The static keyword and its various uses in C++

The keyword `static` is one which has several meanings in C++ that I find very confusing and I can never bend my mind around how its actually supposed to work.

From what I understand there is `static` storage duration, which means that it lasts for the lifetime of the program in the case of a global, but when you're talking about a local, it means that it's zero initialized by default.

The C++ Standard says this for class data members with the keyword `static`:

3.7.1 Static storage duration [basic.stc.static]

> 3 The keyword static can be used to declare a local variable with static storage duration.
>
> 4 The keyword static applied to a class data member in a class definition gives the data member static storage duration.

What does it mean with *local variable*? Is that a function local variable? Because there's also that when you declare a function local as `static` that it is only initialized once, the first time it enters this function.

It also only talks about storage duration with regards to class members, what about it being non instance specific, that's also a property of `static` no? Or is that storage duration?

Now what about the case with `static` and file scope? Are all global variables considered to have static storage duration by default? The following (from section 3.7.1) seems to indicate so:

> 1 All variables which do not have dynamic storage duration, do not have thread storage duration, and are **not local** have static storage duration. The storage for these entities shall last for the duration of the program (3.6.2, 3.6.3)

How does `static` relate to the linkage of a variable?

This whole `static` keyword is downright confusing, can someone clarify the different uses for it English and also tell me *when* to initialize a `static` class member?

c++ static

edited Mar 5 '13 at 22:43

asked Mar 5 '13 at 22:38

Tony The Lion

37.8k ● 45 ● 179 ● 344

9 Answers

Variables:

`static` variables exist for the "lifetime" of the *translation unit that it's defined in*, and:

- If it's in a namespace scope (i.e. outside of functions and classes), then it can't be accessed from any other translation unit. This is known as "internal linkage". (Dont' do this in headers, it's just a terrible idea, you end up with a separate variable in each translation unit, which is crazy confusing)
- If it's a variable *in a function*, it can't be accessed from outside of the function, just like any other local variable. (this is the local they mentioned)
- class members have no restricted scope due to `static`, but can be addressed from the class as well as an instance (like `std::string::npos`). [Note: you can *declare* static members in a class, but they should usually still be *defined* in a translation unit (cpp file), and as such, there's only one per class]

Before any function in a translation unit is executed (possibly after `main` began execution), the variables with static storage duration in that translation unit will be "constant initialized" (to `constexpr` where possible, or zero otherwise), and then non-locals are "dynamically initialized" properly *in the order they are defined in the translation unit* (for things like `std::string="HI";` that aren't `constexpr`). Finally, function-local statics are initialized the first time execution "reaches" the line where they are declared. They are all destroyed in the reverse order of initialization.

The easiest way to get all this right is to make all static variables that are not `constexpr` initialized into function static locals, which makes sure all of your statics/globals are initialized properly when you try to use them no matter what, thus preventing the static initialization order fiasco.

```
T& get_global() {
    static T global = initial_value();
    return global;
}
```

Be careful, because when the spec says namespace-scope variables have "static storage duration" by default, they mean the "lifetime of the translation unit" bit, but that does *not* mean it can't be accessed outside of the file.

Functions

Significantly more straightforward, `static` is often used as a class member function, and only very rarely used for a free-standing function.

A static member function differs from a regular member function in that it can be called without an instance of a class, and since it has no instance, it cannot access non-static members of the class. Static variables are useful when you want to have a function for a class that definitely absolutely does not refer to any instance members, or for managing `static` member variables.

```
struct A {
    A() {++A_count;}
    A(const A&) {++A_count;}
    A(A&&) {++A_count;}
    ~A() {--A_count;}

    static int get_count() {return A_count;}
private:
    static int A_count;
}

int main() {
    A var;

    int c0 = var.get_count(); //some compilers give a warning, but it's ok.
    int c1 = A::get_count(); //normal way
}
```

A `static` free-function means that the function will not be referred to by any other translation unit, and thus the linker can ignore it entirely. This has a small number of purposes:

