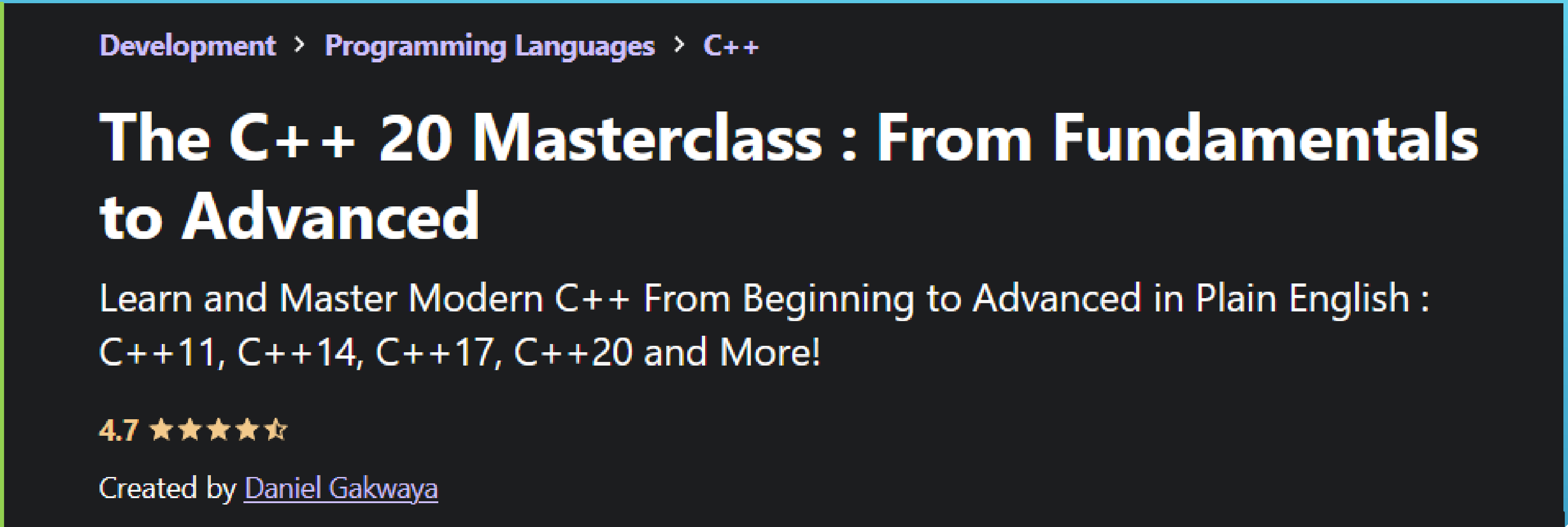
Slides



Development › Programming Languages › C++

The C++ 20 Masterclass : From Fundamentals to Advanced

Learn and Master Modern C++ From Beginning to Advanced in Plain English : C++11, C++14, C++17, C++20 and More!

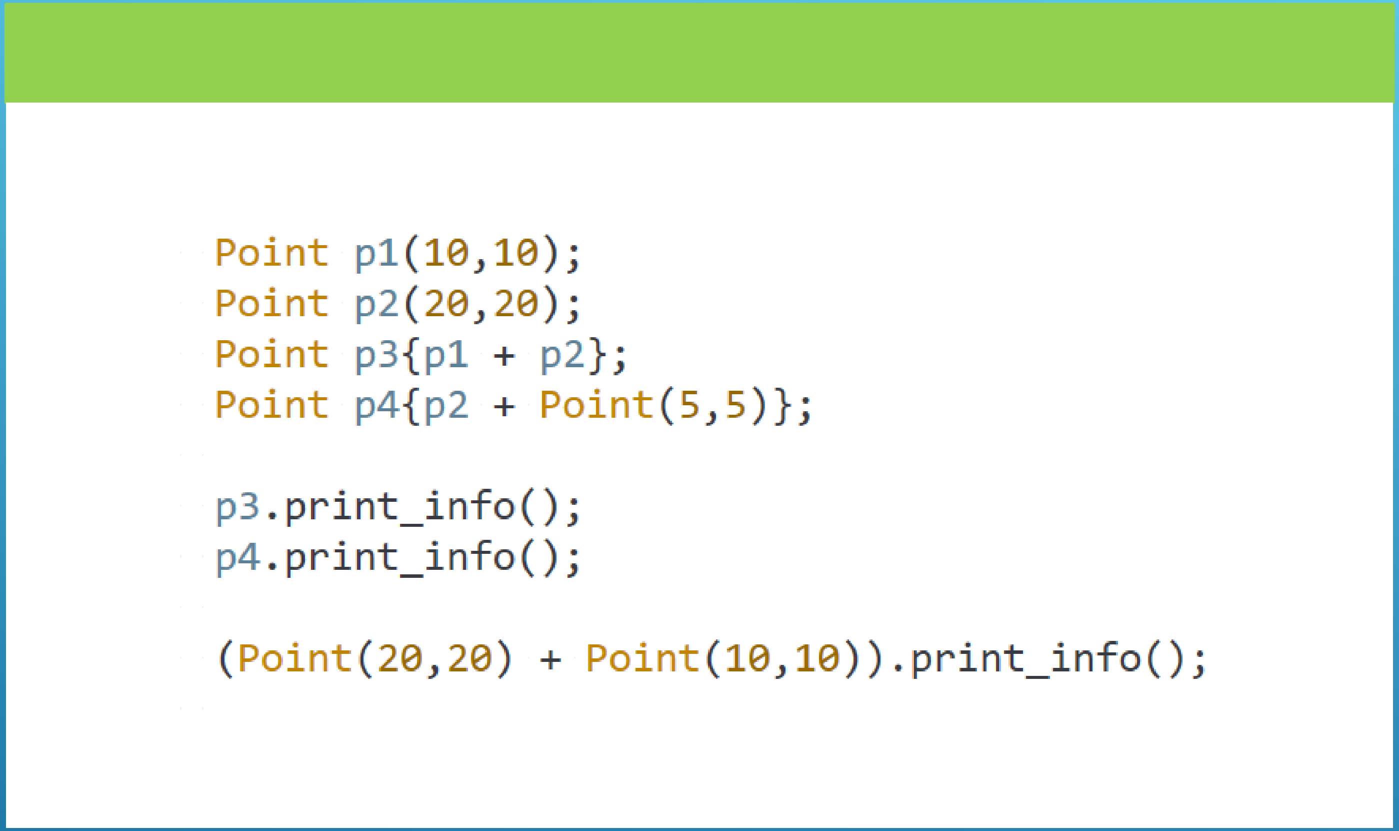
4.7 ★★★★⯪

Created by Daniel Gakwaya

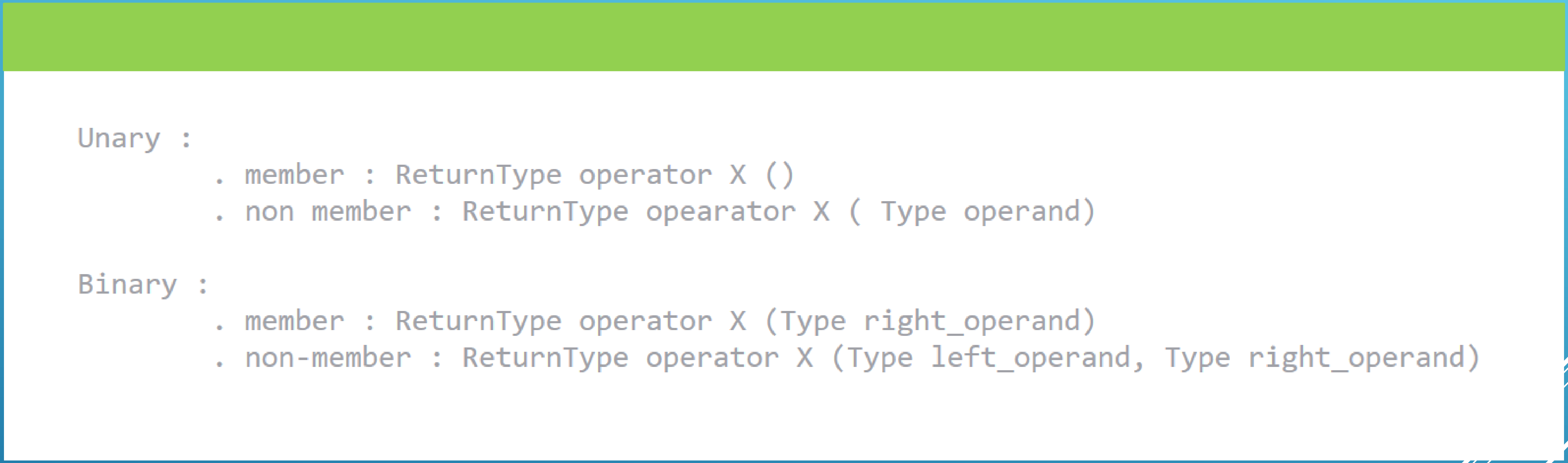
Section : Operator Overloading

Slide intentionally left empty

Operator overloading



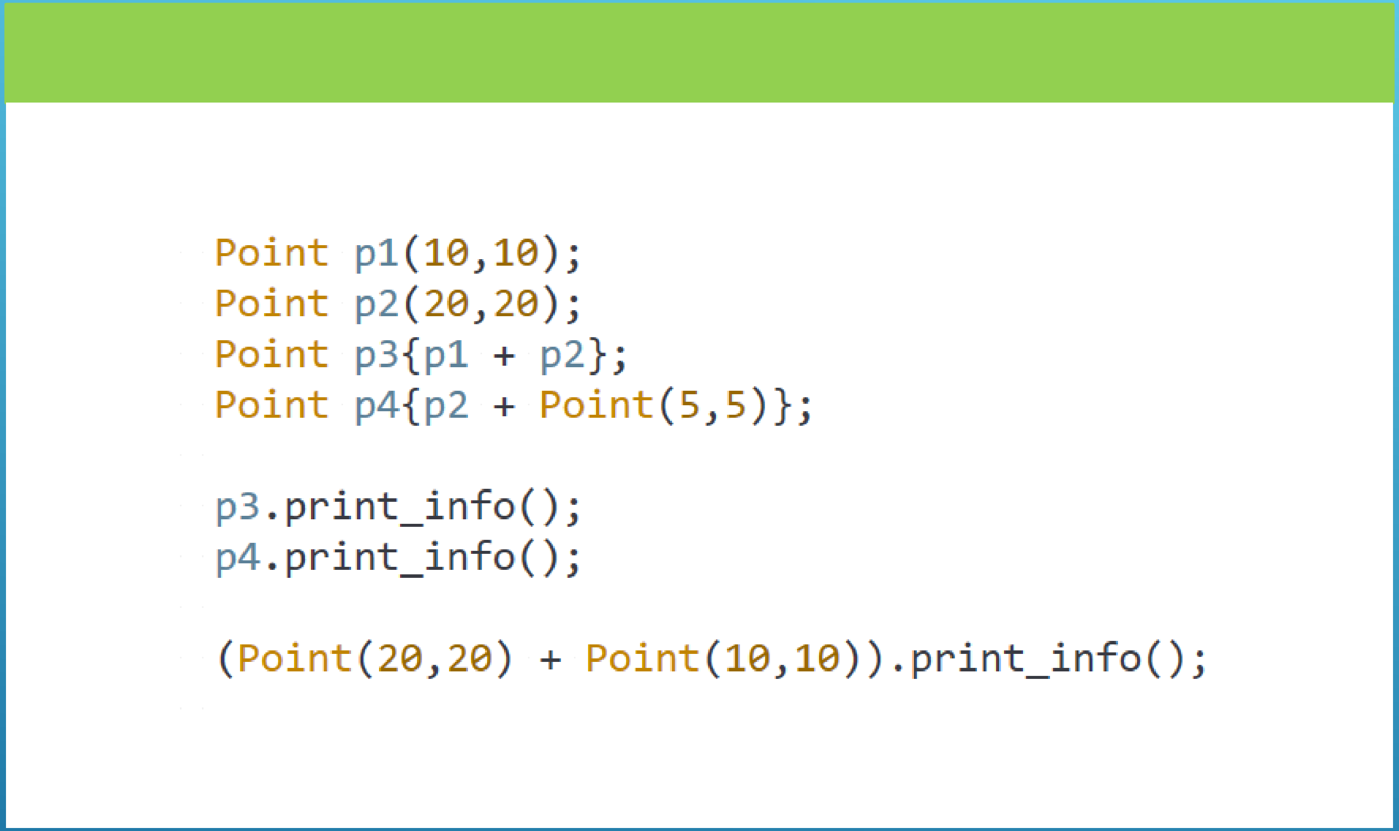
```
Point p1(10,10);
Point p2(20,20);
Point p3{p1 + p2};
Point p4{p2 + Point(5,5)};

p3.print_info();
p4.print_info();

(Point(20,20) + Point(10,10)).print_info();
```



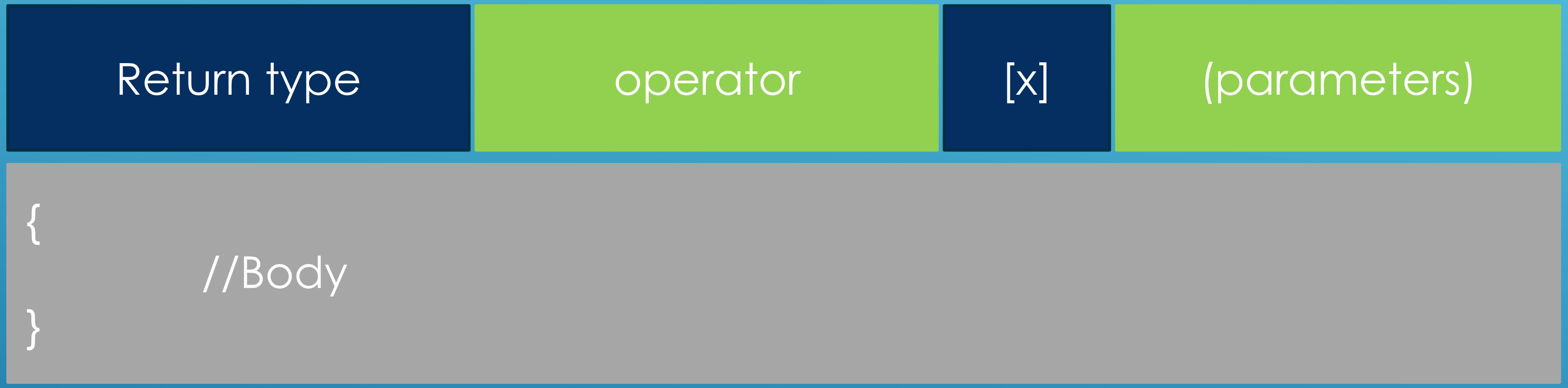
```
Unary :
        . member : ReturnType operator X ()
        . non member : ReturnType opearator X ( Type operand)

Binary :
        . member : ReturnType operator X (Type right_operand)
        . non-member : ReturnType operator X (Type left_operand, Type right_operand)
```

Slide intentionally left empty

Operator+ (as member)

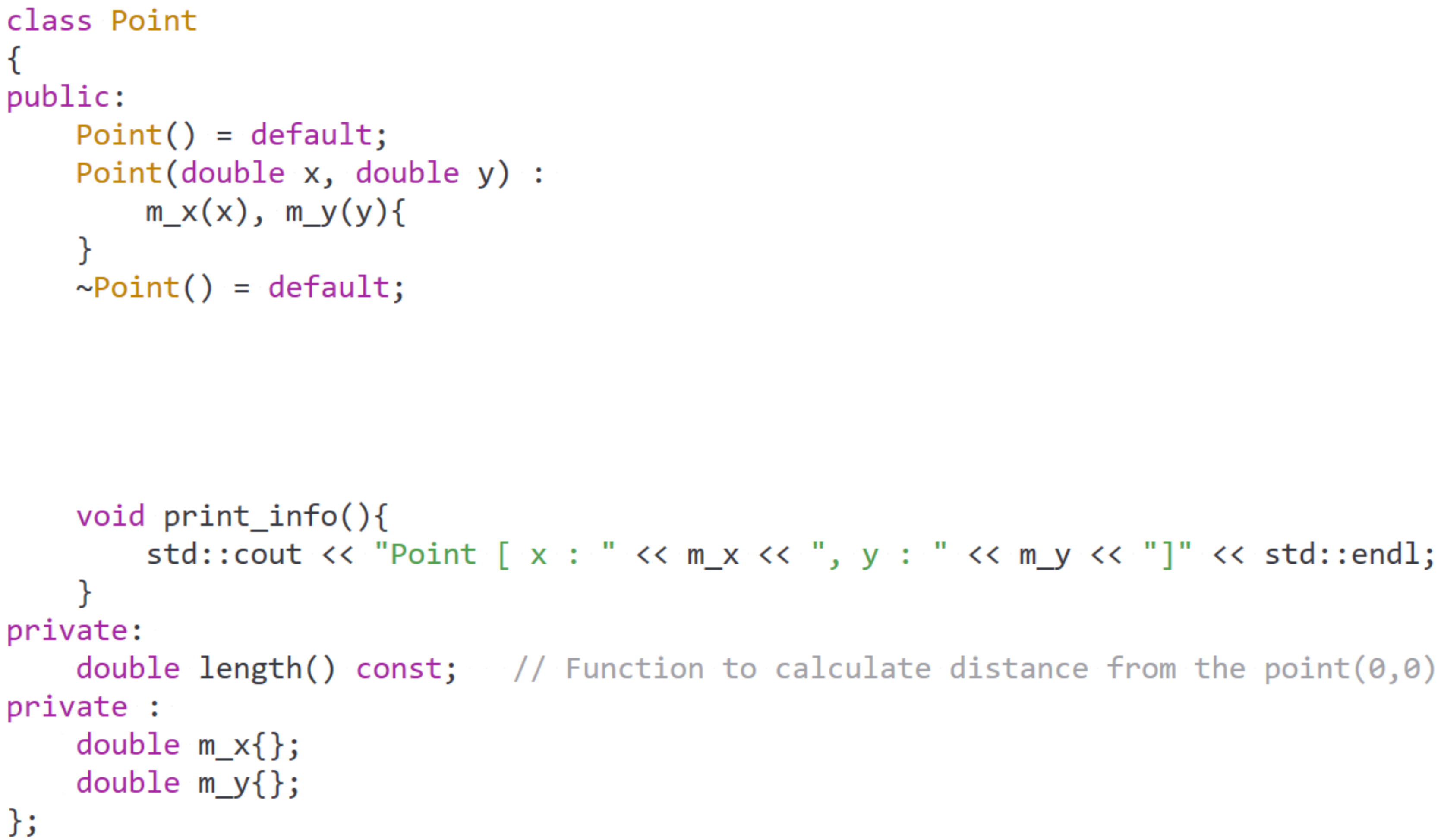


```
Point p1(10,10);
Point p2(20,20);
Point p3{p1 + p2};
Point p4{p2 + Point(5,5)};

p3.print_info();
p4.print_info();

(Point(20,20) + Point(10,10)).print_info();
```

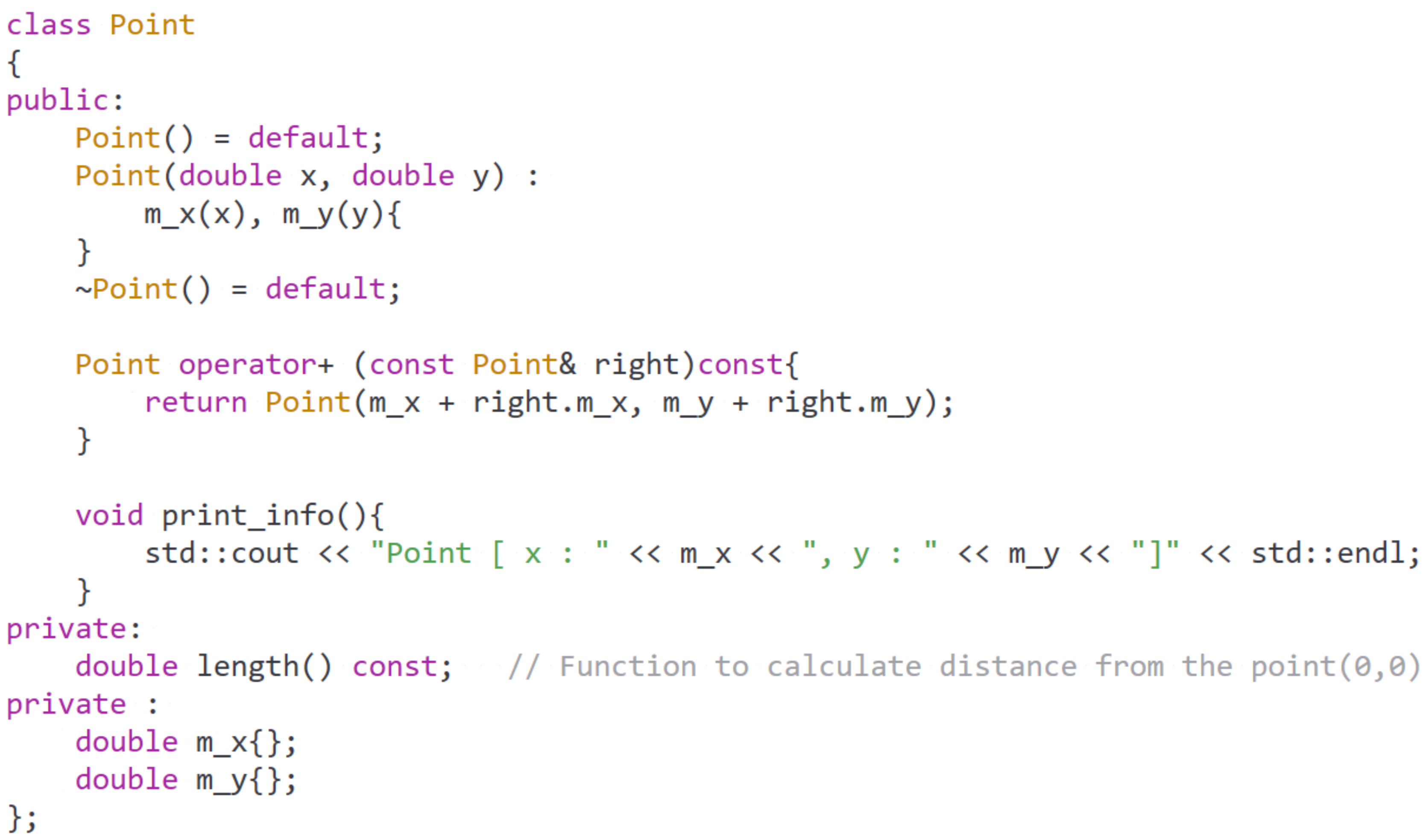


Return type | operator | [x] | (parameters)

```
{
        //Body
}
```



```cpp
class Point
{
public:
    Point() = default;
    Point(double x, double y) :
        m_x(x), m_y(y){
    }
    ~Point() = default;





    void print_info(){
        std::cout << "Point [ x : " << m_x << ", y : " << m_y << "]" << std::endl;
    }
private:
    double length() const;   // Function to calculate distance from the point(0,0)
private :
    double m_x{};
    double m_y{};
};
```



