8.10 — Const class objects and member functions

BY ALEX ON SEPTEMBER 11TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

In lesson **4.13 -- 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 copy, direct, or uniform initialization:

```
const int value1 = 5; // copy initialization
const int value2(7); // direct 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_value = 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 by calling a member function that attempts to change a member variable.

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

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 modify the object or call any non-const member functions (as they may modify the object).

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
};
```

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 definition, the const keyword must be used on both the function prototype in the class definition 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 mem
};
```

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 cannot be marked as const. This is because constructors need to be able to initialize their member variables, and a const constructor would not be able to do so. Consequently, the language disallows const constructors.

Rule: Make any member function that does not modify the state of the class object const, so that it can be called by const objects

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);
```

```
38
39      return 0;
40  }
```

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:

```
1   class Date
2   {
3   private:
4       int m_year;
5       int m_month;
6       int m_day;
7
8   public:
9       Date(int year, int month, int day)
10      {
11          setDate(year, month, day);
12      }
13
14      // setDate() cannot be const, modifies member variables
15      void setDate(int year, int month, int day)
16      {
17          m_year = year;
18          m_month = month;
19          m_day = day;
20      }
21
22      // The following getters can all be made const
23      int getYear() const { return m_year; }
24      int getMonth() const { return m_month; }
25      int getDay() const { return m_day; }
26  };
```

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:

```
1   #include <string>
2
3   class Something
4   {
5   private:
6       std::string m_value;
7
8   public:
9       Something(const std::string &value="") { m_value= value; }
10
11      const std::string& getValue() const { return m_value; } // getValue() for const objects
12      std::string& getValue() { return m_value; } // getValue() for non-const objects
13  };
```

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:

```
1   int main()
```

```
{
    Something something;
    something.getValue() = "Hi"; // calls non-const getValue();

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

    return 0;
}
```

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 non-const version of getValue() will only work with non-const objects, but is more flexible in that we can use it to both read and write m_value (which we do by assigning the string “Hi”).

The const version of getValue() will work with either const or non-const objects, but returns a const reference, to ensure we can’t modify the const object’s data.

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**

C++ TUTORIAL | PRINT THIS POST

130 comments to 8.10 — Const class objects and member functions

« Older Comments 1 2

Ged
December 26, 2019 at 6:02 am · Reply

Just a quick sum up question.

int getYear() const { return m_year; } // Is used when we are working with a const class, but returns a normal int variable?

const int getYear() const { return m_year; } // Is used when we are working with a const class and returns a const int variable?

Also, const in the middle of the function allows us to use non-const class, but treats the function as const and does not allow any changes to be made?

nascardriver
December 28, 2019 at 4:28 am · Reply

```
1 int getYear() const { return m_year; }
```

The member function `getYear` is `const`. It can be used on `const` objects.

```
1 const int getYear() const { return m_year; }
```

`getYear` is non-const, it can only be used on non-const objects. The returned `int` is `const`. That doesn't make sense, because `getYear` is returning by value. `const` return types should be used for references, pointers, etc.

Piyush
January 1, 2020 at 9:51 am · Reply

why is 'getYear' non-const in the second example.

nascardriver
January 2, 2020 at 4:21 am · Reply

It's not, thanks for pointing out my mistake! I wrote another reply to Ged's comment for clarification.

nascardriver
January 2, 2020 at 4:20 am · Reply

Sorry for my last reply, I didn't read your code properly.
Both functions are `const`, they can be used on `const` as well as non-`const` objects.
Neither is allowed to modify members of `Date`.
The second function returns a `const int`, this has no practical use and can prevent return-value-optimization.

Samira Ferdi

November 28, 2019 at 5:04 pm · Reply

Hi, Alex and Nascardriver! What is 'const' actually in terms of class. We make the class const or we make the object const? const class or const object?

nascardriver
November 29, 2019 at 3:12 am · Reply

There is no "const class", only an object of the class or the class's members can be `const`.

hellmet
October 29, 2019 at 6:35 am · Reply

```
1  #include <string>
2
3  class Something
4  {
5  private:
6      std::string m_value;
7
8  public:
9      Something(const std::string &value="") { m_value= value; }
10
11     const std::string& getValue() const { return m_value; } // getValue() for const objects
12     std::string& getValue() { return m_value; } // getValue() for non-const objects
13 };
```

I understand this example is for demonstration, but (at line 11) isn't returning a reference potentially unsafe? The reference can be used to modify the state of the object perhaps putting it in an inconsistent state. If I really wanted to return a reference, it always be by const reference so that I am confident that all state changes to the object are through the methods I write, correct?

nascardriver
October 30, 2019 at 3:55 am · Reply

Yes it is potentially unsafe. A better example would be array element access, but then the example would be bigger.

hellmet
October 30, 2019 at 3:56 am · Reply

Thank you!

