Copy assignment operator

```
/*
      . Topics:
          . Continuing the journey on copy semantics
          . Setting up copy assignment operators (syntax)
          . The copy and swap idiom


*/
```

```
ct13::Pixel p1{0xFF00FF00, 100, 50};
ct13::Pixel p2{0x00FF00FF, 200, 150};

p2 = p1;
```

Copy assignment operator

```
export class Pixel {
 public:
     Pixel(uint32_t color, unsigned int x, unsigned int y);

     // Copy constructor
     Pixel(const Pixel& other);

     // Copy assignment operator
     Pixel& operator=(const Pixel& other);

     ~Pixel();


private:
    uint32_t m_color{0xFF000000};
    Position* m_pos{nullptr}; // Raw pointer for position
    std::unique_ptr<Position> m_smart_pos; // Smart pointer for position
};
```

Copy assignment operator

```cpp
// Copy assignment operator
Pixel& Pixel::operator=(const Pixel& other) {
    if (this == &other) {
        return *this; // Handle self-assignment
    }

    // Copy color
    m_color = other.m_color;

    // Deep copy the raw pointer (m_pos)
    if (m_pos != nullptr) {
        delete m_pos; // Clean up old memory
    }
    m_pos = new Position{*other.m_pos}; // Allocate new memory and copy

    // Deep copy the smart pointer (m_smart_pos)
    m_smart_pos = std::make_unique<Position>(*other.m_smart_pos); // Smart pointers handle their own memory

    fmt::print("Pixel copy-assigned (with both raw and smart pointers)\n");
    return *this;
}
```

Copy assignment operator: The copy and swap idiom

```cpp
// Copy assignment operator using copy-and-swap idiom
Pixel& Pixel::operator=(const Pixel& other) {
    Pixel temp(other); // Create a temporary copy
    swap(temp); // Swap the contents with the temporary copy
    fmt::print("Pixel copy-assigned (with both raw and smart pointers)\n");
    return *this;
}
```

The copy and swap idiom

```
/*
            . Here is how it works:
                . It creates a copy of the object to be assigned.
                . The hard work of memory allocation and deallocation is done in the copy constructor.
                . If something goes wrong during the copy, the original object is left unchanged.
                . Once the copy is successfully created, the swap function swaps the contents of the copy with the original object.
                . We return the original object, which now contains the copied data.
                . The old data is deleted when the copy goes out of scope.
            . We also set up a global swap function to swap the contents of two objects.
            . This global swap function helps the standard library algorithms like std::swap to work with our class.


*/
```

Compiler generated constructors: Update

```
/*
        . If have a user defined copy assignment operator, or destructor,
          the compiler won't generate a copy constructor for you. If you still
          need the compiler to generate a copy constructor, you can explicitly default it.
          eg: Pixel(const Pixel& other) = default;

        . If you have a user defined copy constructor or destructor, the compiler won't
           generate a copy assignment operator for you. If you still need the compiler
           to generate a copy assignment operator, you can explicitly default it.
          eg: Pixel& operator=(const Pixel& other) = default;

        . Think twice before you add a destructor to your class. If you don't need one, don't add one.
          The compiler generated destructor is usually good enough.It affects other constructors that
          the compiler could have generated for you

       . You can also explicitly delete the copy constructor and copy assignment operator
           if you don't want them .
          eg: Pixel(const Pixel& other) = delete;
              Pixel& operator=(const Pixel& other) = delete;


*/
```