More Kinds of Constructors

After the default and parameterized constructors, we will study a few more constructor types that make classes more convenient.

WE'LL COVER THE FOLLOWING

Copy constructors

> The copy constructor allows a class to create an object by copying an existing object.

They expect a constant reference to another instance of the class as their argument.

```
class Account{
public:
    Account(const Account& other);
};
```

All the values of `other` can be copied into the new object. Both objects will have the same values afterward.

Move constructors

> The move constructor allows the data of one object to be transferred completely to another object.

They are a more efficient alternative to the copy constructor since everything is being *moved* instead of *copied*.

They expect a non-constant **rvalue-reference** to an instance of the class as their argument.

```
class Account{ public:
      Account(Account&& other);
};
```

After the move operation, `other` is in a moved-from state. Accessing it will result in undefined behavior. To use it again, we would have to re-initialize it.

Explicit constructors

> The explicit constructor is used to avoid implicit calls to a class's constructor.

Consider the following `Account` constructor:

```
public:
    Account(double b): balance(b){}
private:
    double balance;
    std::string cur;
};
```

An instance can be created like this:

```
Account acc = 100.0;
```

A `double` is being assigned to an `Account` object, but the compiler implicitly calls the constructor that takes a `double` as an argument. Hence, the operation works without any errors.

If the constructor had been made explicit, this statement would not have worked. For this, we would have to use the `explicit` keyword.

```
class Account{
public:

    explicit Account(double b): balance(b){}
    Account (double b, std::string c): balance(b), cur(c){}
private:
    double balance;
    std::string cur;
};
Account account = 100.0; // ERROR: implicit conversion
Account account(100.0); // OK: explicit invocation
Account account = {100.0,"EUR"}; // OK: implicit conversion
```

Now, the assignment operator won’t work as it did before, though it still works for `Account(double b, std::string c)` since it has not been made explicit.

Example

Here’s a complete example showing the use of the `explicit` keyword:

Implicit | Explicit

```
#include <iostream>
#include <string>

class Account{
public:
  Account(double b): balance(b){}
  Account(double b, std::string c):balance(b), cur(c){}

private:
    double balance;
    std::string cur;
};

void strange(Account a){
  std::cout << "It works!" << std::endl;
}

int main(){

   Account account = 100.0; // No ERROR
   Account account1(100.0);
   Account account2 = {100.0, "EUR"};
   strange(100.0);        // No ERROR
   strange(false);

}
```

- In the implicit approach, the assignment operations in lines 20 and 22 do not cause an error.
- In the implicit approach, the function `strange` has an `Account` parameter, but passing it a `double` or `bool` implicitly calls the `Account` constructor, as done in lines 23 and 24.

- When the `explicit` keyword is introduced in the code, implicit constructor calls are restricted.

- Uncomment the lines to observe the error shown by the compiler in the **explicit** code tab.

The next lesson deals with the concept of **instance initializers**.