12.5 — Constructors

ALEX SEPTEMBER 12, 2021

When all members of a class (or struct) are public, we can use**aggregate initialization** to initialize the class (or struct) directly using list-initialization:

```
class Foo
{
public:
    int m_x {};
    int m_y {};
};

int main()
{
    Foo foo { 6, 7 }; // list-initialization

    return 0;
}
```

However, as soon as we make any member variables private, we're no longer able to initialize classes in this way. It does make sense: if you can't directly access a variable (because it's private), you shouldn't be able to directly initialize it.

So then how do we initialize a class with private member variables? The answer is through constructors.

Constructors

A **constructor** is a special kind of class member function that is automatically called when an object of that class is created. Constructors are typically used to initialize member variables of the class to appropriate user-provided values, or to do any setup steps necessary for the class to be used (e.g. open a file or database).

After a constructor executes, the object should be in a well-defined, usable state.

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

1. Constructors must have the same name as the class (with the same capitalization)

2. Constructors have no return type (not even void)

Default constructors and default initialization

A constructor that takes no parameters (or has parameters that all have default values) is called a**default constructor**. The default constructor is called if no user-provided initialization values are provided.

Here is an example of a class that has a default constructor:

```
#include <iostream>

class Fraction
{
private:
    int m_numerator {};
    int m_denominator {};

public:
    Fraction() // default constructor
    {
        m_numerator = 0;
        m_denominator = 1;
    }

    int getNumerator() { return m_numerator; }
    int getDenominator() { return m_denominator; }
    double getValue() { return static_cast<double>(m_numerator) /
m_denominator; }
};

int main()
{
    Fraction frac{}; // calls Fraction() default constructor
    std::cout << frac.getNumerator() << '/' << frac.getDenominator() << '\n';

    return 0;
}
```

This class was designed to hold a fractional value as an integer numerator and denominator. We have defined a default constructor named Fraction (the same as the class).

When the line `Fraction frac{};` executes, the compiler will see that we're instantiating an object of type Fraction with no arguments. It then performs value-initialization of `frac`, that is, the default constructor gets called. Although technically incorrect, this is often called **default initialization**. In this case, the default constructor will be called immediately after memory has been allocated and cleared for the object. The default constructor runs just like a normal function (assigning the values 0 to m_numerator and 1 to m_denominator).

This program produces the result:

```
0/1
```

List-initialization

In the above program, we initialized our class object using list-initialization:

```
Fraction frac {}; // List-initialization with empty set of braces (leads to value-
initialization)
```

We can also initialize class objects using default-initialization:

```
Fraction frac; // Default-initialization, calls default
constructor
```

For the most part, default- and list-initialization of a class object results in the same outcome: the default constructor is called.

Many programmers favor default-initialization over list-initialization for class objects. This is because when using value-initialization, the compiler may zero-initialize the class members before calling the default constructor in certain cases, which is slightly inefficient (C++ programmers don't like paying for features they're not using).

However, favoring default-initialization also comes with a downside: you have to know whether a type will initialize itself, ie. it is a class-type and all members have an initializer, or there is a default-constructor that initializes all member variables. If you see a defined variable without an initializer, you have to think about whether that's a mistake or not (depending on what type the object it is).

For example, the following code causes undefined behavior

```
#include <iostream>

class Fraction
{
private:
    // Removed initializers
    int m_numerator;
    int m_denominator;

public:
    // Removed default-constructor

    int getNumerator() { return m_numerator; }
    int getDenominator() { return m_denominator; }
    double getValue() { return static_cast<double>(m_numerator) /
m_denominator; }
};

int main()
{
    Fraction frac;
    // frac is uninitialized, accessing its members causes undefined behavior
    std::cout << frac.getNumerator() << '/' << frac.getDenominator() << '\n';

    return 0;
}
```

While you might be able to initialize all members in the classes you write, it's not feasible to read the definitions of all classes you use to make sure they do the same.

Favoring value initialization for class objects is simple, consistent, and can help you catch errors, particularly while you are learning.

Best practice

Favor list-initialization over default-initialization for class objects.

Direct- and list-initialization using constructors with parameters

While the default constructor is great for ensuring our classes are initialized with reasonable default values, often times we want instances of our class to have specific values that we provide. Fortunately, constructors can also be declared with parameters. Here is an example of a constructor that takes two integer parameters that are used to initialize the numerator and denominator:

```cpp
#include <cassert>

class Fraction
{
private:
    int m_numerator {};
    int m_denominator {};

public:
    Fraction() // default constructor
    {
         m_numerator = 0;
         m_denominator = 1;
    }

    // Constructor with two parameters, one parameter having a default value
    Fraction(int numerator, int denominator=1)
    {
        assert(denominator != 0);
        m_numerator = numerator;
        m_denominator = denominator;
    }

    int getNumerator() { return m_numerator; }
    int getDenominator() { return m_denominator; }
    double getValue() { return static_cast<double>(m_numerator) /
m_denominator; }
};

```

Note that we now have two constructors: a default constructor that will be called in the default case, and a second constructor that takes two parameters. These two constructors can coexist peacefully in the same class due to function overloading. In fact, you can define as many constructors as you want, so long as each has a unique signature (number and type of parameters).