- Can be used in a cpp file to guarantee that the function is never used from any other file.
- Can be put in a header and every file will have it's own copy of the function. Not useful, since inline does pretty much the same thing.
- Speeds up link time by reducing work
- Can put a function with the same name in each TU, and they can all do different things. For instance, you could put a `static void log(const char*) {}` in each cpp file, and they could each all log in a different way.

edited Oct 22 '17 at 6:09

answered Mar 5 '13 at 22:46

Mooing Duck

44k ● 9 ● 67 ● 119

1 What about class members? Isn't it a third separate case? – Étienne Mar 5 '13 at 22:50

3 @Etienne - static class data members are the same as static global variables except that you can access them from other translation units, and any access (except from member functions) must specify the `classname::` scope. Static class member functions are like global functions but scoped to the class, or like normal members but without `this` (that's not a choice - those two should be equivalent). – Steve314 Mar 5 '13 at 22:53

1 @LuchianGrigore: while I see your point, I'm not sure what wording to use. – Mooing Duck Mar 5 '13 at 22:57

1 @Steve314: I understand what you mean, but when dealing with a so horribly overloaded term as *static*, I wish we were all a bit more careful. In particular all global (really namespace level) variables have static duration, so adding static in *static global variables* may be understood as `namespace A { static int x; }`, which means *internal linkage* and is very different from the behavior of *static class data members*. – David Rodríguez - dribeas Mar 6 '13 at 13:53

1 "If it's in a namespace scope, then it can't be accessed from any other translation unit..." What do you mean if it's in a namespace scope? Isn't that always the case, could you give an example and a counter example? – zehelvion Sep 17 '15 at 15:45

Static storage duration means that the variable resides in the same place in memory through the lifetime of the program.

Linkage is orthogonal to this.

I think this is the most important distinction you can make. Understand this and the rest, as well as remembering it, should come easy (not addressing @Tony directly, but whoever might read this in the future).

The keyword `static` can be used to denote internal linkage **and** static storage, but in essence these are different.

> What does it mean with local variable? Is that a function local variable?

Yes. Regardless of when the variable is initialized (on first call to the function and when execution path reaches the declaration point), it will reside in the same place in memory for the life of the program. In this case, `static` gives it static storage.

> Now what about the case with static and file scope? Are all global variables considered to have static storage duration by default?

Yes, all globals have by definition static storage duration (now that we cleared up what that means). **But** namespace scoped variables aren't declared with `static`, because that would give them internal linkage, so a variable per translation unit.

> How does static relate to the linkage of a variable?

It gives namespace-scoped variables internal linkage. It gives members and local variables static storage duration.

Let's expand on all this:

```
//

static int x; //internal linkage
              //non-static storage - each translation unit will have its own copy
of x
              //NOT A TRUE GLOBAL!

int y;        //static storage duration (can be used with extern)
              //actual global
              //external linkage
struct X
{
   static int x;     //static storage duration - shared between classes
};

void foo()
{
   static int x;     //static storage duration - shared between calls
}
```

> This whole static keyword is downright confusing

Definitely, unless you're familiar with it. :) Trying to avoid adding new keywords to the language, the committee re-used this one, IMO, to this effect - confusion. It's used to signify different things (might I say, probably opposing things).

edited Mar 5 '13 at 23:23

answered Mar 5 '13 at 22:52

Luchian Grigore

187k ● 38 ● 341 ● 502

It's actually quite simple. If you declare a variable as static in the scope of a function, its value is preserved between successive calls to that function. So:

```
int myFun()
{
static int i=5;
i++;
return i;
}
int main()
{
printf("%d", myFun());
printf("%d", myFun());
printf("%d", myFun());
}
```

will show `678` instead of `666`, because it remembers the incremented value.

