8.5 — Constructors

BY ALEX ON SEPTEMBER 5TH, 2007 | LAST MODIFIED BY ALEX ON MAY 21ST, 2018

When all members of a class (or struct) are public, we can initialize the class (or struct) directly using an initialization list or uniform initialization (in C++11):

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

int main()
{
    Foo foo1 = { 4, 5 }; // initialization list
    Foo foo2 { 6, 7 }; // uniform initialization (C++11)

    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 instantiated. Constructors are typically used to initialize member variables of the class to appropriate default or user-provided values, or to do any setup steps necessary for the class to be used (e.g. open a file or database).

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

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:

```cpp
#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; // Since no arguments, 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).

Because we're instantiating an object of type Fraction with no arguments, the default constructor will be called immediately after memory is allocated for the object, and our object will be initialized.

This program produces the result:

```
0/1
```

Note that our numerator and denominator were initialized with the values we set in our default constructor! This is such a useful feature that almost every class includes a default constructor. Without a default constructor, the numerator and denominator would have garbage values until we explicitly assigned them reasonable values (remember: fundamental variables aren't initialized by default).

Direct and uniform 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:

```
#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 just use the direct initialization form of initialization:

```
int x(5); // Direct initialize an integer
```

```
Fraction fiveThirds(5, 3); // Direct initialize a Fraction, calls Fraction(int, int) constructor
```

This particular fraction will be initialized to the fraction 5/3!

In C++11, we can also use uniform initialization:

```
int x { 5 }; // Uniform initialization of an integer
Fraction fiveThirds {5, 3}; // Uniform initialization of a Fraction, calls Fraction(int, int) constructor
```

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

```
Fraction six(6); // calls Fraction(int, int) constructor, second parameter uses default value
```

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

Rule: Use direct or uniform initialization with your classes

Copy initialization using equals with classes

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

```
int x = 6; // Copy initialize an integer
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, uniform 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.

Rule: Do not copy initialize your classes

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 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 lesson **7.7 -- Default parameters**) apply to constructors too. To recap, when defining a function with default parameters, all default parameters must follow any non-default parameters.

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

```cpp
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:

```cpp
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 other 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:

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

This class has no constructor. Therefore, the compiler will generate a constructor that behaves identically to the following:

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

public:
    Date() // implicitly generated constructor
    {
    };
};
```

This particular implicit constructor allows us to create a Date object with no parameters, but doesn't initialize any of the members (because all of the members are fundamental types, and those don't get initialized upon creation).

Although you can't see the implicitly generated constructor, you can prove it exists:

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

    // No constructor provided, so C++ creates a public default constructor for us
};

int main()
{
    Date date; // calls implicit constructor

    return 0;
}
```

The above code compiles, because date object will use the implicit constructor (which is public).

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

```cpp
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
};
```

```cpp

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, 10, 14); // today is initialized to Oct 14th, 2020

    return 0;
}
```

Generally speaking, it’s a good idea to always provide at least one constructor in your class. This explicitly allows you to control how objects of your class are allowed to be created, and will prevent your class from potentially breaking later when you add other constructors.

Rule: Provide at least one constructor for your class, even if it’s an empty default constructor.

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:

```cpp
#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!

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 code the compiler creates does that).

Constructors actually serve two purpose. The primary purpose is to initialize objects that have already been created. The secondary purpose is to allow creation of an object. That is, an object of a class can only be created if a matching constructor can be found. This means that a class without any public constructors can’t be created!

Although the main purpose of a constructor is to initialize an object, whether the constructor actually does an initialization is up to the programmer. It’s perfectly 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).

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

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
```

Hide Solution

```
#include <string>
#include <iostream>
class Ball
{
private:
    std::string m_color;
    double m_radius;

public:
        // Default constructor with no parameters
        Ball()
        {
        m_color = "black";
        m_radius = 10.0;
        }

        // Constructor with only color parameter (radius will use default value)
    Ball(const std::string &color)
    {
        m_color = color;
        m_radius = 10.0;
    }

        // Constructor with only radius parameter (color will use default value)
    Ball(double radius)
    {
        m_color = "black";
        m_radius = radius;
    }

        // Constructor with both color and radius parameters
    Ball(const std::string &color, double radius)
    {
        m_color = color;
        m_radius = radius;
    }

    void print()
    {
        std::cout << "color: " << m_color << ", radius: " << m_radius << '\n';
    }
};

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;
}
```

1b) Update your answer to the previous question to use constructors with default parameters. Use as few constructors as possible.

Hide Solution

```
#include <string>
#include <iostream>
class Ball
{
private:
    std::string m_color;
    double m_radius;

public:
        // Constructor with only radius parameter (color will use default value)
    Ball(double radius)
    {
        m_color = "black";
        m_radius = radius;
    }

        // Constructor with both color and radius parameters
        // handles no parameter, color only, and color + radius cases.
    Ball(const std::string &color="black", double radius=10.0)
    {
        m_color = color;
        m_radius = radius;
    }

    void print()
    {
        std::cout << "color: " << m_color << ", radius: " << m_radius << '\n';
    }
};

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;
}
```

2) What happens if you don’t declare a default constructor?

Hide Solution

If you haven’t defined any other constructors, the compiler will create an empty public default constructor for you. This means your objects will be instantiable with no parameters. If you have defined other constructors (default or otherwise), the compiler will not create a default constructor for you. Assuming you haven’t provided a default constructor yourself, your objects will not be instantiable with no parameters.

8.5a -- Constructor member initializer lists

Index

8.4 -- Access functions and encapsulation

Share this:

260 comments to 8.5 — Constructors

« Older Comments 1 2 3 4

Lim Che Ling
June 19, 2018 at 6:08 pm · Reply

Hi, Alex,

I tried to modify the code into this and found that the output has printed only 'B' (NOT A B)