Piyush
January 2, 2020 at 4:05 am · Reply

you mean line 12 as at line 11 the function is returning by const reference

hellmet
January 2, 2020 at 4:14 am · Reply

Yes!

Maxim Ivan
October 7, 2019 at 3:39 am · Reply

I just don't Understand the need of the & sign after std::string , because if you ommit it , it is giving the same results. could you explain please ?

```
1 std::string& getValue() { return m_value; } // getValue() for non-const objects
```

nascardriver
October 7, 2019 at 4:13 am · Reply

It may look like it's doing the same, but it's not. See lesson 6.11, 7.3 and 7.4a.

Maxim Ivan
October 7, 2019 at 7:48 am · Reply

Thus return is giving back the original value(reference) instead of making a copy ? Am I Right ?
So it’s faster ?

nascardriver
October 7, 2019 at 7:51 am · Reply

You're right. It's faster. And since the reference is non-const, the caller can modify `m_value` when they call `getValue`.

Maxim Ivan
October 7, 2019 at 8:33 am · Reply

Thank you very much really !!!

Parsa
October 3, 2019 at 5:10 pm · Reply

```
1 something.getValue() = "Hi"; // calls non-const getValue();
```

What is "Hi" being assigned to?

nascardriver
October 4, 2019 at 2:30 am · Reply

```
1 something.m_value
```

Because `something.getValue()` return a reference to `something.m_value`.

McDidda
September 10, 2019 at 11:21 am · Reply

hi, I was trying to write my own matrix class using two dimensional array but getting error

```
class Array
{
    private:
        const int m_size;
        void *ptr;
    public:
        Array(int size):m_size(size)
        {
             ptr=new int[m_size][m_size];        //I want to allocate 2d dynamic array
        }
};
```

error:

main.cc: In constructor ‘Array::Array(int)’:
main.cc:11:30: error: array size in new-expression must be constant
ptr=new int[m_size][m_size];
^
main.cc:11:30: error: ‘this’ is not a constant expression
Makefile:3: recipe for target 'a.out' failed
make: *** [a.out] Error 1

nascardriver
September 11, 2019 at 3:09 am · Reply

Allocate the out array first, then allocate the inner arrays.
`ptr` should be `int**`.

Alireza
June 25, 2019 at 12:31 am · Reply

Hello,
I have a question: Why can we access to a const member function which the object isn't const, or vice versa ?

```
#include <iostream>

class Something
{
public:
    std::string m_value;

    Something(): m_value("const") { }

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

    ~Something() {std::cout << "DESTRUCTOR_CALLED\n";}
};



int main()
```

```
{
    const Something something;
    std::cout << something.getValue();

    Something some;
    std::cout << "\n" << some.getValue();

    return 0;
}
```

nascardriver
June 25, 2019 at 9:13 am · Reply

A non-const object is allowed to be modified, so it doesn't matter whether or not the function is const.
The other way around doesn't work. If `Something::getValue` is non-const, line 20 will cause an error.

Nguyen
June 24, 2019 at 2:52 pm · Reply

Hi,

```
class Something
{
public:
    int m_value ;
    Something(): m_value(0) { }
    void resetValue() const { m_value = 0; } // compile error, const functions can't change
};
```

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.

_____

m_value(0) in default constructor
m_value = 0 in void resetValue() const

Looks like the value of m_value is still zero in both.
Does it consider as a change to m_value?

nascardriver
June 25, 2019 at 8:14 am · Reply

Yes, the value is irrelevant.

Nguyen
June 19, 2019 at 9:41 am · Reply

Hello,

"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)."

Could you please give me an example for "a const member function is a member function that guarantees it will not call any non-const member functions"?

Thanks

Alex
June 19, 2019 at 2:40 pm · Reply

In class Something, getValue() is const, but resetValue() is not. Therefore, getValue() could not call resetValue(), as a const function can not call a non-const function.