As for the static members, they preserve their value across instances of the class. So the following code:

```
struct A
{
static int a;
};
int main()
{
```

```
A first;
A second;
first.a = 3;
second.a = 4;
printf("%d", first.a);
}
```

will print 4, because first.a and second.a are essentially the same variable. As for the initialization, see this question.

edited May 23 '17 at 12:10
Community ♦
1 ● 1

answered Mar 5 '13 at 22:52
Maciej Stachowski
1,038 ● 7 ● 13

When you a declare a `static` variable at file scope, then that variable is only available in *that* particular file (technically, the *translation unit, but let's not complicate this too much). For example:

a.cpp

```
static int x = 7;

void printax()
{
    cout << "from a.cpp: x=" << x << endl;
}
```

b.cpp

```
static int x = 9;

void printbx()
{
    cout << "from b.cpp: x=" << x << endl;
}
```

main.cpp:

```
int main(int, char **)
{
    printax(); // Will print 7
    printbx(); // Will print 9

    return 0;
}
```

For a *local* variable, `static` means that the variable will be zero-initialized *and* retain its value between calls:

```
unsigned int powersoftwo()
{
    static unsigned lastpow;

    if(lastpow == 0)
        lastpow = 1;
    else
        lastpow *= 2;

    return lastpow;
}

int main(int, char **)
{
    for(int i = 0; i != 10; i++)
        cout << "2^" << i << " = " << powersoftwo() << endl;
}
```

For *class* variables, it means that there is only a single instance of that variable that is shared among all members of that class. Depending on permissions, the variable can be accessed from outside the class using its fully qualified name.

```
class Test
{
private:
    static char *xxx;

public:
    static int yyy;

public:
    Test()
    {
        cout << this << "The static class variable xxx is at address "
             << static_cast<void *>(xxx) << endl;
        cout << this << "The static class variable yyy is at address "
             << static_cast<void *>(&y) << endl;
    }
};
```

```
// Necessary for static class variables.
char *Test::xxx = "I'm Triple X!";
int Test::yyy = 0;

int main(int, char **)
{
    Test t1;
    Test t2;

    Test::yyy = 666;

    Test t3;
};
```

Marking a non-class function as `static` makes the function only accessible from that file and inaccessible from other files.

a.cpp

```
static void printfilename()
{ // this is the printfilename from a.cpp -
  // it can't be accessed from any other file
    cout << "this is a.cpp" << endl;
}
```

b.cpp

```
static void printfilename()
{ // this is the printfilename from b.cpp -
  // it can't be accessed from any other file
    cout << "this is b.cpp" << endl;
}
```

For class member functions, marking them as `static` means that the function doesn't need to be called on a particular instance of an object (i.e. it doesn't have a `this` pointer).

```
class Test
{
private:
    static int count;

public:
    static int GetTestCount()
    {
        return count;
    };

    Test()
    {
        cout << this << "Created an instance of Test" << endl;
        count++;
    }

    ~Test()
    {
        cout << this << "Destroyed an instance of Test" << endl;
        count--;
    }
};

int Test::count = 0;

int main(int, char **)
{
    Test *arr[10] = { NULL };

    for(int i = 0; i != 10; i++)
        arr[i] = new Test();

    cout << "There are " << Test::GetTestCount() << " instances of the Test
class!" << endl;

    // now, delete them all except the first and last!
    for(int i = 1; i != 9; i++)
        delete arr[i];

    cout << "There are " << Test::GetTestCount() << " instances of the Test
class!" << endl;

    delete arr[0];

    cout << "There are " << Test::GetTestCount() << " instances of the Test
class!" << endl;

    delete arr[9];

    cout << "There are " << Test::GetTestCount() << " instances of the Test
class!" << endl;

    return 0;
}
```

edited Apr 14 '15 at 14:31

korish

71 ● 1 ● 6

answered Mar 5 '13 at 23:04

Nik Bougalis

9,464 ● 1 ● 16 ● 34

Static variables are shared between every instance of a class, instead of each class having their own variable.

```
class MyClass
{
    public:
    int myVar;
    static int myStaticVar;
};

//Static member variables must be initialized. Unless you're using C++11, or it's
an integer type,
//they have to be defined and initialized outside of the class like this:
MyClass::myStaticVar = 0;

MyClass classA;
MyClass classB;
```

