Methods

Let's study class methods in detail.

WE'LL COVER THE FOLLOWING

As we discussed earlier, **methods** are the functions associated with a class or struct. Normal methods can only be called through a class instance. However, `static` methods can be called without an instance as well.

Defining methods

A class method has to be declared inside the class, but can be defined outside using the scope resolution operator, `::`.

```
struct Account{
  double getBalance() const;
  void withdraw(double amt);
  void deposit(double amt) { balance += amt; }
private:
  double balance;
};

double Account::getBalance() const { return balance; }
inline void Account::withdraw(double amt){ balance -= amt; }
```

If it is defined inside the class, the compiler automatically considers it an *inline* function.

`this` pointer

The `this` pointer can be used to access the attributes of a class within a method. Every attribute has an implicit `this` pointer. Because of this, we can directly access the attribute inside a function. But what would happen if the method contains a variable that has the same name as an attribute of the class?

We can easily differentiate between the two by explicitly using `this` to access the attribute. Since `this` is a pointer, we must use the arrow syntax to access members.

```
struct Base{
  int a{1998}; // Member
  void newA(){
    int a{2011}; // Local variable in the method
    std::cout << this->a; // 1998
    std::cout << a; // 2011
  }
};
```

As we can see, `this` is sometimes necessary to access the hidden attributes of a class.

`static` methods

Like static attributes, **static** methods can be used with or without an instance of the class. We can define static methods using the `static` keyword and call them using the scope resolution operator, `::`.

```
class Account{
public:
  static int getDeposits(){ return deposits; }
private:
  static int deposits;
};

...
int total = Account::getDeposits();
```

> **Note**: Static methods do not have `this` pointers. They can only access

> static attributes and methods.

`const` methods

- A **constant** method cannot modify the object that calls it.
- A method can be declared constant by using the `const` keyword.
- Constant objects can only call `const` or `constexpr` methods.
- Constant methods can only change an instance variable if the instance variable is declared `mutable`.

```
class Account{
public:
  double getBalance() const { return balance; }
private:
  double balance;
};

...
const Account acc;
double account = acc.getBalance();
```

Changing the value of `balance` in `getBalance()` would cause an error since it is a `const` method. The `acc` instance has been declared `const`, but an instance does not have to be `const` to use constant methods.

`constexpr` methods

The aim of defining `constexpr` methods is to improve the performance of our code. `constexpr` methods are executed at compile-time and can later be used at runtime without any redundant computations.

It can also be evaluated at runtime if **non-**`constexpr` arguments are provided. One thing to remember is that `constexpr` methods are implicitly `const`.

Such methods can only call other `constexpr` functions, methods, and global variables.

`constexpr` methods can only be called by `constexpr` objects.

`const` vs. `constexpr`

`const` methods are used to increase safety. They restrict modification access to the attributes of the class.

On the other hand, `constexpr` methods are used to increase performance and optimize the program.

In the next lesson, we will observe examples of all the types of methods we've studied here.