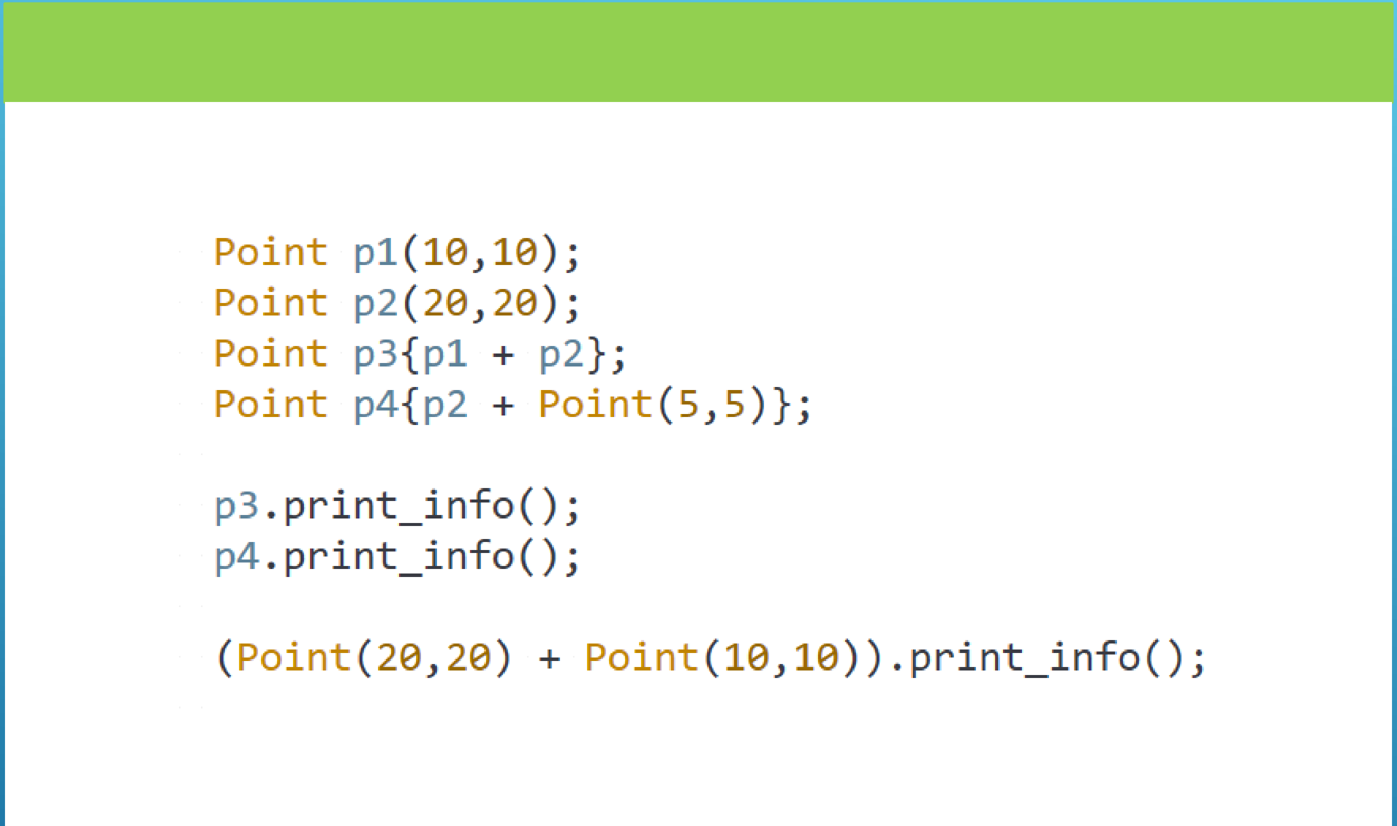
```cpp
class Point
{
public:
    Point() = default;
    Point(double x, double y) :
        m_x(x), m_y(y){
    }
    ~Point() = default;

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

    void print_info(){
        std::cout << "Point [ x : " << m_x << ", y : " << m_y << "]" << std::endl;
    }
private:
    double length() const;   // Function to calculate distance from the point(0,0)
private :
    double m_x{};
    double m_y{};
};
```

```
p1 + p2
```

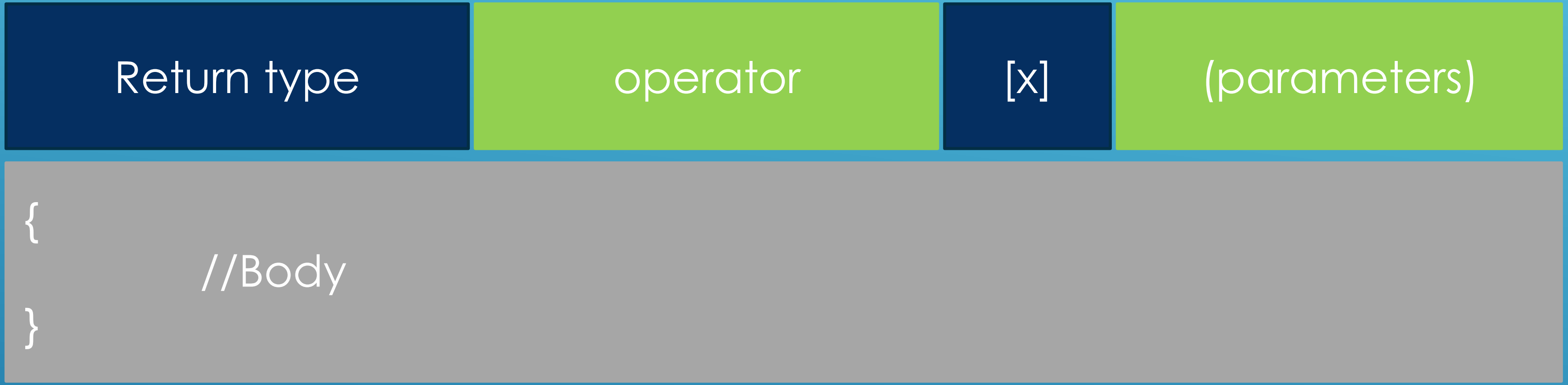
```
p1.operator+(p2)
```

Slide intentionally left empty

Operator+ (as NON member)

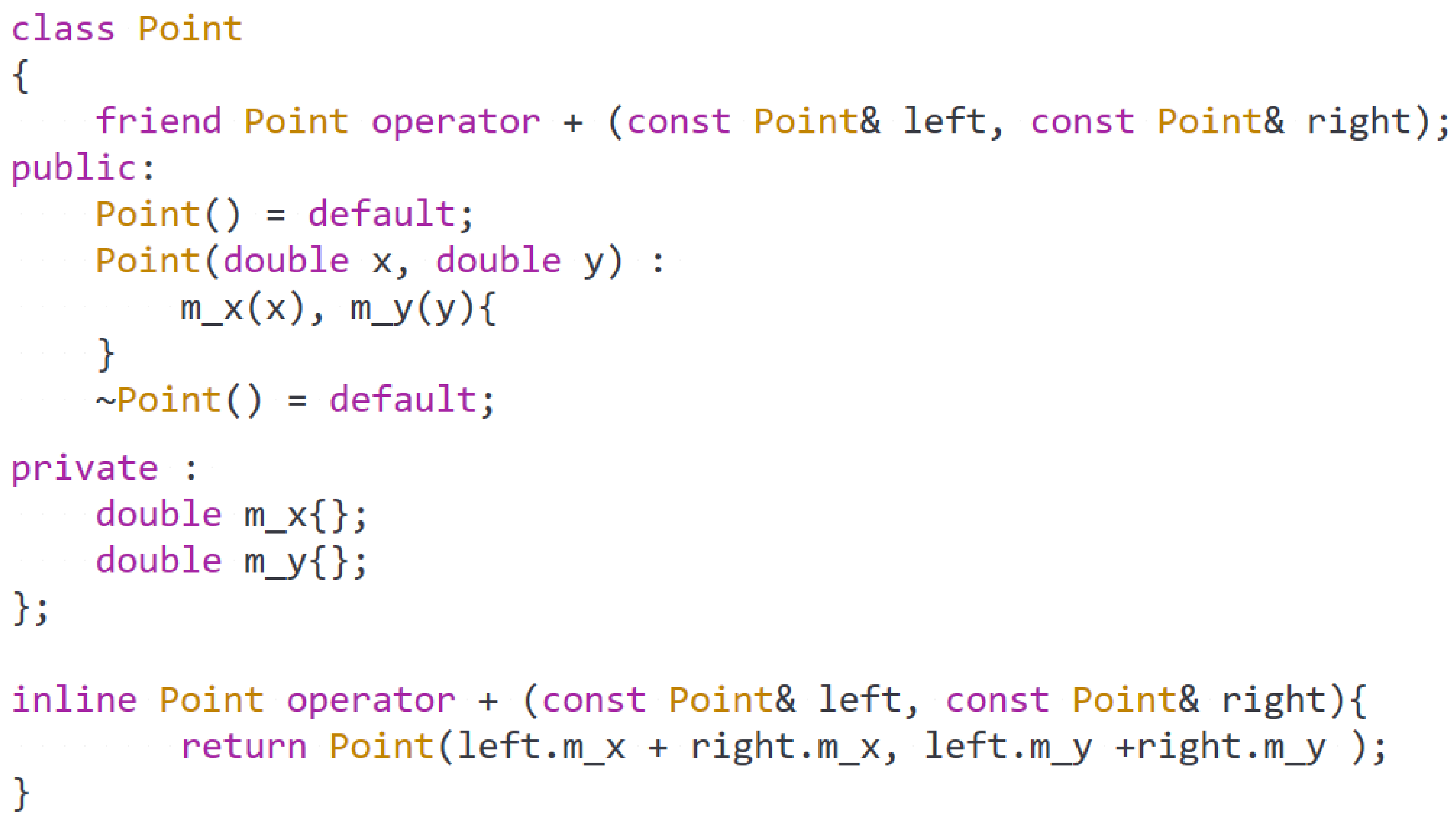


```
Point p1(10,10);
Point p2(20,20);
Point p3{p1 + p2};
Point p4{p2 + Point(5,5)};

p3.print_info();
p4.print_info();

(Point(20,20) + Point(10,10)).print_info();
```



Return type operator [x] (parameters)

```
{
        //Body
}
```



```
class Point
{
    friend Point operator + (const Point& left, const Point& right);
public:
    Point() = default;
    Point(double x, double y) :
        m_x(x), m_y(y){
    }
    ~Point() = default;

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 );
}
```

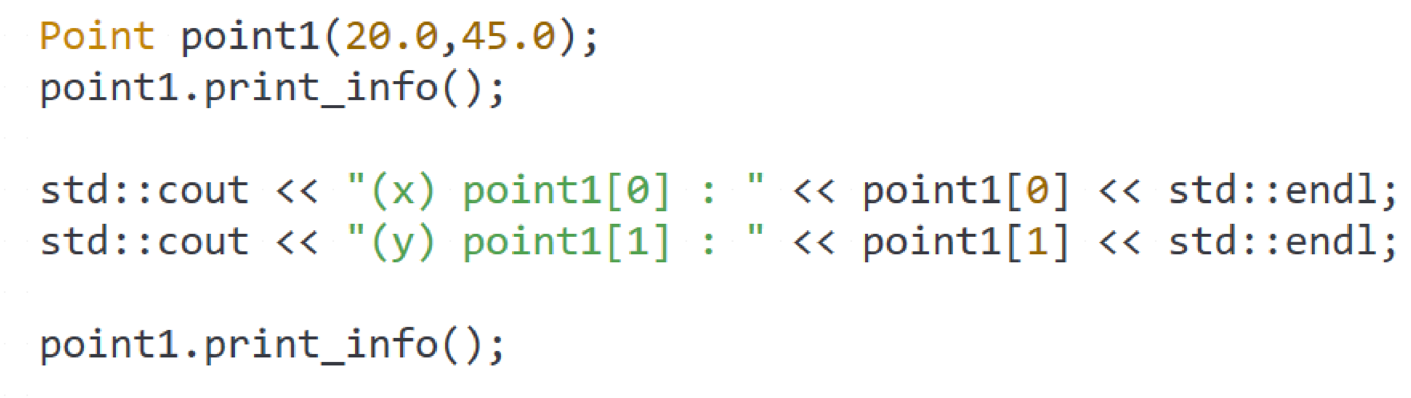
p1 + p2

operator+(p1,p2)

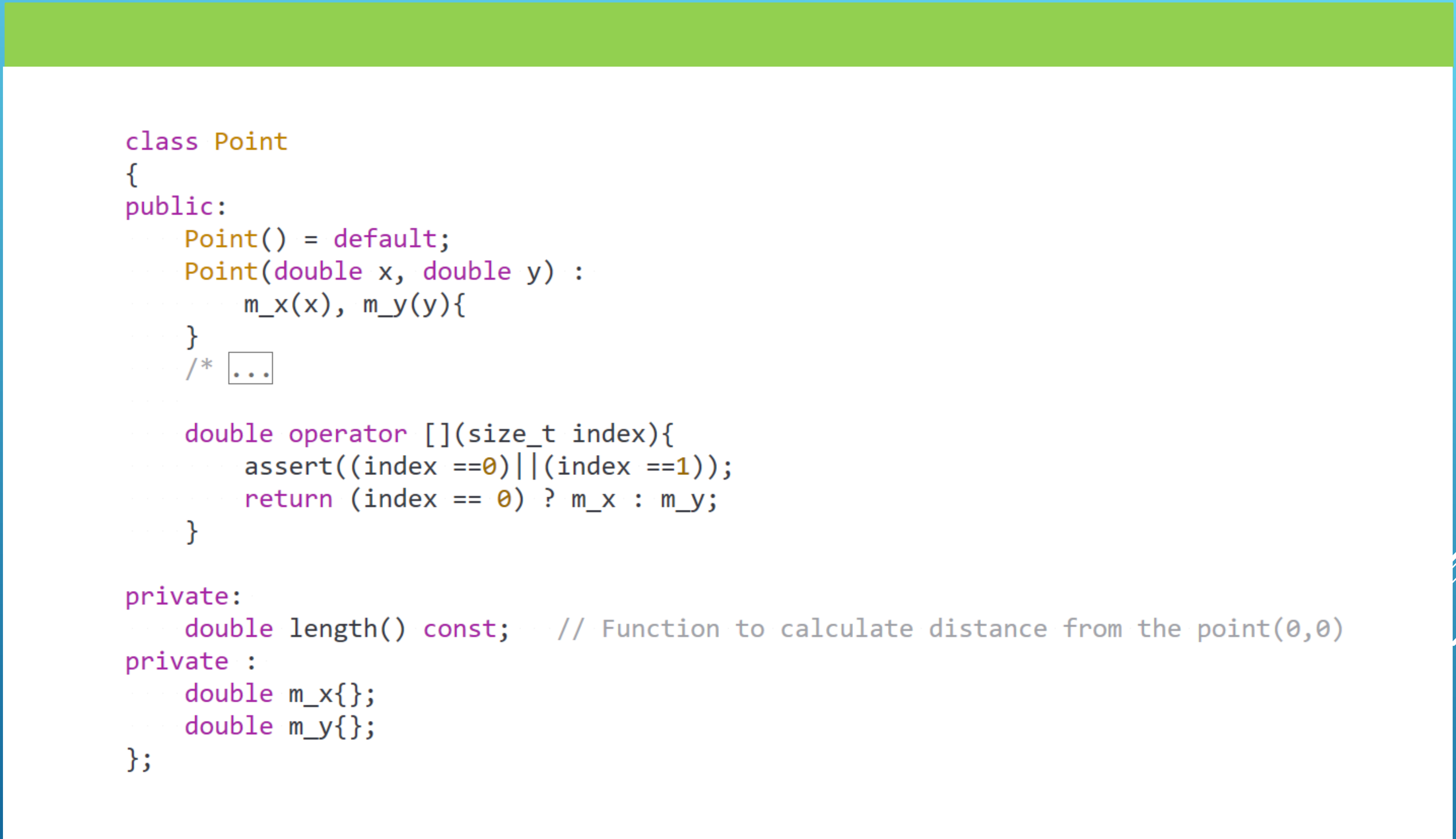
Slide intentionally left empty

Overloading the subscript operator for reading

```
int value = scores[5]
```



```
Point point1(20.0,45.0);
point1.print_info();

std::cout << "(x) point1[0] : " << point1[0] << std::endl;
std::cout << "(y) point1[1] : " << point1[1] << std::endl;

point1.print_info();
```



```cpp
class Point
{
public:
    Point() = default;
    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;
    }

private:
    double length() const;   // Function to calculate distance from the point(0,0)
private :
    double m_x{};
    double m_y{};
};
```

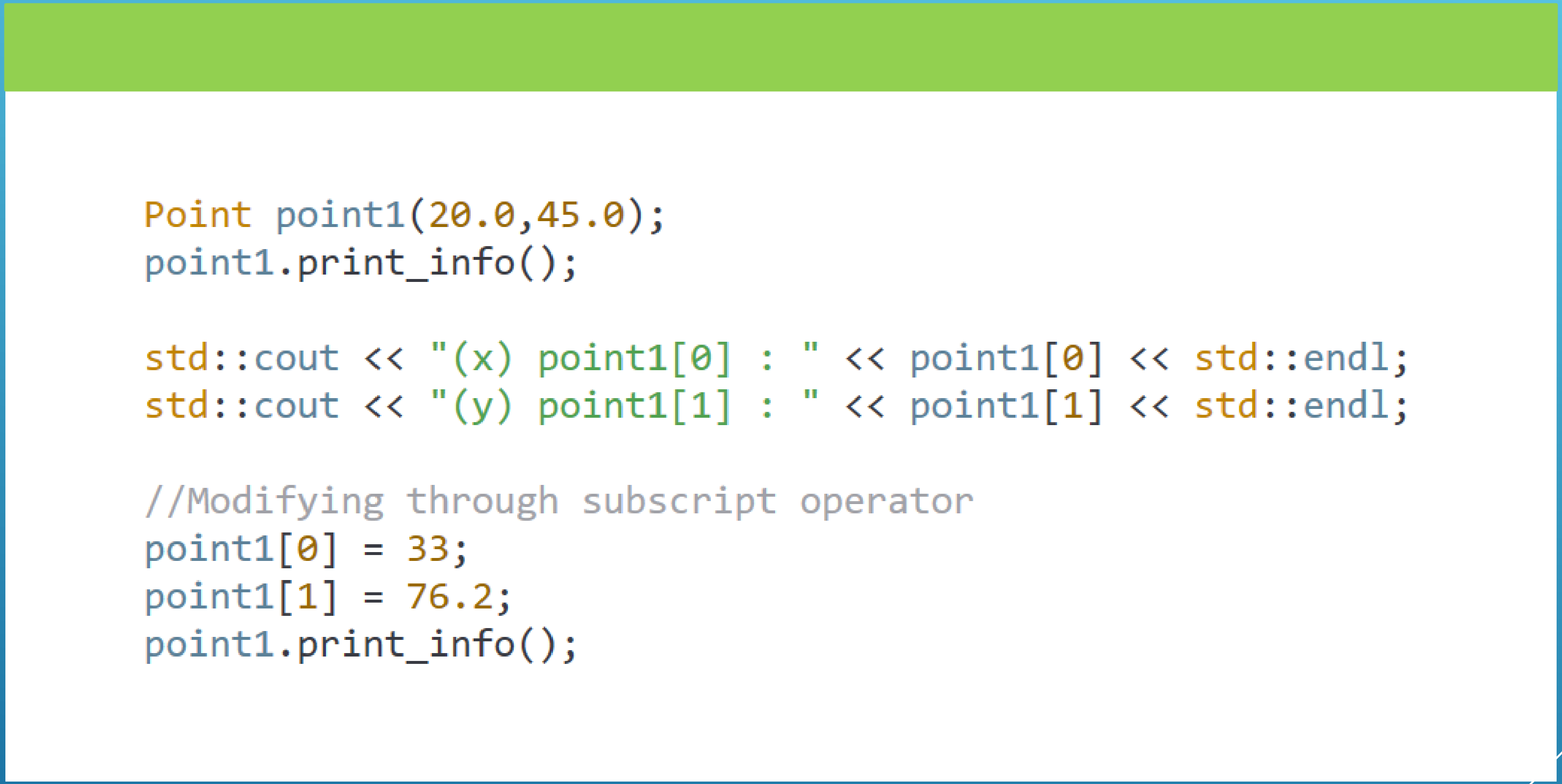
- The subscript operator is a binary operator
- It is one of the operators that MUST be set up as a member function

point1 [0]

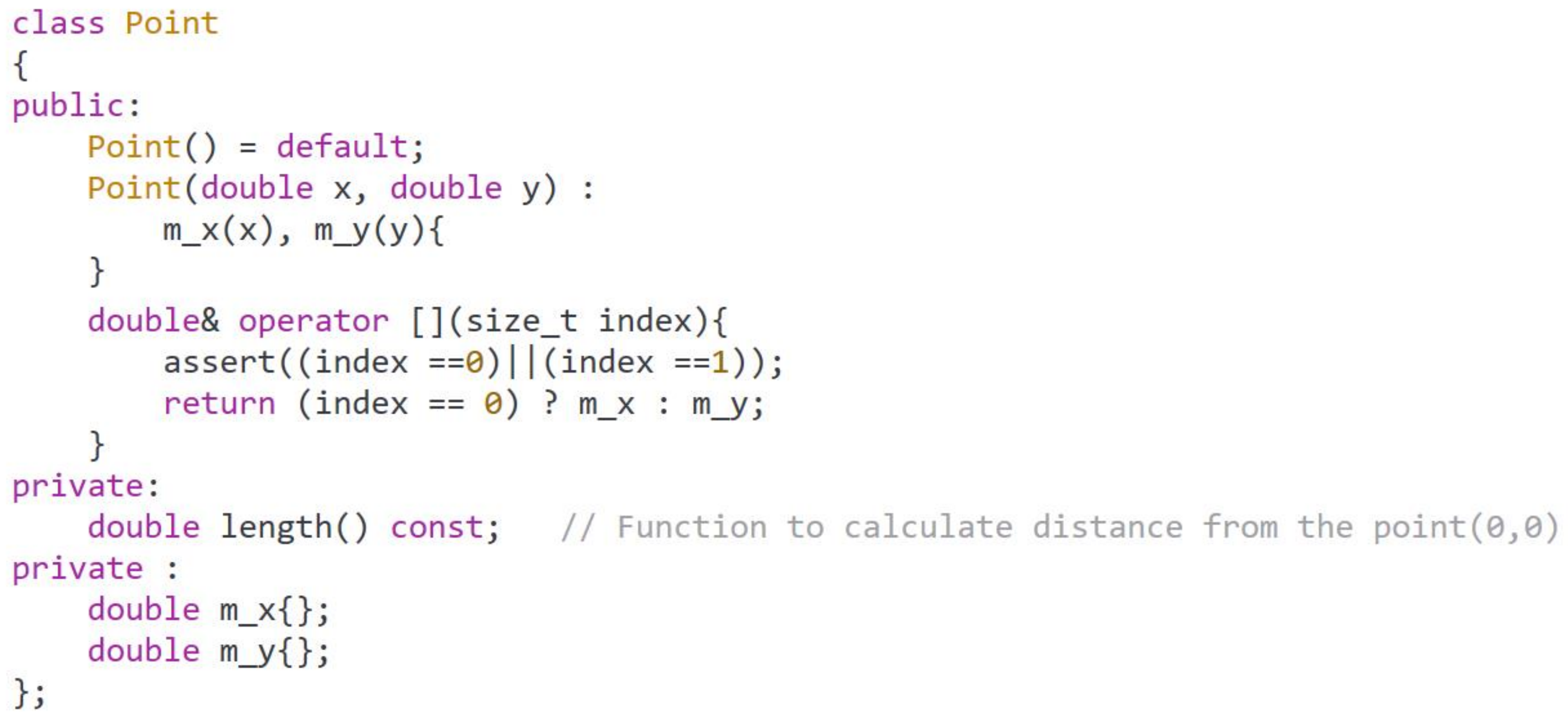
point1.operator[] (0)

