8.10 — Const class objects and member functions

BY ALEX ON SEPTEMBER 11TH, 2007 | LAST MODIFIED BY ALEX ON JUNE 18TH, 2018

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 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.

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 list, bu
t 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 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 member varia
bles.
};
```

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.

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
{
private:
    std::string m_value;

public:
    Something(const std::string &value="") { m_value= value; }

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

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:

```
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