8.10 — Const class objects and member functions

BY ALEX ON SEPTEMBER 11TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2016

In lesson **2.9 -- Const, constexpr, and symbolic constants**, you learned that fundamental data types (int, double, char, etc…) can be made const via the const keyword, and that all const variables must be initialized at time of creation.

In the case of const fundamental data types, initialization can be done through explicit, implicit, or uniform assignment:

```
const int value1 = 5; // explicit initialization
const int value2(7); // implicit initialization
const int value3 { 9 }; // uniform initialization (C++11)
```

Const classes

Similarly, instantiated class objects can also be made const by using the const keyword. Initialization is done via class constructors:

```
const Date date1; // initialize using default constructor
const Date date2(2020, 10, 16); // initialize using parameterized constructor
const Date date3 { 2020, 10, 16 }; // initialize using parameterized constructor (C++11)
```

Once a const class object has been initialized via constructor, any attempt to modify the member variables 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. Consider the following class:

```
class Something
{
public:
    int m_value;

    Something(): m_value(0) { }

    void setValue(int value) { m_value = value; }
    int getValue() { return m_value ; }
};

int main()
{
    const Something something; // calls default constructor

    something.m_nValue = 5; // compiler error: violates const
    something.setValue(5); // compiler error: violates const

    return 0;
}
```

Both of the above lines involving variable something are illegal because they violate the constness of something by either attempting to change a member variable directly, or calling a member function that attempts to change a member variable.

Const member functions

Now, consider the following line of code:

```
    std::cout << something.getValue();
```

Perhaps surprisingly, this will also cause a compile error, even though getValue() doesn't do anything to change a member variable! It turns out that const class objects can only explicitly call *const* member functions, and getValue() has not been marked as a const member function. A **const member function** is a member function that guarantees it will not change any class variables or call any non-const member functions.

To make getValue() a const member function, we simply append the const keyword to the function prototype, after the

parameter list, but before the function body:

```
class Something
{
public:
    int m_value;

    Something() { m_value= 0; }

    void resetValue() { m_value = 0; }
    void setValue(int value) { m_value = value; }

    int getValue() const { return m_value; } // note addition of const keyword after parameter list, but before function body
};
```

Now getValue() has been made a const member function, which means we can call it on any const objects.

For member functions defined outside of the class declaration, the const keyword must be used on both the function prototype in the class declaration and on the function definition:

```
class Something
{
public:
    int m_value;

    Something() { m_value= 0; }

    void resetValue() { m_value = 0; }
    void setValue(int value) { m_value = value; }

    int getValue() const; // note addition of const keyword here
};

int Something::getValue() const // and here
{
    return m_value;
}
```

Futhermore, any const member function that attempts to change a member variable or call a non-const member function will cause a compiler error to occur. For example:

```
class Something
{
public:
    int m_value ;

    void resetValue() const { m_value = 0; } // compile error, const functions can't change member variables.
};
```

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

Note that constructors should not be marked as const. This is because const objects need to be able to initialize their member variables, and a const constructor would not be able to do so.

It's worth noting that a const object is not required to initialize its member variables (that is, it's legal for a const class object to call a constructor that initializes all, some, or none of the member variables)!

rule: Make any member function that does not modify the state of the class object const

Const references

Although instantiating const class objects is one way to create const objects, a more common way is by passing an object

to a function by const reference.

In the lesson on **passing arguments by reference**, we covered the merits of passing class arguments by const reference instead of by value. To recap, passing a class 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 is more performant because it avoids the needless copy. We typically make the reference const in order to ensure the function does not inadvertently change the argument, and to allow the function to work with R-values (e.g. literals), which can be passed as const references, but not non-const references.

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

```
#include <iostream>

class Date
{
private:
    int m_year;
    int m_month;
    int m_day;

public:
    Date(int year, int month, int day)
    {
        setDate(year, month, day);
    }

    void setDate(int year, int month, int day)
    {
        m_year = year;
        m_month = month;
        m_day = day;
    }

    int getYear() { return m_year; }
    int getMonth() { return m_month; }
    int getDay() { return m_day; }
};

// note: We're passing date by const reference here to avoid making a copy of date
void printDate(const Date &date)
{
    std::cout << date.getYear() << "/" << date.getMonth() << "/" << date.getDay() << '\n';
}

int main()
{
    Date date(2016, 10, 16);
    printDate(date);

    return 0;
}
```

The answer is that inside of the printDate function, date is treated as a const object. And with that const date, we're calling functions getYear(), getMonth(), and getDay(), which are all non-const. Since we can't call non-const member functions on const objects, this will cause a compile error.