Slide intentionally left empty

Overloading the subscript operator [Read Write]



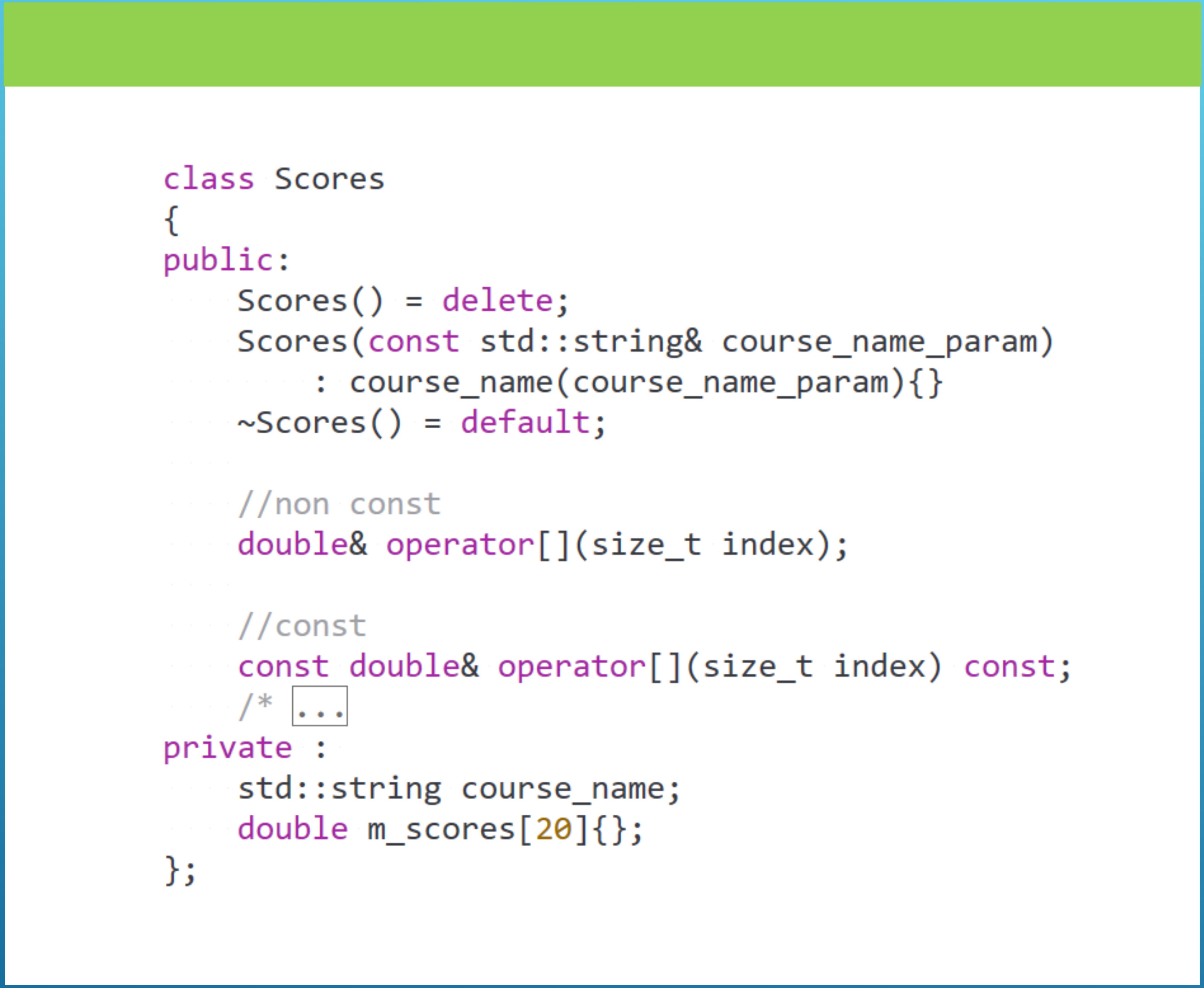
```cpp
Point point1(20.0,45.0);
point1.print_info();

std::cout << "(x) point1[0] : " << point1[0] << std::endl;
std::cout << "(y) point1[1] : " << point1[1] << std::endl;

//Modifying through subscript operator
point1[0] = 33;
point1[1] = 76.2;
point1.print_info();
```



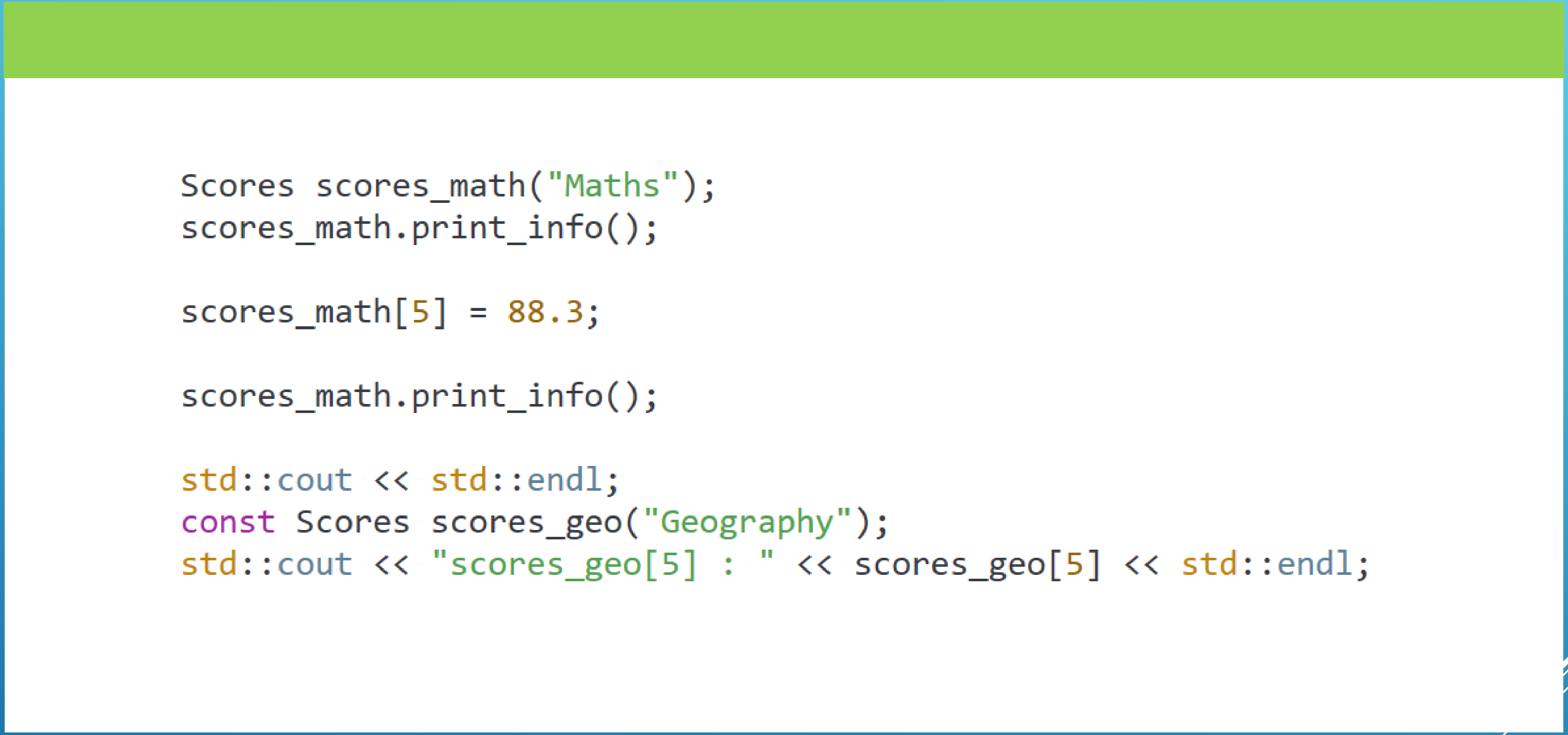
```
class Point
{
public:
    Point() = default;
    Point(double x, double y) :
        m_x(x), m_y(y){
    }
    double& operator [](size_t index){
        assert((index ==0)||(index ==1));
        return (index == 0) ? m_x : m_y;
    }
private:
    double length() const;   // Function to calculate distance from the point(0,0)
private :
    double m_x{};
    double m_y{};
};
```

Slide intentionally left empty

Subscript operator for collection types



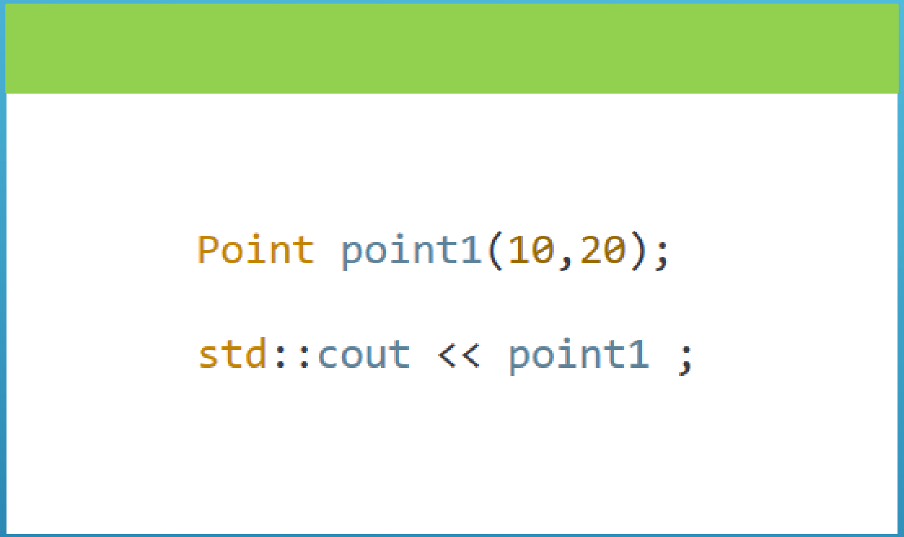
```cpp
class Scores
{
public:
    Scores() = delete;
    Scores(const std::string& course_name_param)
        : course_name(course_name_param){}
    ~Scores() = default;

    //non const
    double& operator[](size_t index);

    //const
    const double& operator[](size_t index) const;
    /* ...
private :
    std::string course_name;
    double m_scores[20]{};
};
```



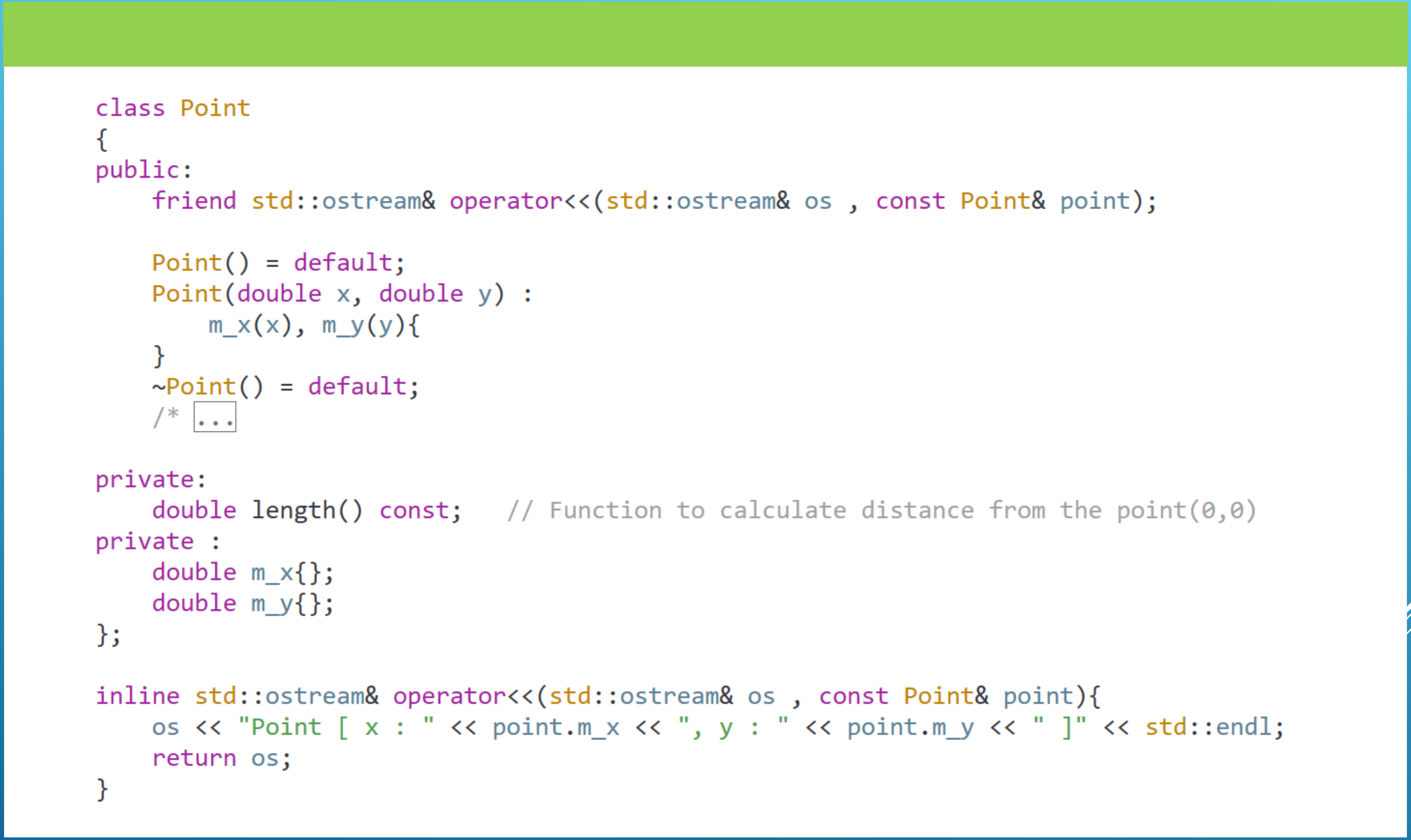
```cpp
Scores scores_math("Maths");
scores_math.print_info();

scores_math[5] = 88.3;

scores_math.print_info();

std::cout << std::endl;
const Scores scores_geo("Geography");
std::cout << "scores_geo[5] : " << scores_geo[5] << std::endl;
```

Slide intentionally left empty

Stream insertion operator

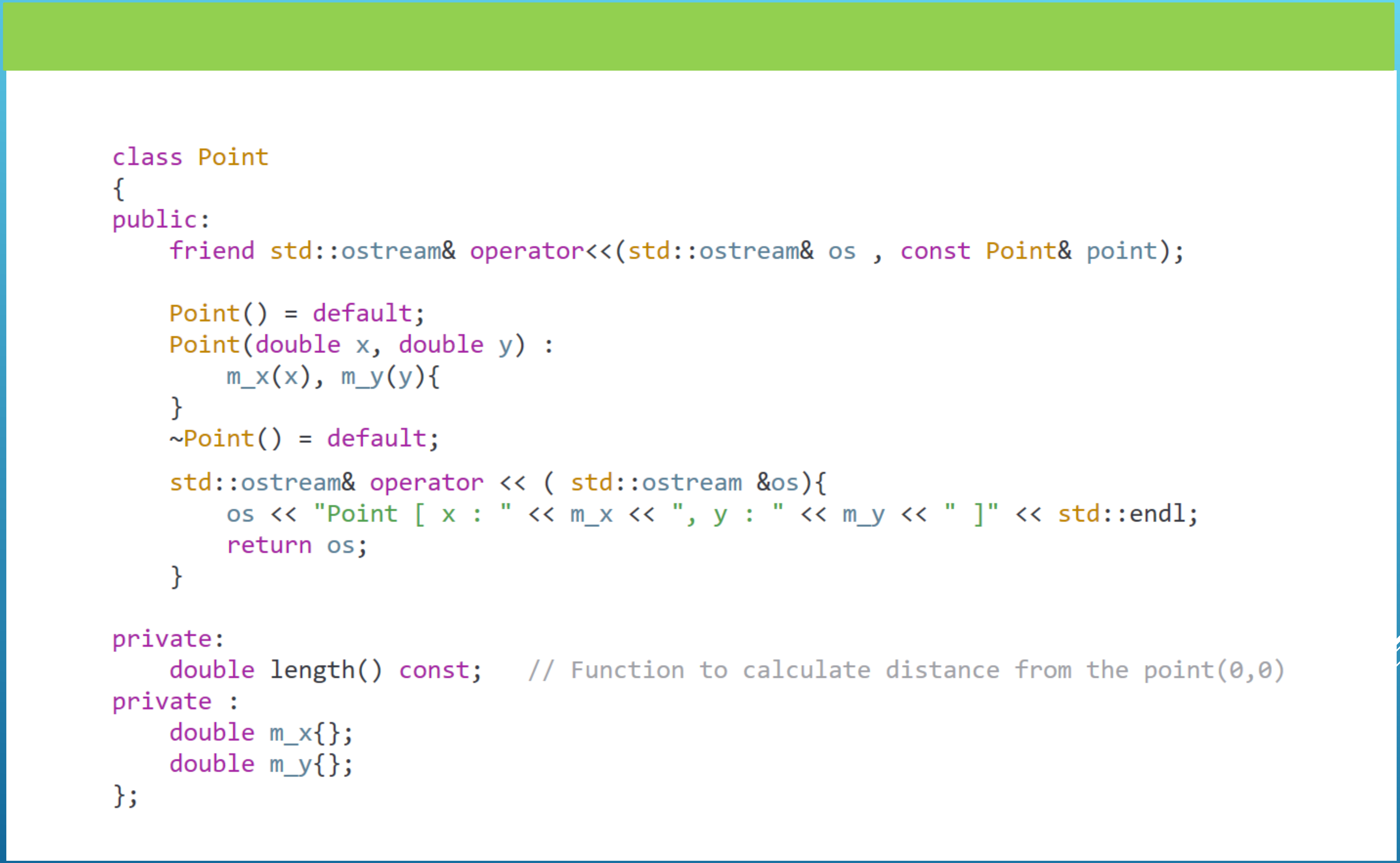


```cpp
Point point1(10,20);

std::cout << point1 ;
```

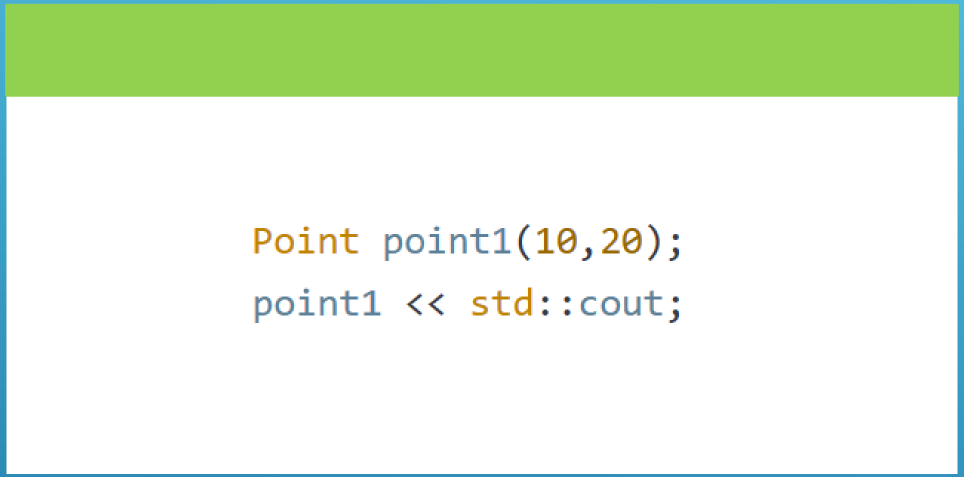


```cpp
class Point
{
public:
    friend std::ostream& operator<<(std::ostream& os , const Point& point);

    Point() = default;
    Point(double x, double y) :
        m_x(x), m_y(y){
    }
    ~Point() = default;
    /* ...

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

inline std::ostream& operator<<(std::ostream& os , const Point& point){
    os << "Point [ x : " << point.m_x << ", y : " << point.m_y << " ]" << std::endl;
    return os;
}
```

Operator << as member function



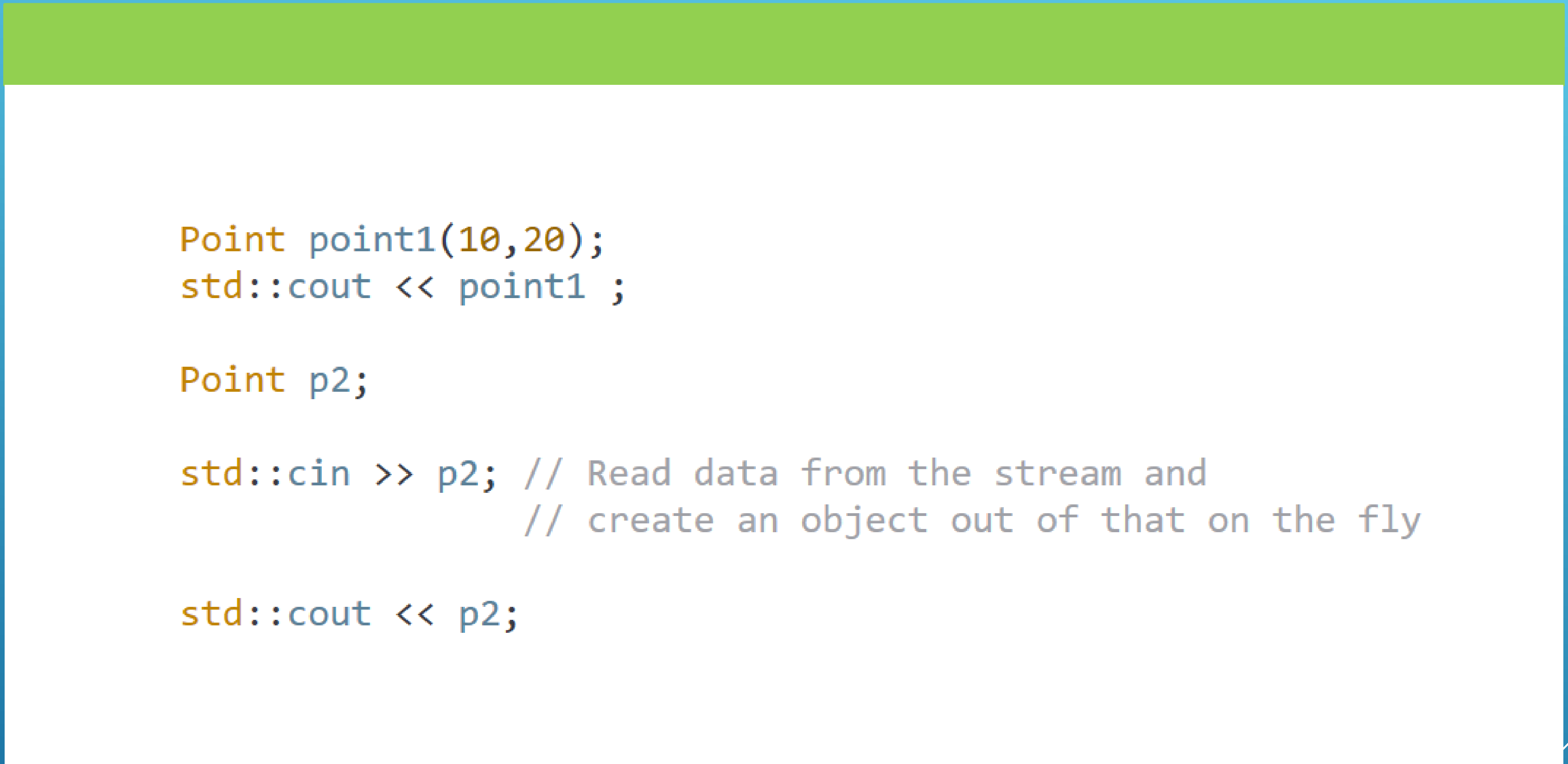
```cpp
class Point
{
public:
    friend std::ostream& operator<<(std::ostream& os , const Point& point);

    Point() = default;
    Point(double x, double y) :
        m_x(x), m_y(y){
    }
    ~Point() = default;
    std::ostream& operator << ( std::ostream &os){
        os << "Point [ x : " << m_x << ", y : " << m_y << " ]" << std::endl;
        return os;
    }

private:
    double length() const;   // Function to calculate distance from the point(0,0)
private :
    double m_x{};
    double m_y{};
};
```



```
Point point1(10,20);
point1 << std::cout;
```

