C++ PITFALLS

- **How Compact Code Can Become Buggy Code: Getting Caught By The Order of Evaluations**
- **The Most Vexing Parse: How to Spot It and Fix It Quickly**
- **The Incredible Const Reference That Isn't Const**
- **What Every C++ Developer Should Know to (Correctly) Define Global Constants**

Fluent {C++}

How Compact Code Can Become Buggy Code: Getting Caught By The Order of Evaluations

Code sprawling over multiple lines of code and getting drowned into low-level details is typically hindering expressiveness. But cramming everything into one single statement is not always the right thing to do either.

As an example, here is a buggy code that was spotted and fixed by my colleague Benoît (context has been obfuscated in the code). And thanks Benoît for bringing up such an important subject.

```
void f(Data const& firstData, int someNumber, std::auto_ptr<Data>
secondData);

std::auto_ptr<Data> data = ... // initialization of data
f(*data, 42, data);
```

Regardless of the questionable design, and even though this code uses `std::auto_ptr` which has been deprecated, the same problem could have been reproduced with an `std::unique_ptr`, though a little more explicitly maybe:

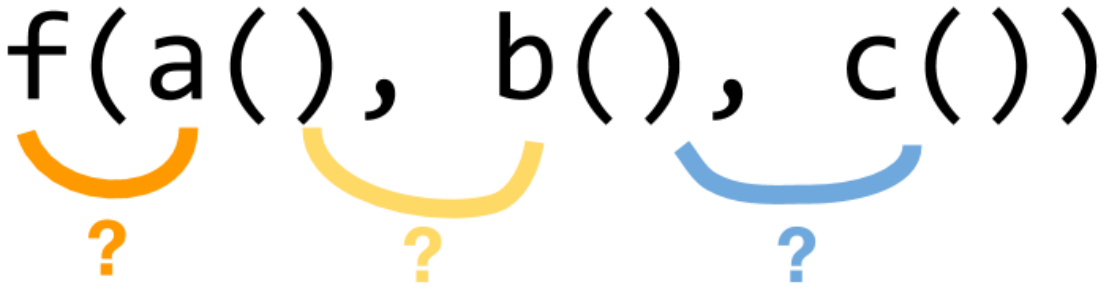
```
void f(Data const& firstData, int someNumber, std::unique_ptr<Data>
secondData);

std::unique_ptr<Data> data = ... // initialization of data
f(*data, 42, move(data));
```

Can you see what can go wrong in these two pieces of code?

In fact, the behaviour was correct for a time, until it broke. And when it broke, it was only on certain platforms and it continued working on others. No need to say that pinning down the source of the issue wasn't easy.

Some leeway for optimization

The problem lies in the passing of arguments to the function `f`. In C++, the **order of evaluation of a function's arguments is unspecified**. Some compilers could decide to evaluate from left to right, others from right to left, and others in a completely different order. This varies from compiler to compiler, and a given compiler can even have different orders of evaluation for different call sites.



In the above case, if the arguments are evaluated from right to left, then `*data` is evaluated **after** the moving of the smart pointer. And moving the smart pointer (or copying it for `auto_ptr`), empties it out, leaving a null pointer inside. Accessing `*data` then causes undefined behaviour (btw, if you want to read more about smart pointer, there is a whole series of posts dedicated to them on Fluent C++).

On the other hand, if the arguments are evaluated from left to right, then `*data` is evaluated **before** the smart pointer has been moved from, so it is still valid at the moment it is accessed.

The reason why the language gives compilers this liberty (and many others) is to let them make optimizations. Indeed, it could be that rearranging the instructions in a specific order would lead to more efficient assembly code. (While I don't doubt it is true, I couldn't find any specific example to illustrate this. Does anyone have one?)

EDIT: As pointed out by Patrice Roy, the unspecified order of evaluation presents another advantage. Fixing an order would leave the possibility to rely on interrelated side effects in the evaluation of the parameters. And this would force us to check inside of the functions what those side effects are in order to understand what the code is doing, which would induce more complexity in the code.

Calls and subcalls

In fact the order of evaluation of arguments can be even more mixed up than the above example.

Consider the following example taken from Item 17 of Scott Meyers' Effective C++:

```
int priority();
void processWidget(std::shared_pointer<Widget> pw, int priority);

processWidget(std::shared_ptr<Widget>(new Widget), priority());
```

(I have taken the liberty to use std::shared_ptr here instead of the book's tr1 component used before C++11 – but the meaning stays unchanged)

The order of evaluation of **all the parameters** is not specified. And even the parameters in the subcalls to the function call. For example, the compiler could generate code that follows this order:

- call `new Widget`,
- call `priority`,
- call the constructor of `std::shared_ptr`!