Each instance of 'MyClass' has their own 'myVar', but share the same 'myStaticVar'. In fact, you don't even need an instance of MyClass to access 'myStaticVar', and you can access it outside of the class like this:

```
MyClass::myStaticVar //Assuming it's publicly accessible.
```

When used inside a function as a local variable (and not as a class member-variable) the static keyword does something different. It allows you to create a persistent variable, without giving global scope.

```
int myFunc()
{
   int myVar = 0; //Each time the code reaches here, a new variable called
'myVar' is initialized.
   myVar++;

   //Given the above code, this will *always* print '1'.
   std::cout << myVar << std::endl;

   //The first time the code reaches here, 'myStaticVar' is initialized. But ONLY
the first time.
   static int myStaticVar = 0;

   //Each time the code reaches here, myStaticVar is incremented.
   myStaticVar++;

   //This will print a continuously incrementing number,
   //each time the function is called. '1', '2', '3', etc...
   std::cout << myStaticVar << std::endl;
}
```

It's a global variable in terms of persistence... but without being global in scope/accessibility.

You can also have static member functions. Static functions are basically non-member functions, but inside the class name's namespace, and with private access to the class's members.

```
class MyClass
{
    public:
    int Func()
    {
        //...do something...
    }

    static int StaticFunc()
    {
        //...do something...
    }
};

int main()
{
   MyClass myClassA;
   myClassA.Func(); //Calls 'Func'.
   myClassA.StaticFunc(); //Calls 'StaticFunc'.

   MyClass::StaticFunc(); //Calls 'StaticFunc'.
   MyClass::Func(); //Error: You can't call a non-static member-function without
a class instance!

   return 0;
}
```

When you call a member-function, there's a hidden parameter called 'this', that is a pointer to the instance of the class calling the function. Static member functions **don't** have that hidden parameter...

they are callable without a class instance, but also cannot access non-static member variables of a class, because they don't have a 'this' pointer to work with. They aren't being called on any specific class instance.

edited Sep 9 '16 at 15:33

answered Mar 5 '13 at 22:42

Jamin Grey

5,914 ● 2 ● 24 ● 35

1 "Assuming it's publicly accessible." - it's not. – Luchian Grigore Mar 5 '13 at 22:43

2 `myStaticVar` needs to be defined also. Kinda important to mention that when answering a question about the semantics of the `static` keyword, don't you think? – Praetorian Mar 5 '13 at 22:46

@Praetorian: Thanks, fixed. – Jamin Grey Mar 5 '13 at 22:57

1 @JaminGrey By "static standalone" I meant static non-member functions, and I write such whenever I need some new functionality only in the current CPP file, and do not wish the linker to have to process an additional symbol. – V-R Sep 9 '16 at 15:52

1 @V-R Odd! I never knew that functionality existed. Thanks for broadening my knowledge! – Jamin Grey Sep 10 '16 at 19:52

In order to clarify the question, I would rather categorize the usage of 'static' keyword in three different forms:

(A). variables

(B). functions

(C). member variables/functions of classes

the explanation follows below for each of the sub headings:

(A) 'static' keyword for variables

This one can be little tricky however if explained and understood properly, it's pretty straightforward.

To explain this, first it is really useful to know about the **scope, duration and linkage** of variables, without which things are always difficult to see through the murky concept of staic keyword

1. Scope : Determines where in the file, the variable is accessible. It can be of two types: (i) **Local or Block Scope**. (ii) **Global Scope**

2. Duration : Determines when a variable is created and destroyed. Again it's of two types: (i) **Automatic Duration** (for variables having Local or Block scope). (ii) **Static Duration** (for variables having Global Scope).

