Static Data Member Initialization

Why must static data member initialization be outside the class?

```
class X
{
public:
      int normalValue = 5; //NSDMI
      static int i;
};

int X::i = 0;
```

Why is the static data member (here "i") only a declaration, not a definition?

c++ c++11 g++

edited Jul 3 '12 at 3:53
Datalore
560 2 11

asked Jul 2 '12 at 20:24
CyberGuy
1,243 7 18

7 Because. ;) That's just how the language is defined. ;) – jalf Jul 2 '12 at 20:28

You are allowed `static const int i = 10 ;` – Charles Beattie Jul 2 '12 at 20:45

2 @CharlesBeattie, that's still not a definition, and sometimes a definition is needed, see gcc.gnu.org/wiki/… – Jonathan Wakely Jul 2 '12 at 20:55

add a comment

7 Answers

It's important to distinguish the *initializer* which says what its initial value is, and the *definition*. This modified code is valid, with the initializer in the class definition:

```
class X
{
public:
  int normalValue = 5;
  static const int i = 0;       // declaration, with initializer
};

const int X::i;                 // definition
```

i.e. What must be outside the class is a definition, not the initialization.

That's because a variable must have an address in memory (unless it's only used in limited situations, such as in compile-time constant expressions.)

A non-static member variable exists inside the object it is a member of, so its address depends on the address of the object that contains it. Every time you create a new `X` you also create a new `X::i` variable. The non-static data member's lifetime begins with the class' constructor. NSDMI syntax doesn't have anything to do with the variable's address in memory, it just allows you to provide an initial value in one place, instead of repeating it in every constructor with an explicit constructor initializer list.

On the other hand, a static member variable is not contained within an instance of the class, it exists independently of any single instance and exists from the start of the program, at a fixed address. In

order for a static member variable (or any other global object) to get a unique address the linker must see exactly one definition of the static variable, in exactly one object file, and assign it an address.

Because a static variable needs exactly one definition in exactly one object file, it doesn't make sense to allow that definition to be provided in the class, since class definitions typically exist in header files and are included in multiple object files. So although you can provide an initializer in the class, you still need to define the static data member somewhere.

You can also look at it like declaring an `extern` variable:

```
namespace X {
  extern int i;
}
```

This declares the variable, but there must be a definition somewhere in the program:

```
int X::i = 0;
```

edited Jul 3 '12 at 21:04

answered Jul 2 '12 at 21:16

Jonathan Wakely
58.7k 2 81 172

1 Does `i` not have to be `const` or `constexpr` for that to work if it is a static data member? – Johannes Schaub - litb Jul 3 '12 at 17:50

Oops, yes! fixed, thanks – Jonathan Wakely Jul 3 '12 at 21:03

add a comment

You need to supply a separate definition for a static data member (if its *odr-used*, as defined in C++11) simply because that definition shall reside somewhere - in one and only one translation unit. Static class data members are basically global objects (global variables) declared in class scope. The compiler wants you to choose a specific translation unit that will hold the actual "body" of each global object. It is you who has to decide which translation unit to place the actual object to.

edited Jul 2 '12 at 21:15

answered Jul 2 '12 at 20:43

AndreyT
155k 16 210 425

i thought it is NON-STATIC DATA member initialization with c++11. – CyberGuy Jul 2 '12 at 20:46

I removed the incorrect "in C++11 all static members can have initializers too" claim from my answer. – AndreyT Jul 2 '12 at 20:53

add a comment

"static" class member is like a globally allocated variable (it is not related to the single class instance), so it must reside in some object file (and to be declared in the ".cpp" file) as a symbol just like any global variable.

Simple class member (non-static) resides in the memory block allocated for the class instance.

answered Jul 2 '12 at 20:32

Viktor Latypov
9,853 2 12 31

add a comment

The simple reason is because classes are usually declared in *header* files, which often are included in multiple cpp files. Static data members have external linkage and must be declared in exactly one translation unit which makes them unfit for being defined inside a class.

As juanchopanza points out the following is allowed:

```
struct A
{
    const static int i = 1;
};
```

However, this is only a *declaration* not a definition. You still need to define it if you are going to use `i` 's address somewhere. For example:

```
f(int);
g(int&);

X<A::i> x; // Okay without definition for template arguments
char a[A::i]; // Okay without definition, just using value as constant expression
&A::i; // Need a definition because I'm taking the address
f(A::i); // Okay without definition for pass by value
g(A::i); // Need a definition with pass by reference
```

edited Jul 2 '12 at 21:08

answered Jul 2 '12 at 20:50

Jesse Good

23.3k 3 33 74

add a comment

Bear in mind that is is possible to initialize the static data member at the point of declaration if it is of const integral type of const enumeration type:

From the C++03 standard, §9.4.2

> If a static data member is of const integral or const enumeration type, its declaration in the class definition can specify a constant-initializer which shall be an integral constant expression (5.19)

```
struct Foo {
  static const int j = 42; // OK
};
```

answered Jul 2 '12 at 20:53

juanchopanza

124k 7 134 240

2 But even then you still need a definition if the variable is odr-used. – Jonathan Wakely Jul 2 '12 at 20:57

add a comment

When the compiler generate binary code from a unit (extreme simplification: a cpp file and all its included headers) it will emit a symbol for the static variable and eventually initialization code for that variable.

It is okay for a static variable symbol to be declared in multiple units, but it is not okay for it to be initialized multiple times.

So you must make sure that the initialization code is only emitted for a single unit. This mean that the static variable must be defined in exactly one unit.

edited Jul 2 '12 at 21:04

answered Jul 2 '12 at 20:41

Gigi

2,988 9 17

2 No, you are confusing the initializer with the definition. The initializer can be given on the declaration which is repeated in every TU. The definition (with or without initializer) must only exist in one place. – Jonathan Wakely Jul 2 '12 at 20:56

Instead of thinking about initialization I find it's more useful to think in terms of addresses. The static variable must have an address, and only *one* address, so it must be defined in exactly one translation unit, so the linker can give it an address. – Jonathan Wakely Jul 2 '12 at 21:03

add a comment

Static Data Member

```
#include<iostream.h>
#include<conio.h>

class static_var
{

static int count; //static member of class
public :

void incr_staticvar()
{
count++;
}

void outputc()
{
cout<<"Value of Static variable Count :- "<<count<<endl;
}
};

int static_function : : count;

void main()
{
clrscr();
static_var obj1,obj2,obj3,obj4;

obj1.incr_staticvar();
obj2.incr_staticvar();
obj3.incr_staticvar();
obj4.incr_staticvar();

cout<<"\nAfter Increment of static variable by Four Different objects is :-\n";

obj1.outputc ( );
obj2.outputc ( );
```

edited Jul 13 '14 at 21:56

Brad Larson ♦
129k 30 281 434

add a comment

answered Apr 20 '14 at 14:02

user3491994
11 4

Not the answer you're looking for? Browse other questions tagged c++ c++11 g++ or ask your own question.