8.5b — Non-static member initialization

BY ALEX ON FEBRUARY 12TH, 2016 | LAST MODIFIED BY ALEX ON AUGUST 18TH, 2017

When writing a class that has multiple constructors (which is most of them), having to specify default values for all members in each constructor results in redundant code. If you update the default value for a member, you need to touch each constructor.

Starting with C++11, it's possible to give normal class member variables (those that don't use the static keyword) a default initialization value directly:

```
class Rectangle
{
private:
    double m_length = 1.0; // m_length has a default value of 1.0
    double m_width = 1.0; // m_width has a default value of 1.0

public:
    Rectangle()
    {
    // This constructor will use the default values above since they aren't overridden here
    }

    void print()
    {
        std::cout << "length: " << m_length << ", width: " << m_width << '\n';
    }
};

int main()
{
    Rectangle x; // x.m_length = 1.0, x.m_width = 1.0
    x.print();

    return 0;
}
```

This program produces the result:

```
length: 1.0, width: 1.0
```

Non-static member initialization (also called in-class member initializers) provides default values for your member variables that your constructors will use if the constructors do not provide initialization values for the members themselves (via the member initialization list).

However, note that constructors still determine what kind of objects may be created. Consider the following case:

```
class Rectangle
{
private:
    double m_length = 1.0;
    double m_width = 1.0;

public:

    // note: No default constructor provided in this example

    Rectangle(double length, double width)
        : m_length(length), m_width(width)
    {
        // m_length and m_width are initialized by the constructor (the default values aren't used)
    }

    void print()
```

```
    {
        std::cout << "length: " << m_length << ", width: " << m_width << '\n';
    }

};

int main()
{
    Rectangle x; // will not compile, no default constructor exists, even though members have default initialization values

    return 0;
}
```

Even though we've provided default values for all members, no default constructor has been provided, so we are unable to create Rectangle objects with no parameters.

If a default initialization value is provided and the constructor initializes the member via the member initializer list, the member initializer list will take precedence. The following example shows this:

```
class Rectangle
{
private:
    double m_length = 1.0;
    double m_width = 1.0;

public:

    Rectangle(double length, double width)
        : m_length(length), m_width(width)
    {
        // m_length and m_width are initialized by the constructor (the default values aren't used)
    }

    void print()
    {
        std::cout << "length: " << m_length << ", width: " << m_width << '\n';
    }

};

int main()
{
    Rectangle x(2.0, 3.0);
    x.print();

    return 0;
}
```

This prints:

```
length: 2.0, width: 3.0
```

Note that initializing members using non-static member initialization requires using either an equals sign, or a brace (uniform) initializer -- the direct initialization form doesn't work here.

Rule: Favor use of non-static member initialization to give default values for your member variables.

Quiz time

1) Update the following program to use non-static member initialization and member initializer lists.

```
#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;
}
```

This program should produce the result:

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

Hide Solution

```
#include <string>
```

```cpp
#include <iostream>
class Ball
{
private:
    std::string m_color = "black";
    double m_radius = 10.0;

public:
        // Default constructor with no parameters (color and radius will use default values)
    Ball()
    {
    }

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

        // Constructor with only radius parameter (color will use default value)
    Ball(double radius):
        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;
}
```

2) Why do we need to declare an empty default constructor in the program above, since all members are initialized via non-static member initialization?

Hide Solution

“Ball def;” will look for a default constructor to handle instantiation of the object. If that default constructor does not exist (even if it is empty), the compiler will throw an error.

8.6 -- Overlapping and delegating constructors