14.9 — Introduction to constructors

When a class type is an aggregate, we can use aggregate initialization to initialize the class type directly:

```
struct Foo // Foo is an aggregate
{
    int x {};
    int y {};
};

int main()
{
    Foo foo { 6, 7 }; // uses aggregate initialization

    return 0;
}
```

Aggregate inititalization does memberwise initialization (members are initialized in the order in which they are defined). So when `foo` is instantiated in the above example, `foo.x` is initialized to `6`, and `foo.y` is initialized to `7`.

Related content

We discuss the definition of an aggregate and aggregate initialization in lesson 13.8 -- Struct aggregate initialization.

However, as soon as we make any member variables private (to hide our data), our class type is no longer an aggregate (because aggregates cannot have private members). And that means we're no longer able to use aggregate initialization:

```
class Foo // Foo is not an aggregate (has private members)
{
    int m_x {};
    int m_y {};
};

int main()
{
    Foo foo { 6, 7 }; // compile error: can not use aggregate initialization

    return 0;
}
```

Not allowing class types with private members to be initialized via aggregate initialization makes sense for a number of reasons:

- Aggregate initialization requires knowing about the implementation of the class (since you have to know what the members are, and what order they were defined in), which we’re intentionally trying to avoid when we hide our data members.
- If our class had some kind of invariant, we’d be relying on the user to initialize the class in a way that preserves the invariant.

So then how do we initialize a class with private member variables? The error message given by the compiler for the prior example provides a clue: “error: no matching constructor for initialization of ‘Foo'”

We must need a matching constructor. But what the heck is that?

Constructors

A **constructor** is a special member function that is automatically called after a non-aggregate class type object is created.

When a non-aggregate class type object is defined, the compiler looks to see if it can find an accessible constructor that is a match for the initialization values provided by the caller (if any).

- If an accessible matching constructor is found, memory for the object is allocated, and then the constructor function is called.
- If no accessible matching constructor can be found, a compilation error will be generated.

Key insight

Many new programmers are confused about whether constructors create the objects or not. They do not -- the compiler sets up the memory allocation for the object prior to the constructor call. The constructor is then called on the uninitialized object.

However, if a matching constructor cannot be found for a set of initializers, the compiler will error. So while constructors don’t create objects, the lack of a matching constructor will prevent creation of an object.

Beyond determining how an object may be created, constructors generally perform two functions:

- They typically perform initialization of any member variables (via a member initialization list)
- They may perform other setup functions (via statements in the body of the constructor). This might include things such as error checking the initialization values, opening a file or database, etc…

After the constructor finishes executing, we say that the object has been “constructed”, and the object should now be in a consistent, usable state.

Note that aggregates are not allowed to have constructors -- so if you add a constructor to an aggregate, it is no longer an aggregate.

Naming constructors

Unlike normal member functions, constructors have specific rules for how they must be named:

- Constructors must have the same name as the class (with the same capitalization). For template classes, this name excludes the template parameters.
- Constructors have no return type (not even `void`).

Because constructors are typically part of the interface for your class, they are usually public.

A basic constructor example

Let’s add a basic constructor to our example above:

```
#include <iostream>

class Foo
{
private:
    int m_x {};
    int m_y {};

public:
    Foo(int x, int y) // here's our constructor function that takes two initializers
    {
        std::cout << "Foo(" << x << ", " << y << ") constructed\n";
    }

    void print() const
    {
        std::cout << "Foo(" << m_x << ", " << m_y << ")\n";
    }
};

int main()
{
    Foo foo{ 6, 7 }; // calls Foo(int, int) constructor
    foo.print();

    return 0;
}
```

This program will now compile and produce the result:

```
Foo(6, 7) constructed
Foo(0, 0)
```

When the compiler sees the definition `Foo foo{ 6, 7 }`, it looks for a matching `Foo` constructor that will accept two `int` arguments. `Foo(int, int)` is a match, so the compiler will allow the definition.

At runtime, when `foo` is instantiated, memory for `foo` is allocated, and then the `Foo(int, int)` constructor is called with parameter `x` initialized to `6` and parameter `y` initialized to `7`. The body of the constructor function then executes and prints `Foo(6, 7) constructed`.

When we call the `print()` member function, you'll note that members `m_x` and `m_y` have value 0. This is because although our `Foo(int, int)` constructor function was called, it did not actually initialize the members. We'll show how to do that in the next lesson.

Constructor implicit conversion of arguments

In lesson 10.1 -- Implicit type conversion, we noted that the compiler will perform implicit conversion of arguments in a function call (if needed) in order to match a function definition where the parameters are a different type:

```
void foo(int, int)
{
}

int main()
{
    foo('a', true); // will match foo(int, int)

    return 0;
}
```

This is no different for constructors: the `Foo(int, int)` constructor will match any call whose arguments are implicitly convertible to `int`:

```
class Foo
{
public:
    Foo(int x, int y)
    {
    }
};

int main()
{
    Foo foo{ 'a', true }; // will match Foo(int, int) constructor

    return 0;
}
```

Constructors should not be const

A constructor needs to be able to initialize the object being constructed -- therefore, a constructor must not be const.

```
#include <iostream>

class Something
{
private:
    int m_x{};

public:
    Something() // constructors must be non-const
    {
        m_x = 5; // okay to modify members in non-const constructor
    }

    int getX() const { return m_x; } // const
};

int main()
{
    const Something s{}; // const object, implicitly invokes (non-const) constructor

    std::cout << s.getX(); // prints 5

    return 0;
}
```

Normally a non-const member function can’t be invoked on a const object. However, because the constructor is invoked implicitly, a non-const constructor can be invoked on a const object.

Constructors vs setters

Constructors are designed to initialize an entire object at the point of instantiation. Setters are designed to assign a value to a single member of an existing object.