How to use const in C++

On the virtues of constness

Sandor Dargo

www.sandordargo.com

How to use const in C++

On the virtues of constness

Sandor Dargo

This book is for sale at http://leanpub.com/cppconst

This version was published on 2021-08-04

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Introduction

Just make everything `const` that you can! That's the bare minimum you could do for your compiler!

This is a piece of advice, many *senior* developers tend to repeat to juniors, while so often even the preaching programmers - we - fail to follow this recommendation.

It's so easy just to declare a variable without making it `const`, even though we know that its value should never change. Of course, our compiler doesn't know it.

The compiler cannot think after all, at least not for the time being.

It's not enough that we fail to comply with our own recommendations, we are also not specific enough. So if others just blindly follow our instructions without much thinking, then they will just mess things up - by implementing what we suggested!

Compilation failures are easy to spot early on, but dangling references or degrading performance due to extra copies are more difficult to identify.

Hopefully, those are caught not later than the code review[1].

Please, don't be mad at the people following your words blindly. If you share pieces of advice without much thinking, if you don't demand critical thinking from yourself, why would you expect more from others?

In this book you are going to learn how to use `const` keyword in the different contexts it might appear:

- `const` functions
- `const` local variables
- `const` member variables
- `const` return types
- `const` parameters

Things that are not discussed even if they are related to constness are `constexpr`, `consteval` and `constinit`.

Before we dive deep, we should discuss what kind of arguments your fellow developers will come up with as objections while you try to turn more and more entities `const` and what you can tell them.

[1] https://www.sandordargo.com/blog/2018/03/28/codereview-guidelines

Arguments against using `const`

People don't like changes.

People despise to change themselves even more! What I propose in this book brings nothing new under the sky, yet, if you apply these techniques, your code will be more expressive and it will be more difficult to accidentally introduce bugs.

At some places, it might be a bit more verbose, but at the same time, it will become more robust, the behaviour of your code will be more in line with your expectations.

This requires changes, and not only on your side. It's not enough to set up a static code analyzer or to implement an additional rule in `clang-tidy`. Applying the proposed rules will probably require a change in the attitude of many developers.

People opposing these changes will come up with a mix of the following 3 three objections:

- `const` brings tons of *visual noise*
- `const` confuses the developer
- `const` doesn't bring any additional gain

They will mix the above ingredients in unique blends. The most important thing to bear in mind is that they are not bad people, often they are not even bad developers. In fact, they might be better than us who just try to bring some more constness to the world.

They simply used to do things in a certain way and they don't want to give up their approaches.

It's normal, it's human.

Give some respect, be humble and explain your points. Use resources such as this book, and the referenced materials, including even the C++ Core Guidelines[2].

With the right approach, you'll definitely gain some hearts and minds. If you are interested in how to successfully introduce technical changes, I highly recommend reading Driving Technical Change by Terrence Ryan[3].

`const` and visual noise

> *If I make all the variables `const` that are not going to change, I'll pollute the code a lot. It'll be much less readable. And I also have to type more.*

[2] https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-const
[3] https://amzn.to/3a5SHkh

I've heard people saying this several times and probably you will too if you decide to make your code more const.

To be fair, they are right to a certain extent.

```
auto numberOfDoors{2u};
```

is indeed shorter than

```
const auto numberOfDoors{2u};
```

First, typing const or typing almost anything is barely going to be an issue. Typing speed is never the bottleneck for the development of any application. You cannot create any meaningful application as fast as you can write.

Can we, should we consider const as visual noise?

Of course, we can! But I don't think we should.

First, let's see a couple of examples of visual noise!

Comments

Imagine these lines of code:

```
struct Car {
  // ...
  int m_horsePower; // Performance in horse power
};
```

or

```
/*
* Gets the performance in horsepower
*/
std::string getHorsepower() {
  // ...
}
```

Similar comments are definitely a visual noise. They are not bringing any new information. They represent the original sin of comments, they merely repeat what the code says instead of revealing the *why*, instead of sharing a bit more about the intention.

Duplicated type information

A similar visual noise is when you are merely duplicating type information. Something auto helps with:

```
int* myInt = new int{42};
auto* myOtherInt = new int{42};

unsigned int num{42u};
auto otherNum{42u};
```

In the previous example, we duplicated type information either by explicitly writing `int` or simply by adding `u` to a number showing that it's an `unsigned int`.

These pieces of information can be replaced with `auto`. I keep this section short, but I'm sure you can come up with many other similar examples.

It's arguable that in some cases using `auto` makes you think more when you are reading the code, but in the above examples adding the type instead of `auto` wouldn't change the lines' meaning. The explicit types don't add information that is not already there. *(Of course, there are other reasons to use `auto`, but they are out of our scope in this book. I'd recommend reading this classic article by Herb Sutter[4].)*

At the same time, adding `const` **does change** the meaning of the above declarations, `const` is not just noise. Without `const` the variables can be changed, but by adding `const` they cannot be altered anymore.