The fix is simple: make getYear(), getMonth(), and getDay() const:

```
class Date
{
private:
    int m_year;
    int m_month;
    int m_day;

```

```
public:
    Date(int year, int month, int day)
    {
        setDate(year, month, day);
    }

    // setDate() cannot be const, modifies member variables
    void setDate(int year, int month, int day)
    {
        m_year = year;
        m_month = month;
        m_day = day;
    }

    // The following getters can all be made const
    int getYear() const { return m_year; }
    int getMonth() const { return m_month; }
    int getDay() const { return m_day; }
};
```

Now in function printDate(), const date will be able to successfully call getYear(), getMonth(), and getDay().

Overloading const and non-const function

Finally, although it is not done very often, it is possible to overload a function in such a way to have a const and non-const version of the same function:

```
#include <string>

class Something
{
public:
    std::string m_value;

    const std::string& getValue() const { return m_value; }
    std::string& getValue() { return m_value; }
};
```

The const version of the function will be called on any const objects, and the non-const version will be called on any non-const objects:

```
Something something;
something.getValue(); // calls non-const getValue();

const Something something2;
something2.getValue(); // calls const getValue();
```

Overloading a function with a const and non-const version is typically done when the return value needs to differ in constness. In the example above, the const version of getValue() returns a const reference, whereas the non-const version returns a non-const reference (which could then be used to assign a value to, for example). This works because the const-ness of the function is considered part of the function's signature, so a const and non-const function which differ only in const-ness are considered distinct.

Summary

Because passing objects by const reference is common, your classes should be const-friendly. That means making any member function that does not modify the state of the class object const!

8.11 -- Static member variables


Index


8.9 -- Class code and header files

Share this:

C++ TUTORIAL | PRINT THIS POST

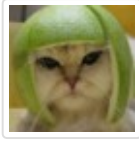
34 comments to 8.10 — Const class objects and member functions


Abhishek
January 13, 2008 at 2:29 am · Reply

ok


Shri
November 7, 2008 at 6:40 am · Reply

In the following statement const is mentioned twice, what does this implies.
const int& GetValue() const { return m_nValue; }


Alex
November 8, 2008 at 11:44 am · Reply

The leftmost const applies to the return value of the function. GetValue() is returning a const reference to an int. If the return value were not const, we could do something like this:

cSomething.GetValue() = 5;

This works because cSomething.GetValue() would return a reference to an int, which is set to m_nValue. So this essentially becomes:

m_nValue = 5;

Obviously this defeats the spirit of the function, so we make the return value const to ensure people don't modify the member variables via assignment this way.

The rightmost const applies to the function. This means the GetValue() can be called on const objects because it's guaranteed not to modify any of the member values.

Tony
January 19, 2009 at 3:45 pm · Reply

So I guess const objects can have their variables changed, so long as they are changed by a constructor or member functions called by the constructor when the object is instantiated?

Alex
February 5, 2009 at 11:15 pm · Reply

What good is a const object if you can't even initialize it? 🙂 Consequently, you are correct.

anandchanduri
April 1, 2009 at 5:21 am · Reply

Alex

Nice explanation … If possible please update the const section with the mutable keyword
as both the topics go along please update the tutorial also 🙂

Anand

Alex
January 13, 2016 at 7:07 pm · Reply

I agree, this would be a good place to discuss the mutable keyword thematically. However, the mutable keyword is used so infrequently, I'm not sure it's even worth the mention at all. I'll revisit the topic later.

LOLY
April 6, 2009 at 8:52 am · Reply

should I understand from ur last reply and Tony's comment that we can modify the const objects ?
I think we can't .. can we ?

Alex
May 1, 2009 at 8:42 pm · Reply

It's better to think about it this way:
Const objects MUST be initialized with a value, but CANNOT have a value assigned to them beyond that (without doing stuff you shouldn't be doing, like casting away the constness of a variable).

