14.4 — Const class objects and const member functions

ALEX[1] FEBRUARY 12, 2025

In lesson 5.1 -- Constant variables (named constants) (https://www.learncpp.com/cpp-tutorial/constant-variables-named-constants/)[2], you learned that objects of a fundamental data type (`int`, `double`, `char`, etc...) can be made constant via the `const` keyword. All const variables must be initialized at time of creation.

```
const int x;      // compile error: not initialized
const int y{};    // ok: value initialized
const int z{ 5 }; // ok: list initialized
```

Similarly, class type objects (struct, classes, and unions) can also be made const by using the `const` keyword. Such objects must also be initialized at the time of creation.

```
struct Date
{
    int year {};
    int month {};
    int day {};
};

int main()
{
    const Date today { 2020, 10, 14 }; // const class type object

    return 0;
}
```

Just like with normal variables, you'll generally want to make your class type objects const (or constexpr) when you need to ensure they aren't modified after creation.

Modifying the data members of const objects is disallowed

Once a const class type object has been initialized, any attempt to modify the data members of the object is disallowed, as it would violate the const-ness of the object. This includes both changing member variables directly (if they are public), or calling member functions that set the value of member variables.

```
struct Date
{
    int year {};
    int month {};
    int day {};

    void incrementDay()
    {
        ++day;
    }
};

int main()
{
    const Date today { 2020, 10, 14 }; // const

    today.day += 1;         // compile error: can't modify member of const object
    today.incrementDay();  // compile error: can't call member function that modifies
member of const object

    return 0;
}
```

Const objects may not call non-const member functions

You may be surprised to find that this code also causes a compilation error:

```
#include <iostream>

struct Date
{
    int year {};
    int month {};
    int day {};

    void print()
    {
        std::cout << year << '/' << month << '/' << day;
    }
};

int main()
{
    const Date today { 2020, 10, 14 }; // const

    today.print();  // compile error: can't call non-const member function

    return 0;
}
```

Even though `print()` does not try to modify a member variable, our call to `today.print()` is still a const violation. This happens because the `print()` member function itself is not declared as const. The compiler won't let us call a non-const member function on a const object.

Const member functions

To address the above issue, we need to make `print()` a const member function. A **const member function** is a member function that guarantees it will not modify the object or call any non-const member functions (as they may modify the object).

Making `print()` a const member function is easy -- we simply append the `const` keyword to the function prototype, after the parameter list, but before the function body:

```
#include <iostream>

struct Date
{
    int year {};
    int month {};
    int day {};

    void print() const // now a const member function
    {
        std::cout << year << '/' << month << '/' << day;
    }
};

int main()
{
    const Date today { 2020, 10, 14 }; // const

    today.print();  // ok: const object can call const member function

    return 0;
}
```

In the above example, `print()` has been made a const member function, which means we can call it on const objects (such as `today`).

For advanced readers

For member functions defined outside of the class definition, the `const` keyword must be used on both the function declaration in the class definition, and on the function definition outside the class definition. We show an example of this in lesson 15.2 -- Classes and header files (https://www.learncpp.com/cpp-tutorial/classes-and-header-files/)[3].

Constructors may not be made const, as they need to initialize the members of the object, which requires modifying them. We cover constructors in lesson 14.9 -- Introduction to constructors (https://www.learncpp.com/cpp-tutorial/introduction-to-constructors/)[4].

A const member function that attempts to change a data member or call a non-const member function will cause a compiler error to occur. For example:

```
struct Date
{
    int year {};
    int month {};
    int day {};

    void incrementDay() const // made const
    {
        ++day; // compile error: const function can't modify member
    }
};

int main()
{
    const Date today { 2020, 10, 14 }; // const

    today.incrementDay();

    return 0;
}
```

In this example, `incrementDay()` has been marked as a const member function, but it attempts to change `day`. This will cause a compiler error.

Const member functions can modify non-members (such as local variables and function parameters) and call non-member functions per usual. The const only applies to members.

Key insight

A const member function may not: modify the implicit object, call non-const member functions.
A const member function may: modify objects that aren't the implicit object, call const member functions, call non-member functions.

Const member functions may be called on non-const objects

Const member functions may also be called on non-const objects:

```
#include <iostream>

struct Date
{
    int year {};
    int month {};
    int day {};

    void print() const // const
    {
        std::cout << year << '/' << month << '/' << day;
    }
};

int main()
{
    Date today { 2020, 10, 14 }; // non-const

    today.print();  // ok: can call const member function on non-const object

    return 0;
}
```

Because const member functions can be called on both const and non-const objects, if a member function does not modify the state of the object, it should be made const.

Best practice

A member function that does not (and will not ever) modify the state of the object should be made const, so that it can be called on both const and non-const objects.