3. Linkage: Determines whether a variable can be accessed (or linked) in another file. Again (and luckily) it is of two types: (i) **Internal Linkage** (for variables having Block Scope and Global Scope) (ii) **External Linkage** (for variables having only for Global Scope)

Let's refer an example below for better understanding:

```
//main file
#include <iostream>

int global_var1; //has global scope
const global_var2(1.618); //has global scope

int main()
{
//these variables are local to the block main.
//they have automatic duration, i.e, they are created when the main() is
//  executed and destroyed, when main goes out of scope
 int local_var1(23);
 const double local_var2(3.14);

 {
/* this is yet another block, all variables declared within this block are
 have local scope limited within this block. */
// all variables declared within this block too have automatic duration, i.e,
/*they are created at the point of definition within this block,
 and destroyed as soon as this block ends */
   char block_char1;
   int local_var1(32) //NOTE: this has been re-declared within the block,
//it shadows the local_var1 declared outside

 std::cout << local_var1 <<"\n"; //prints 32

  }//end of block
  //local_var1 declared inside goes out of scope

 std::cout << local_var1 << "\n"; //prints 23

 global_var1 = 29; //global_var1 has been declared outside main (global scope)
```

```
 std::cout << global_var1 << "\n"; //prints 29
 std::cout << global_var2 << "\n"; //prints 1.618

 return 0;
}  //local_var1, local_var2 go out of scope as main ends
//global_var1, global_var2 go out of scope as the program terminates
//(in this case program ends with end of main, so both local and global
//variable go out of scope together
```

(Readers please be patient: This is the foundation needed to understand the usage of 'static' keyword for various cases in variables)

Now comes the concept of Linkage. When a global variable defined in one file is intended to be used in another file, the linkage of the variable plays an important role.

The Linkage of global variables is specified by the keywords: (i) **static** , and, (ii) **extern**

(Now you get the explanation ;-))

static keyword can be applied to variables with local and global scope, and in both the cases, they mean different things. I will first explain the usage of 'static' keyword in variables with global scope (where I also clarify the usage of keyword 'extern') and later the for those with local scope.

1. Static Keyword for variables with global scope

Global variables have static duration, meaning they don't go out of scope when a particular block of code (for e.g main()) in which it is used ends . Depending upon the linkage, they can be either accessed only within the same file where they are declared (for static global variable), or outside the file even outside the file in which they are declared (extern type global variables)

In the case of a global variable having extern specifier, and if this variable is being accessed outside the file in which it has been initialized, it has to be forward declared in the file where it's being used, just like a function has to be forward declared if it's definition is in a file different from where it's being used.

In contrast, if the global variable has static keyword, it cannot be used in a file outside of which it has been declared.

(see example below for clarification)

eg:

```
//main2.cpp
 static int global_var3 = 23;  /*static global variable, cannot be
                                 accessed in anyother file */
 extern double global_var4 = 71; /*can be accessed outside this file
linked to main2.cpp */
 int main() { return 0; }
```

main3.cpp

```
//main3.cpp
#include <iostream>

int main()
{
   extern int gloabl_var4; /*this variable refers to the gloabal_var4
                            defined in the main2.cpp file */
  std::cout << global_var4 << "\n"; //prints 71;

  return 0;
}
```

now any variable in c++ can be either a const or a non-const and for each 'const-ness' we get two case of default c++ linkage, in case none is specified:

(i) **If a global variable is non-const, its linkage is extern by default**, i.e, the non-const global variable can be accessed in another .cpp file by forward declaration using the extern keyword (in other words, non const global variables have external linkage (with static duration of course)). Also usage of extern keyword in the original file where it has been defined is redundant. In this case ***to make a non-const global variable inaccessible to external file, use the specifier 'static' before the type of the variable***.