And if the call to `priority` throws an exception, the `Widget` will leak because it hasn't been stored into the shared pointer yet. For this reason, Scott Meyers advises to store `new`ed objects in smart pointers in standalone statements. But even this wouldn't fix the code at the beginning.

Striking a balance

Leaving some room to the compiler to make optimizations is certainly a good thing, but too much liberty creates a risk of programs not believing the way a programmer would think they would. For this reason, **some rules are necessary** to strike a balance between optimization and easiness of use for the developer.

Some rules have always been there in C++, and even in C. For example calling `&&`, `||` or `,` on two booleans always evaluates the left hand side first, and (if necessary) the right hand side afterwards.

Some codes actually relies on this, for example:

```
void f(const int * pointer)
{
   if (pointer && *pointer != 0)
   {
        ...
```

In this code, the pointer is suspected to be null, so it is checked before being dereferenced (whether this is a good practice or not is debatable, but it is another debate). This code relies on the fact that `pointer` will always occur before `*pointer != 0`. Otherwise the purpose of performing the check at all would be defeated.

By the way, for this reason Scott Meyers advises against overloading `operator&&`, `operator||` and `operator,` on custom types, so that they keep a behaviour consistent with native types (see Item 7 of More Effective C++).

Also, in the expression

```
a ? b : c
```

`a` is, quite naturally, required to evaluate before `b` and `c`.

More rules with Modern C++

C++11, C++14 and C++17 have added more rules to fix the order of the evaluation of various subparts of an expression. However, the order of evaluation of a function's parameters still remains unspecified. It was considered to fix it, but this proposition was finally rejected.

You may wonder what has been added then. In fact there are a lot of cases where the relative order of evaluation could matter. Take the simple example of calling a function with only one argument. The function itself may be the result of an evaluation. For example:

```
struct FunctionObject
```

```
{
    FunctionObject() { /* Code #1 */ }
    void operator()(int value) {}
};

int argument()
{
    /* Code #2 */
}

// Main call
FunctionObject()(argument());
```

Before C++17, the relative order between `Code #1` and `Code #2` was unspecified. And C++17 changes this by ensuring that the determination of the function to call occurs **before** the evaluation of its arguments. In fact modern C++ adds quite a few new rules, that can be found here.

Keep an eye out

As a closing note, I think that one has to be wary of compressed code that use interdependent arguments, and avoid using it when possible. Indeed, some innocuous code can turn out to be the source of a hard-to-diagnose bug. For instance, in the following line of code:

```
a[i] = i++;
```

the behaviour is **undefined** before C++17. Not even unspecified, *undefined*. This means that the outcomes are not limited to the various possible orders of evaluation. The outcome can be anything, including a immediate (or later) crash of the application. Indeed, it is only in C++17 that the evaluation of the right hand side of an assignement is required to occur before the one of the left hand side.

With the increased rhythm of the evolution of the language, we are likely to have compiler upgrades much more often than before, each time risking to change the way the code is generated and optimized. Let's be wary of this sort of cleverness in code.

The Most Vexing Parse: How to Spot It and Fix It Quickly

Everyone has their little defaults. You know, that little something that they do from time to time and that gets on your nerves, even though they're otherwise nice people?

For C++, one of these little annoyances is the most vexing parse, well, as its name suggests.

I think this is a topic related to expressiveness of code, because it's a case where the code doesn't do what it appears to do at first. However, the most vexing parse doesn't go too far into the product because it causes a compilation error (which is the vexing part).

When you don't know about the most vexing parse you can waste a lot of time because of it. But what make the most vexing parse particularly obnoxious is that it can make you waste time **even if you know about it.** I've known it for years and I spent 15 minutes of not the best time of my life chasing after a mysterious compile error the other day, only to find our most vexing friend lurking happily beneath it.

And then I looked for ways to identify it more quickly. So I'm writing this article to help you track it down and root it out of your code, without spending too much time on it.

The Most Vexing Parse

If you already know what the most vexing parse is about, you can skip over to the next section.

The expression was coined by Scott Meyers, who talks about it in details in Item 6 of Effective STL. It comes from a rule in C++ that says that anything that *could* be considered as a function declaration, the compiler should parse it as a function declaration. And even if such an expression could be interpreted as something else. And even if this something else would seems a lot more natural to a programmer.

For instance, consider the following code:

```
struct B
{
    explicit B(int x){}
};

struct A
{
    A (B const& b){}
    void doSomething(){}
};

int main()
{
    int x = 42;

    A a(B(x));

    a.doSomething();
}
```

This code does not compile.

