level (Zelda is now an RPG in this example) to scale the Keese's health
You could write Monster::Keese::m_health += 1.2 * levelLink to scale all Keese (normal, fire, etc.) instead of each one individually

This example is probably better with Oblivion, but I've already typed it

prabhu
October 12, 2018 at 9:34 pm · Reply

Hi Alex/Nascardriver,

I have a question on inline initialization on static member variable.
When we initialize like
static int a; , we have to define the variable outside the class , and duration/scope of the variable will be persist till end of class and the static variable belong to class rather then object, so it will get updated every time its called through object or scope resolution operator. I am ok with the above points.

if i define & declaration like const int a =10 ;
here I should initialize the content in the declaration itself since its const and it will not allow us to update the value of a .I am ok with the above points here also

But If i declare and define like static const int a = 10 means ,
what are the properties , it will take forward from static and const.

Alex
October 15, 2018 at 9:28 am · Reply

> static const int a = 10

This behaves like a static that you can not change the value of.

Udit
October 1, 2018 at 1:23 am · Reply

The lookup table example could be:

```
#include <iostream>
#include <vector>

using namespace std;

class Table{
private:
    static vector<int> m_enteries;
    static int id_generator;
    int m_id;
public:
    Table():m_id{++id_generator}{};
    const int get_id(void){
        return m_id;
    }
    void push(int num){
        m_enteries.push_back(num);
    }
    static void get_enteries(void){
        for(int i=0;i<m_enteries.size();++i){
            cout << m_enteries[i] << " ";
        }
        cout << endl;
    }
};

int Table::id_generator=0;
vector<int> Table::m_enteries{};

int main(int argc, char const *argv[]){
    Table obj1;
    Table obj2;
    fprintf(stdout, "obj1 id: %d\nobj2 id: %d\n", obj1.get_id(), obj2.get_id());
    obj1.push(2);
    obj2.push(3);
    Table::get_enteries();
    return 0;
}
```

Micah
August 10, 2018 at 10:59 pm · Reply

Why does it print backwards if you don't initialize a new std::cout call?
output for this is:

0
1
3 2

```
#include "stdafx.h"
#include <iostream>

using namespace std;

int addOne()
{
    static int s_num = -1;
```

```
    return ++s_num;
}




int main()
{

    cout << addOne() << "\n";
    cout << addOne() << "\n";
    cout << addOne() << " " << addOne() << "\n";    /* Prints backwards? Why ? */

    return 0;
}
```

nascardriver
August 11, 2018 at 6:02 am · Reply

Hi Micah!
The evaluation order of multiple << is undefined.

Micah
August 11, 2018 at 12:32 pm · Reply

Thanks, I see, then different statements guarantee evaluation order.
Okay then a couple questions if you don't mind..
If they are undefined in evaluation order, why do they evaluate backwards? Does the computer just have to assume how to print and evaluate with the operator? And if so, why does it assume the same way each time?
Would that not be an update-able fix to reverse the process?
And, is there another way to send multiple calls like this to the console on one line? Or is this just a qwerk of C++?

nascardriver
August 12, 2018 at 1:22 am · Reply

The compiler decides the evaluation order. Yours appears to evaluate the statement backwards. As long as you're using the same compiler, you'll get the same results.

> Would that not be an update-able fix to reverse the process?
Reversing the order is probably easy, but the c++ standard doesn't state a specific order, so there is no reason to choose one way over the other.

> is there another way to send multiple calls like this to the console on one line?
Store the values in temporaries before printing them

```
// * Don't use "using namespace"
// * Initialize your variables with uniform initialization

int a{ addOne() };
int b{ addOne() };
std::cout << a << " " << b << '\n';
```

The behavior of this code is the same across all compilers.