Fraz
August 2, 2009 at 7:45 pm · Reply

```
class Something
{
public:
    int m_nValue;

    Something() { m_nValue = 0; }

    void ResetValue() { m_nValue = 0; }
    void SetValue(int nValue) { m_nValue = nValue; }

    int GetValue() { return m_nValue; }
};

int main()
{
    const Something cSomething; // calls default constructor

    cSomething.m_nValue = 5; // violates const - Compile Error
    cSomething.ResetValue(); // violates const - No error, its working
    cSomething.SetValue(5); // violates const - No error, its working

    return 0;
}
```

Im using Borland C++ 5.0 compiler and it allows non constant member functions to call constant datamembers. would u explain why above two violated rules are not caught by my compiler?

C
October 11, 2009 at 8:34 am · Reply

Scary. I compiled this code using g++ and all the errors were caught.

```
class Something
{
public:
    int m_nValue;

    Something() { m_nValue = 0; }

    void ResetValue() { m_nValue = 0; }
    void SetValue(int nValue) { m_nValue = nValue; }

    int GetValue() { return m_nValue; }
};

int main()
{
    const Something cSomething; // calls default constructor

    cSomething.m_nValue = 5; // violates const - Compile Error
    cSomething.ResetValue(); // violates const - No error, its working
    cSomething.SetValue(5); // violates const - No error, its working

    return 0;
}
```

Alex
January 13, 2016 at 6:25 pm · Reply

Because your compiler sucks? 🙂

Anand Kumar
August 4, 2009 at 5:23 am · Reply

Hello Alex,

I am quite new to C++, Whatever I have read that Overloading the function doesn't depend on Return type of function, Overloading the function only depend on Signature of fucntion(i.e. no of args or types of args.)

Here one thig is confusing to me, Overloading function GetValue

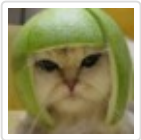
```
1 const int& GetValue() const
```

and

```
1 int& GetValue()
```

Please help me over here.

Thanks a lot.
Anand.



Alex
January 13, 2016 at 6:26 pm · Reply

The const-ness of the function itself is considered part of the function's signature. Thus, a const and non-const function are considered different even if their parameters are identical.

C
October 11, 2009 at 8:41 am · Reply

Thanks, Alex. It may be good to mention sneaky tricks you can do to get around const-ness. Example:

```
1  class Something
2  {
3          public:
4                  int m;
5                  int *buffer;
6
7                  Something() {m = 0; buffer = 0;}
8
9                  void Reset() {m = 0;}
10                 void Set(int n) {m = n;}
11                 void Cheat(int index, int value) const
12                 {
13                         buffer[index] = value;
14                 }
15
16                 int Get() {return m;}
17 };
18
19 int main()
20 {
21         const Something thing;
22         thing.Cheat(2,3);
23
24         return 0;
25 }
```

I got this from Stephen Dewhurst's "C++ Common Knowledge". This code will compile, since Cheat() is modifying what buffer refers to, not buffer itself, even though Cheat() is declared const!

Alex
January 13, 2016 at 6:28 pm · Reply

Interesting. However, instead of cheating, you could always use the mutable keyword to make member variable m editable via the Cheat() function.

gans
August 13, 2012 at 6:09 am · Reply

Hi Alex,
I strange thing about Const object:

lets say I have a class A, and no default or parameterised constructor. but in the main I try to create a const object of this.

const A a;
it does not throw any compilation error.
but if i do it for built in datatype like const int i, it tells us initialise at the time of creation.

Thanks,

Alex
January 13, 2016 at 6:30 pm · Reply

Yes, because class A has no constructors, the instantiation of variable a will call a default constructor that the compiler creates for you, and that satisfies the object's requirement to be initialized (even if it does nothing!)

abhi4556
July 17, 2013 at 9:41 pm · Reply

const class objects can only call const member functions
Note: that constructors should not be marked as const. This is because const objects should initialize their member variables, and a const constructor would not be able to do so.

I think the above lines contradict each other. Please explain.