So how do we use this constructor with parameters? It's simple! We can use list or direct initialization:

```cpp
Fraction fiveThirds{ 5, 3 }; // List initialization, calls Fraction(int, int)
Fraction threeQuarters(3, 4); // Direct initialization, also calls Fraction(int,
int)
```

As always, we prefer list initialization. We'll discover reasons (Templates and std::initializer_list) to use direct initialization when calling constructors later in the tutorials. There is another special constructor that might make brace initialization do something different, in that case we have to use direct initialization. We'll talk about these constructors later.

Note that we have given the second parameter of the constructor with parameters a default value, so the following is also legal:

```cpp
Fraction six{ 6 }; // calls Fraction(int, int) constructor, second parameter uses default value
of 1
```

Default values for constructors work exactly the same way as with any other functions, so in the above case where we ca`six{ 6 }`, the `Fraction(int, int)` function is called with the second parameter defaulted to value 1.

Best practice

Favor brace initialization to initialize class objects.

Copy initialization using equals with classes

Much like with fundamental variables, it's also possible to initialize classes using copy initialization:

```
Fraction six = Fraction{ 6 }; // Copy initialize a Fraction, will call Fraction(6, 1)
Fraction seven = 7; // Copy initialize a Fraction.  The compiler will try to find a way to convert 7 to
a Fraction, which will invoke the Fraction(7, 1) constructor.
```

However, we recommend you avoid this form of initialization with classes, as it may be less efficient. Although direct-initialization, list-initialization, and copy-initialization all work identically with fundamental types, copy-initialization does not work the same with classes (though the end-result is often the same). We'll explore the differences in more detail in a future chapter.

Reducing your constructors

In the above two-constructor declaration of the Fraction class, the default constructor is actually somewhat redundant. We could simplify this class as follows:

```
#include <cassert>

class Fraction
{
private:
    int m_numerator {};
    int m_denominator {};

public:
    // Default constructor
    Fraction(int numerator=0, int denominator=1)
    {
        assert(denominator != 0);

        m_numerator = numerator;
        m_denominator = denominator;
    }

    int getNumerator() { return m_numerator; }
    int getDenominator() { return m_denominator; }
    double getValue() { return static_cast<double>(m_numerator) /
m_denominator; }
};

```

Although this constructor is still a default constructor, it has now been defined in a way that it can accept one or two user-provided values as well.

```
Fraction zero; // will call Fraction(0, 1)
Fraction zero{}; // will call Fraction(0, 1)
Fraction six{ 6 }; // will call Fraction(6, 1)
Fraction fiveThirds{ 5, 3 }; // will call Fraction(5,
3)
```

When implementing your constructors, consider how you might keep the number of constructors down through smart defaulting of values.

A reminder about default parameters

The rules around defining and calling functions that have default parameters (described in lesson8.12 -- Default arguments) apply to constructors too. To recap, when defining a function with default parameters, all default parameters must follow any non-default parameters, i.e. there cannot be non-defaulted parameters after a defaulted parameter.

This may produce unexpected results for classes that have multiple default parameters of different types. Consider:

```
class Something
{
public:
 // Default constructor
 Something(int n = 0, double d = 1.2) // allows us to construct a Something(int, double), Something(int), or Something()
 {
 }
};

int main()
{
 Something s1 { 1, 2.4 }; // calls Something(int, double)
 Something s2 { 1 }; // calls Something(int, double)
 Something s3 {}; // calls Something(int, double)

 Something s4 { 2.4 }; // will not compile, as there's no constructor to handle Something(double)

 return 0;
}
```

With `s4`, we've attempted to construct a `Something` by providing only a `double`. This won't compile, as the rules for how arguments match

with default parameters won't allow us to skip a non-rightmost parameter (in this case, the leftmost int parameter).

If we want to be able to construct a `Something` with only a `double`, we'll need to add a second (non-default) constructor:

```
class Something
{
public:
 // Default constructor
 Something(int n = 0, double d = 1.2) // allows us to construct a Something(int, double), Something(int), or Something()
 {
 }

 Something(double d)
 {
 }
};

int main()
{
 Something s1 { 1, 2.4 }; // calls Something(int, double)
 Something s2 { 1 }; // calls Something(int, double)
 Something s3 {}; // calls Something(int, double)

 Something s4 { 2.4 }; // calls Something(double)

 return 0;
}
```

An implicitly generated default constructor

If your class has no constructors, C++ will automatically generate a public default constructor for you. This is sometimes called an **implicit constructor** (or implicitly generated constructor).

Consider the following class:

```
class Date
{
private:
    int m_year{ 1900 };
    int m_month{ 1 };
    int m_day{ 1 };

    // No user-provided constructors, the compiler generates a default constructor.
};

int main()
{
    Date date{};

    return 0;
}
```

The Date class has no constructors. Therefore, the compiler will generate a default constructor that allows us to create a `Date` object without arguments.

