The `inline` directive

Luca Formaggia

A.A. 2019/2020

1 What **inline** means?

Well, in modern C++ the directive **inline** does not mean inline... Is it confusing? Don't say no: it is confusing. Moreover, C++17 has also added **inline** for variables, which makes the term even more confusing. However, it is an important directive, and it also simplifies the handling of static member variables, so it is better to understand what it does.

In modern C++ the directive **inline** basically means *ignore the one definition rule (ODR)*. So what? Well let's start about *functions*, the only objects to which the original **inline** directive could be applied before the C++17 extension to variables.

2 **inline** and *inlined* functions

Let's first recall the ODR rule with a simple (and incorrect) example.

2.1 The ODR rule

In this header file I **define** a function without declaring it **inline**.

Listing 1: aheader.hpp

```
#ifndef HH_MYHEADER_HH
#define HH_MYHEADER_HH
// this should be inline or not defined here!
double fun(const double & x){return x/5.;}
#endif
```

This is wrong :no function definition should be contained in header files, unless they are **inline**, or not fully-specialized function templates, or **constexpr** functions (which are **inline** by default). Let's see why.

Now I have another function that uses `fun()` declared in `a.hpp`

Listing 2: a.hpp

```
#ifndef HH_AA_HH
#define HH_AA_HH
// I will use fun()
double fun5(); // pure declaration
#endif
```

Here I am doing the right thing, the header file contains just a declaration of the function, not its definition, which is, rightly so, in a source file:

Listing 3: a.cpp

```
#include "a.hpp"
#include "aheader.hpp" // I need fun()
// Definition of fun5()
double fun5(){return fun(5.0);}
```

And now the main, which uses both `fun()` and `fun5`.

Listing 4: main.cpp

```
#include <iostream> //just to do something
#include "a.hpp" // I use fun5()
#include "aheader.hpp" // I use also fun()
int main()
{
  double x = fun(50.9);
  double y = fun5();
  std::cout<<x<<"␣"<<y<<std::endl;
}
```

Now I happily compile. I separate compilation from linking to understand better what happens:

```
g++ -std=c++17 -Wall   -c -o main.o main.cpp
g++ -std=c++17 -Wall   -c -o a.o a.cpp
```

All fine, I have produced the object files, no warning, no errors. For the compiler (which treats each translation unit separately) everything is ok! But now, the linker has to produce the executable:

```
g++   main.o a.o   -o main
```

and I get a **linking error**:

```
a.o: In function ‘fun(double const&)’:
a.cpp:(.text+0x0): multiple definition of ‘fun(double const&)’
main.o:main.cpp:(.text+0x0): first defined here
```

OOps, if you compile adding the option `-g`, which eliminates optimization and loads the debugger, we get a more understandable message:

```
a.o: In function ‘fun(double const&)’:
aheader.hpp:3: multiple definition of ‘fun(double const&)’
main.o:aheader.hpp:3: first defined here
```

Anyway, the information I have here is that `fun(double const&)` is defined twice. In the second case the compiler gives me also the information that the definition is in line 3 of `aheader.hpp`.

Let’s have a more detailed look, so we understand better what’s happening. In `aheader.hpp` I have the **definition** of `fun()`. The file is included in `a.cpp` and indeed I can use the command `nm` to verify it:

```
nm --demangle a.o | grep fun
... T fun(double const&)
..
```

I recall that `T` means that the symbol is defined in that file. But `aheader.hpp` is included also in `main.cpp`, and indeed

```
 nm --demangle main.o | grep fun
... T fun(double const&)
...
```

The linker finds two definitions, the ODR rule applies: **it's an error**.

2.1.1 What happens if I inline the function?

Well, I correct `aheader.hpp` as follows

Listing 5: aheader.hpp (second version)

```cpp
#ifndef HH_MYHEADER_HH
#define HH_MYHEADER_HH
// Now is inline
inline double fun(const double & x){return x/5.;}
#endif
```

remove all object files to recompile from scratch. **Now everything is fine!**.

Indeed, with **inline** we are telling the linker that it can ignore the ODR rule. Not only that, the compiler may now actually inline the function!

2.2 inline directive and inlined functions

An **inline** function, and in general any function whose definition can be in a header file: constexpr function, in-class defined methods, function templates, is eligible to be inlined. And this was indeed the original meaning, now lost, of the **inline** directive!

What does it mean? Well, may be a picture is better than many words:

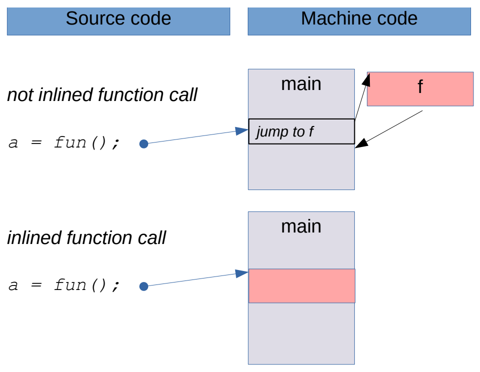

Figure 1: inlined function call vs. standard function call mechanism

The standard mechanism is to store the machine code of the function body in memory. Then, in correspondence of a function call, the program jumps to that memory location, executes the function and returns. In the case of an inlined function, instead, the code is injected directly into the machine code of your executable.