(ii) **If a global variable is const, its linkage is static by default**, i.e a const global variable cannot be accessed in a file other than where it is defined, (in other words, const global variables have internal linkage (with static duration of course)). Also usage of static keyword to prevent a const global variable from being accessed in another file is redundant. Here, ***to make a const global variable have an external linkage, use the specifier 'extern' before the type of the variable***

Here's a summary for global scope variables with various linkages

```
//globalVariables1.cpp

// defining uninitialized vairbles
int globalVar1; //  uninitialized global variable with external linkage
```

```
static int globalVar2; // uninitialized global variable with internal linkage
const int globalVar3; // error, since const variables must be initialized upon
declaration
const int globalVar4 = 23; //correct, but with static linkage (cannot be accessed
outside the file where it has been declared*/
extern const double globalVar5 = 1.57; //this const variable ca be accessed
outside the file where it has been declared
```

Next we investigate how the above global variables behave when accessed in a different file.

```
//using_globalVariables1.cpp (eg for the usage of global variables above)

// Forward declaration via extern keyword:
 extern int globalVar1; // correct since globalVar1 is not a const or static
 extern int globalVar2; //incorrect since globalVar2 has internal linkage
 extern const int globalVar4; /* incorrect since globalVar4 has no extern
                         specifier, limited to internal linkage by
                         default (static specifier for const variables) */
 extern const double globalVar5; /*correct since in the previous file, it
                           has extern specifier, no need to initialize the
                       const variable here, since it has already been
                       legitimately defined perviously */
```

2. Static Keyword for variables with Local Scope

Earlier, I mentioned that variables with local scope have automatic duration, i.e they come to exist when the block is entered (be it a normal block, be it a function block) and cease to exist when the block ends, long story short, ***variables with local scope have automatic duration***

If **static** specifier is applied to a local variable within a block, it **changes the duration of the variable from automatic to static**

lets take a look at an example.

```
//localVarDemo.cpp
 #include <iostream>

 int localNextID()
{
  int tempID = 1;  //tempID created here
  return tempID++; //copy of tempID returned and tempID incremented to 2
} //tempID destroyed here, hence value of tempID lost

int newNextID()
{
  static int newID = 0;//newID has static duration, with internal linkage
  return newID++; //copy of newID returned and newID incremented by 1
}  //newID doesn't get destroyed here :-)


int main()
{
  int employeeID1 = nextID();  //employeeID1 = 1
  int employeeID2 = nextID();  // employeeID2 = 1 again (not desired)
  int employeeID3 = newNextID(); //employeeID3 = 0;
  int employeeID4 = newNextID(); //employeeID4 = 1;
  int employeeID5 = newNextID(); //employeeID5 = 2;
  return 0;
}
```

this concludes my explanation for the static keyword applied to variables. pheww!!!

B. 'static' keyword used for functions

in terms of functions, the static keyword has a straightforward meaning. Here, it **refers to linkage of the function** Normally all functions declared within a cpp file have external linkage by default, i.e a function defined in one file can be used in another cpp file by forward declaration.

using a static keyword before the function declaration limits its linkage to internal , i.e a static function cannot be used within a file outside of its definition.

C. Staitc Keyword used for member variables and functions of classes

1. 'static' keyword for member variables of classes

I start directly with an example here

```
#include <iostream>

class DesignNumber
{
  private:

      static int m_designNum;  //design number
      int m_iteration;     // number of iterations performed for the design

  public:
    DesignNumber() {     }  //default constructor

   int  getItrNum() //get the iteration number of design
```

```
   {
      m_iteration = m_designNum++;
      return m_iteration;
   }
     static int m_anyVariable;  //public static variable
};
int DesignNumber::m_designNum = 0; // starting with design id = 0
                    // note : no need of static keyword here
                    //causes compiler error if static keyword used
int DesignNumber::m_anyNumber = 99; /* initialization of inclass public
                                  static member  */
enter code here

int main()
{
   DesignNumber firstDesign, secondDesign, thirdDesign;
   std::cout << firstDesign.getItrNum() << "\n";  //prints 0
   std::cout << secondDesign.getItrNum() << "\n"; //prints 1
   std::cout << thirdDesign.getItrNum() << "\n";  //prints 2

   std::cout << DesignNumber::m_anyNumber++ << "\n";  /* no object
                                      associated with m_anyNumber */
   std::cout << DesignNumber::m_anyNumber++ << "\n"; //prints 100
   std::cout << DesignNumber::m_anyNumber++ << "\n"; //prints 101

   return 0;
}
```