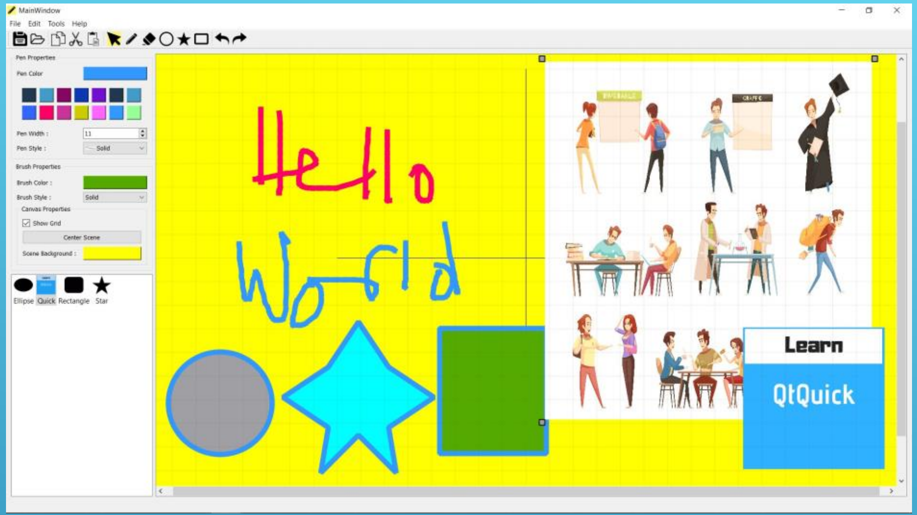
Slide intentionally left empty

Stream extraction operator



```cpp
Point point1(10,20);
std::cout << point1 ;

Point p2;

std::cin >> p2; // Read data from the stream and
                // create an object out of that on the fly

std::cout << p2;
```

```
// stream insertion <<
inline std::ostream& operator<<(std::ostream& os , const Point& point){
    os << "Point [ x : " << point.m_x << ", y : " << point.m_y << " ]" << std::endl;
    return os;
}

// stream extraction
inline std::istream& operator>>(std::istream& is ,  Point& point){
    double x;
    double y;

    std::cout << "Please type in the coordinates for the point" << std::endl;
    std::cout << "order [x,y], separated by spaces : ";

    is >> x >> y ;
    point.m_x = x;
    point.m_y = y;

    return is;
}
```

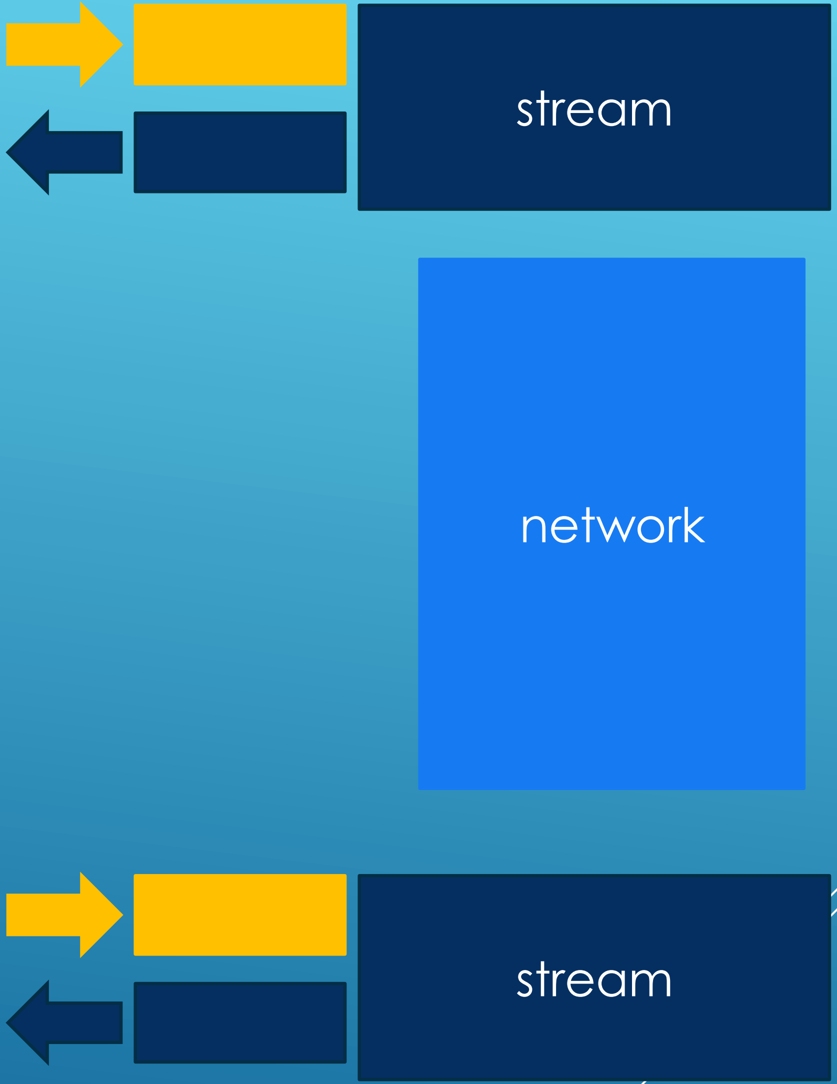


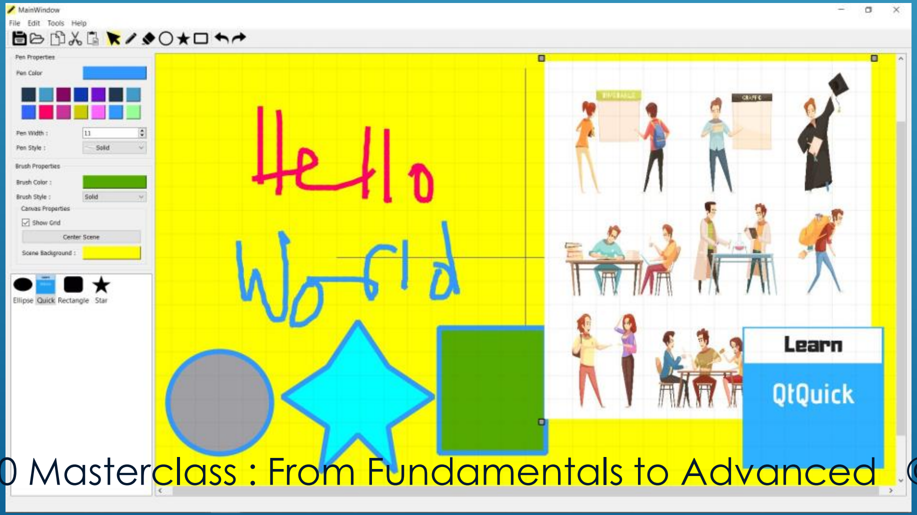
Pen Width : 11

Pen Style : Solid

Brush Style : Solid

☑ Show Grid





Pen Width : 11

Pen Style : Solid

Brush Style : Solid

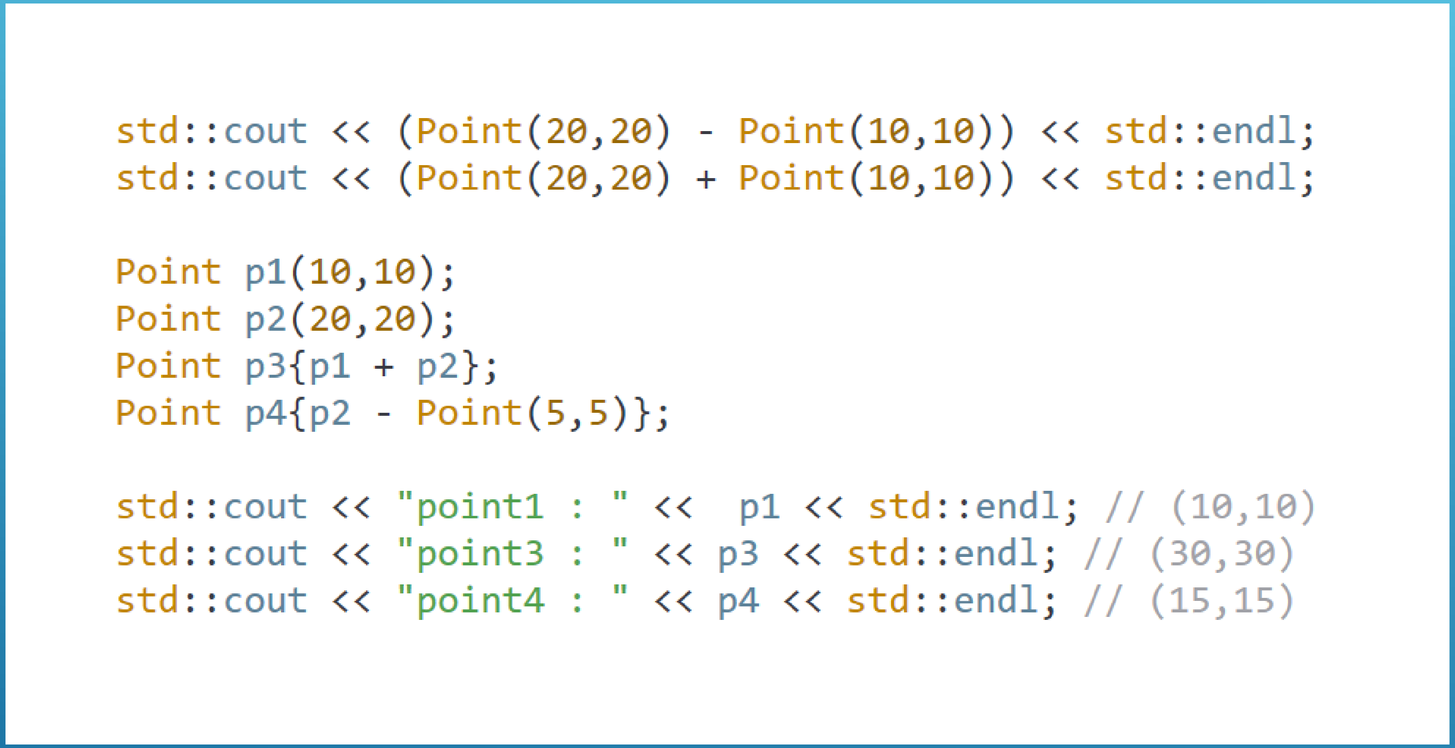
☑ Show Grid

Slide intentionally left empty

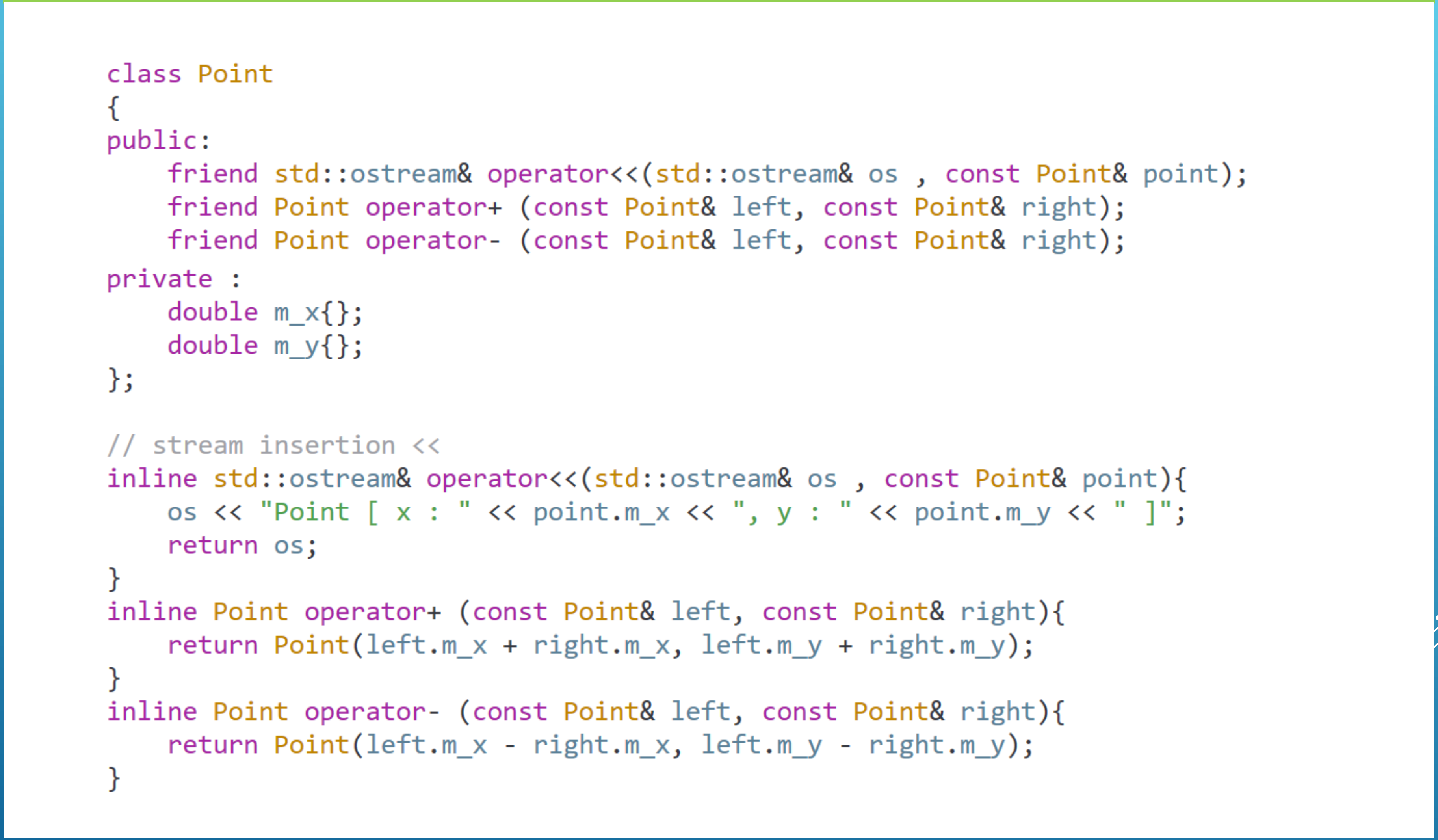
Arithmetic Operators

Just because you can overload an operator doesn’t mean you should

Point : Arithmetic Operators



```
std::cout << (Point(20,20) - Point(10,10)) << std::endl;
std::cout << (Point(20,20) + Point(10,10)) << std::endl;

Point p1(10,10);
Point p2(20,20);
Point p3{p1 + p2};
Point p4{p2 - Point(5,5)};

std::cout << "point1 : " <<  p1 << std::endl; // (10,10)
std::cout << "point3 : " << p3 << std::endl; // (30,30)
std::cout << "point4 : " << p4 << std::endl; // (15,15)
```



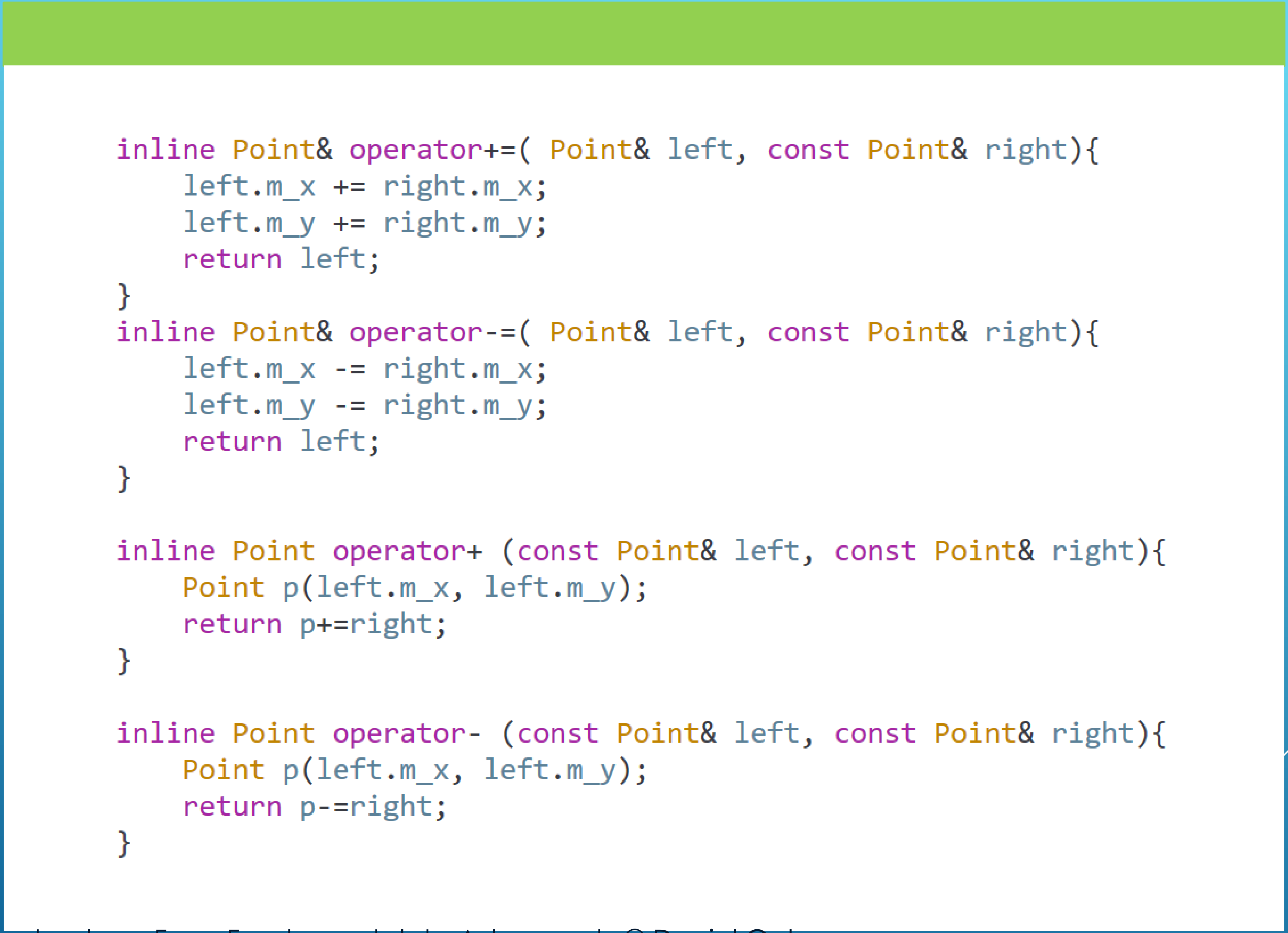
```cpp
class Point
{
public:
    friend std::ostream& operator<<(std::ostream& os , const Point& point);
    friend Point operator+ (const Point& left, const Point& right);
    friend Point operator- (const Point& left, const Point& right);
private :
    double m_x{};
    double m_y{};
};

// stream insertion <<
inline std::ostream& operator<<(std::ostream& os , const Point& point){
    os << "Point [ x : " << point.m_x << ", y : " << point.m_y << " ]";
    return os;
}
inline Point operator+ (const Point& left, const Point& right){
    return Point(left.m_x + right.m_x, left.m_y + right.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);
}
```

Slide intentionally left empty

Compound operators - Reusing Operators

Point : Arithmetic Operators

+ can be implemented in terms of +=

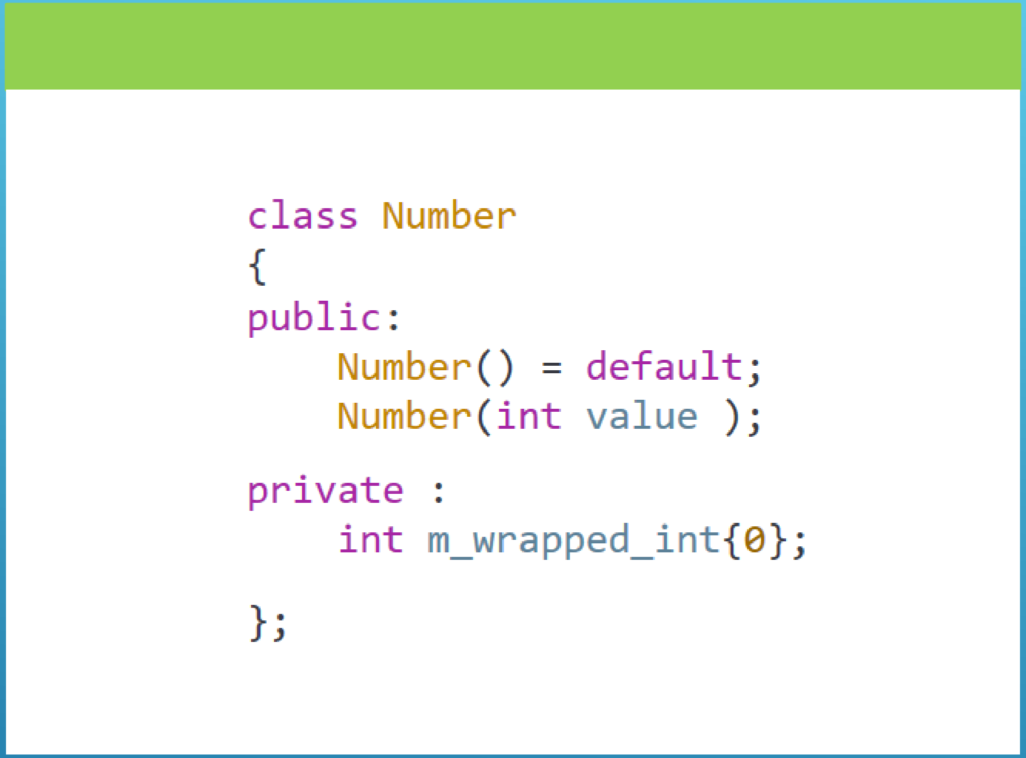


```cpp
inline Point& operator+=( Point& left, const Point& right){
    left.m_x += right.m_x;
    left.m_y += right.m_y;
    return left;
}
inline Point& operator-=( Point& left, const Point& right){
    left.m_x -= right.m_x;
    left.m_y -= right.m_y;
    return left;
}

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

inline Point operator- (const Point& left, const Point& right){
    Point p(left.m_x, left.m_y);
    return p-=right;
}
```