Indeed, the following line:

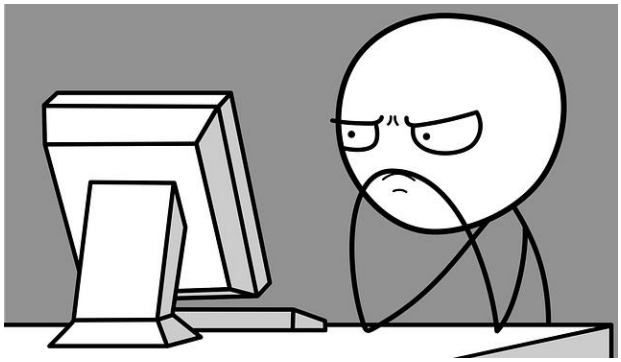
```
A a(B(x));
```

is interpreted as a **function declaration**: it would be a function called `a`, that takes by value a parameter of type `B` called `x` and that returns an object of type `A` by value.

In fact, it could be even more fun: if `B` has a default constructor then the expression:

```
A a(B());
```

is parsed as a function that returns an `A` and that takes a function that returns a `B` and takes no parameter. How fun is that?



And what makes it difficult to diagnose is that **the compilation error occurs a different line**:

```
a.doSomething();
```

Indeed, this line doesn't make sense since we can't call `doSomething` on the *function* `a`.

Thanks to Patrice Roy for his advice on picking use cases and to David Forgeas for noticing the fun case.

A degenerate case

Here is a simpler case where the code doesn't compile, that you probably have come across at some point:

```
struct A
{
    void doSomething(){}
};
int main()
{
    A a();

    a.doSomething();
}
```

This code does not compile because the line

```
A a();
```

is interpreted as a function declaration: it would be a function called `a`, that takes no parameters (hence the empty parentheses) and that return an object of type `A` by value.

How vexing is that?

A vexation in action

Let's take a larger example, which was provided to me by my colleague and friend Jonathan and which I thank for such a great example. I've trimmed it down a bit so that it doesn't take you long to read.

What makes this example interesting is that the most vexing parse is hidden into code that looks perfectly reasonable. There is a compilation error on **line 38**. Can you see the problem?

I do suggest that you make the effort to search for it right now. It's good practice that will make you more acute to analyse this type of compilation error.

```
#include <map>

class Date
{
public:
   Date(int year, int month, int day);
   // ...
};

using FutureDate = Date;   // the original example used NamedType
here
using OptionExpiry = Date; // but I removed it for simplification

class KeyInstrument
{
public:
   KeyInstrument(const FutureDate & futureDate, const OptionExpiry &
optionExpiry);
   bool operator<(const KeyInstrument &other) const;
   // ...
};

enum class StatusInstrument
{
    Ok,
    NegativeFwdFwdVol
};

using PairStatus = std::pair<KeyInstrument, StatusInstrument>;
using StatusCalib = std::map<KeyInstrument, StatusInstrument>;
```

```
int main()
{
    Date date1(2017, 12, 02);
    Date date2(2018, 03, 30);

    KeyInstrument key(FutureDate(date1), OptionExpiry(date2));

    StatusCalib status;
    status.insert(PairStatus(key, StatusInstrument::Ok));
}
```

The root cause of the problem is in fact happening on **line 35**, in that line of code that looks completely innocuous:

```
KeyInstrument key(FutureDate(date1), OptionExpiry(date2));
```

In fact it could be parsed as a function. This function would be called `key` and take 2 parameters: a `FutureDate` parameter, called `date1` (the surrounding parentheses don't matter here), and an `OptionExpiry` parameter called `date2`. And it doesn't matter that `date1` and `date2` also happen to be local variables in `main`. The names of the parameters are local to the function declaration and hide the variables outside.

Annoying, isn't it?

Now let's see the tools we have to quickly detect the most vexing parse, and how to fix that code.

The blood trail of the most vexing parse

clang

clang provides by far the most elegant detection mechanism that I have found. The compiler emits a warning, `-Wvexing-parse`, when it sees you fall into the trap of the most vexing parse. It's as simple as that.

I looked around but couldn't find documentation about how exactly this warning performs its check. From what I've experimented, it seems to fire off whenever there is a function declaration inside of another function.

