Explicit constructors

```
/*
       . Topics:
            . Reducing untintended implicit conversions


*/
```

Explicit constructors

```
export class Pixel {
public:
    // Default constructor
    Pixel() = default;

    // Constructor with color only, explicit to prevent implicit conversions
    explicit(true) Pixel(uint32_t color) : Pixel(color, 0, 0) {
        fmt::print("Body of the one param constructor\n");
    }

    // Constructor with all arguments
    Pixel(uint32_t color, unsigned int x, unsigned int y)
        : m_color{color}, m_pos_x{x}, m_pos_y{y} {
        fmt::print("Three-argument constructor\n");
    }

    // Copy constructor
    Pixel(const Pixel& other)
        : Pixel(other.m_color, other.m_pos_x, other.m_pos_y) {
        fmt::print("Body of the copy constructor\n");
    }

private:
    uint32_t m_color{0xFF000000};
    unsigned int m_pos_x{0};
    unsigned int m_pos_y{0};
};
```

Explicit constructors

```
uint32_t color = 0xFF00FF00;
ct11::print_pixel(color);    //If the one param constructor is explicit, this will not compile
```

Explicit constructors

```
/*
    . We don't want the compiler to convert a uint32_t to a Pixel object.
    . You can do that :
        . using the explicit keyword as shown below
            . explicit(true) Pixel(uint32_t color) : Pixel(color, 0, 0) {};
            . explicit Pixel(uint32_t color) : Pixel(color, 0, 0) {};
    . explicit(true) is the same as explicit
    . You can explicitly allow implicit conversions by using the explicit(false) keyword.
    . The true/false parameter really becomes useful in generic programming scenarios
*/
```

Constructor and compiler behavior

```
/*
        1. You have no constructors in your class.
            . The compiler generates:
                . Default constructor. You can do something like Pixel p;
                . Copy constructor. You caqn do something like Pixel p1 = p2;
            . Things you can do:
                                Pixel a;          // Default constructor
                                Pixel b{a};       // Copy constructor
*/
```

Constructor and compiler behavior

```
/*
        2. You have a default constructor only:
            . The compiler generates:
                . Copy constructor. You can do something like Pixel p1 = p2;
            . Things you can do:
                                Pixel a;          // Default constructor
                                Pixel b{a};       // Copy constructor
*/
```

Constructor and compiler behavior

```
/*
        3. You have a copy constructor only:
            . The compiler generates:
                . No other constructors. You can't create other objects without making copies,
                . But you can't create an object to copy from in the first, place, so you can't create anything
            . Things you can do:
                                Pixel b{a};       // Copy constructor (if 'Pixel a' exists)
*/
```

Constructor and compiler behavior

```
/*
        4. You have a single or multi-argument non-copy constructor:
            . The compiler generates:
                . Copy constructor. You can do something like Pixel p1 = p2;
            . Things you can do:
                                Pixel a{0xFF000000};  // Constructor with arguments
                                Pixel b{a};
*/
```

Constructor and compiler behavior

```
/*
        5. You have a default constructor, and a single or multi-argement non-copy constructor:
            . The compiler generates:
                . Copy constructor. You can do something like Pixel p1 = p2;
            . Things you can do:
                                Pixel a;              // Default constructor
                                Pixel b{0xFF000000};  // Constructor with arguments
                                Pixel c{a};           // Copy constructor
*/
```