Home

PUBLIC

Stack Overflow

Tags

Users

Jobs

TEAMS

Create Team

C++11 allows in-class initialization of non-static and non-const members. What changed?

Before C++11, we could only perform in-class initialization on static const members of integral or enumeration type. Stroustrup discusses this in his C++ FAQ, giving the following example:

```
class Y {
  const int c3 = 7;           // error: not static
  static int c4 = 7;          // error: not const
  static const float c5 = 7;  // error: not integral
};
```

And the following reasoning:

> So why do these inconvenient restrictions exist? A class is typically declared in a header file and a header file is typically included into many translation units. However, to avoid complicated linker rules, C++ requires that every object has a unique definition. That rule would be broken if C++ allowed in-class definition of entities that needed to be stored in memory as objects.

However, C++11 relaxes these restrictions, allowing in-class initialization of non-static members (§12.6.2/8):

> In a non-delegating constructor, if a given non-static data member or base class is not designated by a *mem-initializer-id* (including the case where there is no *mem-initializer-list* because the constructor has no *ctor-initializer*) and the entity is not a virtual base class of an abstract class (10.4), then

- otherwise, if the entity is a variant member (9.5), no initialization is performed;
- otherwise, the entity is default-initialized (8.5).

Section 9.4.2 also allows in-class initialization of non-const static members if they are marked with the `constexpr` specifier.

So what happened to the reasons for the restrictions we had in C++03? Do we just simply accept the "complicated linker rules" or has something else changed that makes this easier to implement?

c++ class c++11 initialization class-members

asked Dec 1 '12 at 18:35

Joseph Mansfield
86.7k 14 181 275

5 Nothing happened. Compilers have grown smarter with all these header-only templates so that is relatively easy extension now. – Öö Tiib Dec 1 '12 at 18:54

Interestingly enough on my IDE when I select pre C++11 compilation I'm allowed to initialise non-static const integral members – Dean P Oct 20 at 23:10

3 Answers

The short answer is that they kept the linker about the same, at the expense of making the compiler still more complicated than previously.

I.e., instead of this resulting in multiple definitions for the linker to sort out, it still only results in one definition, and the compiler has to sort it out.

It also leads to somewhat more complex rules for the *programmer* to keep sorted out as well, but it's

```
class X {
    int a = 1234;
public:
    X() = default;
    X(int z) : a(z) {}
};
```

Now, the extra rules at this point deal with what value is used to initialize `a` when you use the non-default constructor. The answer to that is fairly simple: if you use a constructor that doesn't specify any other value, then the `1234` would be used to initialize `a` -- but if you use a constructor that specifies some other value, then the `1234` is basically ignored.

For example:

```
#include <iostream>

class X {
    int a = 1234;
public:
    X() = default;
    X(int z) : a(z) {}

    friend std::ostream &operator<<(st
        return os << x.a;
    }
};

int main() {
    X x;
    X y{5678};

    std::cout << x << "\n" << y;
    return 0;
}
```

Result:

```
1234
5678
```

answered Dec 1 '12 at 18:51

Jerry Coffin
381k 48 460 900

1 Seems like this was quite possible before. It just made the job of writing a compiler harder. Is that a fair statement? – allyourcode Aug 28 '13 at 23:53

7 @allyourcode: Yes and no. Yes, it made writing the compiler harder. But no, because it *also* made writing the C++ specification quite a bit harder. – Jerry Coffin Aug 29

I guess that reasoning might have been written before templates were finalized. After all the "complicated linker rule(s)" necessary for in-class initializers of static members was/were already necessary for C++11 to support static members of templates.

Consider

```
struct A { static int s = ::ComputeSom


// vs.

template <class T>
struct B { static int s; }

template <class T>
int B<T>::s = ::ComputeSomething();

// or

template <class T>
void Foo()
{
    static int s = ::ComputeSomething(
    s++;
    std::cout << s << "\n";
}
```

The problem for the compiler is the same in all three cases: in which translation-unit should it emit the definition of `s` and the code necessary to initialize it? The simple solution is to emit it everywhere and let the linker sort it out. That's why the linkers already supported things like `__declspec(selectany)`. It just wouldn't have been possible to implement C++03 without it. And that's why it wasn't necessary to extend the linker.

To put it more bluntly: I think the reasoning given in the old standard is just plain wrong.

UPDATE

As Kapil pointed out, my first example isn't even allowed in the current standard (C++14). I left it in anyway, because it IMO is the hardest case

edited May 30 '16 at 16:38

answered Jan 29 '16 at 18:29

Paul Groke
4,722 1 21 27

Shame this didn't get any upvotes, as many of the C++11 features are similar in that compilers already included the necessary capability or optimizations. – Alex Court Feb 9 '16 at 16:10

@AlexCourt I wrote this answer recently. The question and Jerry's answer are from 2012 though. So I guess that's why my answer didn't receive much attention. – Paul Groke Feb 9 '16 at 17:08

This will not complie "struct A { static int s = ::ComputeSomething(); }" because only static const can be initialized in class – Kapil May 30 '16 at 9:15

@Kapil You're right. Thanks. I updated my answer. – Paul Groke May 30 '16 at 16:38

In theory `So why do these inconvenient restrictions exist?...` reason is valid but it can rather be easily bypassed and this is exactly what C++ 11 does.

When you *include* a file, it simply includes the file and disregards any initialization. The members are initialized only when you *instantiate* the class.

In other words, the initialization is still tied with constructor, just the notation is different and is more convenient. If the constructor is not called, the values are not initialized.

If the constructor is called, the values are initialized with in-class initialization if present or the constructor can override that with own initialization. The path of

edited Jun 6 '16 at 16:57

answered Jun 6 '16 at 15:56

zar
4,446 6 52 101