14.4 — Const class objects and const member functions

learncpp.com/cpp-tutorial/const-class-objects-and-const-member-functions/

In lesson 5.1 -- Constant variables (named constants), 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.

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.

A 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:

```
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 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, 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.

```
#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.