Things that could be done simpler

For the category of *things that could be done simpler* qualify all those things that could be done in an easier way without losing any meaning, any understandability. Just look for any overcomplicated *smart* code in your codebases.

Adding `const` to a variable, to a function, etc. is not something like that. You cannot get the same effects simpler.

Having seen all the above, we can agree that `const` is not a visual noise. It adds information, it reveals intention, and there is no easier way to achieve the same effects.

`const` is confusing for the developer

> *"Lots of `const`s would confuse the developers."*

Let's think about this considering different perspectives.

First, let's just focus on the case where `const` is used. It cannot confuse any developer. If it does, - with all respect - the developer should learn more, he should learn the meaning of using `const` in that context.

`const` does not introduce an overly complex structure that is often referred to as smart code.

[4] https://herbsutter.com/2013/08/12/gotw-94-solution-aaa-style-almost-always-auto/

const simply reveals the intentions of the developers both to the reader and to the compiler.

How could it possible confuse anybody?

I tell you how.

By its lack.

When you decide to use const in your code wherever it makes sense, you have to make a choice:

- will you use const only in the code you write?
- or will you go in and update all the existing codebase?

In most corporate circumstances, the second option is not a viable one. You can only focus on the code you add, the code you edit. Maybe a bit on the surroundings. This also means that your codebase - in terms of using const - will lose its consistency.

Indeed, the lack of const - or any similar keyword - can carry two meanings as Kate Gregory explained[5]:

- maybe the author of the code omitted it on purpose
- or the person simply didn't put it as it was not in his habit.

Kate Gregory suggests inferring the answer from the surrounding code. If const is widely used, you can assume that it was omitted on purpose. On the other hand, if it barely appears in your code, the lack of it has no meaning.

When you start using it in an existing codebase where it was not used before, you lose the ability to infer the intentions.

Does this mean that you should keep the consistency and not use const at all?

No, that is what Jon Kalb would call a foolish consistency[6]. In this case, being consistent means never improving.

Who would desire that?

It doesn't matter anyway

You'll also encounter some developers who will be reluctant to mark variables - or functions, parameters, etc - const *"because it doesn't matter anyway"*.

I usually show them this video[7]. Let me sum up what Jason Turner[8] shared at CppCon 2016 in his talk *Rich Code For Tiny Computers*.

[5] https://www.youtube.com/watch?v=-Hb-9TUyjoo
[6] http://slashslash.info/2018/02/a-foolish-consistency/
[7] https://youtu.be/zBkNBP00wJE?t=1604
[8] https://github.com/lefticus

In the example he brought, there is a static array of 16 colours, where each colour is represented by a `Color` object taking an id and the red, green and blue components encoded as integers.

Then there is some simple operation on this array of colours. According to Compiler Explorer[9], the compiler needs about 350 instructions.

He asked the audience if there was some best practice that we could use to reduce the required overhead. Some suggested to use `constexpr` some others a recommended precomputed data, but the solution was even simpler.

Jason only had to add `const` to the declaration and the number of instructions went down from about 350 to 5. No typo there, 350 to around 5.

Changing `static std::array<Color, 16> colors {{...}}` to `static const std::array<Color, 16> colors {{...}}` changed the required number of instructions by about 70 times.

> *"So if you don't currently use `const` anywhere you can, I bet you will after this talk!"*

What else to say?

Even if it won't bring a noticeable change all the time... even if it won't bring a major performance benefit in the waste majority of cases, you have nothing to lose and you can only win if you use `const` the right way.

Now that we agree that using `const` is a good idea, it doesn't create visual noise, it doesn't confuse the developer and it might benefit our code, let's see how to use it correctly.

Key takeaways

You'll meet developers who don't consider frequently using `const` a virtue and they will argue against this practice. We've seen their most common arguments and how to respond to them:

- `const` is not a visual noise as it doesn't repeat information and there is no way to express it in an easier way
- `const` is not confusing and not starting using it just because it was not used before would be a *foolish consistency*
- `const` - if used correctly - will not harm performance, but can help a lot the compiler to optimize

[9] https://godbolt.org/

`const` local variables

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconst.

`const` member variables

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconst.

Why would you want to have `const` members?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconst.

Some unexpected implications

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconst.

Can we find a workaround?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconst.

`const` functions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconst.

Characteristics of `const` functions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconst.

`const` overloads

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconst.

`const` return types

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconst.

Returning `const` objects by value

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconst.

Returning `const` references

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconst.

Return `const` pointers

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconst.

east `const` vs `const` west

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconst.

```
int * const func() const
```

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconst.

```
const int * func() const
```

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconst.

```
const int * const func() const
```

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconst.

`const` parameters

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconst.

`const` primitve data type parameters

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconst.

`const` class type parameters

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconst.

When is `const` ignored?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconst.

const and smart pointers

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconst.

`const` and smart pointers as pointers

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconst.

`const std::unique_ptr<T>`

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconst.

`std::unique_ptr<const T>`

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconst.

const std::unique_ptr<const T>

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconst.

`const` and smart pointers as objects

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconst.

`const` rvalue references

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconst.

What are rvalue references?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconst.

Binding rules

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconst.

When to use `const` rvalue references?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconst.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/cppconst.