So the annoying analysis of this compilation error doesn't even start, since the compiler pinned it down for you, and displays it in its output:

```
main.cpp:34:22: error: parentheses were disambiguated as a function
declaration [-Werror,-Wvexing-parse]
    KeyInstrument key(FutureDate(date1), OptionExpiry(date2));
                     ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

How nice is that?

gcc

For gcc, I couldn't find any such warning, but in some cases there you can find a hint in the error message. Here what gcc (7.2.0) outputs when fed our example code:

```
main.cpp: In function 'int main()':
main.cpp:37:55: error: no matching function for call to
'std::pair<KeyInstrument, StatusInstrument>::pair(KeyInstrument
(&)(FutureDate, OptionExpiry), StatusInstrument)'
     status.insert(PairStatus(key, StatusInstrument::Ok));
                                                       ^
```

Not really informative. But there is one thing worth noting here: the `(&)`, in the second line of the message. It doesn't occur that often in error messages, but it's easy to overlook. This symbol means that the error is about a function being misused. That's a hint for the most vexing parse.

Okay, it's a small hint, a heuristic rather than anything else, but it can point you to the right direction. So my advice to you is this: **if you don't immediately understand a compilation error, and you see `(&)` inside it, think about the most vexing parse**. It could be something else, but it could also be this. And the hardest thing about the most vexing parse it to think about it, not to fix it. Knowing this would have saved me 15 minutes of my time to do something more enjoyable than hunting down a most vexing parse.

However the `(&)` doesn't appear in all the compilation errors related to the most vexing parse with gcc. The first example of this post doesn't trigger it for example:

```
struct A
{
    void doSomething(){}
};

int main()
{
    A a();

    a.doSomething();
}
```

And I couldn't find anything that could help diagnose it faster with MSVC.

How to fix the most vexing parse

So now you know what the most vexing parse is, and you also know some ways to track it down. Let's see how to fix it now, and make our code compile.

In C++11

C++11 brings uniform initialization, which consists in calling constructors with curly braces `{}` instead of parentheses `()`. Applied to our example, it gives the following result:

```
KeyInstrument key(FutureDate{date1}, OptionExpiry{date2});
```

There is no longer an ambiguity. A function doesn't declare its parameters with curly braces, so there is no way the compiler could parse this as a function. Therefore this is parsed as a construction of an object, `key`, like we would have expected in the first place.

In fact, if you always use uniform initialization, the most vexing parse never happens. But whether to code like this or not is a whole topic, discussed in details in Item 7 of Effective Modern C++.

Another possibility (and thanks Christophe for showing this to me) is to use the "`auto` to stick" syntax:

```
auto key = KeyInstrument(FutureDate(date1), OptionExpiry(date2));
```

I'm also preparing an article on the "`auto` to stick" syntax, that should be released in the next few weeks.

Before C++11

When uniform initialization wasn't available yet, the solution to fix the most vexing parse was to add an extra pair of parentheses around one of the arguments to the constructor:

```
KeyInstrument key((FutureDate(date1)), OptionExpiry(date2));
```

This makes the expression impossible to parse as a function declaration. But this isn't as elegant as the solution with the uniform initialization.

And when there is no parameters to surround, just omit all the parentheses:

```
A a;
```

The Incredible Const Reference That Isn't Const

While working on the NamedType library I came across a situation that left me stunned in bewilderment: **a const reference that allows modification of the object it refers to**. Without a `const_cast`. Without a `mutable`. Without anything up the sleeve.

How can this be? And how to enforce the `const` in that const reference?

A const reference that isn't const

Here is a description of a situation where this strange reference comes up. Consider the following wrapper class:

```
template<typename T>
class Wrapper
{
public:
    Wrapper(T const& value) : value_(value) {}

