Move Semantics

```
/*
      . Topics:
          . Stealing from temporary objects
          . What is a temporary
          . Taking addresses of temporaries and seeing what happens
          . rvalue references
          . lvalues and rvalues passed to functions
          . Add move semantics to the Canvas class
          . return values and copy ellision
          . const overloads


*/
```

Move Semantics: The basics

```
int x{ 5 };// x,y and z are all lvalues, they have a memory address we
int y{ 10 };//  can retrieve and use later on ,
int z{ 20 };// as long as the variables are in scope.

int *ptr = &x;

z = (x + y);
fmt::println("z : {}", z);
// fmt::println( "&(x+y) : {}" , (&(x+y)) );



double result = add(10.1, 20.2);// The result of add(10.1,20.2), is stored in some memory
                                // location for a short time, before it's assigned to result, and after it's copied
                                // into result, the memory is reclaimed by the system.
                                // add(10.1,20.2) is (evaluates to) an rvalue

// fmt::println( "address of add(10.1,20.2) : {}" , &(add(10.1,20.2)) );//Error
fmt::println("result is : {}", result);

// Grab the addresses for later use

// int * ptr1 = &(x + y); // Compiler error. The error clearly says what's wrong here
// int * ptr2 = &add(10.1,20.2); // Compiler error. Can only take address of an lvalue,
//  add(10.1 , 20.2) evaluates to an rvalue.
// int* ptr3 = &45; // Compiler error.
int *ptr4 = &x;// OK. x is an lvalue, so we can grab its address
```

Move Semantics: Rvalue References

```
int x{ 5 };
int y{ 10 };

int &&result = x + y;   // If a temporary is referenced by an rvalue reference,
                        // it's lifetime is extended to the lifetime of the reference.

double &&outcome = add(10.1, 20.2);

fmt::println("result : {}", result);
fmt::println("outcome : {}", outcome);
```

Move Semantics: Rvalue References

```
int x{ 5 };
int y{ 10 };

int &&result = x + y;   // If a temporary is referenced by an rvalue reference,
                        // it's lifetime is extended to the lifetime of the reference.

double &&outcome = add(10.1, 20.2);

fmt::println("result : {}", result);
fmt::println("outcome : {}", outcome);
```

Move Semantics: Rvalue and Lvalue overloads

```cpp
//Two functions, one takes an lvalue reference, the other takes an rvalue reference
void print_position(Position &pos){
    fmt::print("Position(lvalue ref): x = {}, y = {}\n", pos.x, pos.y);
}

void print_position(Position &&pos){
    fmt::print("Position(rvalue ref): x = {}, y = {}\n", pos.x, pos.y);
}

int main(){
    print_position(Position{ 10, 20 }); // The rvalue overload is called
}
```

Move Semantics: Rvalue and Lvalue overloads

```cpp
//Two functions, one takes an lvalue reference, the other takes an rvalue reference
void print_position(Position &pos){
    fmt::print("Position(lvalue ref): x = {}, y = {}\n", pos.x, pos.y);
}

void print_position(Position &&pos){
    fmt::print("Position(rvalue ref): x = {}, y = {}\n", pos.x, pos.y);
}

int main(){
    Position pos{ 30, 40 };
    print_position(pos); // The lvalue reference overload is called here
}
```

Move Semantics: Rvalue and Lvalue overloads

```cpp
void print_position(Position &&pos){
    fmt::print("Position(rvalue ref): x = {}, y = {}\n", pos.x, pos.y);
}

int main(){
    Position pos{ 30, 40 };
    print_position(pos); // Error!
}
```

Move Semantics: Rvalue and Lvalue overloads

```
void print_position(Position &&pos){
    fmt::print("Position(rvalue ref): x = {}, y = {}\n", pos.x, pos.y);
}

int main(){
    Position pos{ 30, 40 };
    print_position(std::move(pos)); // std::move casts an lvalue to an rvalue reference
                                    // it doesn't do any moving. Casting to an rvalue
                                    // helps the compiler find an overload taking an
                                    // rvalue reference, and if the object class supports
                                    // move semantics, we will move the data. We'll see
                                    // how to enable move semantics shortly.
}
```

Move Semantics: Tricky Rvalues

```cpp
/*
    . Inside the rvalue overload, we have to use std::move again, because as soon as
        an rvalue is given the parameter name, it becomes an lvalue.
*/
void print_position(Position &pos){
    fmt::print("Position(lvalue ref): x = {}, y = {}\n", pos.x, pos.y);
}

void normalize_position(Position&& pos){
    fmt::print("Position(rvalue ref): x = {}, y = {}\n", pos.x, pos.y);
}

void print_position(Position &&pos){
    fmt::print("Position(rvalue ref): x = {}, y = {}\n", pos.x, pos.y);
    //normalize_position(pos); //pos is named here, so it's an lvalue, we need to cast it to an rvalue reference
    //normalize_position(std::move(pos)); //pos is now an rvalue reference
}

int main(){
    Position pos{ 30, 40 };
    print_position(std::move(pos)); // The rvalue reference overload is called here
}
```