Usman
June 17, 2019 at 12:52 pm · Reply

What will be the working of the class if it has a constant function and instantiated to a static object??

Ishak
April 15, 2019 at 3:27 pm · Reply

Excuse me but how did we create a constant object using the second way (the constant reference way), if I can still change the object named date with the void function named setDate()?
Thanks in advance

nascardriver
April 16, 2019 at 3:41 am · Reply

You can't change a const object.

```
1  Date d{ 1, 2, 3};
2
3  const Date &ref{ d };
4
5  ref.getDay(); // Ok, @getDay is marked const
6  ref.setDate(3, 2, 1); // Illegal, @setDate is non-const.
```

Qing Lu
January 28, 2019 at 3:10 pm · Reply

"that is, it’s legal for a const class object to call a constructor that initializes all, some, or none of the member variables" Can you explain this with an example?
I tried to interprete this with the following code but it generated the error message "error: invalid use of ‘myClass::myClass’"

```
1  #include <iostream>
2
3  class myClass
4  {
5  private:
6      int m_value1;
7  public:
8      myClass(int value=0):m_value1(value){}
9      void print() const
10     {
11         std::cout << m_value1 << '\n';
12     }
13 };
14
15 int main()
16 {
```

```
17      const myClass myObject;
18      myObject.print();
19      myObject.myClass(2);
20      myObject.print();
21      return 0;
22  }
```

nascardriver
January 29, 2019 at 8:41 am · Reply

* Line 6, 8, 17: Initialize your variables with uniform initialization.

You cannot call a constructor on an existing object. A constructor can only be called during initialization.

```
1  const myClass myObject;
2  const myClass myOtherObject{ 4 };
```

The sentence only says, that @myClass::myClass can initialize @m_value1, but it doesn't have to. If there were more member variables, the constructor could initialize all, some, or none of those variables.

Donlod
October 1, 2018 at 11:30 am · Reply

Quick question:
Why is this allowed:

```
1  int getNumber() const
2  {
3      return number;
4  }
```

but this not:

```
1  int& getNumber() const
2  {
3      return number;
4  }
```

Error C2440 'return': cannot convert from 'const int' to 'int &'

The return value must be a const int& to work. Why? In the first (working) example the return value is also not a const int.

nascardriver
October 2, 2018 at 8:41 am · Reply

Version 1 creates a copy of @number. The caller can modify that number without affecting the @number object in the class.
Version 2 returns a reference to @number, giving the caller direct access to a local variable. Since that reference isn't const, the caller could modify the value, but the object @getNumber was called on is const, so it's illegal to modify the object.

Andi
September 14, 2018 at 2:27 am · Reply

Hi Alex,
From what I've learned about the hidden "*this" pointer. The compiler does this conversion:

```
int getYear() { return m_year; } // this gets converted
int getYear(Date* const this) { return this->m_year; } // into this with "*this" pointing to
```

Now I think making a member function const does something to "*this". Possibly this conversion?

```
int getYear() const { return m_year; }
int getYear(const Date* const this) { return this->m_year; } // "*this" not only points on a
```

Is this correct?

In the final section on "Overloading const and non-const function" we also have a const return type of std::string&. Meaning at the end of the day, the compiler compiles this:

```
const std::string& getValue(const Something* const this) { return this->m_value; }
```

I found this part of the chapter a bit confusing as the const return type had nothing to do with the overloading of const and non-const function itself. Doing this for a function that returns by value

```
int getYear() { return m_year; }
int getYear() const { return m_year; }
```

makes it easier to understand the overloading concept at first.

nascardriver
September 14, 2018 at 7:30 am · Reply

> Is this correct?
Correct

Kumar
August 13, 2018 at 10:45 pm · Reply

Hi Alex,
Can you please mention the following?:
1) When would we generally want to declare a function as "const". I mean example, the lesson covers why we would need to.
2) Example of when we would want to declare a const class object, besides passing a const reference to a function.

I am having trouble thinking of examples for this in real life. All the other chapter subsections have these examples/tips which makes it easier to understand and remember when to do what.
I think this Subsection would need an update on that. The "What, why, how?" is covered but the "when" may need an update.

Love the lessons, thank you for your hardwork!