In this example, the static variable m_designNum retains its value and this single private member variable (because it's static) is shared b/w all the variables of the object type DesignNumber

Also like other member variables, static member variables of a class are not associated with any class object, which is demonstrated by the printing of anyNumber in the main function

const vs non-const static member variables in class

(i) non-const class static member variables In the previous example the static members (both public and private) were non constants. ISO standard forbids non-const static members to be initialized in the class. Hence as in previous example, they must be initalized after the class definition, with the caveat that the static keyword needs to be omitted

(ii) const-static member variables of class this is straightforward and goes with the convention of other const member variable initialization, i.e the const static member variables of a class *can be* initialized at the point of declaration and they can be initialized at the end of the class declaration with one caveat that the keyword const needs to be added to the static member when being initialized after the class definition.

I would however, recommend to initialize the const static member variables at the point of declaration. This goes with the standard C++ convention and makes the code look cleaner

for more examples on static member variables in a class look up the following link from learncpp.com
http://www.learncpp.com/cpp-tutorial/811-static-member-variables/

2. 'static' keyword for member function of classes

Just like member variables of classes can ,be static, so can member functions of classes. Normal member functions of classes are always associated with a object of the class type. In contrast, static member functions of a class are not associated with any object of the class, i.e they have no *this pointer.

Secondly since the static member functions of the class have no *this pointer, they can be called using the class name and scope resolution operator in the main function (ClassName::functionName();)

Thirdly static member functions of a class can only access static member variables of a class, since non-static member variables of a class must belong to a class object.

for more examples on static member functions in a class look up the following link from learncpp.com

http://www.learncpp.com/cpp-tutorial/812-static-member-functions/

edited Aug 15 '17 at 13:10

answered Apr 16 '17 at 21:06

gajendra
189 ● 1 ● 9

1 1) Before c++17 only integral static const member varibles can be in-class initialized, for example `struct Foo{static const std::string name = "cpp";};` is error, `name` must be defined outside the class; with inline variables intruduced in c++17 one can code: `struct Foo{static inline const std::string name = "cpp";};` 2) Public static member /member functions can be accessed by class name with scope resolution operator and also an instance with dot operator(eg: instance.some_static_method()) – oz1 Jan 10 at 6:58

I'm not a C programmer so I can't give you information on the uses of static in a C program properly, but when it comes to Object Oriented programming static basically declares a variable, or a function or

a class to be the same throughout the life of the program. Take for example.

```
class A
{
public:
    A();
    ~A();
    void somePublicMethod();
private:
    void somePrivateMethod();
};
```

When you instantiate this class in your Main you do something like this.

```
int main()
{
   A a1;
   //do something on a1
   A a2;
   //do something on a2
}
```

These two class instances are completely different from each other and operate independently from one another. But if you were to recreate the class A like this.

```
class A
{
public:
    A();
    ~A();
    void somePublicMethod();
    static int x;
private:
    void somePrivateMethod();
};
```

Lets go back to the main again.

```
int main()
{
   A a1;
   a1.x = 1;
   //do something on a1
   A a2;
   a2.x++;
   //do something on a2
}
```

Then a1 and a2 would share the same copy of int x whereby any operations on x in a1 would directly influence the operations of x in a2. So if I was to do this

```
int main()
{
   A a1;
   a1.x = 1;
   //do something on a1
   cout << a1.x << endl; //this would be 1
   A a2;
   a2.x++;
   cout << a2.x << endl; //this would be 2
   //do something on a2
}
```

Both instances of the class A share static variables and functions. Hope this answers your question. My limited knowledge of C allows me to say that defining a function or variable as static means it is only visible to the file that the function or variable is defined as static in. But this would be better answered by a C guy and not me. C++ allows both C and C++ ways of declaring your variables as static because its completely backwards compatible with C.

answered Mar 5 '13 at 22:52

David Tr

116 • 9

> What does it mean with local variable? Is that a function local variable?

Yes - Non-global, such as a function local variable.

> Because there's also that when you declare a function local as static that it is only initialized once, the first time it enters this function.

