What does 'const static' mean in C and C++?

```
const static int foo = 42;
```

I saw this in some code here on StackOverflow and I couldn't figure out what it does. Then I saw some confused answers on other forums. My best guess is that it's used in C to hide the constant `foo` from other modules. Is this correct? If so, why would anyone use it in a C++ context where you can just make it `private` ?

c++ c

edited Jun 14 '15 at 21:27
Alexis Wilke
8,277 2 30 62

asked Oct 7 '08 at 6:59
c0m4
1,938 5 29 36

11 Answers

It has uses in both C and C++.

As you guessed, the `static` part limits its scope to that compilation unit. It also provides for static initialization. `const` just tells the compiler to not let anybody modify it. This variable is either put in the data or bss segment depending on the architecture, and might be in memory marked read-only.

All that is how C treats these variables (or how C++ treats namespace variables). In C++, a member marked `static` is shared by all instances of a given class. Whether it's private or not doesn't affect the fact that one variable is shared by multiple instances. Having `const` on there will warn you if any code would try to modify that.

If it was strictly private, then each instance of the class would get its own version (optimizer notwithstanding).

edited May 23 at 12:26
Community ♦
1 1

answered Oct 7 '08 at 7:05
Chris Arguin
9,040 3 24 42

The original example is talking about a "private variable". Therefore, this is a mebmer and static has *no* affect on the linkage. You should remove "the static part limits it's scope to that file". – Richard Corden Oct 7 '08 at 9:15

The "special section" is known as the data segment, which it shares with all the other global variables, such as explicit "strings" and global arrays. This is opposing to the code segment. – spoulson Oct 7 '08 at 10:02

@Richard - what makes you think it's a member of a class? There's nothing in the question that says it is. *If* it's a member of a class, then you're right, but if it's just a variable declared at global scope, then Chris is right. – Graeme Perrow Oct 7 '08 at 10:42

1 The original poster mentioned private as a possible better solution, but not as the original problem. – Chris Arguin Oct 8 '08 at 4:44

@Graeme, OK so it's not "definitely" a member - however, this answer is making statements which only apply to namespace members and those statements are wrong for members variables. Given the amount of votes that error may confuse someone not very familiar with the language - it should be fixed. – Richard Corden Oct 8 '08 at 10:07

A lot of people gave the basic answer but nobody pointed out that in C++ `const` defaults to `static` at `namespace` level (and some gave wrong information). See the C++98 standard section 3.5.3.

First some background:

Translation unit: A source file after the pre-processor (recursively) included all its include files.

Static linkage: A symbol is only available within its translation unit.

External linkage: A symbol is available from other translation units.

At `namespace` level

This includes the global namespace aka global variables.

```
static const int sci = 0; // sci is explicitly static
const int ci = 1;         // ci is implicitly static
extern const int eci = 2; // eci is explicitly extern
extern int ei = 3;        // ei is explicitly extern
int i = 4;                // i is implicitly extern
static int si = 5;        // si is explicitly static
```

At function level

`static` means the value is maintained between function calls.
The semantics of function `static` variables is similar to global variables in that they reside in the program's data-segment (and not the stack or the heap), see this question for more details about `static` variables' lifetime.

At `class` level

`static` means the value is shared between all instances of the class and `const` means it doesn't change.

edited May 23 at 12:18
Community ♦
1 1

answered Oct 7 '08 at 12:44
Motti
62k 28 146 212

4 +1 for including the function level. – rkjnsn Mar 20 '12 at 18:30

2 At function level: static is not reduntant with const, they can behave differently `const int *foo(int x) {const int b=x;return &b};` versus `const int *foo(int x) {static const int b=x;return &b};` – Hanczar Jun 23 '12 at 18:19

1 The question is about both C and C++ so you should include a note about `const` only implying `static` in the latter. – Nikolai Ruhe Jun 6 '13 at 13:05

1 @Motti `const` doesn't imply static at the function level, that would be a concurrency nightmare (const != constant expression), everything at the function level is implicitly `auto` . Since this question is tagged [c] as well, I should mention that a global level `const int` is implicitly `extern` in C. The rules you have here perfectly describe C++, however. – Ryan Haining Sep 25 '15 at 18:12

1 And in C++, in all three cases, `static` indicates that the variable is static duration (only one copy exists, which lasts from the program's beginning until its end), and has internal/static linkage if not otherwise specified (this is overridden by the function's linkage for local static variables, or the class' linkage for static members). The main differences are in what this implies in each situation where `static` is valid. – Justin Time Jul 27 '16 at 21:08

That line of code can actually appear in several different contexts and alghough it behaves approximately the same, there are small differences.