When the generated default constructor is called, members will still be initialized if they have non-static member initializers.

If your class has any other constructors, the implicitly generated constructor will not be provided. For example:

```
class Date
{
private:
    int m_year{ 1900 };
    int m_month{ 1 };
    int m_day{ 1 };

public:
    Date(int year, int month, int day) // normal non-default constructor
    {
        m_year = year;
        m_month = month;
        m_day = day;
    }

    // No implicit constructor provided because we already defined our own constructor
};

int main()
{
    Date date{}; // error: Can't instantiate object because default constructor doesn't exist and the compiler won't generate one
    Date today{ 2020, 1, 19 }; // today is initialized to Jan 19th, 2020

    return 0;
}
```

If your class has another constructor and you want to allow default construction, you can either add default arguments to every parameter of a constructor with parameters, or explicitly define a default constructor.

There's a third option as well: you can use the default keyword to tell the compiler to create a default constructor for us anyway:

```
class Date
{
private:
    int m_year{ 1900 };
    int m_month{ 1 };
    int m_day{ 1 };

public:
    // Tell the compiler to create a default constructor, even if
    // there are other user-provided constructors.
    Date() = default;

    Date(int year, int month, int day) // normal non-default constructor
    {
        m_year = year;
        m_month = month;
        m_day = day;
    }
};

int main()
{
    Date date{}; // date is initialized to Jan 1st, 1900
    Date today{ 2020, 10, 14 }; // today is initialized to Oct 14th, 2020

    return 0;
}
```

Using `= default` is longer than writing a constructor with an empty body, but expresses better what your intentions are (To create a default constructor), and it's safer, because it can zero-initialize members even if they have not been initialized at their declaration. `= default` also works for other special constructors, which we'll talk about in the future.

Best practice

If you have constructors in your `class` and need a default constructor that does nothing (e.g. because all your members are initialized using non-static member initialization), use `= default`.

Classes containing classes

A `class` may contain other classes as member variables. By default, when the outer class is constructed, the member variables will have their default constructors called. This happens before the body of the constructor executes.

This can be demonstrated thusly:

```
#include <iostream>

class A
{
public:
    A() { std::cout << "A\n"; }
};

class B
{
private:
    A m_a; // B contains A as a member variable

public:
    B() { std::cout << "B\n"; }
};

int main()
{
    B b;
    return 0;
}
```

This prints:

```
A
B
```

When variable `b` is constructed, the `B()` constructor is called. Before the body of the constructor executes, `m_a` is initialized, calling the `class A` default constructor. This prints "A". Then control returns back to the `B` constructor, and the body of the B constructor executes.

This makes sense when you think about it, as the `B()` constructor may want to use variable `m_a` -- so `m_a` had better be initialized first!

The difference to the last example in the previous section is that `m_a` is a `class`-type. `class`-type members get initialized even if we don't explicitly initialize them.

In the next lesson, we'll talk about how to initialize these class member variables.

Constructor notes

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.

Constructors actually serve two purposes. First, constructors determine who is allowed to create an object. That is, an object of a class can only be created if a matching constructor can be found.

Second, constructors can be used to initialize objects. Whether the constructor actually does an initialization is up to the programmer. It's syntactically valid to have a constructor that does no initialization at all (the constructor still serves the purpose of allowing the object to be created, as per the above).

However, much like it is a best practice to initialize all local variables, it's also a best practice to initialize all member variables on creation of the object. This can be done via a constructor or via non-static member initialization.

Best practice

Always initialize all member variables in your objects.

Finally, constructors are only intended to be used for initialization when the object is created. You should not try to call a constructor to re-initialize an existing object. While it may compile, the results will not be what you intended (instead, the compiler will create a temporary object and then discard it).

Quiz time

Question #1

Write a `class` named Ball. Ball should have two private member variables with default values: `m_color` ("black") and `m_radius` (10.0). `Ball` should provide constructors to set only `m_color`, set only `m_radius`, set both, or set neither value. For this quiz question, do not use default parameters for your constructors. Also write a function to print out the color and radius of the ball.

The following sample program should compile:

```
int main()
{
 Ball def{};
 def.print();

 Ball blue{ "blue" };
 blue.print();

 Ball twenty{ 20.0 };
 twenty.print();

 Ball blueTwenty{ "blue", 20.0
};
 blueTwenty.print();

 return 0;
}
```

and produce the result:

```
color: black, radius: 10
color: blue, radius: 10
color: black, radius: 20
color: blue, radius: 20
```

Show Solution

b) Update your answer to the previous question to use constructors with default parameters. Use as few constructors as possible.

Show Solution

Question #2

What happens if you don't declare a default constructor?

Show Solution

Next lesson
12.6 Constructor member initializer lists

Back to table of contents

Previous lesson
12.4 Access functions and encapsulation

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ coc

Name*

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies: POST COMMENT

DP N N FOUT

Newest