C++ - Breaking The Rules With Inline Variables and Functions

28 Feb 2019 on C++ (/tag/cpp/)

I've had some free time these days while visiting my parents and little sister in their home in the Colombian Andes. At almost 3000 meters above sea level, it's a very warm place despite the freezing mountain temperatures; full of nostalgic memories and friendly faces. Chaotic but always welcoming.

I finally got around to organize the ever-growing collection of notes I've gathered in the last years. Scrolling over the 130 notes with the labels *programming* and *cpp*, a particular one caught my attention: *C++ - Inline Variables and Functions*. The note brought back a discussion I had regarding C++17's newly introduced feature: inline variables, which made me realize how misleading and poorly understood C++'s `inline` specifier is. In this post I'll visit the points listed in the note concerning `inline`: when to use it, how to use it, and what it actually means. Chances are you will be surprised.

Inline Expansion

Inline expansion is a fundamental optimization mechanism available to compilers. It relies on the idea that performing a function call is relatively expensive: it requires jumping to a new subroutine, passing function arguments and copying return values. Inlining means suppressing a call to a function by copying the function's instructions directly into the caller's body. In other words, similarly to a preprocessor macro, the body of the called function is *expanded* in-place, becoming part of the caller routine and instead of a separate one.

The repercussion of inlining in the final program can be very hard to foresee. In addition to suppressing the function call overhead, inlining enables a wide range of extra optimizations that would otherwise be very difficult to perform across function calls. As a bottom line, the more contiguous code a compiler can see, the more optimizations it may apply. Don't forget, however, that inlining makes a copy of the function's body for every call. As a consequence, in addition to the obvious increase in program size, the duplication of instructions makes the program inherently cache-unfriendly.

I guess what I want to say is: performance must be measured and not assumed. Inlining can greatly increase performance, but also may end up causing more harm than good.

Inline Functions, Or Are They?

In C++, you can declare inline functions:

```
inline int f(int a, int b)
{
        return a + b;
}
```

Function `f` has a little problem: it probably doesn't do what you think. Not even close. In fact, you can forget everything I wrote earlier about inline expansion, the `inline` keyword has practically very little to do with actual inlining.

Don't get me wrong, though, the compiler is still allowed to inline a call to `f`, but it's not forced to. In fact, even if `f` were not declared `inline`, the compiler might still perform an inline expansion.

The `inline` keyword, as specified by the standard, has essentially to do with something called the *One Definition Rule*.

The One Definition Rule And How To Break It

The first point in the note reads:

> Functions and variables declared inline may be defined multiple times in a program.

So, apparently, if you declare a function or variable (with external linkage) as `inline`, it may be defined multiple times in the same program. This is essentially a violation of the *One Definition Rule*.

What does the *One Definition Rule* (*ODR*) say? Well, basically, that you can only define stuff once: functions, variables, classes, enumeration, etc. No more. No less.

The *ODR* is not just valid at compilation unit level, but also at program level. This is fairly reasonable: what are your compiler and linker supposed to do if they encounter two different implementations of the same function in your program?

Inline functions and variables are an exception to the *One Definition Rule*: they may be defined multiple times in the program. Beware that this is strictly limited to program scope: within a single compilation unit the *ODR* can't be violated, i.e. every single class, function, template or variable used must be defined only once.

Being able to define functions and variables multiple times seems pretty crazy, but, despite being horribly dangerous, crazy things are often very useful.

With Great Power Comes Great Undefined Behavior

Consider that I wrote that inline functions and variables *"may be defined multiple times"* and not *"multiple definitions may exist"*. This brings me to the next bullet point in the note:

> All definitions of an inline function an variables in the entire program must be identical.

This also has the implication that a definition of an inline function or variable must exist in every translation unit where it is used and declared `inline`.

The intention is to technically allow for multiple definitions but practically not. **By making sure that all definitions are equivalent, the program behaves as if it were only one**. Having different definitions of an inline function or variable with external linkage in the same program results in undefined behavior.

Header-only Library Developers Best Friend

The most important usage of the `inline` keyword is when defining a (non-static) function or variable in a header file:

```
// my_header.h
inline int f(int a, int b)
{
        return a + b;
} // f has external linkage (non-static)

inline std::string MyGlobalVariable{"This is so cool"}; // has external linkage
```

Failing to declare a non-static (i.e. with external linkage) function or variable in a header as `inline` may result in nasty multiple-definition errors when linking, given that only inline functions and variables can break the *ODR*, and, not declared static, have external linkage.

Header-only libraries make use of this mechanism extensively. Definitions of non-templated functions and variables can be shipped in header files and be included in different compilation units without having to be defined in a separate source file. This is strategy also ensures that all definitions of the same function are identical.

C++17 Inline Variables

The C++17 standard extends the concept of inlining to variables and not just functions.