However, *it is up to the compiler to decide whether to inline or not.* The compiler uses some heuristics to decide if inlining would produce a more efficient code. If the function is too complicated or too long, it may decide that it is better not to inline it.

2.2.1 Is there a way to discover if the function has been actually inlined?

Yes, if you run `nm --demangle` on your executable (or object file) and you discover that there is no symbol corresponding to that function it means that it has been inlined.

2.2.2 Is inlining good?

In general yes. It makes, however, the debugging of that function practically impossible. Indeed, the debugger relies on the standard way of calling functions and inlined functions remain inaccessible to it.

If you do not activate optimization (`-g` or `-O0`) the compiler may be more cautious in inlining. So, sometimes functions are inlined only if you activate optimization (`-OX` with X greater than 0). This often happens with instances of function templates.

2.3 When to use **inline** with functions

Typically, you use **inline** only on simple functions, few lines of code that does not make use of other complex functions.

Remember that you may gain run-time efficiency only if the function is actually inlined. While, compilation time is increased when you have the function definition in a header file since all translation units including it generate the corresponding machine code. So using **inline** with complex functions (and thus storing the definition in a header file) is useless.

2.3.1 Redundant use of **inline**

What happens if I do

Listing 6: A useless **inline**

```
inline constexpr double five(const double & x){return 5.;}
```

or

Listing 7: Another useless **inline**

```
struct Foo
{
  inline double getX(){return MyX;}
  ..
};
```

inline is here useless since **constexpr** functions and in-class defined methods are implicitly **inline**. But it is not an error, C++ allows for some redundancies.

3 inline variables

This is new stuff (I am still learning it) introduced in C++17 to simplify things. The reason is the the ODR rules applies also to variable, but normally is less relevant, since usualy variables live in a scope: they are created in a scope and eliminated when exiting the scope. But with some noteble exceptions: the so-called global and namespace variables as well as **static member variables**. The basic reason is that those variables "live" for the whole duration of your program, so they are a little special. Unfortunately, in the standard before c++17 integral static member variables where treated differently that other types, but we can ignore this distinction, for simplicity.

I have already written a note on Piazza on the subject. I repeat it here for completeness.

3.1 Static member variables and the inline directive

I recall that static members express properties common to all objects of the class, and they can be accesses also without the need of an object, just by using the qualified name. Since C++17 it is better to declare **inline** non-const static member variable. This way, you avoid the hassle of the out-of-class definition necessary before c++17. Some details in the following.

3.1.1 Non-const static member variables

Before C++17:

Listing 8: In a header file

```
class Foo
{
public:
static double x; // this is only a declaration
};
```

Listing 9: In the corresponding source file

```
double Foo::x=0.; // Out of class definition (+ initialization)
```

Since C++17:

Listing 10: In the header file

```
class Foo
{
public:
inline static double x=0; //definition and initialization
};
```

Simpler and less prone to errors.

3.1.2 Const static member variables

In this case, first consider using **constexpr**, unless you really need a variable (i.e. something that you want to get the address of at a later stage). Example:

Listing 11: A static constexpr

```
class Point2D{
 public:
 ...
 static constexpr unsigned int nDimensions=2;
};
```

You do not need **inline** here, it's implicit.

If you want a variable, the rules for the non-const version applies also here. So since C++17 you do

Listing 12: A static const variable

```
class Point2D{
 public:
 ...
 inline const unsigned int nDimensions=2;
};
```

3.2 Global and Namespace variable

Let's now talk about global or namespace variables. This type of variables are meant to store resources made at disposal everywhere in your program. They should be used with care (better avoid them). So this part of the discussion may also be skipped.

Yet, you are already using a lot some namespace variables, without knowing it! For instance, `std::cout` is a *variable in the namespace std* of type `std::ostream` and it provides access to the terminal (unless redirected) for writing.

Namespace variables are variables defined in a namespace, global variable are defined outside any scope. Before C++17 you had to declare them in a header file as **extern** and then define them in a source file (like static member variables, with slightly different syntax). Rather complicated stuff. Just for completeness:

Before C++17:

Listing 13: In header file `variables.hpp`

```
extern double globalVar;
namespace Variables{
  extern std::vector<double> aUsefulVector;
}
```

Here, I am declaring a global variable and a namespace variable. **extern** here basically means: hey, this is only a declaration, not a definition. The definition should be found in a source file:

Listing 14: File `variables.cpp`

```
 #include "Variables.hpp"
 double globalVar=0.; //initialised to 0
 namespace Variables{
   std::vector aUsefulVector; // may be initialized to something
 }
```

Now, in all translation units that includes `variables.hpp` I can use globalVar and Variables::aUsefulVector. Exactly as I can use std::cout when I include the header iostream

Since C++17: I do everything in the header file:

Listing 15: In header file `variables.hpp`. C++17 version

```
inline double globalVar=0.0;
namespace Variables{
  inline std::vector<double> aUsefulVector;
}
```

That's all, folks.