Slide intentionally left empty

Custom Type Conversions

Set up the transformations you want to support from your type to other types

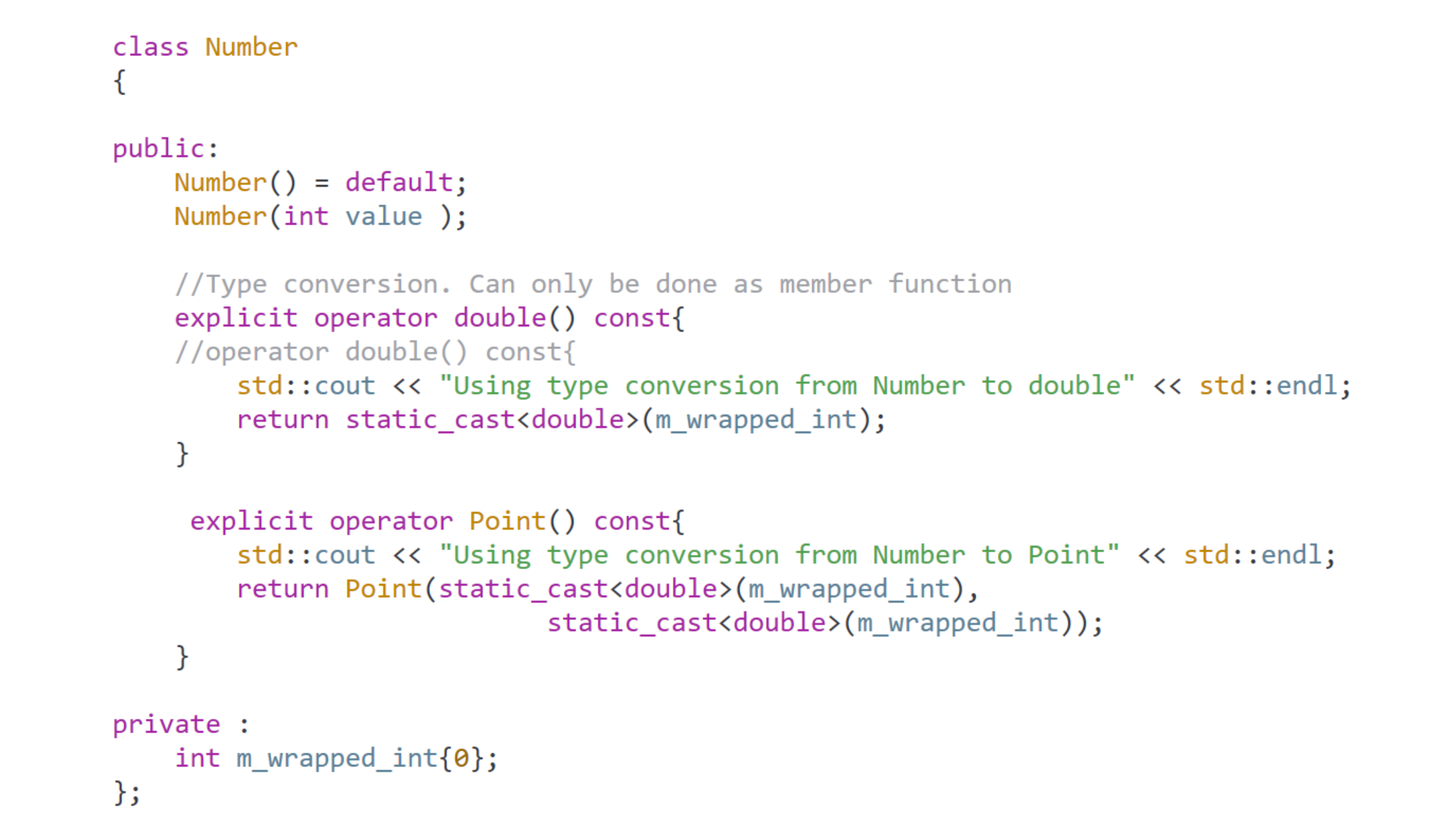


```cpp
class Number
{
public:
    Number() = default;
    Number(int value );

private :
    int m_wrapped_int{0};

};
```

Conversion :

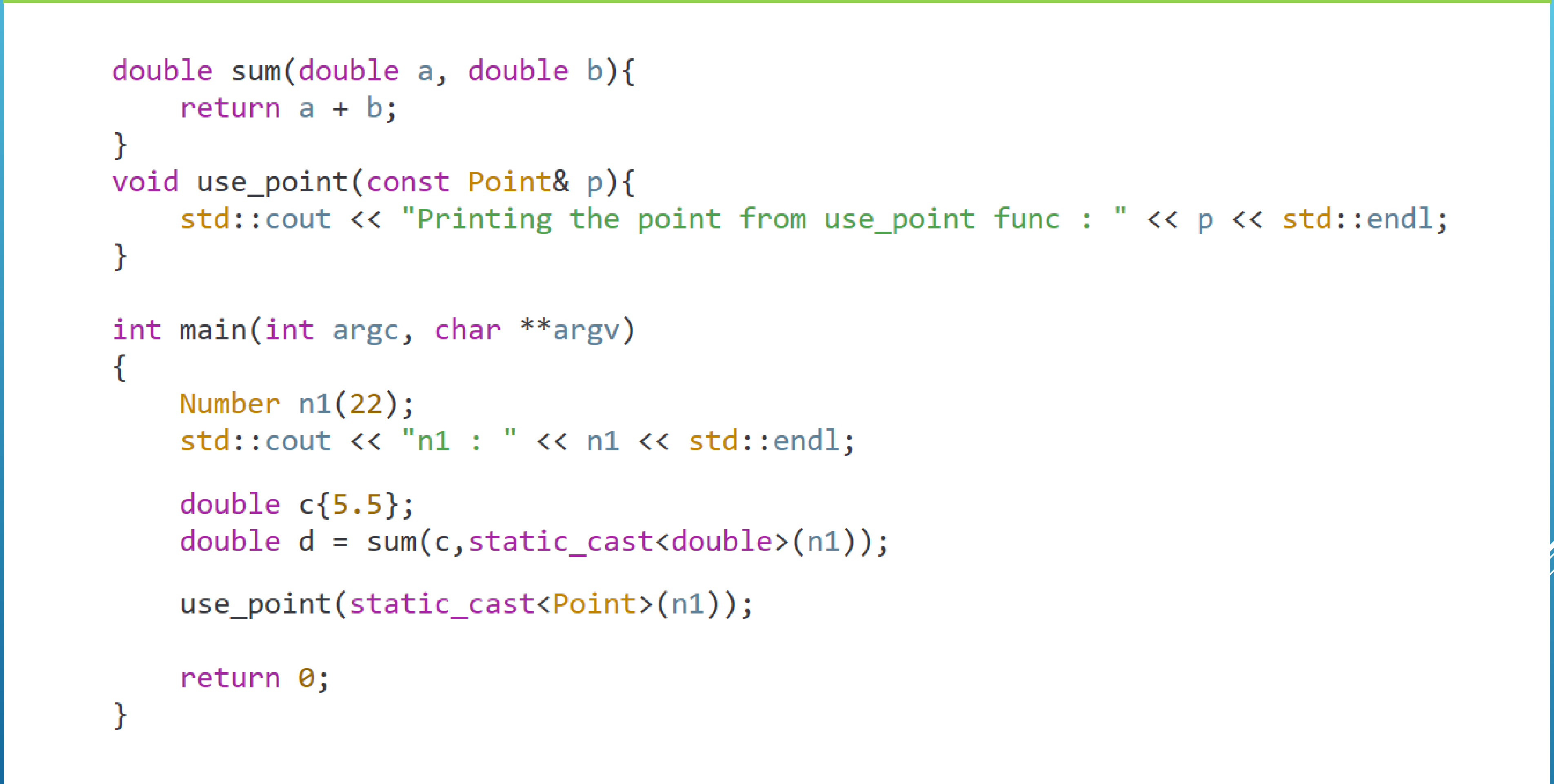
- From Number to double
- From Number to Point

Setting up type conversions



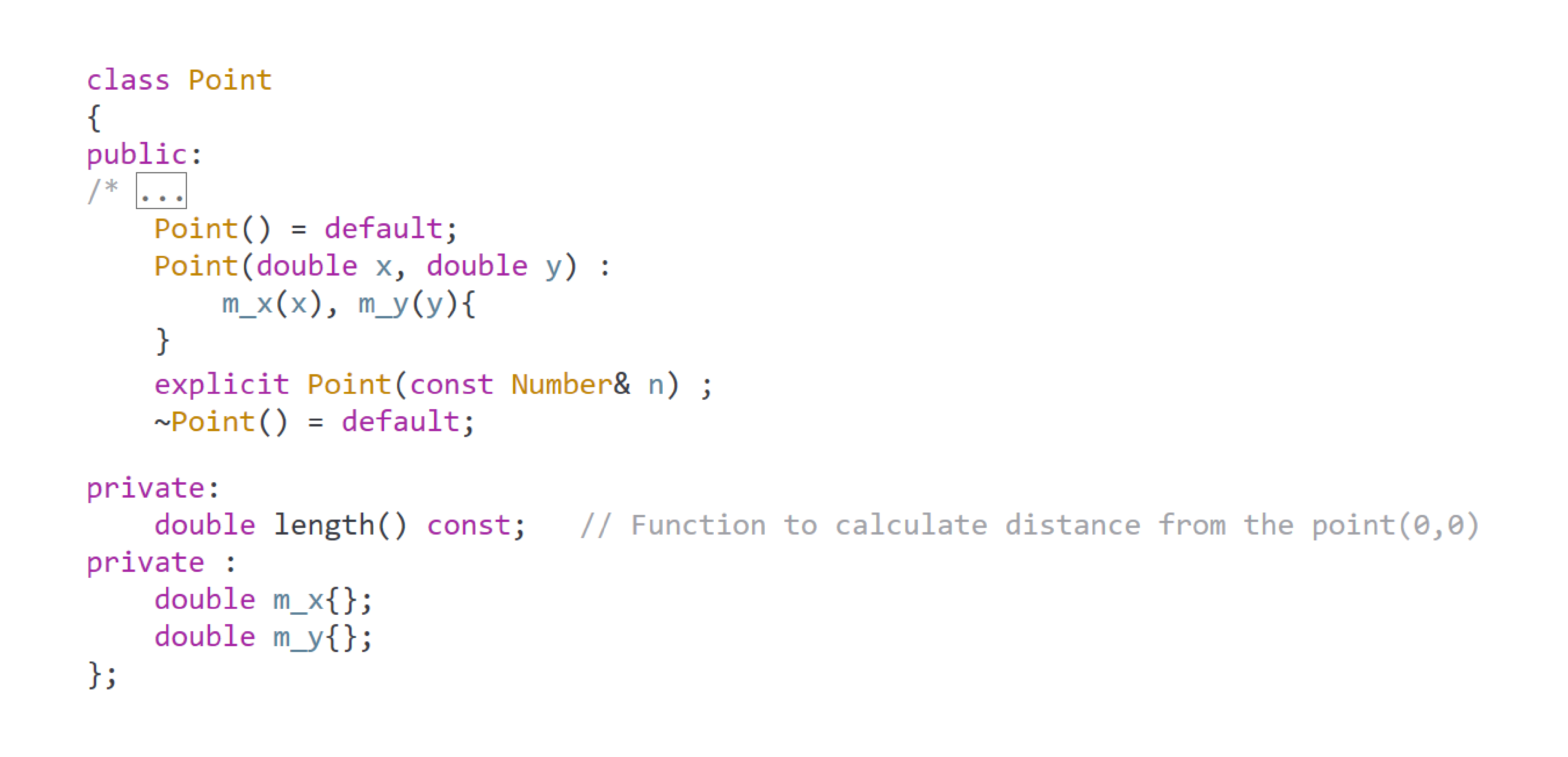
```cpp
class Number
{

public:
    Number() = default;
    Number(int value );

    //Type conversion. Can only be done as member function
    explicit operator double() const{
    //operator double() const{
        std::cout << "Using type conversion from Number to double" << std::endl;
        return static_cast<double>(m_wrapped_int);
    }

     explicit operator Point() const{
        std::cout << "Using type conversion from Number to Point" << std::endl;
        return Point(static_cast<double>(m_wrapped_int),
                            static_cast<double>(m_wrapped_int));
    }

private :
    int m_wrapped_int{0};
};
```

Using type conversions



```
double sum(double a, double b){
    return a + b;
}
void use_point(const Point& p){
    std::cout << "Printing the point from use_point func : " << p << std::endl;
}

int main(int argc, char **argv)
{
    Number n1(22);
    std::cout << "n1 : " << n1 << std::endl;

    double c{5.5};
    double d = sum(c,static_cast<double>(n1));

    use_point(static_cast<Point>(n1));

    return 0;
}
```

Ambiguity on some compilers



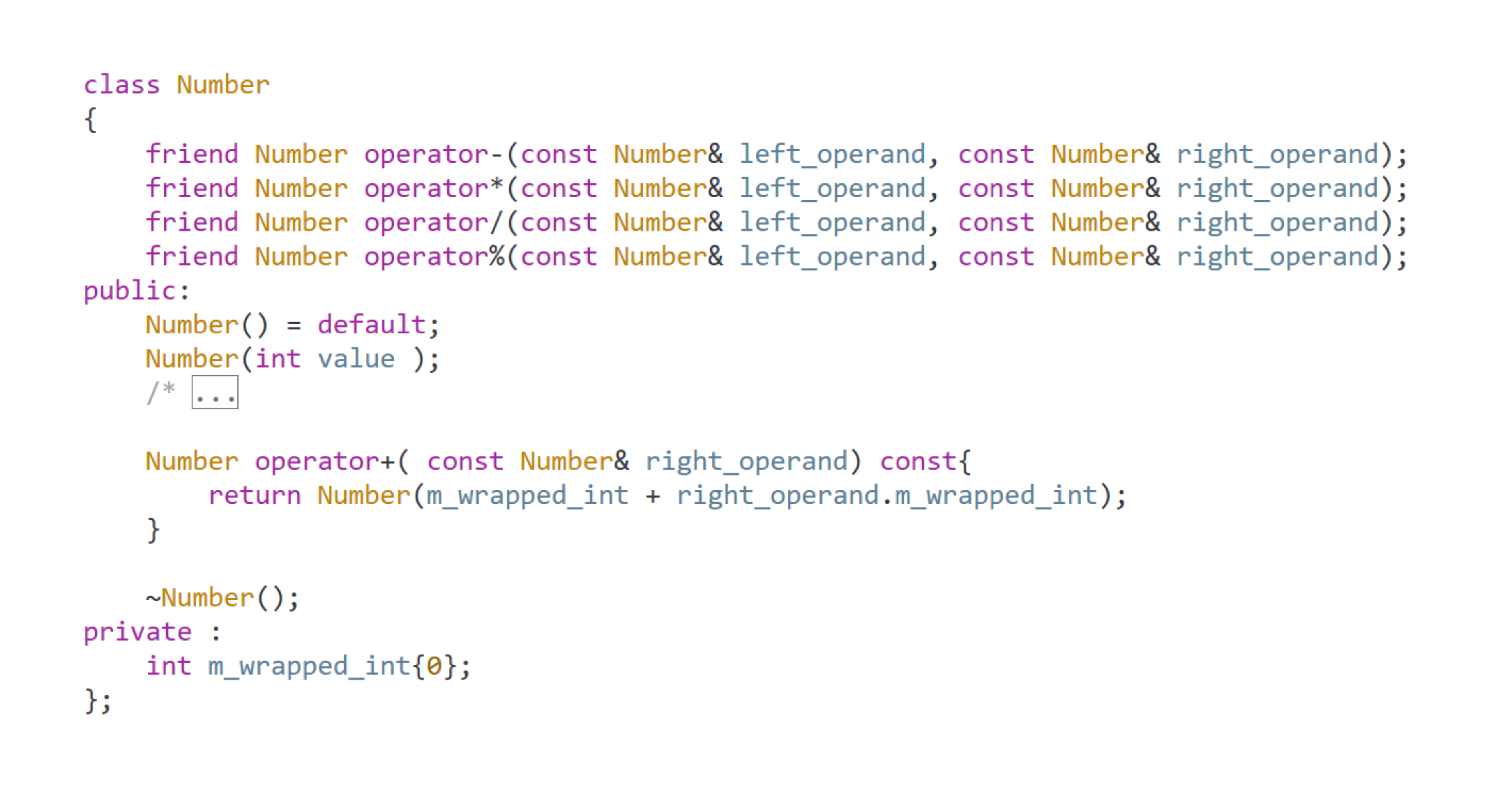
```cpp
class Point
{
public:
/* ...
    Point() = default;
    Point(double x, double y) :
        m_x(x), m_y(y){
    }
    explicit Point(const Number& n) ;
    ~Point() = default;

private:
    double length() const;   // Function to calculate distance from the point(0,0)
private :
    double m_x{};
    double m_y{};
};
```

Slide intentionally left empty

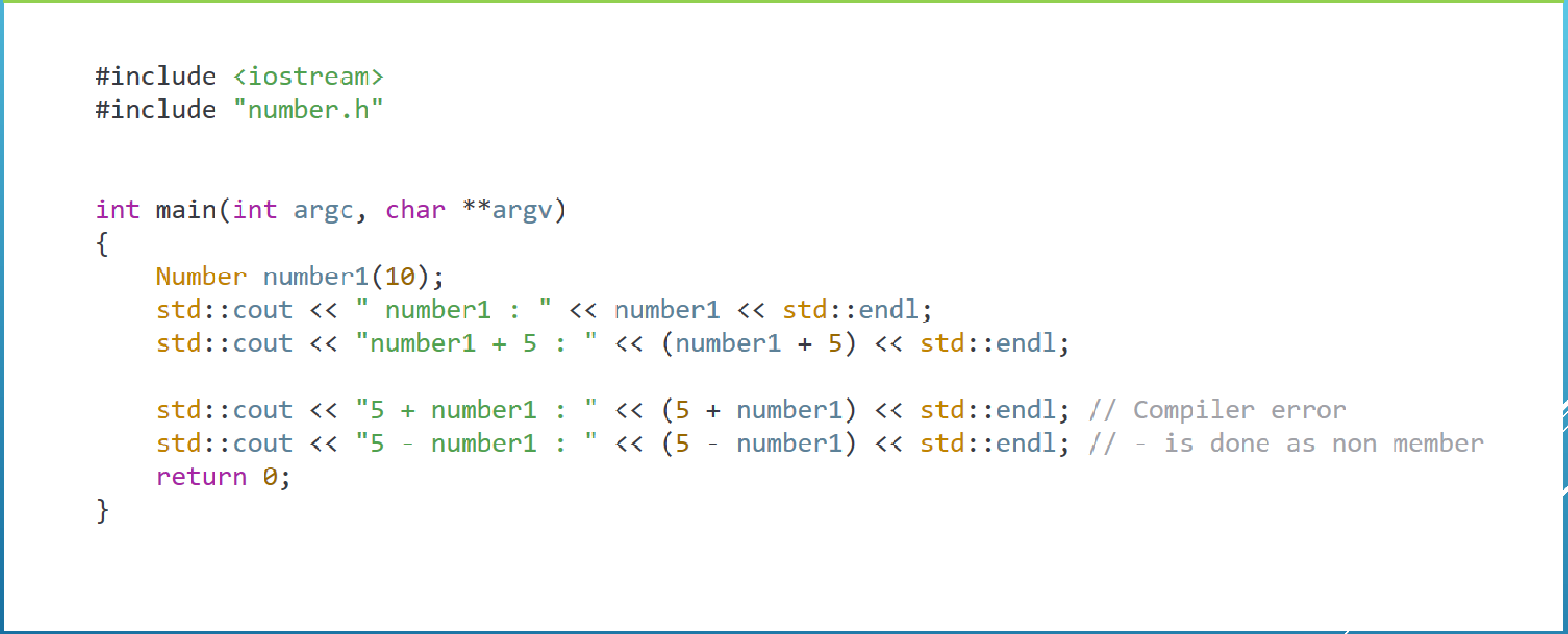
Implicit Conversions with Overloaded binary operators

When a binary operator is implemented as a member function, the left operand is never implicitly converted