    T const& get() const { return value_; }
```

```
private:
    T value_;
};
```

This `Wrapper` class stores a value of type `T`, and gives access to it via the `get()` method (independently from that, if you're wondering if "get" is a good name, you might want to read this detailed discussion). The important part is that the `get()` method only gives a **read-only** access to the stored value, because it returns a const reference.

To check that, let's instantiate the template with `int`, for instance. Trying to modify the stored value through the `get()` method fails to compile (try to click the Run button below).

Which is just fine, because we want `get()` to be read-only.

But let's now instantiate `Wrapper` with `int&`. One use case for this could be to avoid a copy, or keep track of potential modifications of `a` for example. Now let's try to modify the stored value through the const reference returned by `get()`:

And if you hit the RUN buttong you will see that... it compiles! And it prints the value modified through the const reference, like nothing special had happened.

Isn't this baffling? Don't you feel the reassuring impression of safety provided by `const` collapsing around us? Who can we trust?

Let's try to understand what is going on with this reference.

Const reference or reference to const?

The crux of the matter is that our a reference is a **const reference**, but not a reference to const.

To undersand what that means, let's decompose step by step what is happening in the template instantiation.

The `get()` method returns a `const T&`, with `T` coming from `template T`. In our second case, `T` is `int&`, so `const T&` is `const (int&) &`.

Let's have a closer look at this `const (int&)`. `int&` is a reference that refers to an `int`, and is allowed to modify that `int`. But `const (int&)` is a reference `int&` that is `const`, meaning that the **reference itself** cannot be modified.

What does it mean to modify a reference, to begin with? In theory, it would mean making it refer to another object, or not refer to anything. But both those possibilities are illegal in C++: a reference cannot rebind. It refers to the same object during the whole course of it life.

So being `const` doesn't say much for a reference, since they always *are* `const`, since they cannot rebind. This implies that `const (int&)` is effectively the same type as `int&`.

Which leaves us with `get()` returning `(int&) &`. And by the rules of references collapsing, this is `int&`.

So `get` returns an `int&`. Its interface says `T const&`, but it's in fact a `int&`. Not really expressive code, is it?

How to make the const reference a reference to const

Now that we're past the first step of understanding the issue, how can we fix it? That is, how can we make `get()` return a reference to the stored value that does not allow to modify it?

One way to do this is to explicitly insert a `const` **inside** of the reference. We can do this by stripping off the reference, adding a `const`, and putting the reference back. To strip off the reference, we use `std::remove_reference` in C++11, or the more convenient `std::remove_reference_t` in C++14:

As you can see if you click on Run, the expected compilation error now appears when trying to modify the value stored in `Wrapper`.

I think it is useful to understand what is going on with this reference, because this is code that looks like it's safe, but is really not, and this is the most dangerous kind.

You can play around with the code in the above playgrounds embedded in the page. And a big thanks to stack overflow user rakete1111 for helping me understand the reasons behind this issue.

What Every C++ Developer Should Know to (Correctly) Define Global Constants

Constant values are an everyday tool to make code more expressive, by putting names over values.

For example, instead of writing `10` you can write `MaxNbDisplayedLines` to clarify your intentions in code, with `MaxNbDisplayedLines` being a constant defined as being equal to `10`.

Even though defining constants is such a basic tool to write clear code, their definition in C++ can be tricky and lead to surprising (and even, unspecified) behaviour, in particular when making a constant accessible to several files.

Everything in this article also applies to global variables as well as global constants, but global variables are a bad practice contrary to global constants, and we should avoid using them in the first place.

Thanks a lot to Patrice Roy for reviewing this article and helping me with his feedback!

Declaring a global constant: the natural but incorrect way

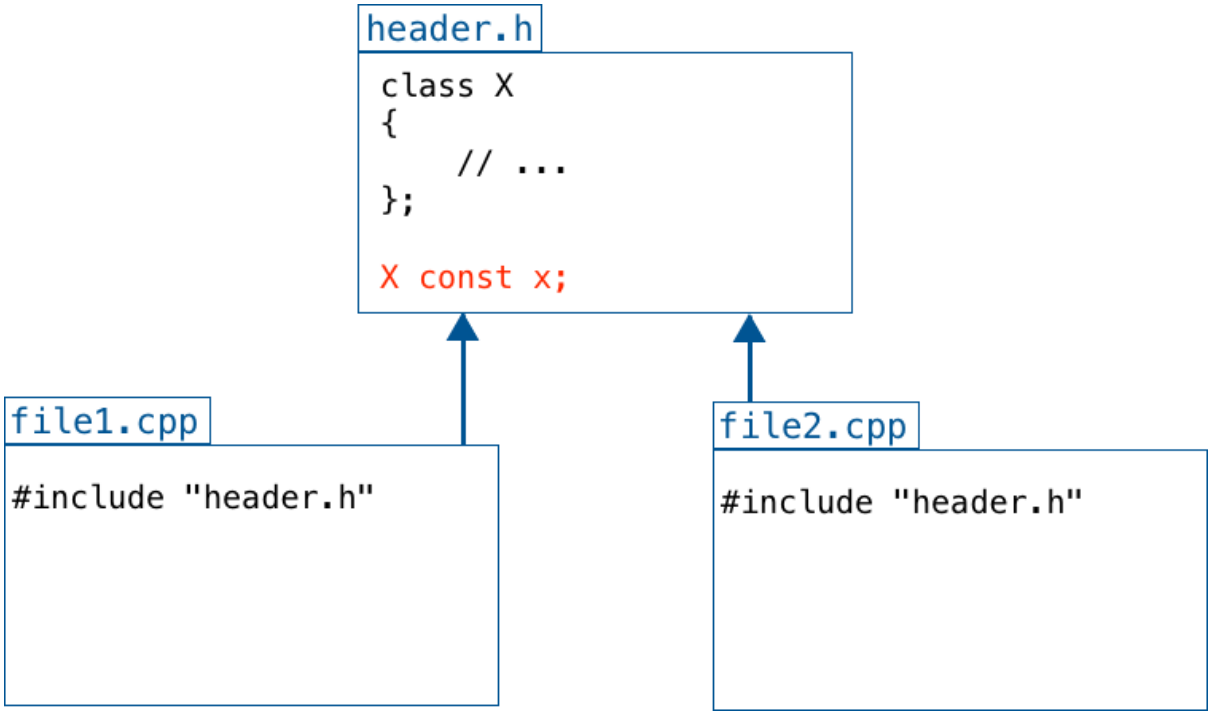
To define a constant of type `X`, the most natural way is this:

```
X const x;
```

Note: Maybe it would seem more natural for you to read `const X x`. Even though I'm an *East const* person, none of the contents of this post has anything to do with putting const before or after the type. Everything here holds with `const X x` (friendly hat tip to the folks on the West side of the `const`).

This works ok (assuming that `X` has a default constructor) when `X` is defined and used only inside a `.cpp` file.

But what if `X` is defined this way in a header file, which is `#include`d in several `.cpp` files?



This code compiles, but **doesn't define a global constant!**

Rather, it defines **two** global constants. How so? The preprocessor `#include` directives essentially copy-paste the code of `header.h` into each `.cpp` file. So after the preprocessor expansion, each of the two `.cpp` file contains:

```
X const x;
```

Each file has its own version of `x`. This is a problem for several reasons:

- for global variables, it is undefined behaviour (objects must be defined only once in C++),
- for global constants, since they have internal linkage we're having several independent objects created. But their order of initialisation is undefined, so it's *unspecified* behaviour,
- it uses more memory,

- if the constructor (or destructor) of `X` has side effects, they will be executed twice.

Strictly speaking, the undefined behaviour makes the last two reasons rather theoretical, because in undefined behaviour anything can happen. But if the two objects *are* created, then they *would* consume more memory and two constructors (and destructors) would be called.

Really?

Given that writing `X const x` is such a natural thing to do (another hat tip to the `const` Westerners), you may doubt that such problems could appear. I doubted that too.

Let's make a simple test to observe it with our own eyes: let's add a side effect in the constructor of `X`:

```
class X
{
public:
    X(){ std::cout << "X constructed\n"; }
};
```

With this addition, here is what our program with the two `.cpp` files outputs:

```
X constructed
X constructed
```

Wow. This was real. `x` is constructed twice.

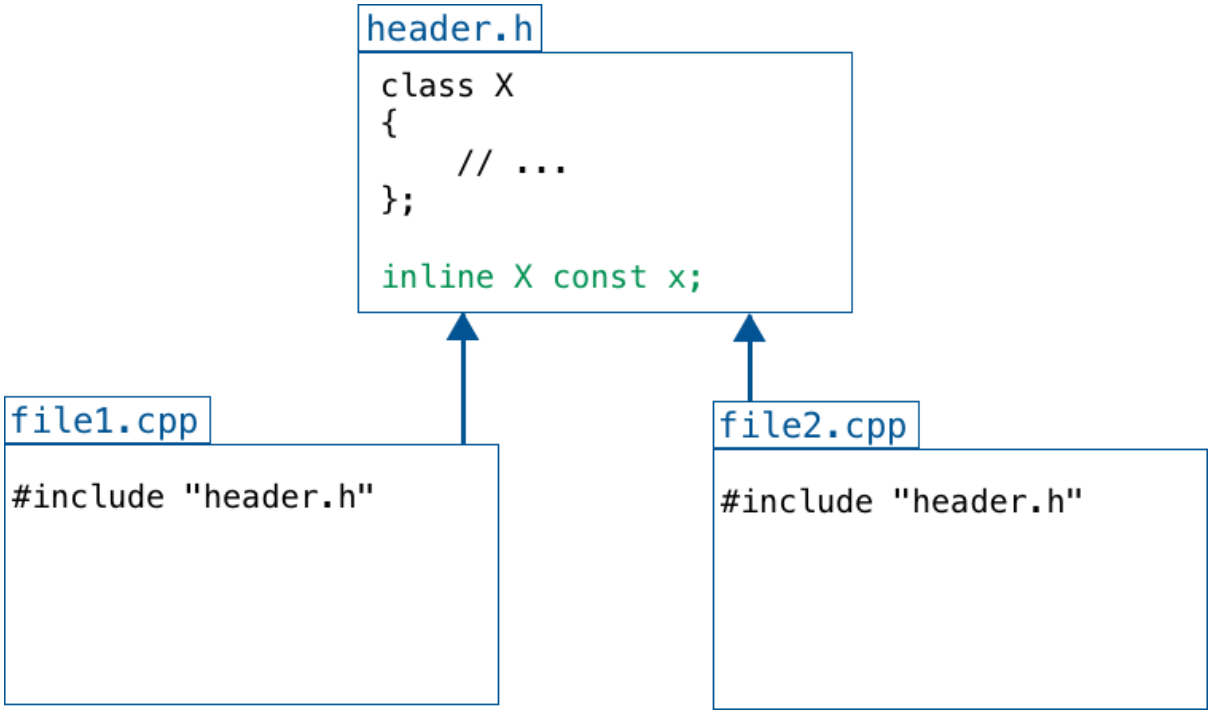
How to fix it then?

In C++17: inline variables

C++17 offers a "simple" solution to this. (I write "simple" between quotes because even if it is simpler than the solution before C++17, the real simplest way should be the natural above way. Which doesn't work. This feature of C++ makes the language a little harder to learn).

The solution in C++17 is to add the `inline` keyword in the definition of `x`:

```
inline X const x;
```



This tells the compiler to not to define the object in every file, but rather to collaborate with the linker in order to place it in only one of the generated binary files.

Note that this usage of `inline` has (to my knowledge, correct me if I'm wrong in the comments section) nothing to do with copying code at call site, like with `inline` functions.

With this change our program now correctly outputs:

```
X constructed
```

inline and class constants

Constants inside of a class, declared `static`, have the same scope as global constants, and `inline` simplified their definition in C++17 too.

Before C++17, we had to follow the annoying pattern of declaring the `static` in the class definition, and define it outside in only one `cpp` file:

```
// header file
class X
{
   static std::string const S;
};
```

```
// in one cpp file
std::string const X::S = "Forty-Two";
```

With `inline`, we can define it and declare it at the same time:

```
// header file
class X
{
   static inline std::string const S = "Forty-Two";
};

// cpp file
// nothing!
```

But not everyone compiles their code in C++17, at least at the time of this writing. How to share a global constant across multiple files before C++17?

Before C++17: the extern keyword

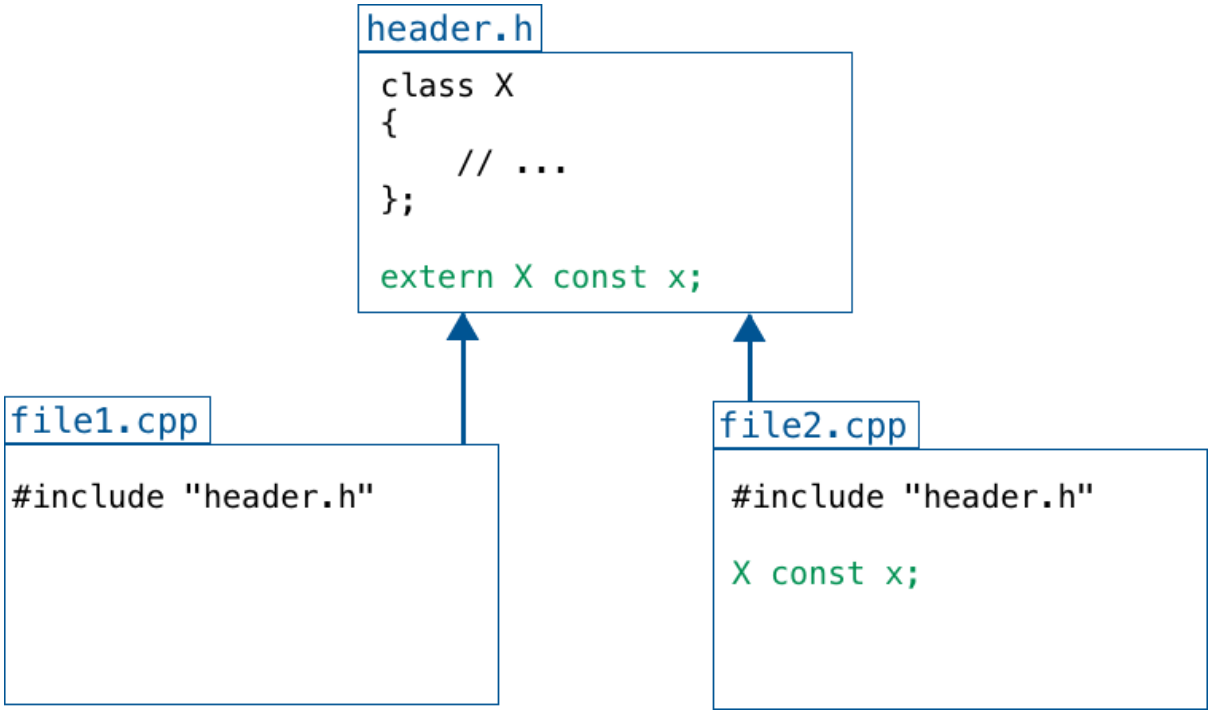
Before C++17, one way to fix the problem is to use the `extern` keyword in the header file:

```
extern X const x;
```

It looks somewhat similar to `inline`, but its effect is very different. With `extern`, the above code is a **declaration**, and not a definition. With `inline`, it was a definition. This declaration informs all the `#include`ing files of the existence and type of `x`.

Even if C++ requires a unique definition of each object, it allows multiple declarations.

However, to use `x` we need to define it somewhere. This can be done in any of the `.cpp` files. You are the one to decide in which file in makes more sense to define it, given the meaning of your global constant, but it will work with any files:

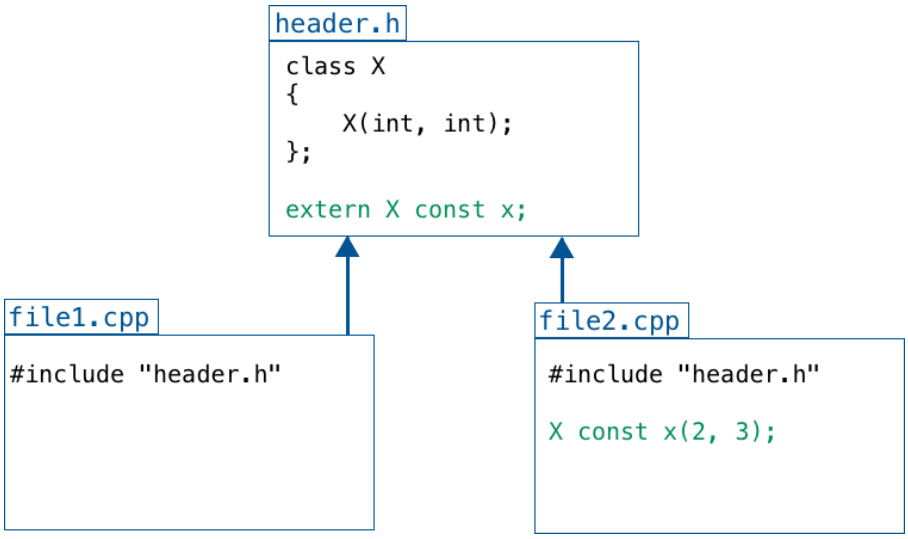


This way our program outputs:

```
X constructed
```

`x` is constructed only once.

And since the line in the header is only a declaration, it doesn't contain the call to the constructor. This shows when the constructor of `X` can accept values:



Note how the declaration in the header file doesn't take constructor arguments, while the definition in the `.cpp` file does.

Note that for this to work, there needs to be **exactly one** definition of `x`. Indeed, if there is no definition we get an undefined external symbol error, and if there is more than one there is a duplicate external symbol.

As for constants inside of classes, there are no other solution than resorting to the annoying pattern of defining the constant outside of the class in one cpp file.

static is not a good solution

`static` has several meanings in C++. When we're not talking about a class constant, declaring an object or function `static` defines it only in the compiled file where it is written.

```
// cpp file

static X const x; // not accessible to other files

static int f(int x) // not accessible to other files
{
    return x * 42;
}
```

Is declaring our object `static` in the header an alternative then? Not really, as it leaves a part of the problem unsolved:

If we declared our object `static` like this in the header file:

```
// header.h

static X const x;
```

Then each file that `#include` it would have its own object `x`. There wouldn't be a violation of the ODR, because there would be as many `x` as compiled files that `#include` the header, but each one would only have its own definition.

The problem with `static` is the fact that there would be several `x` instead of one. It's a shame to execute the constructor and destructor of `X` for each instance, and in the (unlikely,

unrecommended) case of the constructor relying on global variables, each instance of the "constant" `x` could be defined differently and have its own value.

Note that putting `x` in an anonymous namespace would have the same effect as declaring it `static`.

The cart before the horse

To understand how to declare global constants in C++, you need to have some understanding of how a C++ program in built: preprocessing, compiling, linking.

At one point you need to master the build process of C++ anyway, but it may seem a bit surprising that such a basic feature as global constants have this pre-requisite. Anyway, that's how it is, and it's a good thing to master both anyway!