Be careful about what member functions you apply `const` to. Once a member function is made const, that function can be called on const objects. Later removal of `const` on a member function will break any code that calls that member function on a const object.

Const objects via pass by const reference

Although instantiating const local variables is one way to create const objects, a more common way to get a const object is by passing an object to a function by const reference.

In lesson 12.5 -- Pass by lvalue reference (https://www.learncpp.com/cpp-tutorial/pass-by-lvalue-reference/)[5], we covered the merits of passing class type arguments by const reference instead of by value. To recap, passing a class type argument by value causes a copy of the class to be made (which is slow) -- most of the time, we don't need a copy, a reference to the original argument works just fine and avoids making a copy. We typically make the reference const to allow the function to accept const lvalue arguments and rvalue arguments (e.g. literals and temporary objects).

Can you figure out what's wrong with the following code?

```
#include <iostream>

struct Date
{
    int year {};
    int month {};
    int day {};

    void print() // non-const
    {
        std::cout << year << '/' << month << '/' << day;
    }
};

void doSomething(const Date& date)
{
    date.print();
}

int main()
{
    Date today { 2020, 10, 14 }; // non-const
    today.print();

    doSomething(today);

    return 0;
}
```

The answer is that inside of the `doSomething()` function, `date` is treated as a const object (because it was passed by const reference). And with that const `date`, we're calling non-const member function `print()`. Since we can't call non-const member functions on const objects, this will cause a compile error.

The fix is simple: make `print()` const:

```
#include <iostream>

struct Date
{
    int year {};
    int month {};
    int day {};

    void print() const // now const
    {
        std::cout << year << '/' << month << '/' << day;
    }
};

void doSomething(const Date& date)
{
    date.print();
}

int main()
{
    Date today { 2020, 10, 14 }; // non-const
    today.print();

    doSomething(today);

    return 0;
}
```

Now in function `doSomething()`, `const date` will be able to successfully call const member function `print()`.

Member function const and non-const overloading

Finally, although it is not done very often, it is possible to overload a member function to have a const and non-const version of the same function. This works because the const qualifier is considered part of the function's signature, so two functions which differ only in their const-ness are considered distinct.

```cpp
#include <iostream>

struct Something
{
    void print()
    {
        std::cout << "non-const\n";
    }

    void print() const
    {
        std::cout << "const\n";
    }
};

int main()
{
    Something s1{};
    s1.print(); // calls print()

    const Something s2{};
    s2.print(); // calls print() const

    return 0;
}
```

This prints:

```
non-const
const
```

Overloading a function with a const and non-const version is typically done when the return value needs to differ in constness. This is pretty rare.

Next lesson
14.5 Public and private members and access specifiers

6

Back to table of contents

7

Previous lesson
14.3 Member functions

8

9

Leave a comment...

Name*

Email*

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/[11] are connected to your provided email address.

Notify me about replies:

POST COMMENT

264 COMMENTS

Newest

howard roark

May 8, 2025 7:49 am PDT

if I'm gonna be able to do that why am i use const with member function. okey we cannot modify implicit object particularly but
it's not include objects. Maybe helpful just pass by reference ? I'm confused about that.

```
#include <iostream>

struct Date
{
int year{};
int month{};
int day{};

void print() const // now a const member function
{
std::cout << year << '/' << month << '/' << day;
}
};

int main()
{
Date today{ 2020, 10, 14 }; // const

today.day++;

today.print(); // ok: const object can call const member function

return 0;
}
```

0 Reply

vitrums
April 22, 2025 8:03 am PDT

Few chapters later it became apparent that I had to return back to this chapter and double check if I'd accidentally missed the vital info on what happens to transitive constness. I.e. in the context of struct with data members of program-defined type, when our struct was instantiated as `const`, would it entail true immutability? I think for anyone with background in Java this question is incredibly meaningful. Unfortunately, this chapter doesn't give a clear answer. So it will be nice to have a paragraph showcasing why data members themself aren't required to be declared `const`. Because in contrast (in Java) without `final` specifier on every single data member including `final` in deep structure of transitive dependencies through reference type members immutability isn't feasible.

0 Reply

Max
April 14, 2025 8:55 pm PDT

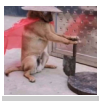
constness

3 Reply

Biscuit
April 5, 2025 7:40 pm PDT

Lock in gang

1 Reply



LechugaPlayer
April 5, 2025 6:33 am PDT

in lesson 14.3 the compiler wanted me to make the function `print()` const, so I tried `const void print()` and `void const print()` and none of them worked, now I understand why

1 Reply

antonipanomixali
February 11, 2025 5:33 pm PST

Hey Alex! What happens with the following code???