Implicit conversions don’t work for the left operand



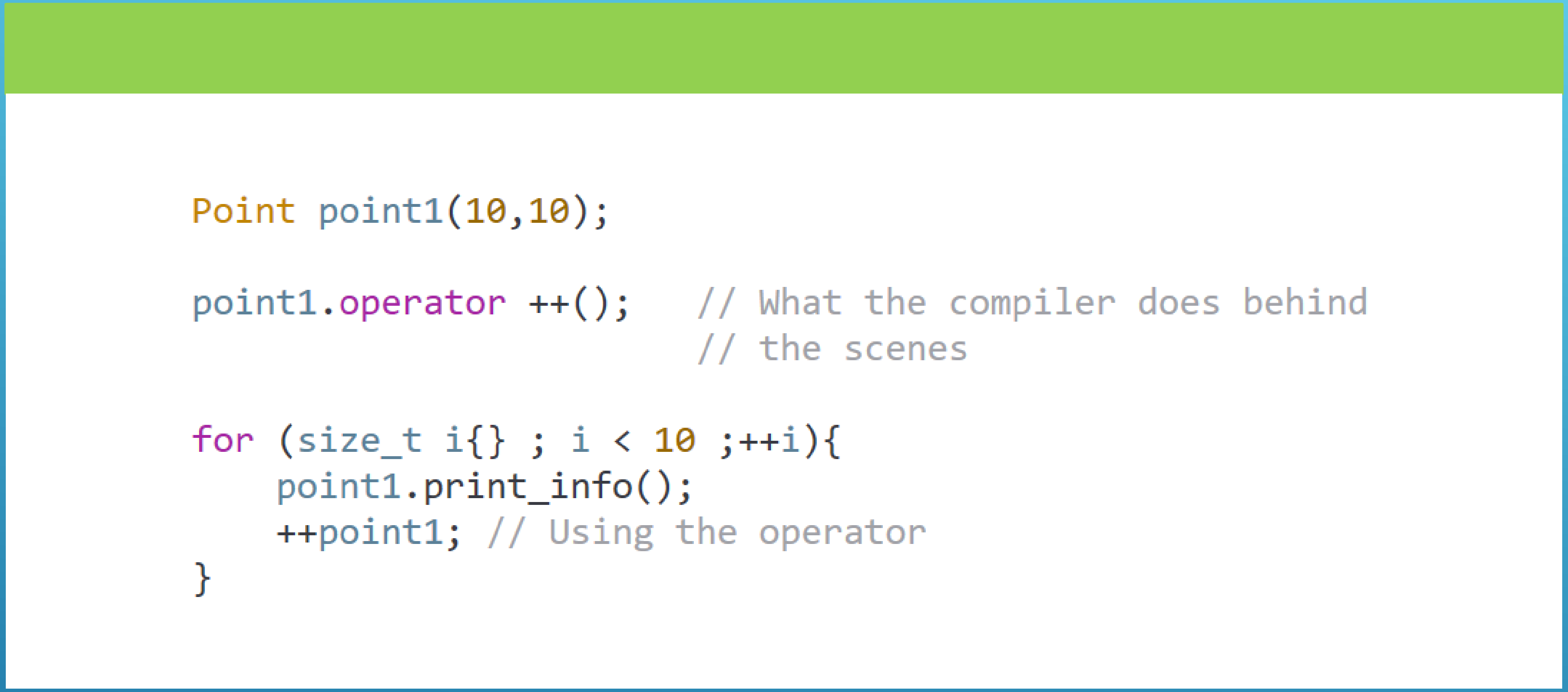
```cpp
class Number
{
    friend Number operator-(const Number& left_operand, const Number& right_operand);
    friend Number operator*(const Number& left_operand, const Number& right_operand);
    friend Number operator/(const Number& left_operand, const Number& right_operand);
    friend Number operator%(const Number& left_operand, const Number& right_operand);
public:
    Number() = default;
    Number(int value );
    /* ...

    Number operator+( const Number& right_operand) const{
        return Number(m_wrapped_int + right_operand.m_wrapped_int);
    }

    ~Number();
private :
    int m_wrapped_int{0};
};
```



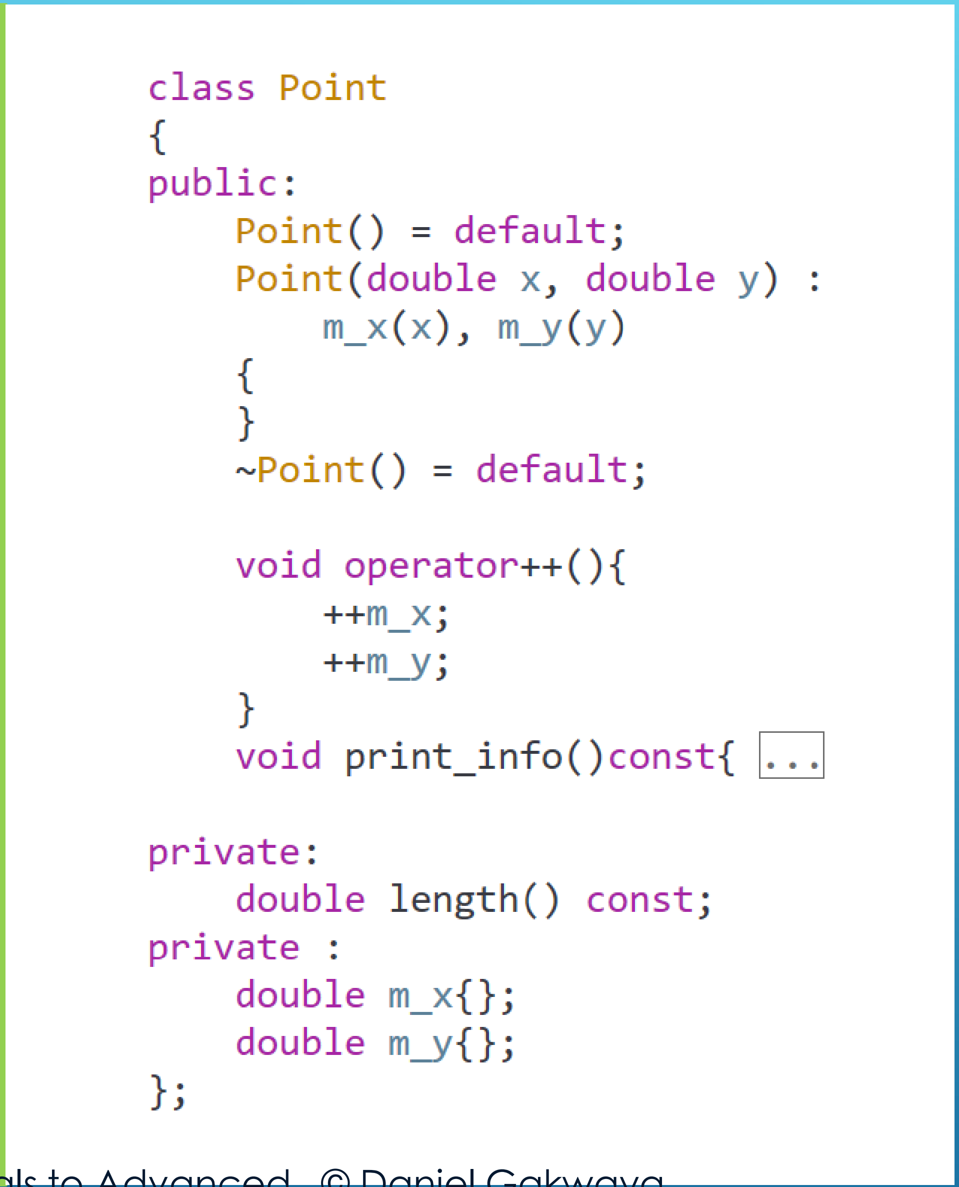
```cpp
#include <iostream>
#include "number.h"


int main(int argc, char **argv)
{
    Number number1(10);
    std::cout << " number1 : " << number1 << std::endl;
    std::cout << "number1 + 5 : " << (number1 + 5) << std::endl;

    std::cout << "5 + number1 : " << (5 + number1) << std::endl; // Compiler error
    std::cout << "5 - number1 : " << (5 - number1) << std::endl; // - is done as non member
    return 0;
}
```

Slide intentionally left empty

Overloading the prefix ++ operator



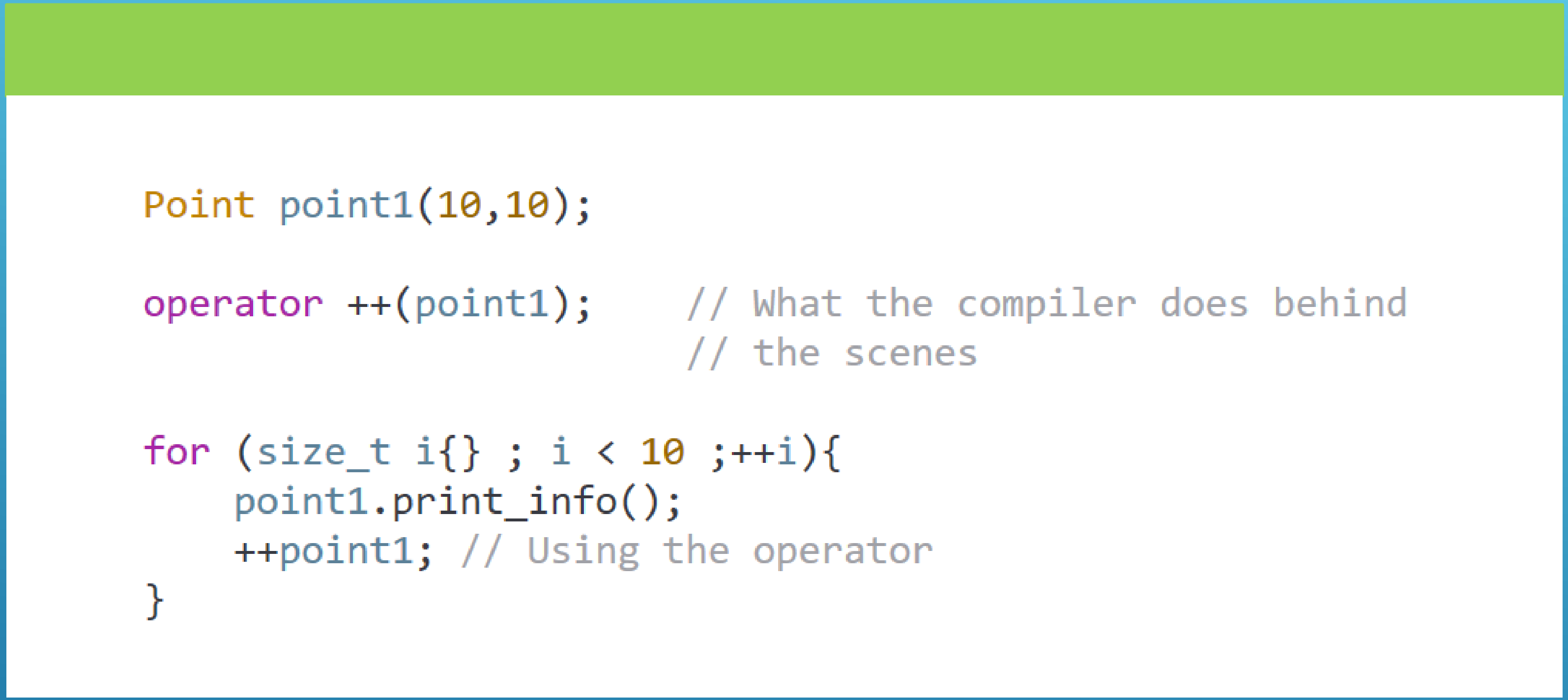
```cpp
Point point1(10,10);

point1.operator ++();   // What the compiler does behind
                        // the scenes

for (size_t i{} ; i < 10 ;++i){
    point1.print_info();
    ++point1; // Using the operator
}
```



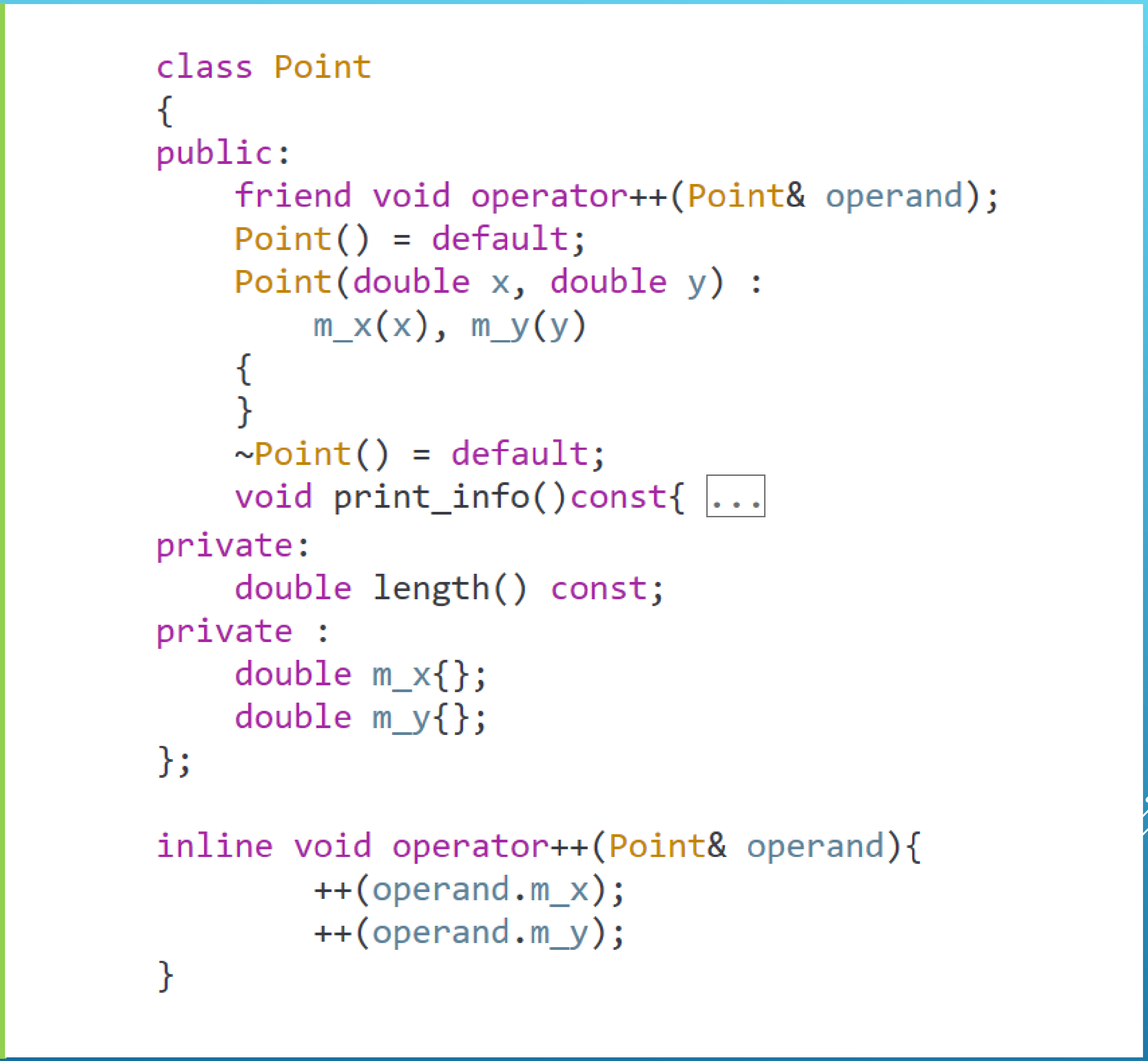
```cpp
class Point
{
public:
    Point() = default;
    Point(double x, double y) :
        m_x(x), m_y(y)
    {
    }
    ~Point() = default;

    void operator++(){
        ++m_x;
        ++m_y;
    }
    void print_info()const{ ...

private:
    double length() const;
private :
    double m_x{};
    double m_y{};
};
```

Slide intentionally left empty

Overloading the prefix ++ operator as non member



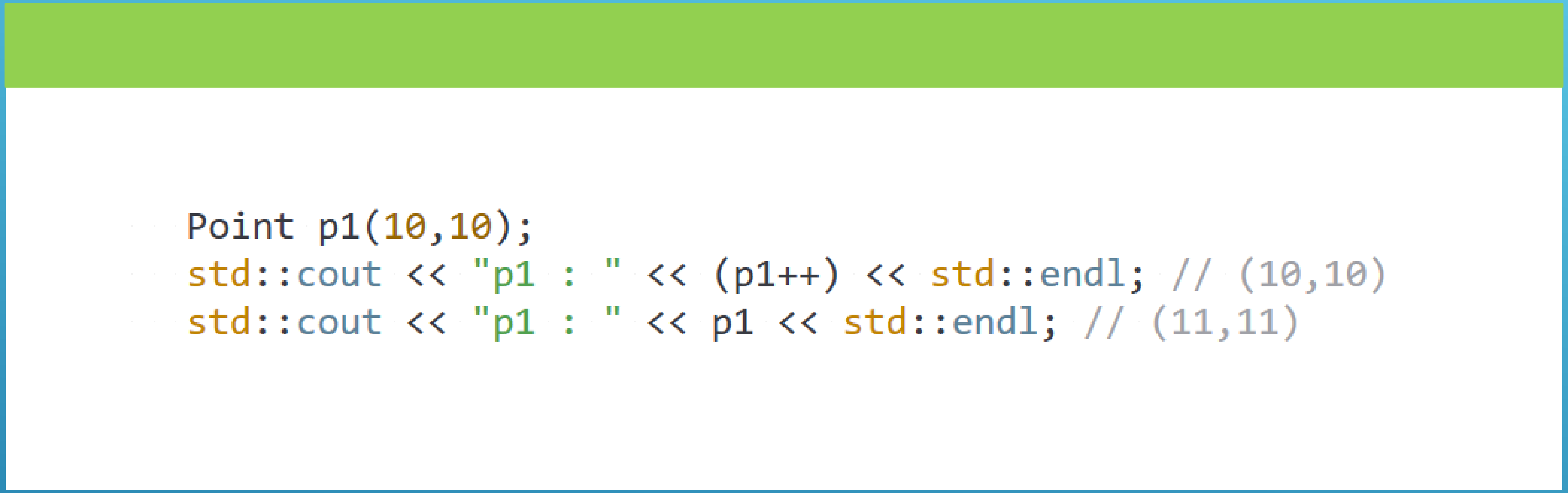
```cpp
Point point1(10,10);

operator ++(point1);     // What the compiler does behind
                         // the scenes

for (size_t i{} ; i < 10 ;++i){
    point1.print_info();
    ++point1; // Using the operator
}
```



```
class Point
{
public:
    friend void operator++(Point& operand);
    Point() = default;
    Point(double x, double y) :
        m_x(x), m_y(y)
    {
    }
    ~Point() = default;
    void print_info()const{ ...
private:
    double length() const;
private :
    double m_x{};
    double m_y{};
};

inline void operator++(Point& operand){
        ++(operand.m_x);
        ++(operand.m_y);
}
```

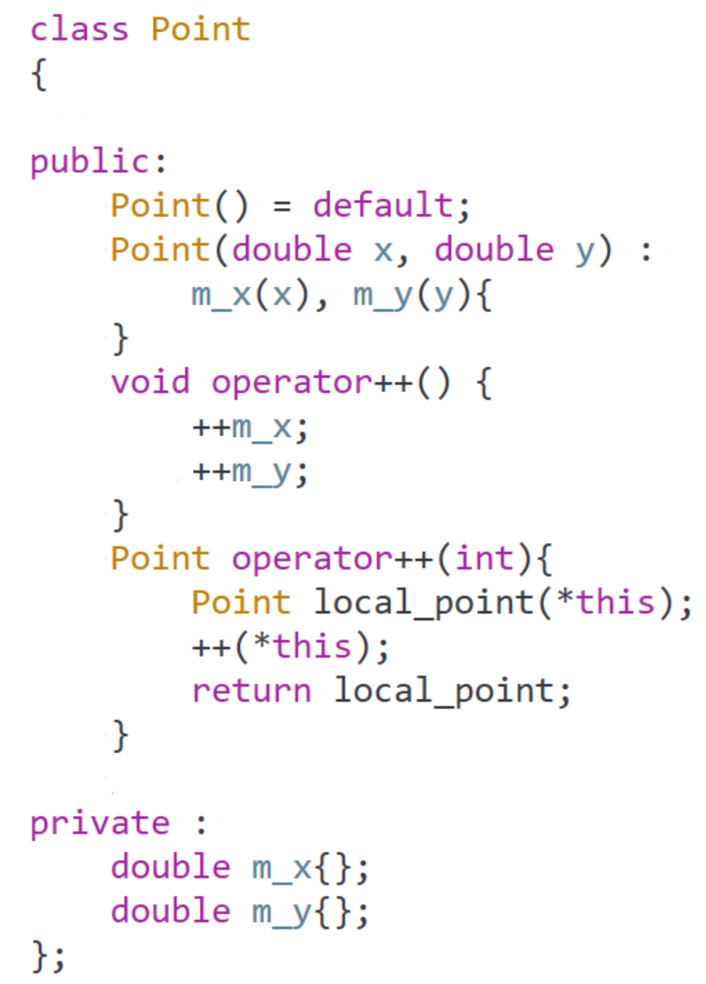
Slide intentionally left empty

Unary postfix increment operator



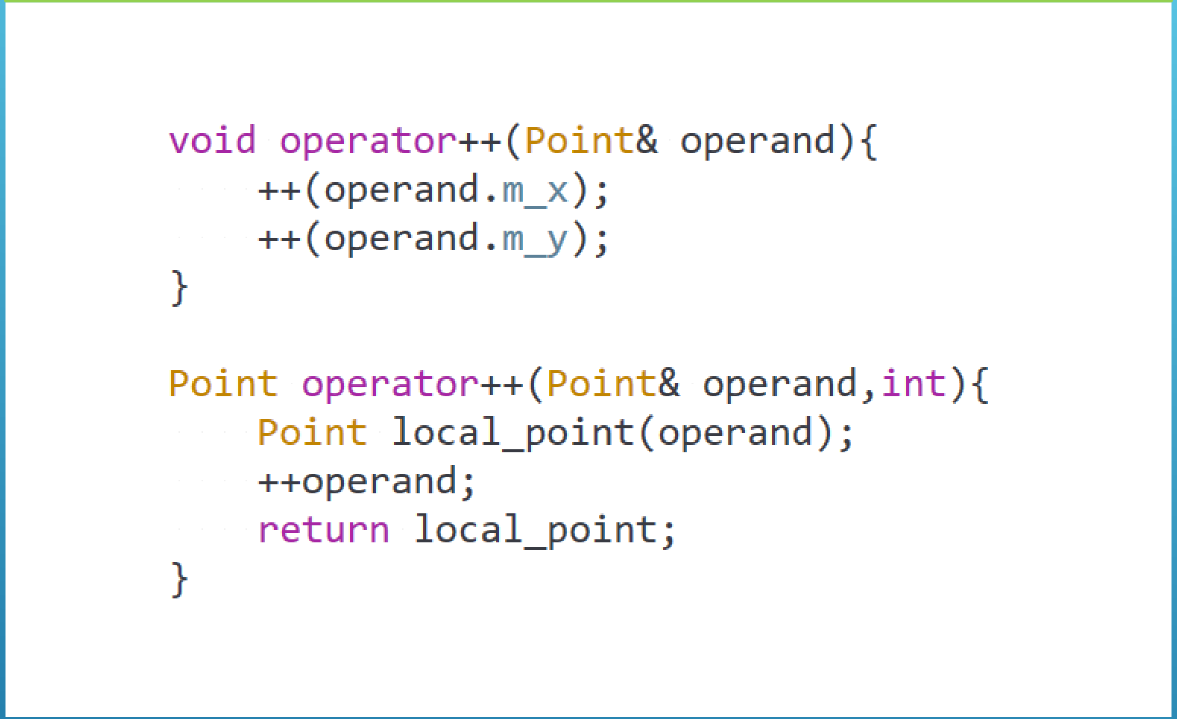
```cpp
Point p1(10,10);
std::cout << "p1 : " << (p1++) << std::endl; // (10,10)
std::cout << "p1 : " << p1 << std::endl; // (11,11)
```