Right.

> It also only talks about storage duration with regards to class members, what about it being non instance specific, that's also a property of static no? Or is that storage duration?

```
class R { static int a; }; // << static lives for the duration of the program
```

that is to say, all instances of `R` share `int R::a` -- `int R::a` is never copied.

> Now what about the case with static and file scope?

Effectively a global which has constructor/destructor where appropriate -- initialization is not deferred until access.

> How does static relate to the linkage of a variable?

For a function local, it is external. Access: It's accessible to the function (unless of course, you return it).

For a class, it is external. Access: Standard access specifiers apply (public, protected, private).

`static` can also specify internal linkage, depending on where it's declared (file/namespace).

> This whole static keyword is downright confusing

It has too many purposes in C++.

> can someone clarify the different uses for it English and also tell me when to initialize a static class member?

It's automatically initialized before `main` if it's loaded and has a constructor. That might sound like a good thing, but initialization order is largely beyond your control, so complex initialization becomes very difficult to maintain, and you want to minimize this -- if you must have a static, then function local scales much better across libraries and projects. As far as data with static storage duration, you should try to minimize this design, particularly if mutable (global variables). Initialization 'time' also varies for a number of reasons -- the loader and kernel have some tricks to minimize memory footprints and defer initialization, depending on the data in question.

edited Mar 5 '13 at 23:18

answered Mar 5 '13 at 23:06

justin

94.9k ● 12 ● 155 ● 206

Static Object: We can define class members static using static keyword. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.

A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it can be initialized outside the class as done in the following example by redeclaring the static variable, using the scope resolution operator :: to identify which class it belongs to.

Let us try the following example to understand the concept of static data members:

```
#include <iostream>

using namespace std;

class Box
{
   public:
      static int objectCount;
      // Constructor definition
      Box(double l=2.0, double b=2.0, double h=2.0)
      {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;
         // Increase every time object is created
         objectCount++;
      }
      double Volume()
      {
         return length * breadth * height;
      }
   private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};
```

```
// Initialize static member of class Box
int Box::objectCount = 0;

int main(void)
{
   Box Box1(3.3, 1.2, 1.5);    // Declare box1
   Box Box2(8.5, 6.0, 2.0);    // Declare box2

   // Print total number of objects.
   cout << "Total objects: " << Box::objectCount << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Constructor called.
Constructor called.
Total objects: 2
```

Static Function Members: By declaring a function member as static, you make it independent of any particular object of the class. A static member function can be called even if no objects of the class exist and the static functions are accessed using only the class name and the scope resolution operator ::.

A static member function can only access static data member, other static member functions and any other functions from outside the class.

Static member functions have a class scope and they do not have access to the this pointer of the class. You could use a static member function to determine whether some objects of the class have been created or not.

Let us try the following example to understand the concept of static function members:

```
#include <iostream>

using namespace std;

class Box
{
   public:
      static int objectCount;
      // Constructor definition
      Box(double l=2.0, double b=2.0, double h=2.0)
      {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;
         // Increase every time object is created
         objectCount++;
      }
      double Volume()
      {
         return length * breadth * height;
      }
      static int getCount()
      {
         return objectCount;
      }
   private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};

// Initialize static member of class Box
int Box::objectCount = 0;

int main(void)
{

   // Print total number of objects before creating object.
   cout << "Inital Stage Count: " << Box::getCount() << endl;

   Box Box1(3.3, 1.2, 1.5);    // Declare box1
   Box Box2(8.5, 6.0, 2.0);    // Declare box2

   // Print total number of objects after creating object.
   cout << "Final Stage Count: " << Box::getCount() << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Inital Stage Count: 0
Constructor called.
Constructor called.
Final Stage Count: 2
```

answered May 8 '15 at 8:49

Nima Soroush

4,964 ● 3 ● 34 ● 38

1 It would be right to mention that this paradigms were taken from tutorialspoint.com/cplusplus/cpp_static_members.htm – BugShotGG May 21 '16 at 21:12