Operator overloading

```
/*
        .Overload operator+ as a member
            . Inside the class
            . outside
        .Overload operator+ as a non-member
            . Inside
            . Outside

        .verloading the subscript operator for Point
            . Can only be a member

        .Subscript operator reading and writing

        .Subscript operator for collection types

        . Multi dimensional subscript operator

*/
```

operator+ as a member

```
op_overloading_1::Point p1(10, 10);
op_overloading_1::Point p2(20, 20);
op_overloading_1::Point p3{ p1 + p2 };// p1.operator+(p2)
p3.print_info();
```

operator+ as a member

```
//#1: Overload operator+ as a member: Inline implementation
export class Point
{
public:
Point(double x, double y) : m_x(x), m_y(y) {}

// Member
Point operator+(const Point& right_operand){
       return Point(this→m_x + right_operand.m_x ,
                    this→m_y + right_operand.m_y );
}

Point operator+(const Point &right_operand);

void print_info() { fmt::println("Point [ x: {}, y: {} ]", m_x, m_y); }

private:
double length() const;// Function to calculate distance from the point(0,0)

private:
double m_x{};
double m_y{};
};

//Implementation

double Point::length() const { return sqrt(pow(m_x - 0, 2) + pow(m_y - 0, 2) * 1.0); }
```

operator+ as a member

```
//#1: Overload operator+ as a member: Outside implementation
export class Point
{
public:
Point(double x, double y) : m_x(x), m_y(y) {}

Point operator+(const Point &right_operand);

void print_info() { fmt::println("Point [ x: {}, y: {} ]", m_x, m_y); }

private:
double length() const;// Function to calculate distance from the point(0,0)

private:
double m_x{};
double m_y{};
};

//Implementation

double Point::length() const { return sqrt(pow(m_x - 0, 2) + pow(m_y - 0, 2) * 1.0); }

Point Point::operator+(const Point &right_operand)
{
return Point(this->m_x + right_operand.m_x, this->m_y + right_operand.m_y);
}
```

operator+ as a non-member

```
//#2: Overload operator+ as a non_member
export class Point
{
friend Point operator+(const Point &left, const Point &right);
public:
Point(double x, double y) : m_x(x), m_y(y) {}

void print_info() { fmt::println("Point [ x : {}, y: {}s]", m_x, m_y); }

private:
double length() const;// Function to calculate distance from the point(0,0)

private:
double m_x{};
double m_y{};
};

// inline Point operator+(const Point &left, const Point &right)
// {
//   return Point(left.m_x + right.m_x, left.m_y + right.m_y);
// }

//Implemantations
double Point::length() const { return sqrt(pow(m_x - 0, 2) + pow(m_y - 0, 2) * 1.0); }

Point operator+(const Point& left, const Point& right){
        return Point(left.m_x + right.m_x , left.m_y + right.m_y );
}
```

overloading the subscript operator

```
op_overloading_3::Point p1(10, 20);
fmt::println( "p1.x : {}" , p1[0] ); // x coordinate : 10
fmt::println("p1.x : {}", p1.operator[](0));// x coordinate : 10
fmt::println("p1.y : {}", p1[1]);// y coordinate : 20
```

overloading the subscript operator

```
//#3: Overload the subscript operator for Point
 // It should be set up as a member function. If you don't do this, you will get a compiler error.
 export class Point
 {
 public:
 Point() = default;
 Point(double x, double y) : m_x(x), m_y(y) {}
 ~Point() = default;

 double operator[](size_t index) const
 {
     assert((index == 0) || (index == 1));
     if (index == 0) {
     return m_x;
     } else {
     return m_y;
     }
}

void print_info() { fmt::println("Point [ x : {}, y: {}] ", m_x, m_y); }

private:
double length() const;// Function to calculate distance from the point(0,0)

private:
double m_x{};// 0
double m_y{};// 1
};

//Implementation
double Point::length() const { return sqrt(pow(m_x - 0, 2) + pow(m_y - 0, 2) * 1.0); }
```

overloading the subscript operator

```
//#3: Overload the subscript operator for Point
 // It should be set up as a member function. If you don't do this, you will get a compiler error.
 export class Point
 {
 public:
 Point() = default;
 Point(double x, double y) : m_x(x), m_y(y) {}
 ~Point() = default;

 double operator[](size_t index) const
 {
     assert((index == 0) || (index == 1));
     if (index == 0) {
     return m_x;
     } else {
     return m_y;
     }
}

void print_info() { fmt::println("Point [ x : {}, y: {}] ", m_x, m_y); }

private:
double length() const;// Function to calculate distance from the point(0,0)

private:
double m_x{};// 0
double m_y{};// 1
};

//Implementation
double Point::length() const { return sqrt(pow(m_x - 0, 2) + pow(m_y - 0, 2) * 1.0); }
```

operator[] for reading and writing

```
//#4: Overload the subscript operator for reading and writting
export class Point
{
public:
Point(double x, double y) : m_x(x), m_y(y) {}

double &operator[](size_t index)
{
    assert((index == 0) || (index == 1));
    return (index == 0) ? m_x : m_y;
}

void print_info() { fmt::println("Point [ x : {}, y: {}] ", m_x, m_y); }

private:
double length() const;// Function to calculate distance from the point(0,0)

private:
double m_x{};// 0
double m_y{};// 1
};

//Implementations
double Point::length() const { return sqrt(pow(m_x - 0, 2) + pow(m_y - 0, 2) * 1.0); }
```

operator[] for a collection

```
export class Scores
{
public:
Scores() = delete;
Scores(const std::string &course_name) : m_course_name{ course_name } {}

//These two implementations allow us to use the subscript operator,
//both const and non-const objects
double &operator[](size_t index);
double operator[](size_t index) const;


void print_info() const
{
    fmt::print("{} : [", m_course_name);
    for (size_t i{}; i < 20; ++i) { fmt::print("{}  ", m_scores[i]); }
    fmt::println("]");
}

private:
std::string m_course_name;
double m_scores[20]{};
};
```

multi-dimensional operator[] (C++23)

```
op_overloading_6::Array3d v;
v[3, 2, 1] = 42;
fmt::print("v[3,2,1] = {}\n", v[3, 2, 1]);
```

multi-dimensional operator[] (C++23)

```
//Multi dimensional array subscript operator (Works on Visual C++ and Clang 18 at the time of this writting)
export class Array3d {
private:
    static constexpr int X = 2;// Width
    static constexpr int Y = 3;// Depth
    static constexpr int Z = 4;// Height
    int m[X * Y * Z]{};

public:
    // Multidimensional subscript operator for accessing elements (C++23)
    int& operator[](std::size_t z, std::size_t y, std::size_t x) {
        assert(x < X && y < Y && z < Z);
        return m[z * Y * X + y * X + x]; // Row major order. This is how the elements are stored in memory.
                                         // This formula is learnt in your Linear Algebra class.
    }
};
```