Members



```cpp
class Point
{

public:
    Point() = default;
    Point(double x, double y) :
        m_x(x), m_y(y){
    }
    void operator++() {
        ++m_x;
        ++m_y;
    }
    Point operator++(int){
        Point local_point(*this);
        ++(*this);
        return local_point;
    }

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

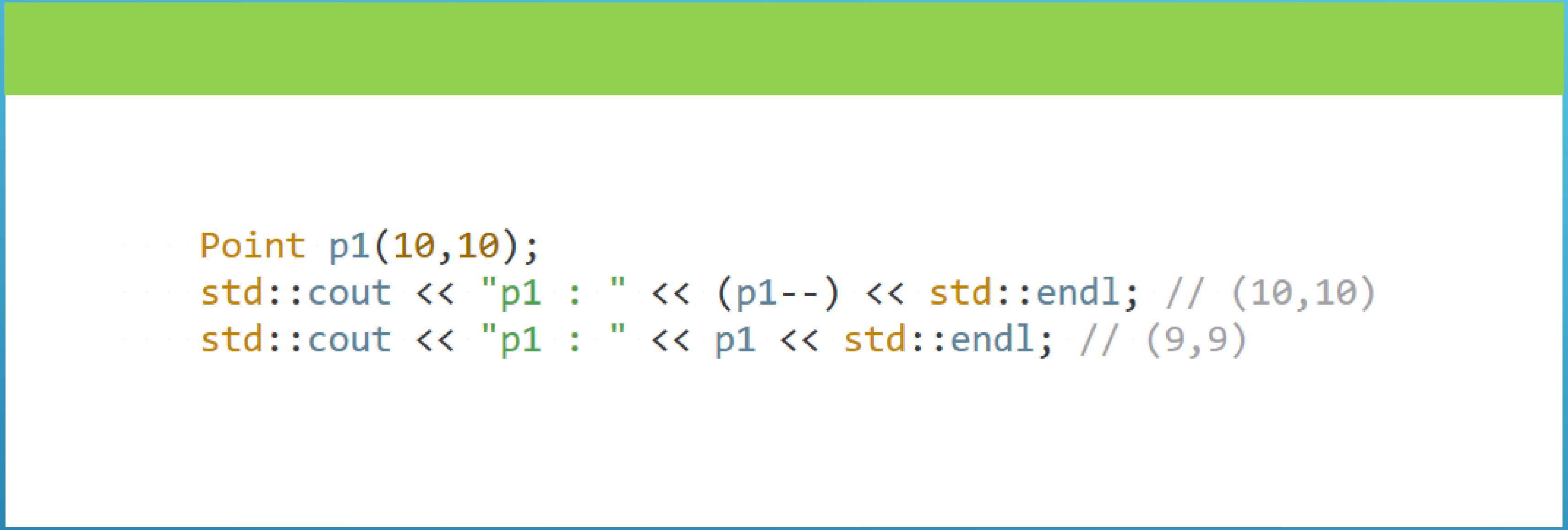
Non members



```
void operator++(Point& operand){
    ++(operand.m_x);
    ++(operand.m_y);
}

Point operator++(Point& operand,int){
    Point local_point(operand);
    ++operand;
    return local_point;
}
```

Slide intentionally left empty

Prefix/postfix decrement operator

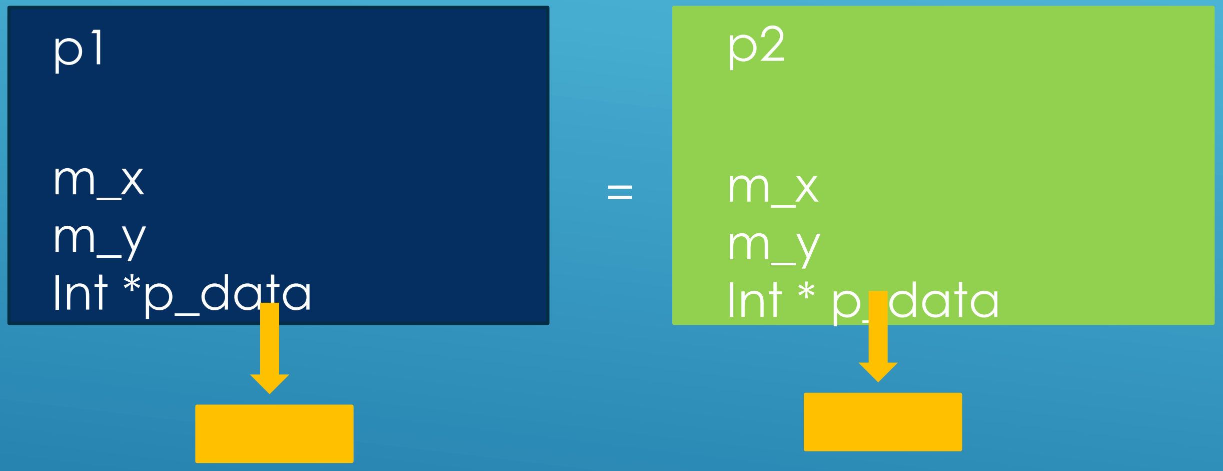


```cpp
    Point p1(10,10);
    std::cout << "p1 : " << (p1--) << std::endl; // (10,10)
    std::cout << "p1 : " << p1 << std::endl; // (9,9)
```

Slide intentionally left empty

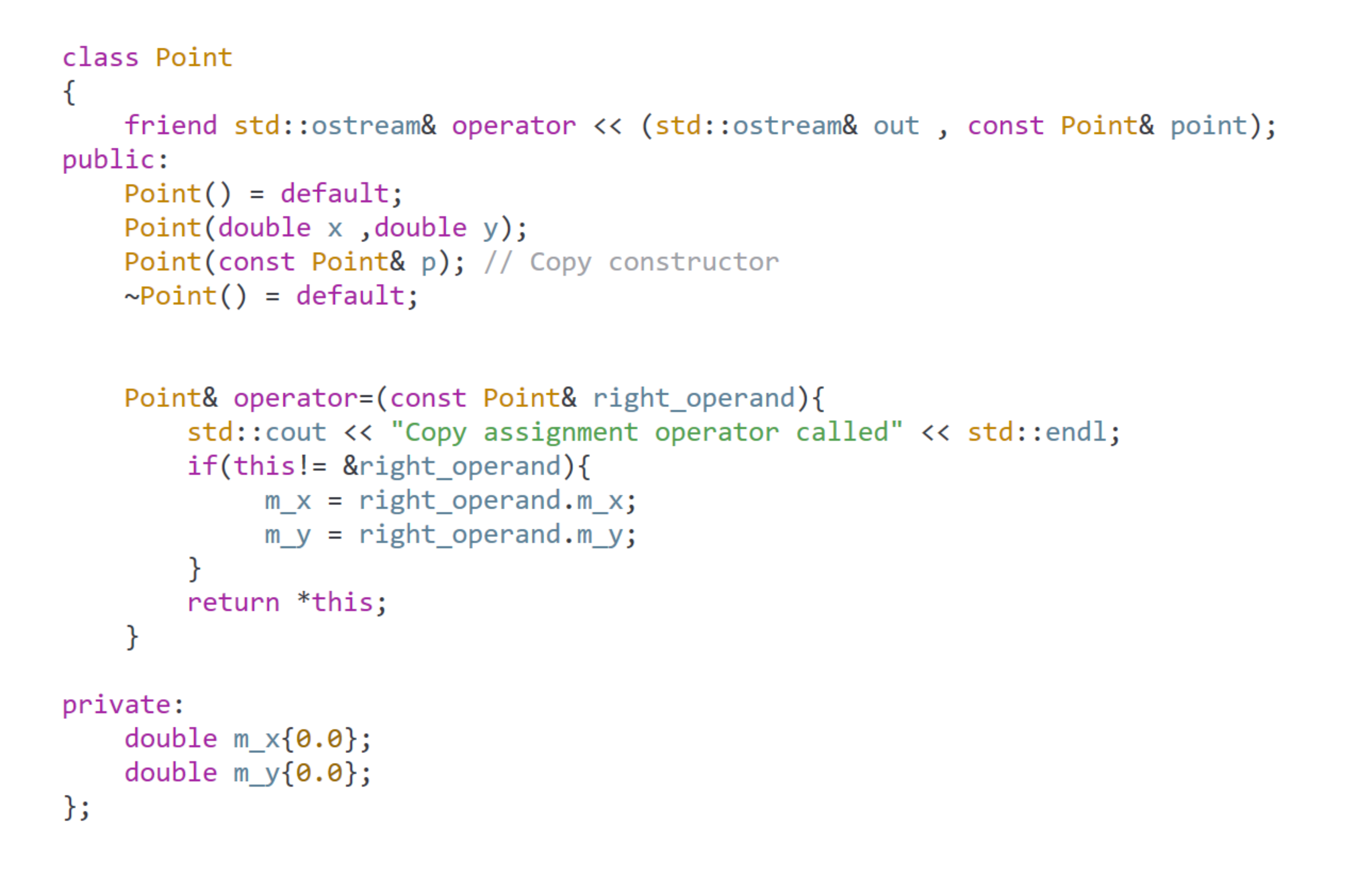
Copy assignment operator

Copy assignment operator



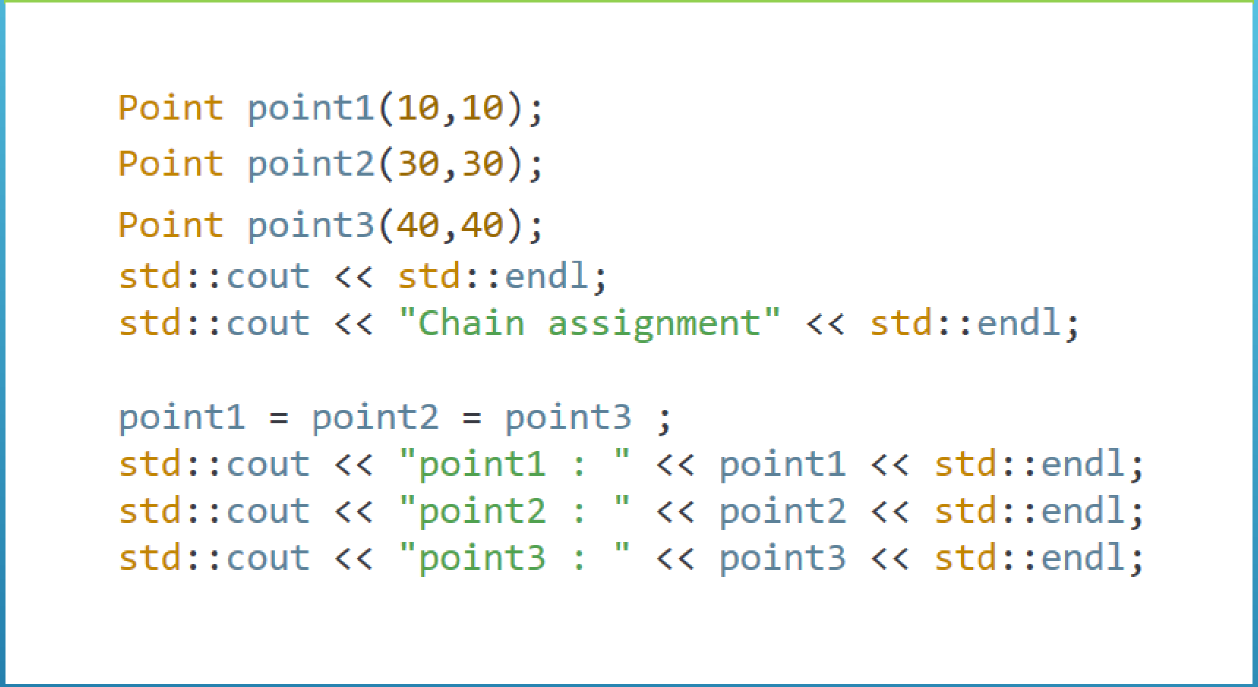
If you have no custom copy assignment operator in place, the compiler is going to generate one for you

The compiler generated one is going to do member wise copy



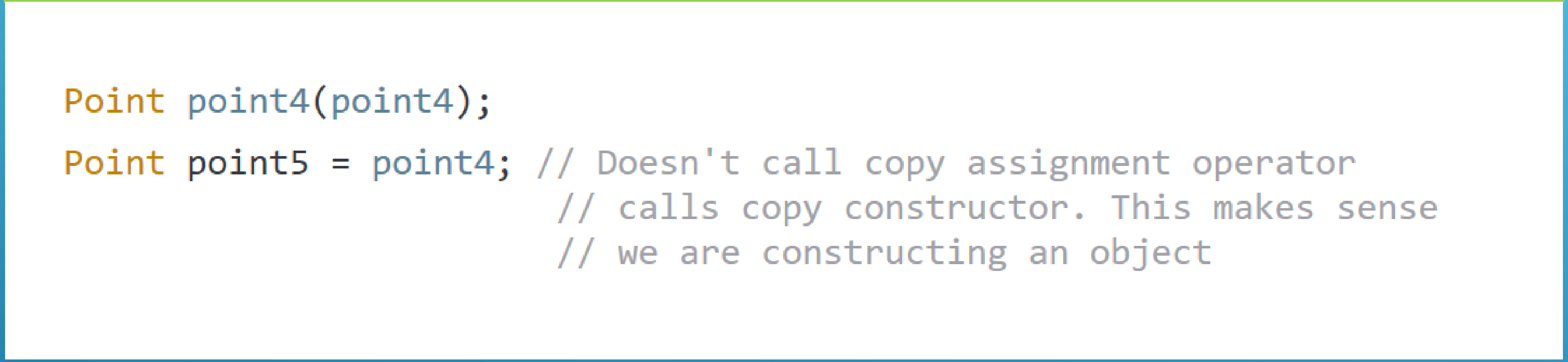
```cpp
class Point
{
    friend std::ostream& operator << (std::ostream& out , const Point& point);
public:
    Point() = default;
    Point(double x ,double y);
    Point(const Point& p); // Copy constructor
    ~Point() = default;


    Point& operator=(const Point& right_operand){
        std::cout << "Copy assignment operator called" << std::endl;
        if(this!= &right_operand){
             m_x = right_operand.m_x;
             m_y = right_operand.m_y;
        }
        return *this;
    }

private:
    double m_x{0.0};
    double m_y{0.0};
};
```

Operator chaining



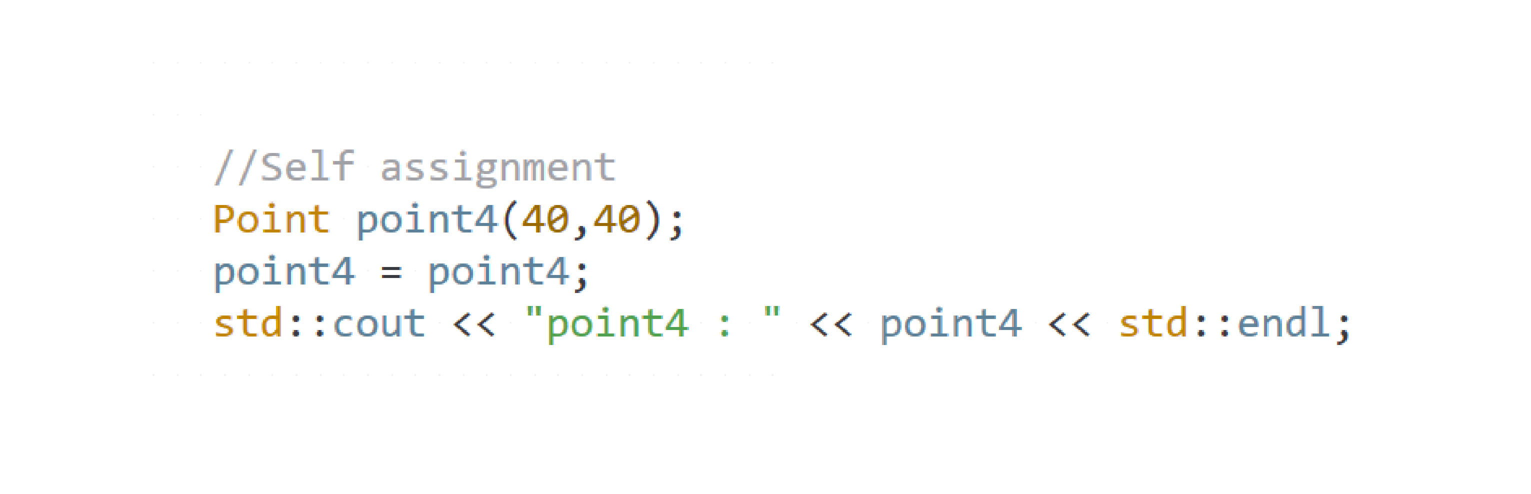
```
Point point1(10,10);
Point point2(30,30);
Point point3(40,40);
std::cout << std::endl;
std::cout << "Chain assignment" << std::endl;

point1 = point2 = point3 ;
std::cout << "point1 : " << point1 << std::endl;
std::cout << "point2 : " << point2 << std::endl;
std::cout << "point3 : " << point3 << std::endl;
```

Watch out!



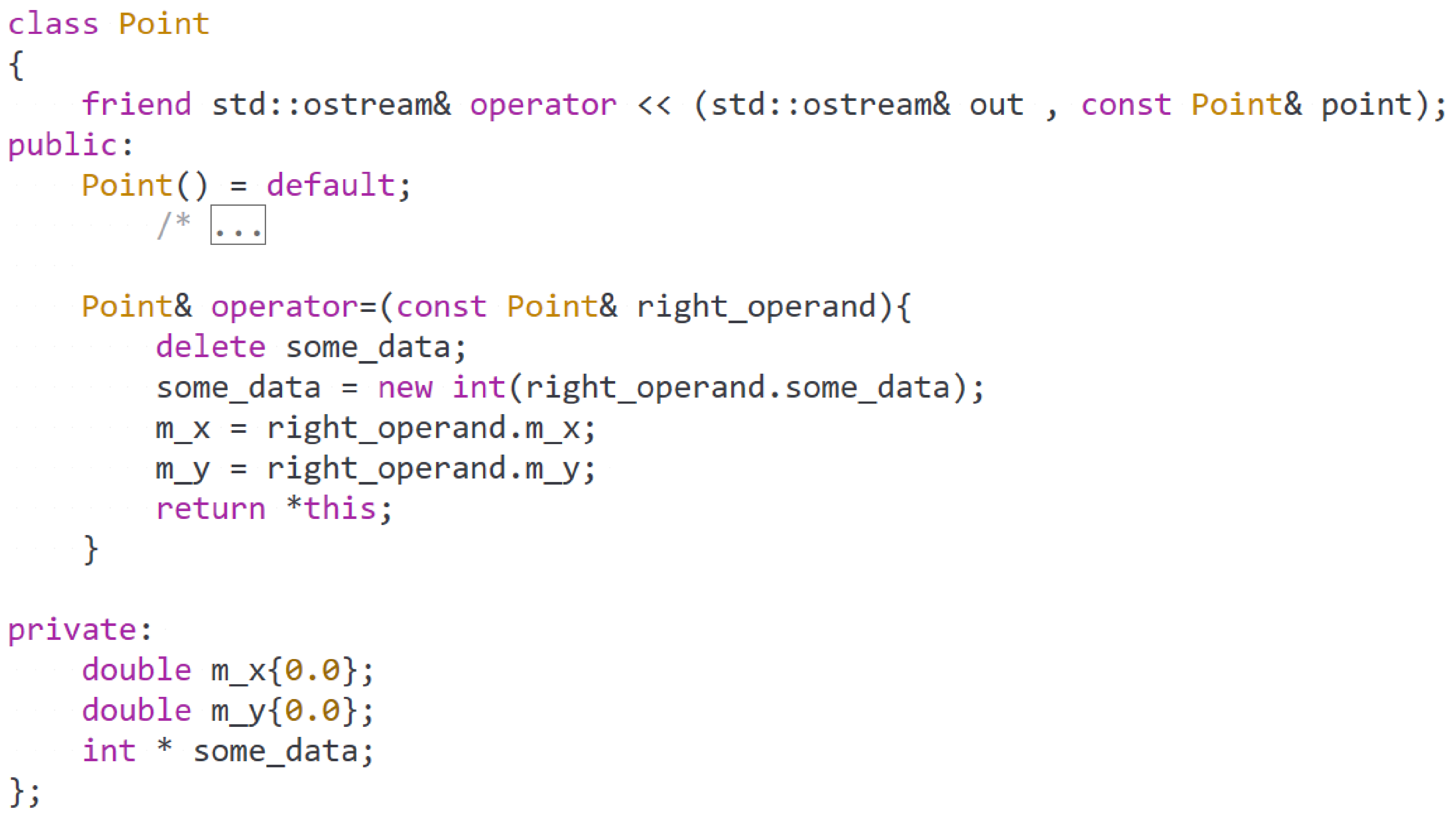
```
Point point4(point4);
Point point5 = point4; // Doesn't call copy assignment operator
                        // calls copy constructor. This makes sense
                        // we are constructing an object
```

Self assignment



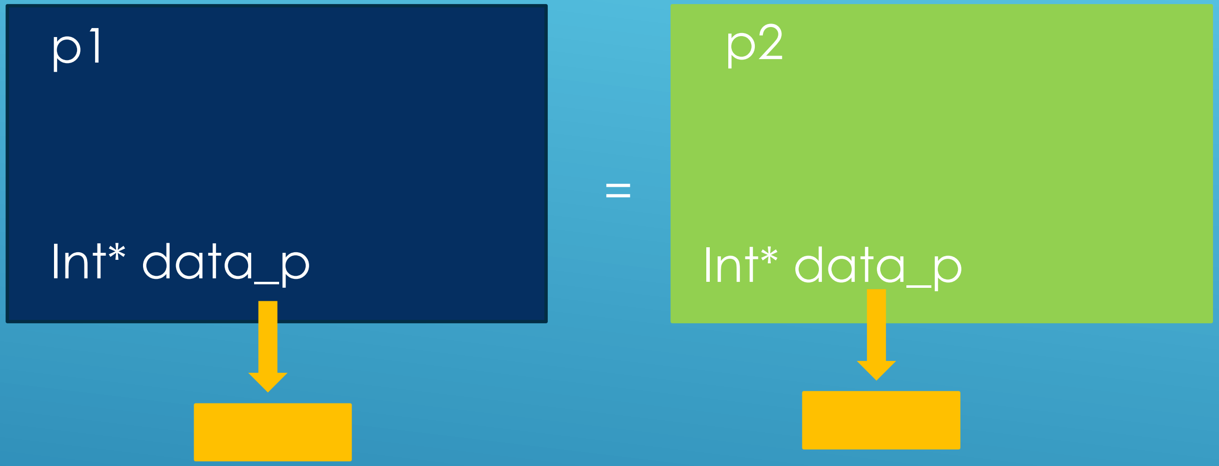
```
    //Self assignment
    Point point4(40,40);
    point4 = point4;
    std::cout << "point4 : " << point4 << std::endl;
```

Why check for self assignment



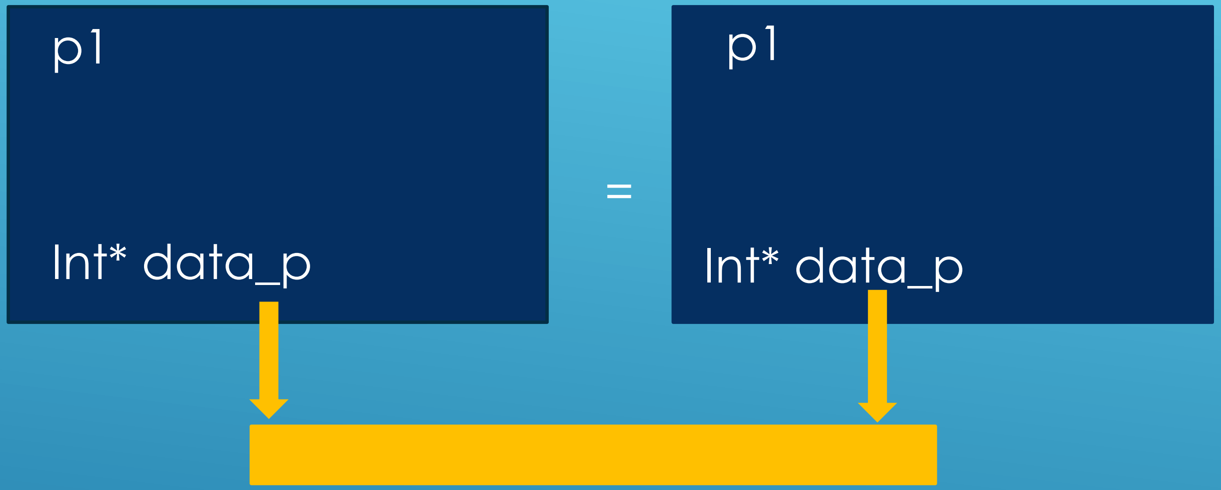
```
class Point
{
    friend std::ostream& operator << (std::ostream& out , const Point& point);
public:
    Point() = default;
        /* ...