#include <iostream>

class Pair{
public:
int first;
int second;

```
void print(){
std::cout << "pair(" << first << ", " << second << ")\n";
}
bool isEqual(Pair p) const{
p.second = 6; // changes the const pair p3
return((first == p.first) && (second == p.second));
}
};

int main (int argc, char *argv[]) {
Pair p1;
Pair p2;

p1.first = 2;
p1.second = 6;

p2 = {3, 6};

p1.print();
p2.print();

//std::cout << (p1.isEqual(p1) ? "equal\n" : "not equal\n");

const Pair p3 = {2, 2};
std::cout << (p1.isEqual(p3) ? "equal\n" : "not equal\n"); //

return 0;
}
```

Somehow we managed to change the const Pair p3 from p1's member function which changes the second member of the passed argument?

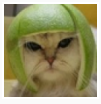
1 Reply

antonipanomixali

Reply to antonipanomixali [12] February 11, 2025 5:38 pm PST

Is it because p3 is passed by value?

0 Reply



Alex Author

Reply to antonipanomixali [13] February 12, 2025 9:25 am PST

`const p3` is copied to non-const function parameter `p`. Since `p` is non-const, the function can modify it. The function itself is marked as const, but this only means it can't modify the data members of the implicit object.

2 Reply

Cpp Learner

December 12, 2024 11:39 am PST

Even if a member function that doesn't involve any member variable needs to be inside the function, there's no point to not make it static?

I mean look at this

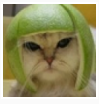
```
void print()
    {
        std::cout << "non-const\n";
    }

    void print() const
    {
        std::cout << "const\n";
    }
```

%100 in all the time, such non member variable functions needs to be static and no point to add const since it doesn't involve any member variable anyway

```
struct Something
{
    static void print()
    {
        std::cout << "non-const\n";
    }
};
```

1 Reply



Alex Author

Reply to Cpp Learner [14] December 19, 2024 3:05 am PST

Static functions can't have different behavior for const and non-const objects. In this case, that's what we want.

Use static functions when you intend to call them as `ClassName::functionName`, and member functions when you intend to call them as `object.functionName`.

0 Reply

Cop learner

Reply to Alex [15] December 19, 2024 1:10 pm PST

Yes so we shouldn't make static functions const right?

0 Reply

Alex Author

Reply to Cop learner [16] December 29, 2024 4:11 pm PST

If I recall correctly, the language doesn't allow cv-qualifiers (such as const) on static member functions.

0 Reply

Cpp learner

Reply to Cop learner [16] December 21, 2024 11:57 pm PST

Not necessary right.....static functions can modify member variables. Depends on what your function does.

0 Reply

abdullah hatem

July 26, 2024 2:45 pm PDT

hey alex i want to ask can i use the keyword `volatile` to call a specific overloaded function in the class ?

Last edited 11 months ago by abdullah hatem

0 Reply

Alex Author

Reply to abdullah hatem [17] July 27, 2024 11:54 am PDT

Volatile works just like const, so generally if you can overload something on const you can overload it on volatile. This includes reference parameters and member functions, but not value parameters.

1 Reply

Dmitriy aka dimonidze

July 10, 2024 2:20 am PDT

How much I read textbooks and other articles \ manuals, I do not understand why the authors are so attached to the transfer by reference to const. The ability to change the object passed to the function is not a bug, but a feature; since the constancy of the transferred object is determined by the author of the function, I believe that he has the right to transfer the object as it seems to him the most correct in a particular situation

It just works:

```cpp
#include <iostream>

struct Date
{
    int year{};
    int month{};
    int day{};

    void print() // non-const
    {
        std::cout << year << '/' << month << '/' << day;
    }
};

void doSomething(Date& date) //non-const
{
    date.print();
}

int main()
{
    Date today{ 2020, 10, 14 }; // non-const
    today.print();

    doSomething(today);

    return 0;
}
```

Last edited 11 months ago by Dmitriy aka dimonidze

0 Reply

Alex Author

Reply to Dmitriy aka dimonidze [18] July 12, 2024 1:41 pm PDT

A few reasons we prefer pass by const reference over pass by non-const reference (when the function doesn't modify the object passed in):

- Non-const references can only bind to modifiable lvalues. Const references can bind to non-modifiable lvalues, modifiable lvalues, and rvalues. This makes them much more flexible.
- Const references promise not to modify the argument, which is useful information to the caller.

If the function does modify the object passed in, then pass by non-const reference is correct.