Alex
August 17, 2018 at 3:49 pm · Reply

1) Const class objects can only explicitly call const member functions, therefore you should mark
2) I added the following line, "Just like with normal variables, you'll generally want to make your class objects const when you need to ensure they aren't modified after creation."

Examples are fairly easy to imagine -- basically, any class object that might be used as a constant value would benefit from being made const. e.g:

```
const std::string helloWorld { "Hello, world!" }; // should be const since this std::s
```

```
const Fraction half { 1, 2 }; // should be const since the half fraction should always
```

hrmn
July 7, 2018 at 11:57 am · Reply

"The const version of getValue() will work with either const or non-const objects",
suppose both getvalue(const and non const) not changing member vars , and are little different ,one print "const" other "non const" ,
which will get called from non const object.

thanks.

nascardriver
July 8, 2018 at 7:56 am · Reply

Hi hrmn!

Non-const objects prefer the non-const functions.

```
#include <iostream>

class C
{
public:
  void fn(void) const
  {
    std::cout << "const" << std::endl;
  }

  void fn(void)
  {
    std::cout << "non-const" << std::endl;
  }
};

int main(void)
{
  C c{ };
  const C c2{ };

  c.fn();
  c2.fn();
  // If we have a non-const object and want to call the const overload
  const_cast<const C &>(c).fn();

  return 0;
}
```

Output
non-const
const
const

hrmn
July 8, 2018 at 8:00 am · Reply

it clear things,

thanks that was helpful.

Aakash
June 14, 2018 at 9:58 pm · Reply

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

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

//difference : rule->Rule

Peter Baum
May 18, 2018 at 12:26 pm · Reply

Section: Const references

As noted previously, "performant" is not a real word. Also, saying that something performs better doesn't say in what way it performs better. Are we executing faster?
Are we saving memory? Are we making the code easier to understand?

Aron
April 2, 2018 at 6:23 am · Reply

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

I understand the usage of two const, as you replied to someone "The leftmost const is part of the function's return type, indicating that a const reference is returned. The rightmost const indicates that the member function itself is const. This means it won't change the state of the object and can be called by const objects of the class."

However, why will the compiler issue an error when I remove the first const? Const object could still call this function because a const is added between the function name and the function body even after i remove the first const.

nascardriver
April 2, 2018 at 6:49 am · Reply

Hi Aron!

Removing the first const causes an error, because it will allow direct access to @m_value.

```
1 // Allowed if the return value of @getValue is not const
2 obj.getValue() = "Hello World";
```

But const functions aren't allowed to modify members so you're getting an error.

ankit mahlawat
December 15, 2017 at 9:17 am · Reply

I have similar doubt(as of jules) but with const part of getvalue function i.e.

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

now when you call its function in main i.e

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

it will return const string refrence but you are not taking that refrence anywhere in any const variable isn't it wrong. I think it should be done like this

```
const string s=something2.getValue(); // calls const getValue();
```

Alex
December 16, 2017 at 5:26 pm · Reply

It's perfectly okay to ignore a return value. You are not obligated to print it or assign it to anything.

Ishak
July 18, 2018 at 8:19 am · Reply

Is the leftmost const a must? or is the rightmost const enough to overload the function?

Alex
July 24, 2018 at 2:40 pm · Reply

The leftmost const is part of the return type, and return types aren't considered when determining whether a function is unique or not.

Rohde Fischer
December 13, 2017 at 2:15 am · Reply

Wouldn't this be a good place to cover the advantages of immutability, and how immutability heavily improves ones design, especially in regard to concurrency (even though you haven't covered that yet)? And also cover the tradeoffs with immutability, for instance some people might be coding for embedded devices, where the cost of extra instances etc. might not be acceptable

Alex
December 13, 2017 at 10:33 am · Reply

Yes, it might be worth a mention. I'll add it to my to-do list. Thanks for the suggestion!

Jules
October 20, 2017 at 5:17 am · Reply

Hello,

I don't understand how this:

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

can be called with:

```
something.getValue() = "Hi"
```

With the assignation of a value?

Thanks.

Alex
October 21, 2017 at 12:06 pm · Reply

getValue() returns a reference to m_value, so the return value can be treated as if it were m_value itself. So something.getValue() = "Hi" is essentially the same as something.m_value = "Hi".