Namespace Scope

```
// foo.h
static const int i = 0;
```

' `i` ' will be visible in every translation unit that includes the header. However, unless you actually use the address of the object (for example. ' `&i` '), I'm pretty sure that the compiler will treat ' `i` ' simply as a type safe `0` . Where two more more translation units take the ' `&i` ' then the address will be different for each translation unit.

```
// foo.cc
static const int i = 0;
```

' `i` ' has internal linkage, and so cannot be referred to from outside of this translation unit. However, again unless you use its address it will most likely be treated as a type-safe `0` .

One thing worth pointing out, is that the following declaration:

```
const int i1 = 0;
```

is **exactly** the same as `static const int i = 0` . A variable in a namespace declared with `const` and not explicitly declared with `extern` is implicitly static. If you think about this, it was the intention of the C++ committee to allow `const` variables to be declared in header files without always needing the `static` keyword to avoid breaking the ODR.

Class Scope

```
class A {
public:
  static const int i = 0;
};
```

In the above example, the standard explicitly specifies that ' `i` ' does not need to be defined if its address is not required. In other words if you only use ' `i` ' as a type-safe 0 then the compiler will not define it. One difference between the class and namespace versions is that the address of ' `i` ' (if used in two ore more translation units) will be the same for the class member. Where the address is used, you must have a definition for it:

```
// a.h
class A {
public:
  static const int i = 0;
};

// a.cc
#include "a.h"
const int A::i;            // Definition so that we can take the address
```

answered Oct 7 '08 at 9:42
Richard Corden
17k 6 49 76

nice explanation! – Baiyan Huang Mar 11 '11 at 1:55

2 +1 for pointing out that static const is same as just const in namespace scope. – Plumenator Jun 1 '11 at 9:09

There is actually no difference between placing in "foo.h" or in "foo.cc" since .h is simply included when compiling translation unit. – Mikhail Jul 28 '11 at 15:38

2 @Mikhail: You're correct. There is an assumption that a header can be included in multiple TUs and so it was useful to talk about that separately. – Richard Corden Aug 1 '11 at 17:44

It's a small space optimization.

When you say

```
const int foo = 42;
```

You're not defining a constant, but creating a read-only variable. The compiler is smart enough to use 42 whenever it sees foo, but it will also allocate space in the initialized data area for it. This is done because, as defined, foo has external linkage. Another compilation unit can say:

extern const int foo;

To get access to its value. That's not a good practice since that compilation unit has no idea what the value of foo is. It just knows it's a const int and has to reload the value from memory whenever it is used.

Now, by declaring that it is static:

```
static const int foo = 42;
```

The compiler can do its usual optimization, but it can also say "hey, nobody outside this compilation unit can see foo and I know it's always 42 so there is no need to allocate any space for it."

I should also note that in C++, the preferred way to prevent names from escaping the current compilation unit is to use an anonymous namespace:

```
namespace {
    const int foo = 42; // same as static definition above
}
```

edited Jul 12 '16 at 17:36

answered Oct 7 '08 at 10:37
Ferruccio
73.4k 34 186 275

1 ,u mentioned without using static "it will also allocate space in the initialized data area for it". and by using static "no need to allocate any space for it".(from where the compiler is picking the val then ?) can you explain in term of heap and stack where the variable is getting stored .Correct me if I am interpretting it wrong . – N.Nihar Mar 31 '15 at 4:09

@N.Nihar - The static data area is a fixed size chunk of memory which contains all data which has static linkage. It's "allocated" by the process of loading the program into memory. It's not part of the stack or heap. – Ferruccio Mar 31 '15 at 10:10

It's missing an 'int'. It should be:

```
const static int foo = 42;
```

In C and C++, it declares an integer constant with local file scope of value 42.