Aditi
July 31, 2018 at 8:09 pm · Reply

Thank you for the great examples . They were really easy to understand and explained the concept very well :)

Dr. Khalid Alharbi
June 20, 2018 at 1:45 am · Reply

Hi Alex,

```
1 "Note that this static member definition is not subject to access controls: you can define a
```

I think static member definition and initialization is not subject to access controls,but static members that are under private section cannot be accessed directly using the class name or through objects of the class. We need to use static member functions to access (i.e., read and/or write) static data members.
The following code works fine, but if you try to run the comments you will receive an error: ‘int Something::s_value’ is private within this context.

[code]

```
#include <iostream>
using namespace std;
class Something
{
    public:
    Something()
    {
       setvalue();

    }
    static int getvalue()
    {
       return s_value;
    }
    static void setvalue()
    {
        ++s_value;
    }
     private:
     static int s_value;
};

int Something::s_value=0 ;

int main()
{
   Something first;
   Something second;

   //first.s_value = 2;
  // cout << first.s_value << '\n';
  //cout << second.s_value << '\n';
  //cout << Something::s_value << '\n';
  cout << Something::getvalue() << '\n';
```

```
    return 0;
}
[code]
```

Peter Baum
May 18, 2018 at 1:06 pm · Reply

The static member definition and initialization looks just like a global to me in terms of where it is defined and its function. The only thing slightly different is that it is namespace qualified by the class name. Is there anything else that differentiates it from an ordinary global?

nascardriver
May 19, 2018 at 3:35 am · Reply

Hi Peter!

Access modifiers apply to static members.

April
March 18, 2018 at 5:26 pm · Reply

How is it that the first, second, and third objects can access the getId member function if it is const and the objects themselves are not?

nascardriver
March 19, 2018 at 2:12 am · Reply

Hi April!

Non-const objects can access const-functions (The object is allowed to be modified, the function doesn't modify anything, everything is fine).
Const objects can't access non-const functions (The object is not allowed to be modified, the function wants to modify the object, not allowed).

Arnav Borborah
February 20, 2018 at 5:17 pm · Reply

You have written:

"Second, as of C++11, static constexpr members of any type can be initialized inside the class definition:"

Does this apply for classes without constexpr constructors? How would they be initialized at compile time? (I presume they won't since classes such as std::vector use runtime dynamic allocation)

nascardriver
February 21, 2018 at 6:30 am · Reply

Hi Arnav!

I'm not sure if I got your question right since what you wrote isn't related to your quote.

The quote states that this is allowed

```
1 class CConst
2 {
```

```
    // Can be initialized here, whereas a static int has do be intialized outside
    // of this class
    static constexpr int c_iMyInt{ 123 };
};
```

What (I think) you asked is having a constexpr object of a type without a constexpr constructor

```
class CNonConst
{
public:

  int iMember{ 0 };

  /* constexpr */ CNonConst(int i) : iMember(i) // Uncomment to make it work
  {

  }
};

// Not allowed, because CNonConst::CNonConst(int) is not a constexpr
constexpr CNonConst nonConst{ CNonConst(33) };
```

Alex
February 22, 2018 at 11:51 am · Reply

constexpr variables aren't initialized at compile time (they are initialized at run-time). However, the value they are initialized with must be known at compile time.

C++ doesn't currently support const (or constexpr) constructors in any case.

Arnav Borborah
February 22, 2018 at 3:18 pm · Reply

Alex, C++ actually [does support `constexpr` constructors](http://en.cppreference.com/w/cpp/language/constexpr). (Scroll down a bit to see it).

nascardriver, I'm asking that if I am allowed to do:

```
class CConst
{
  // Can be initialized here, whereas a static int has do be intialized outside
  // of this class
  static constexpr CNonConst test{ 123 };
};
```

When CNonConst does NOT have a constexpr constructor. I am confused about why it says "any type" when this is not allowed for classes without a constexpr constructor.

Alex
February 26, 2018 at 11:39 am · Reply

1) I stand corrected!
2) I see what you're getting at. I'll amend the text to be more specific that this applies only to types that can actually be created as constexpr.

« Older Comments 1 2