Adding move semantics to Canvas

```
//Adding a move constructor and a move assignment operator
export class Canvas {
public:
    Canvas(std::size_t width, std::size_t height);
    Canvas(const Canvas& src);
    ~Canvas();

    Canvas& operator=(const Canvas& rhs);

    //Add move semantics
    Canvas(Canvas&& other) noexcept; // Move constructor
    Canvas& operator=(Canvas&& other) noexcept;    // Move assignment operator

    void modify_pixel(std::size_t x, std::size_t y, const Pixel& pixel);
    Pixel& retrieve_pixel(std::size_t x, std::size_t y);

    void swap(Canvas& other) noexcept;

    void print() const;

private:
    void verify_coordinate(std::size_t x, std::size_t y) const;

    std::size_t m_width{ 0 };
    std::size_t m_height{ 0 };
    Pixel** m_pixels{ nullptr };
};
```

Adding move semantics to Canvas

```
//Move constructor
/*
Canvas::Canvas(Canvas&& other) noexcept
    : m_width{other.m_width}, m_height{other.m_height}, m_pixels{other.m_pixels} {
    other.m_width = 0;
    other.m_height = 0;
    other.m_pixels = nullptr;
}
*/

//Move constructor in terms of std::exchange
/*
Canvas::Canvas(Canvas&& other) noexcept
    : m_width{std::exchange(other.m_width, 0)},
      m_height{std::exchange(other.m_height, 0)},
      m_pixels{std::exchange(other.m_pixels, nullptr)} {}
*/

//Move constructor in terms of the swap method
Canvas::Canvas(Canvas&& other) noexcept{
    fmt::println("Move constructor");
    swap(other); // Steal the data from other and let it die.
}
```

Adding move semantics to Canvas

```cpp
//Move assignment operator.
Canvas& Canvas::operator=(Canvas&& other) noexcept {
    if (this != &other) {
        // Free the existing resource
        for (std::size_t i = 0; i < m_width; ++i) {
            delete[] m_pixels[i];
        }
        delete[] m_pixels;

        // Move the data
        m_width = other.m_width;
        m_height = other.m_height;
        m_pixels = other.m_pixels;

        // Reset the other object
        other.m_width = 0;
        other.m_height = 0;
        other.m_pixels = nullptr;
    }
    return *this;
}
```

Adding move semantics to Canvas

```
//Move assignment operator through the swap method
Canvas& Canvas::operator=(Canvas&& other) noexcept {
    fmt::println("Move assignment operator");
    if (this == &other) {
        return *this;
    }
    swap(other);
    //Whatever the current object was pointing to is moved into the temporary
    //which should die after a while when it's no longer in use. The destructor
    //will release the memory
    return *this;
}
```

Adding move semantics to Canvas

```cpp
//Move assignment operator in terms of std::exchange
Canvas& Canvas::operator=(Canvas&& other) noexcept {
    if (this =/= &other) {
        // Free the existing resource
        for (std::size_t i = 0; i < m_width; ++i) {
            delete[] m_pixels[i];
        }
        delete[] m_pixels;

        // Move the data using std::exchange
        m_width = std::exchange(other.m_width, 0);
        m_height = std::exchange(other.m_height, 0);
        m_pixels = std::exchange(other.m_pixels, nullptr);
    }
    return *this;
}
```

Move Semantics: Testing out

```cpp
raw::Canvas make_canvas(std::size_t width, std::size_t height){
    return raw::Canvas(width, height);
}

int main(){

    std::vector<raw::Canvas> canvases;
    canvases.reserve(3);    // This reserves the capacity, not the size.

    for(size_t i{0} ; i < 3 ; ++i){
        fmt::println("Iteration: {}", i);
        canvases.push_back(make_canvas(10,10)); // temporaries returned from the function
                                                // are moved into the vector because now
                                                // we support move semantics. They could
                                                // have been copied otherwise.

        fmt::println("");
    }

    raw::Canvas c(10,10);
    c = make_canvas(20,20); // move semantics

    raw::Canvas c2 (30,30);
    c = c2;                 // copy semantics
}
```

Returning objects, move semantics and copy ellision

```
export raw::Canvas returning_objects_with_move_semantics(){

    //1. Mandatory Copy/Move Ellision C++17
    /*
    return raw::Canvas{10,10}; // nameless temporary returned: Mandatory Copy/Move Elision (C++17)
    */

    //2. Non mandatory Copy/Move Ellision: Named return value optimization
    /*
    raw::Canvas c1{10,10};
    return c1;
    */

    //3. std::move and its effects on return statements. Turns off ellision
    raw::Canvas c2{10,10};
    return std::move(c2);
}
```

Move semantics and const overloads

```
/*
    . One of the few places where passing by value is recommended.
    . You should mostly pass by const reference to avoid copies though
    . This only applies to cases where the parameter is copied anyway.
    . This also won't work on Polymprphic types where passing by value
        causes for the object to be sliced off.
*/

export class Scores{
    public:

        //Two overloads
        /*
        void set_scores(const std::vector<int>& scores){
            m_scores = scores;
        }
        void set_scores(std::vector<int>&& scores){
            m_scores = std::move(scores);
        }
        */

        //One function taing the vector by value
        void set_scores(std::vector<int> scores){
            m_scores = std::move(scores);
        }

    private:
        std::vector<int> m_scores;
};
```