Why 42? If you don't already know (and it's hard to believe you don't), it's a refernce to the **Answer to Life, the Universe, and Everything**.

edited Oct 7 '08 at 7:07

answered Oct 7 '08 at 7:02
Kevin
11.5k 13 48 56

Yes, thanks. I editet the question – c0m4 Oct 7 '08 at 7:06

1 I do know about thhgttg :-) – c0m4 Oct 7 '08 at 7:07

Thanks... now everytime... for the rest of my life...when I see 42, I will always thing about this. haha – Inisheer Oct 7 '08 at 7:08

This is proof-positive that the universe was created by being with 13 fingers (the question and answer actually match in base 13). – paxdiablo Oct 7 '08 at 7:42

Its the mice. 3 toes on each foot, plus a tail gives you base 13. – KeithB Oct 7 '08 at 11:57

In C++,

```
static const int foo = 42;
```

is the preferred way to define & use constants. I.e. use this rather than

```
#define foo 42
```

because it doesn't subvert the type-safety system.

answered Oct 7 '08 at 7:49
paxos1977
51.9k 20 73 110

This ia s global constant visible/accessible only in the compilation module (.cpp file). BTW using static for this purpose is deprecated. Better use an anonymous namespace and an enum:

```
namespace
{
  enum
  {
     foo = 42
  };
}
```

answered Oct 7 '08 at 7:06
Roskoto
799 1 10 26

this would force the compiler to not treat foo as a constant and as such hinders optimization. – Nils Pipenbrinck Oct 7 '08 at 7:13

enums values are always constant so I don't see how this will hinder any optimizations – Roskoto Oct 7 '08 at 7:17

ah - true.. my error. thought you've used a simple int-variable. – Nils Pipenbrinck Oct 7 '08 at 7:31

Roskoto, I'm not clear what benefit the `enum` has in this context. Care to elaborate? Such `enums` are usually only used to prevent the compiler from allocating any space for the value (though modern compilers don't need this `enum` hack for it) and to prevent the creation of pointers to the value. – Konrad Rudolph Oct 7 '08 at 7:55

Konrad, What exactly problem you see in using an enum in this case? Enums are used when you need constant ints which is exactly the case. – Roskoto Oct 7 '08 at 8:12

To all the great answers, I want to add a small detail:

If You write plugins (e.g. DLLs or .so libraries to be loaded by a CAD system), then *static* is a life saver that avoids name collisions like this one:

1. The CAD system loads a plugin A, which has a "const int foo = 42;" in it.
2. The system loads a plugin B, which has "const int foo = 23;" in it.
3. As a result, plugin B will use the value 42 for foo, because the plugin loader will realize, that there is already a "foo" with external linkage.

Even worse: Step 3 may behave differently depending on compiler optimization, plugin load mechanism, etc.

I had this issue once with two helper functions (same name, different behaviour) in two plugins. Declaring them static solved the problem.

answered Oct 8 '08 at 7:57
Black
3,663 1 14 27

Yes, it hides a variable in a module from other modules. In C++, I use it when I don't want/need to change a .h file that will trigger an unnecessary rebuild of other files. Also, I put the static first:

```
static const int foo = 42;
```

Also, depending on its use, the compiler won't even allocate storage for it and simply "inline" the value where it's used. Without the static, the compiler can't assume it's not being used elsewhere and can't inline.

answered Oct 7 '08 at 7:06
Jim Buck
16k 8 43 68

Making it private would still mean it appears in the header. I tend to use "the weakest" way that works. See this classic article by Scott Meyers: http://www.ddj.com/cpp/184401197 (it's about functions, but can be applied here as well).

answered Oct 7 '08 at 7:07
yrp
3,939 1 19 10

According to C99/GNU99 specification:

- `static`
  - is storage-class specifier
  - objects of file level scope by default has **external** linkage
  - objects of file level scope with static specifier has **internal** linkage
- `const`
  - is type-qualifier (is a part of type)
  - keyword applied to immediate left instance - i.e.
    - `MyObj const * myVar;` - unqualified pointer to const qualified object type
    - `MyObj * const myVar;` - const qualified pointer to unqualified object type
  - Leftmost usage - applied to the object type, not variable

- `const MyObj * myVar;` - unqualified pointer to const qualified object type

THUS:

`static NSString * const myVar;` - constant pointer to immutable string with internal linkage.

Absence of the `static` keyword will make variable name global and might lead to name conflicts within the application.

edited May 11 '16 at 17:24

answered May 11 '16 at 17:13

Alexey Pelekh

421 3 12