Alex
January 13, 2016 at 6:33 pm · Reply

You don't explicitly call a constructor, it's called implicitly when the object is instantiated. I've updated the lesson to note that const class objects can only explicitly call const member functions.

Abhishek
February 25, 2015 at 2:05 am · Reply

Please correct the below mentioned code as it won't be compiled due to uninitialized const:
Something cSomething;
cSomething.GetValue(); // calls non-const GetValue();

const Something cSomething2;

cSomething2.GetValue(); // calls const GetValue();

Janez
March 14, 2015 at 5:50 am · Reply

I still don't get that private default constructor. What does it do, why is it there?

Chris
August 26, 2015 at 11:06 am · Reply

Hi, really loving the tutorials but I'm a bit confused on the overloading section of this one.
In the section on overloading const functions you wrote this:

```
const int& GetValue() const { return m_nValue; }
int& GetValue() { return m_nValue; }
```

Is the first const on the first line required? I found that my program wouldn't work if I failed to include the second const, but the first one appeared to have no effect.

Alex
August 26, 2015 at 1:19 pm · Reply

It should be required. In a const member function, the member variables are treated as const. You shouldn't be able to return a non-const reference to a const variable.

Quang
September 17, 2015 at 7:14 am · Reply

Hi Alex thank you for the tutorial!i have 1 question:
- In the last example, why do we use "void PrintDate(const Date &cDate)" instead of just "void PrintDate(const Date cDate)".
I try to re-study 7.3 but its pretty unclear to me. Can you somehow use an example to make me understand? Ty so much

Alex
September 17, 2015 at 3:26 pm · Reply

void PrintDate(const Date cDate) passes Date by value. This means PrintDate gets a copy of the Date argument.
void PrintDate(const Date &cDate) passes Date by const reference. This means cDate is a const reference to the actual Date argument.

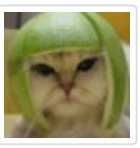
They'll both work, but passing by const reference is faster for classes, because we don't have to make a copy of the Date class every time when the function is called.

Gopal
November 9, 2015 at 2:44 am · Reply

```
1 Something cSomething;
2 cSomething.GetValue(); // calls non-const GetValue();
3
4 const Something cSomething2;
5 cSomething2.GetValue(); // calls const GetValue();
```

Overloading a function with a const and non-const version is typically done when the return value needs to differ in constness. In the example above, the const version of GetValue() returns a const reference, whereas the non-const version returns a non-const reference.

-- I think non-const version returns a non-const "valve" not a non-const "refernce". Correct me if im wrong.

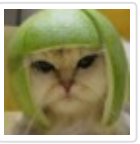


Alex
November 9, 2015 at 10:27 am · Reply

You're correct. I fixed the wording.

Danny
December 28, 2015 at 2:13 pm · Reply

I am totally lost after reading this tutorial…. I think I should come back tomorrow after getting enough sleep

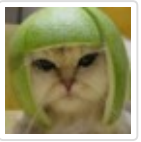


Alex
December 28, 2015 at 4:56 pm · Reply

Sure. Get a good night's sleep, and if you still have any questions after reading it, ask away.

Danny
December 29, 2015 at 1:47 am · Reply

This is what I get after reading again….once a constant object is created, it invokes the default constructor created by the programmer(or parameterized if any), which in turn initializes the data members to constant values and cannot be changed henceforth. Am I right or something…..



Alex
December 31, 2015 at 12:57 pm · Reply

Yes. When a class object is created, the appropriate constructor is called, which can initialize any const (or reference) parameters that need to have values upon initialization. In the case of const values, those can't be changed later because they're const.

Meghana
January 23, 2016 at 6:59 am · Reply

int m_value;

const int& getValue() const { return m_value; }
int& getValue() { return m_value; }

Should the return type be int& or int?

Alex
January 23, 2016 at 9:56 am · Reply

It could be either. In this case, if the intent were to use getValue() as an access function, it would probably be better to return an int or const int, since returning fundamental variables by value is fine.

But imagine we were returning some class object instead, and we didn't want to make a copy of for performance reasons. In that case, it would make sense to have the non-const version return a non-const reference, and the const version return a const reference.

I've updated the example to use a std::string instead of an int, since that's more reflective of a case you'd want to use a reference instead of a return by value.