11 Reply

Swaminathan R

May 31, 2024 1:39 am PDT

Hi Alex, my query in the comments.

```cpp
#include <iostream>
struct MyStruct
{
    int a;
    int doSomething() const
    {
        return ++a; // ERROR: the compiler is smart enough to know that we are
modifying something!
    }
    int doNothing()
    {
        return a; // Then why can't the compiler know we aren't changing anything here
(and allow the call in main)?
    }
};
int main()
{
    const MyStruct a {};
    a.doNothing();
}
```

Last edited 1 year ago by Swaminathan R

0 Reply

shayan ahmad

Reply to Swaminathan R [19] March 8, 2025 7:49 pm PST

no i think here you are calling a non const function from a const objetc ,whihc is not allowed

Last edited 3 months ago by shayan ahmad

0 Reply

groovyest

Reply to Swaminathan R [19] June 1, 2024 5:05 am PDT

maybe the compiler doesnt care if u change anything or not since it doesnt see a const keyword

0 Reply

Swaminathan R

Reply to Swaminathan R [19] May 31, 2024 2:31 am PDT

I think I understand it. Correct me if wrong.

Unlike this code, we may have had the class definition in some other file, and included only a header with declaration in this TU. In that case, the compiler wouldn't know what's inside the function.

1 Reply

Aaheed

Reply to Swaminathan R [20] June 25, 2024 2:46 am PDT

Maybe I did not understand your reply, but isn't it necessary for every code file (or TU) to have the complete definition of a class before using it; and when we include a class from a header, we are actually including the definition rather than just a declaration?

As far as I know, forward declarations work only for functions(and maybe variables), not for classes, since the compiler has to know the complete definition of a class.

0 Reply

Swaminathan

Reply to Aaheed [21] June 25, 2024 2:59 am PDT

Consider three files.

```cpp
//Declaration.cpp
struct MyStruct
{
    int a;
    int doNothing();  //only declaration.
};
```

```cpp
//Definition.cpp
#include "Declaration.cpp"

int MyStruct::doNothing() //defining here.
{
 return ++a;
}
```

```cpp
//main.cpp
#include "Declaration.cpp"

int main()
{
    const MyStruct a {};
    a.doNothing();
}
```

when "main.cpp" is compiled it will only include "Declaration.cpp" which doesn't know whether doNothing() is modifying the object or not.. In this case, the compiler is unsure, and hence throws an error. But if it is part of the function declaration, let's say with a const modifier, then it knows in the main.cpp

Last edited 1 year ago by Swaminathan

1 Reply

Aaheed

Reply to Swaminathan [22] June 25, 2024 4:37 am PDT

Nice explanation, but I just want to point out that it is a member function declaration and not a class declaration.

0 Reply

Alex Author

Reply to Swaminathan R [20] June 2, 2024 1:22 pm PDT

Exactly.

1 Reply

shayan ahmad

Reply to Alex [23] March 8, 2025 7:54 pm PST

alex,i think he is calling a non const memebr function from a const object,which voilets a const correctness?
the example he mentioned is i think nothing to do with the main above code problem?

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/constant-variables-named-constants/
3. https://www.learncpp.com/cpp-tutorial/classes-and-header-files/
4. https://www.learncpp.com/cpp-tutorial/introduction-to-constructors/
5. https://www.learncpp.com/cpp-tutorial/pass-by-lvalue-reference/
6. https://www.learncpp.com/cpp-tutorial/public-and-private-members-and-access-specifiers/
7. https://www.learncpp.com/
8. https://www.learncpp.com/cpp-tutorial/member-functions/
9. https://www.learncpp.com/const-class-objects-and-const-member-functions/
10. https://www.learncpp.com/cpp-tutorial/static-member-variables/
11. https://gravatar.com/
12. https://www.learncpp.com/cpp-tutorial/const-class-objects-and-const-member-functions/#comment-607625
13. https://www.learncpp.com/cpp-tutorial/const-class-objects-and-const-member-functions/#comment-607626
14. https://www.learncpp.com/cpp-tutorial/const-class-objects-and-const-member-functions/#comment-605193
15. https://www.learncpp.com/cpp-tutorial/const-class-objects-and-const-member-functions/#comment-605444
16. https://www.learncpp.com/cpp-tutorial/const-class-objects-and-const-member-functions/#comment-605468
17. https://www.learncpp.com/cpp-tutorial/const-class-objects-and-const-member-functions/#comment-600127
18. https://www.learncpp.com/cpp-tutorial/const-class-objects-and-const-member-functions/#comment-599384
19. https://www.learncpp.com/cpp-tutorial/const-class-objects-and-const-member-functions/#comment-597763
20. https://www.learncpp.com/cpp-tutorial/const-class-objects-and-const-member-functions/#comment-597764
21. https://www.learncpp.com/cpp-tutorial/const-class-objects-and-const-member-functions/#comment-598779
22. https://www.learncpp.com/cpp-tutorial/const-class-objects-and-const-member-functions/#comment-598781
23. https://www.learncpp.com/cpp-tutorial/const-class-objects-and-const-member-functions/#comment-597866
24. https://g.ezoic.net/privacy/learncpp.com