Consider a class that defines a static member of type `std::string`. In C++14, you would need to first declare it in the class:

```
// my_header.h
#pragma once
#include <string>

struct SomeClass
{
        static std::string myStaticString;
};
```

And then define it in a separate compilation unit:

```
//my_src.cpp
#include #my_header.h"
std::string SomeClass::myStaticString{"This is annoying"};
```

In C++14, defining the variable in-class will result in a compiler or linker error. In C++17, however, the following becomes a completely valid header:

```
// my_header.h
#pragma once

struct SomeClass
{
        static inline std::string myStaticString{"This is cool"};
};
```

Out-of-class definition is also possible:

```
// my_header.h
#pragma once

struct SomeClass
{
        static std::string myStaticString;
};

inline std::string SomeClass::myStaticString{"This is cool"};
```

Defining `myStaticString` multiple times in different compilation units is allowed, given that it is declared inline.

This works the same for `static const` members of a class. Which, previously to C++17, had to also be defined separately (with the exception of literal types) in a source file.

With C++17 inline variables, you can initialize them, even *at runtime*, in a header file:

```
// my_header.h
#pragma once

struct SomeClass
{
        static const int myRandomInt;
};

inline const int myRandomInt = generateRandomInt(); // calculates a random number at runtim
```

Note that if `generateRandomInt` were a `constexpr` function, one could have written:

```
// my_header.h
#pragma once

constexpr int generateRandomInt()
{
        // calculate some random value at compile time
}

struct SomeClass
{
        static constexpr int myRandomInt = generateRandomInt();
};
```

This is also perfectly valid, because `constexpr` functions and variables are implicitly inline.

To be fair, in-class multiple definition of `constexpr` variables was already possible in C++14. Inline variables, therefore, extend the same capabilities to general constants with static storage duration (i.e.

declared static) that are either not known at compile time or are not of a literal type.

Technically Many, Practically One

There is a very important property of inline functions and variables that we haven't discussed yet.

Consider the example above where we initialize a `static const` member of type `int` at runtime by calculating a random value:

```
// my_header.h
#pragma once
#include <cstdlib>
#include <ctime>

inline int generateRandomInt()
{
         std::cout << "Calculating random initialization value" << '\n';
         std::srand(std::time(nullptr));
         return std::rand();
}

struct SomeClass
{
        static const int myRandomInt;
};

inline const int myRandomInt = generateRandomInt(); // calculates a random number at runtim
```

Now we write two compilation units where this header is included:

```
//my_src1.cpp
#include "my_header.h"
#include <iostream>

void printRandomIntFromSrc1()
{
        std::cout << "Value read from src1: " << SomeClass::myRandomInt << '\n';
}
```

And an identical one:

```
//my_src2.cpp
#include "my_header.h"
#include <iostream>

void printRandomIntFromSrc2()
{
        std::cout << "Value read from src2: " << SomeClass::myRandomInt << '\n';
}
```

Which output do you expect if we call both functions above?

```
//main.cpp
#include "my_header1.h"
#include "my_header2.h"

int main()
{
        printRandomIntFromSrc1();
        printRandomIntFromSrc2();
}
```

First execution:

```
Calculating random initialization value
Value read from src1: 1168843399
Value read from src2: 1168843399
```

Second execution:

```
Calculating random initialization value
Value read from src1: 1083431764
Value read from src2: 1083431764
```

Both compilation units see the same generated random value for `SomeClass::myRandomInt`, and the value is computed at runtime (every time that the executable is run) exactly once. This is a crucial property of inline function and variables and the last bullet point in the note:

> inline variables and functions with external linkage share all the same address in every translation unit

Even though, theoretically, the variable `myRandomInt` is initialized multiple times (every time the header is included in a compilation unit), in reality there is only one definition in the program, initialized once and shared by all compilation units that use it.

Wrapping Up

- Inline expansion might be performed regardless of the `inline` declaration of a function.
- Inline functions and variables (with external linkage) may be defined multiple times in the same program, but not compilation unit.
- Inline function and variables must be defined in every compilation unit where they are used and declared inline.
- All definitions must be identical. Different definitions result in undefined behavior.
- `constexpr` implies `inline`
- `inline` functions and variables with external linkage share all the same address (exists only once in practice).

About

Hola! My name is Pablo and I am a software engineer living in Munich. My first line of code was written 12 years ago while attempting to customize an add-on for my World of Warcraft UI in Lua. A passion for programming has been growning in me ever since and now, at 27 years old, I write software for a living. This blog explores a variety of topics that I've stumbled across in my journey as a software developer. Even though mostly written for my own understanding, my hope is that the curious programmer may find my writings entertaining.

Related Posts

std::variant Doesn't Let Me Sleep (/2018/06/26/std-variant/) 26 Jun 2018

It's Time To Do CMake Right (/2018/02/19/its-time-to-do-cmake-right/) 19 Feb 2018

Understandig Virtual Tables in C++ (/2017/06/10/understanding-virtual-tables/) 10 Jun 2017