Sam Meachem
March 5, 2018 at 5:25 am · Reply

Is this a flaw in C++ then? Because m_value is a private element of the class, doesn't this mean that the reference has different qualities and essentially breaks the properties of the class? Or are references supposed to work this way, forgive me, this is my second run through these tutorials and I've only made it to chapter 9 on the first pass.

nascardriver
March 5, 2018 at 10:45 am · Reply

This is intentional. You can limit access to the returned data using the const keyword, this will prevent the caller from making any changed to the data.

```
1  int &getValue(void)
2  {
3    return this->m_i;
4  }
5
6  const int &getValueConst(void)
7  {
8    return this->m_i;
9  }
10
11 ...
12
13 something.getValue() = 8; // Allowed
14 something.getValueConst() = 8; // Disallowed, return value is read-only
```

Alex
March 6, 2018 at 3:20 pm · Reply

Yup, making a member private only means the member can't be accessed _directly_ from outside the class. However, it can still be accessed indirectly (through a pointer or reference) if you expose public methods which return those.

You can also limit exposure by returning members of fundamental types by value instead of reference.

Nguyen
June 21, 2019 at 1:57 pm · Reply

"You can also limit exposure by returning members of fundamental types by value instead of reference."

Sounds to me I can't return a private member variable by value but it is ok that I can return a private member variable by reference?

```
#include <iostream>
using namespace std;

class Something
{
  int m_value = 555;

public:

    int getValue() { return m_value ; }
};

int main()
{
    Something something;
    std::cout<<something.getValue();

    return 0;
}
```

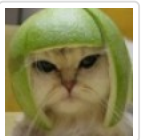
In my example program, it is understood that main() is not a member of Something; therefore, it does not have access to m_value private member variable. Then, why getValue() can return something.m_value to main() without any problem?

nascardriver
June 22, 2019 at 1:52 am · Reply

`main` doesn't have access to `something.m_value`, `Something::getValue` returns a copy of `m_value`. `main` can't modify it and it doesn't notice if `something.m_value` gets modified.

ujjwal jain
September 9, 2017 at 1:53 pm · Reply

Hi Alex,
It is mentioned there we can not create const constructor and const object can only call const function so when we create a object like ex: const Example objEx; how this is able to initialize via non const constructor?



Alex
September 9, 2017 at 8:33 pm · Reply

Const variables still need to be initialized:

```
const int x = 5;
```

Const classes are no different -- when you create a const object, the constructor has to be non-const so you can initialize the class to whatever value is appropriate.

When a class object is created, the constructor is called implicitly (by the code that creates the object) -- it's smart enough to know it should always invoke a non-const constructor.

rara

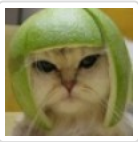
August 20, 2017 at 10:45 pm · Reply

hi Alex,

there is no return in getValue() function but member variable "m_something" has value "hi", how can "m_something" has value ?, is this default assignment to the first member variable ?,

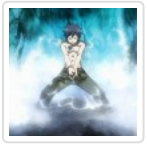
-sorry bad english-

```
#include <string>
#include <iostream>
class Something
{
private:
    std::string m_something;
    std::string m_value;
public:
    Something(const std::string &value="") {  }

    std::string& getValue() { /* no return */ }

    void Print(){std::cout << m_something;}
};

int main()
{
    Something something;

    something.getValue() = "hi";

        something.Print();

    return 0;
}
```



Alex
August 21, 2017 at 10:01 pm · Reply

If your compiler doesn't prevent this (which it really should), the C++ standard has this to say: "Flowing off the end of a function is equivalent to a return with no value; this results in undefined behavior in a value-returning function."

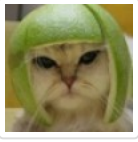


blaze077
May 21, 2017 at 8:45 am · Reply

Hi, could you tell me why we need to use const twice in the const function?

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

I conjecture that the second const (right before the function body) is what distinguishes the const function from the non-const function during method overloading but I do not understand the use of the first const (right before the function type).
Thanks.



Alex
May 21, 2017 at 3:09 pm · Reply

The leftmost const is part of the function's return type, indicating that a const reference is returned. The rightmost const indicates that the member function itself is const. This

means it won't change the state of the object, and can be called by const objects of the class.

blaze077
May 21, 2017 at 3:12 pm · Reply

Thanks, got it.

« Older Comments 1 2