    Point& operator=(const Point& right_operand){
        delete some_data;
        some_data = new int(right_operand.some_data);
        m_x = right_operand.m_x;
        m_y = right_operand.m_y;
        return *this;
    }

private:
    double m_x{0.0};
    double m_y{0.0};
    int * some_data;
};
```

Why check for self assignment



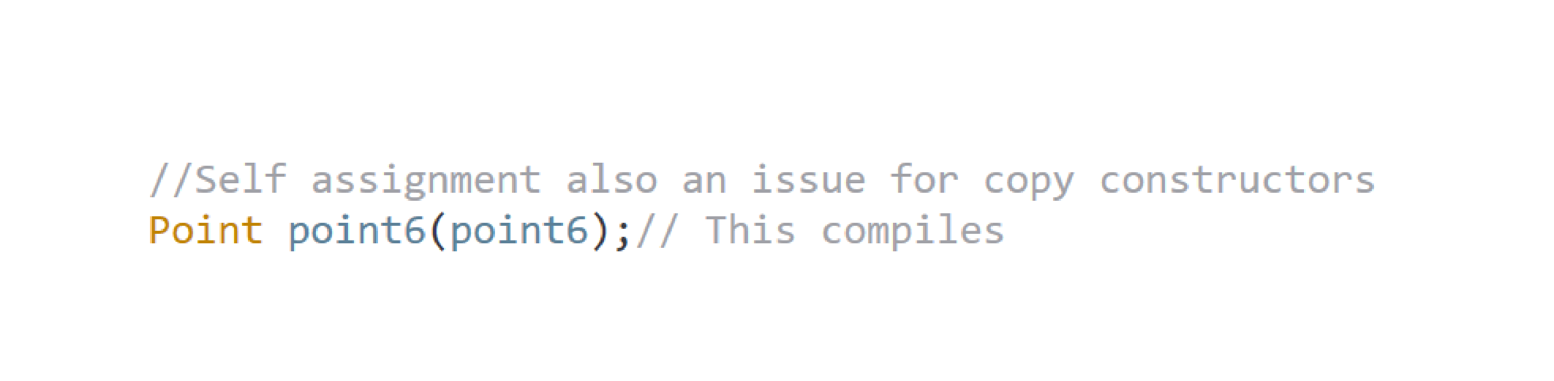
- Release memory in p1
- Allocate new dynamic memory for data_p
- Copy in data from p2

Why check for self assignment



- Release memory in p1
- Allocate new dynamic memory for data_p
- Copy in data from p1

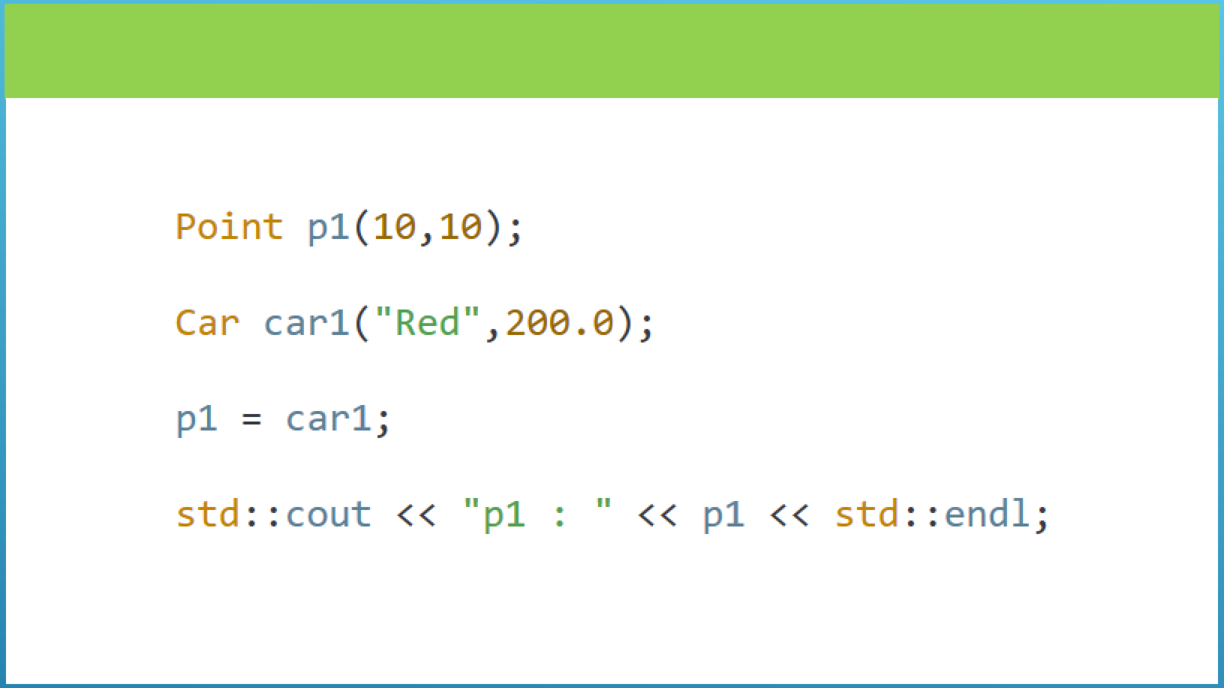
Self assignment with copy constructors



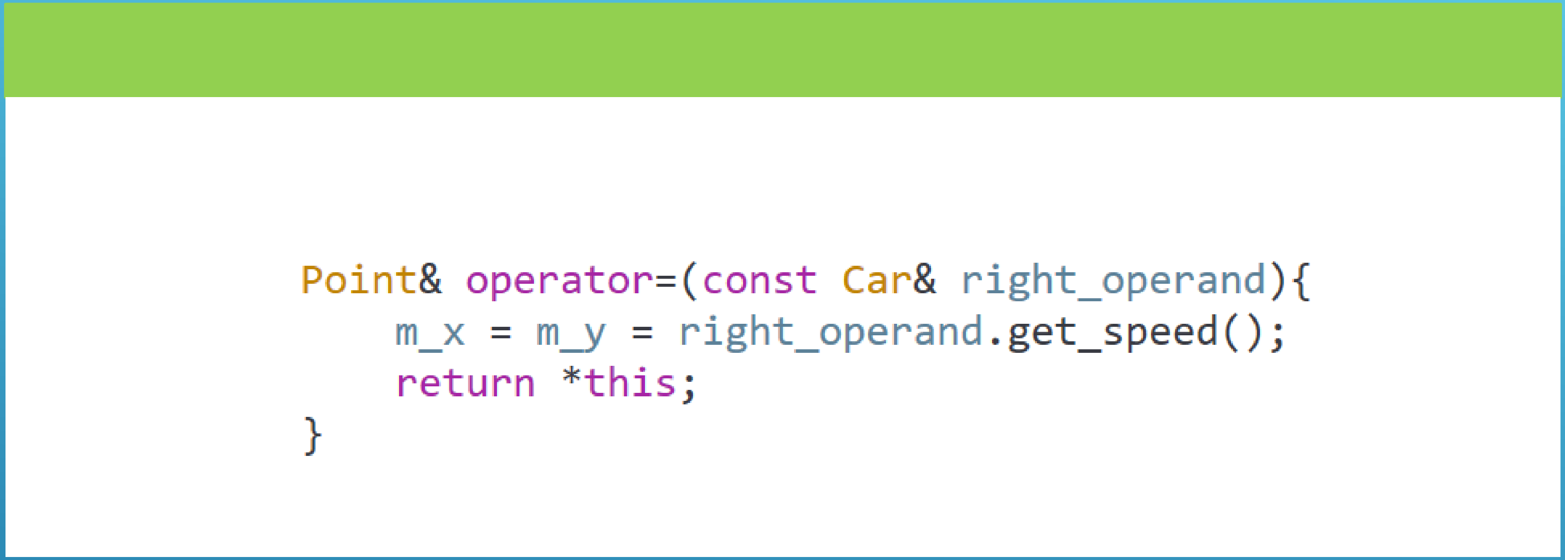
```
//Self assignment also an issue for copy constructors
Point point6(point6);// This compiles
```

Slide intentionally left empty

Copy assignment operator for other types



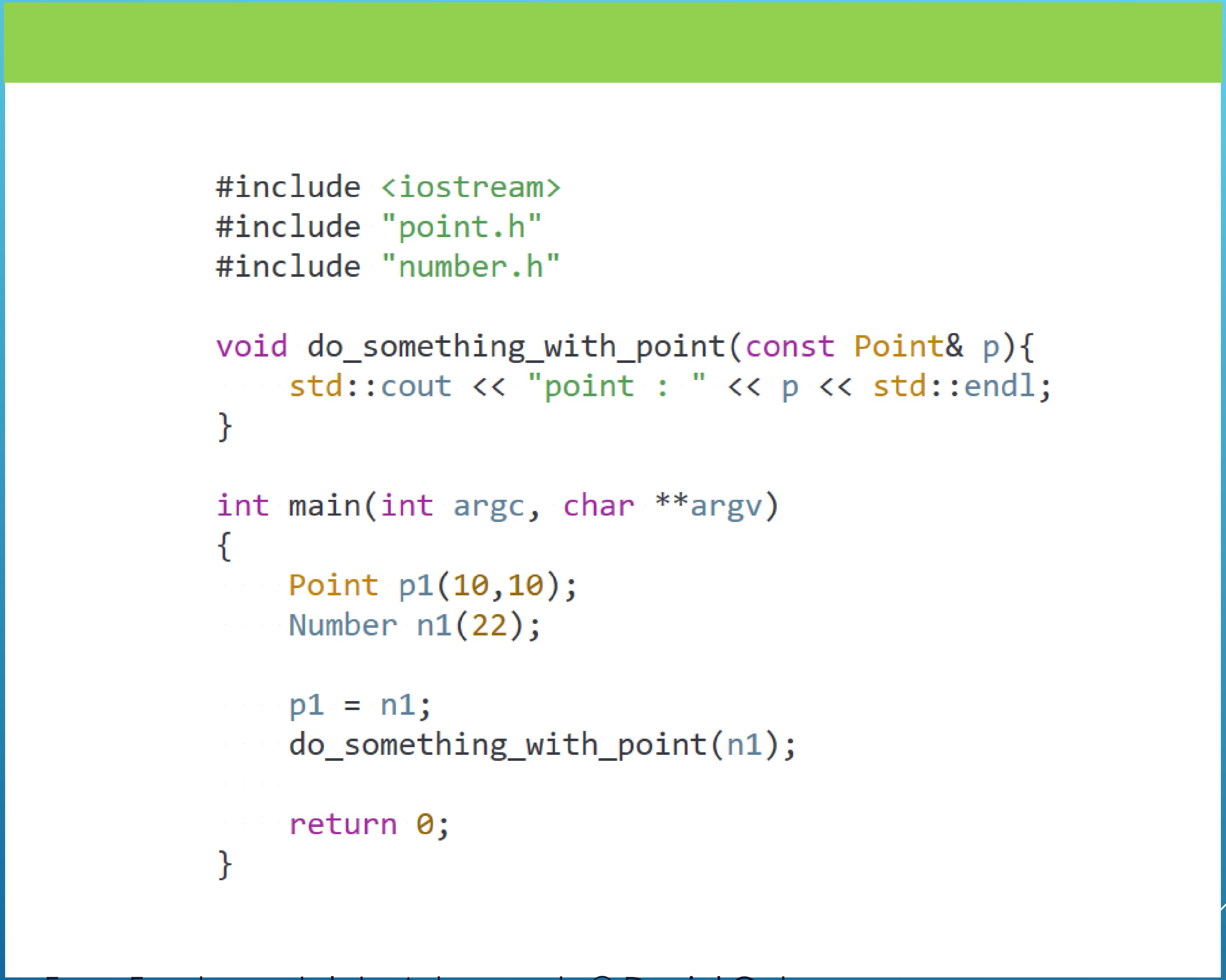
```
Point p1(10,10);

Car car1("Red",200.0);

p1 = car1;

std::cout << "p1 : " << p1 << std::endl;
```



```cpp
Point& operator=(const Car& right_operand){
    m_x = m_y = right_operand.get_speed();
    return *this;
}
```

Slide intentionally left empty

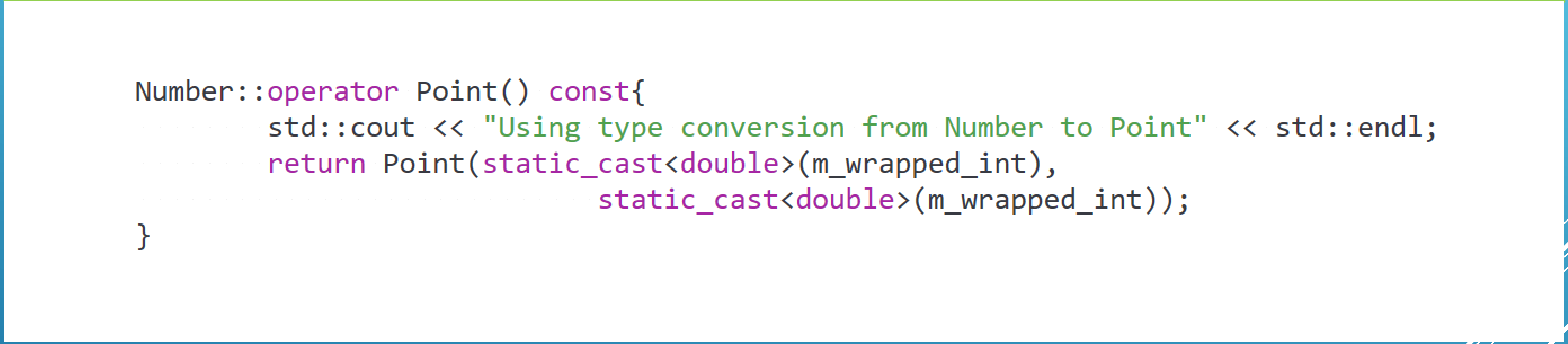
Custom Type conversions : A Recap



```cpp
#include <iostream>
#include "point.h"
#include "number.h"

void do_something_with_point(const Point& p){
    std::cout << "point : " << p << std::endl;
}

int main(int argc, char **argv)
{
    Point p1(10,10);
    Number n1(22);

    p1 = n1;
    do_something_with_point(n1);

    return 0;
}
```

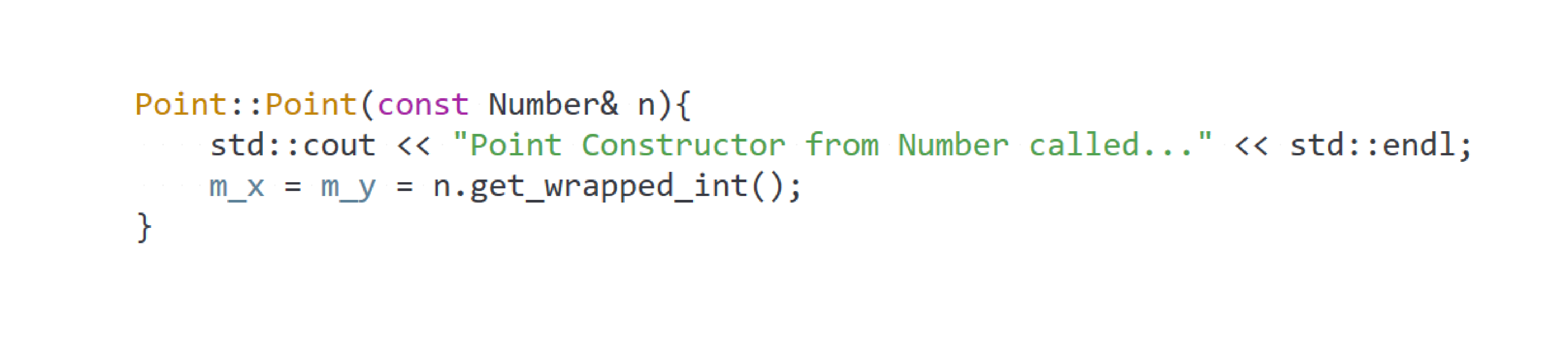
- Type Conversion Operator : Number -> Point
- Constructor taking Number in : Number -> Point
- Copy assignment operator for Number : Number -> Point

Type conversion operator



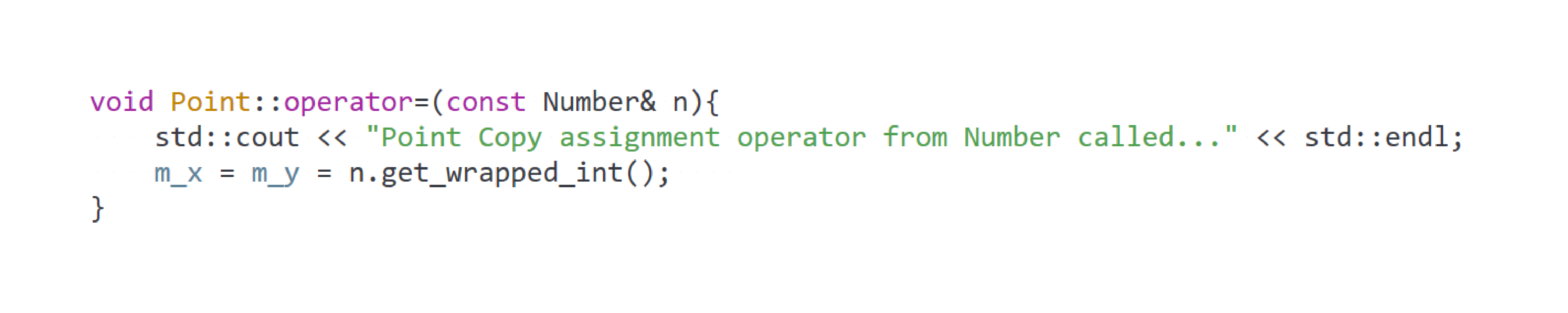
```
Number::operator Point() const{
        std::cout << "Using type conversion from Number to Point" << std::endl;
        return Point(static_cast<double>(m_wrapped_int),
                            static_cast<double>(m_wrapped_int));
}
```

Constructor taking in a Number



```cpp
Point::Point(const Number& n){
    std::cout << "Point Constructor from Number called..." << std::endl;
    m_x = m_y = n.get_wrapped_int();
}
```

Copy assignment operator for Number

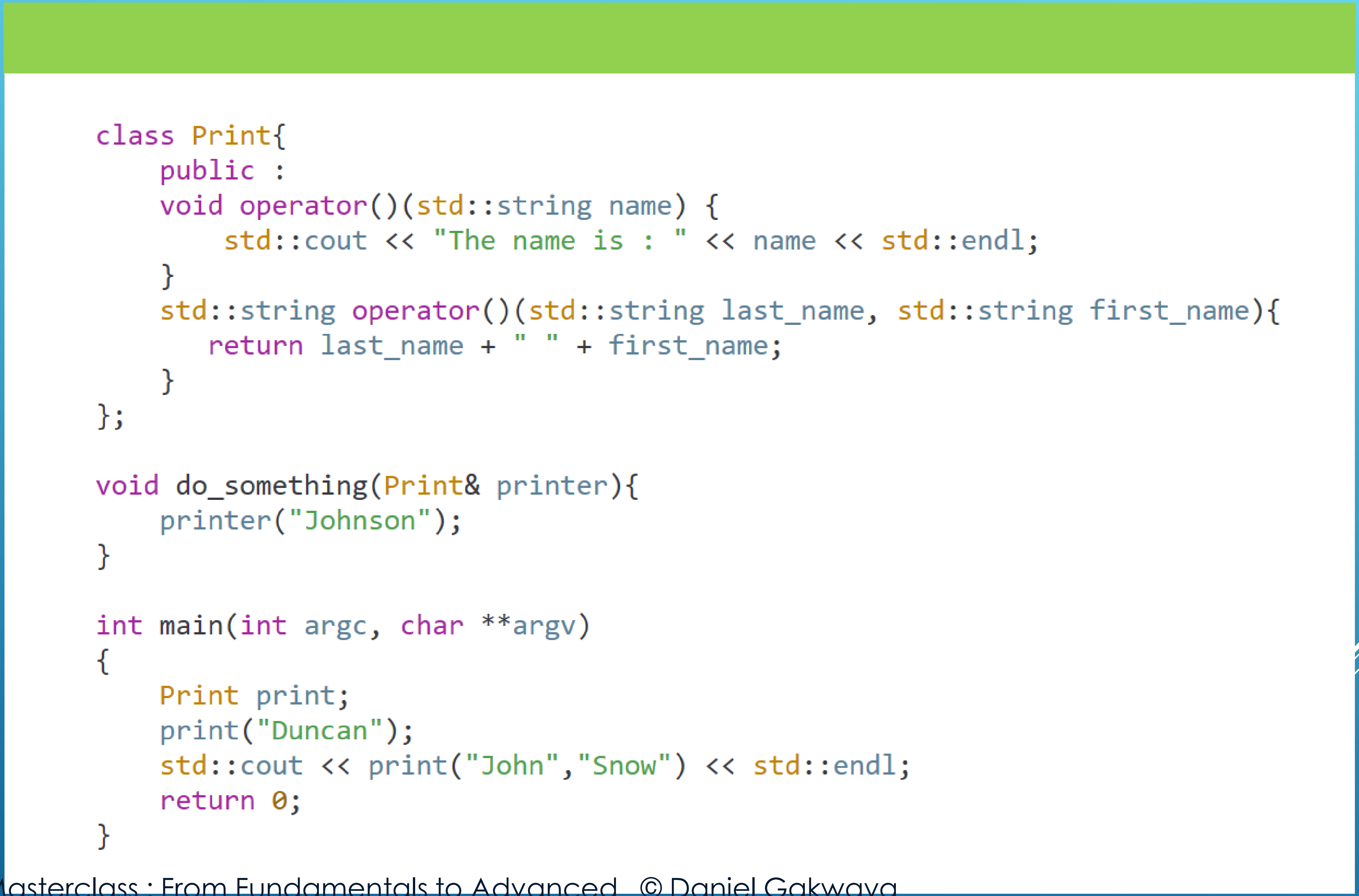


```cpp
void Point::operator=(const Number& n){
    std::cout << "Point Copy assignment operator from Number called..." << std::endl;
    m_x = m_y = n.get_wrapped_int();
}
```

Slide intentionally left empty

Functors

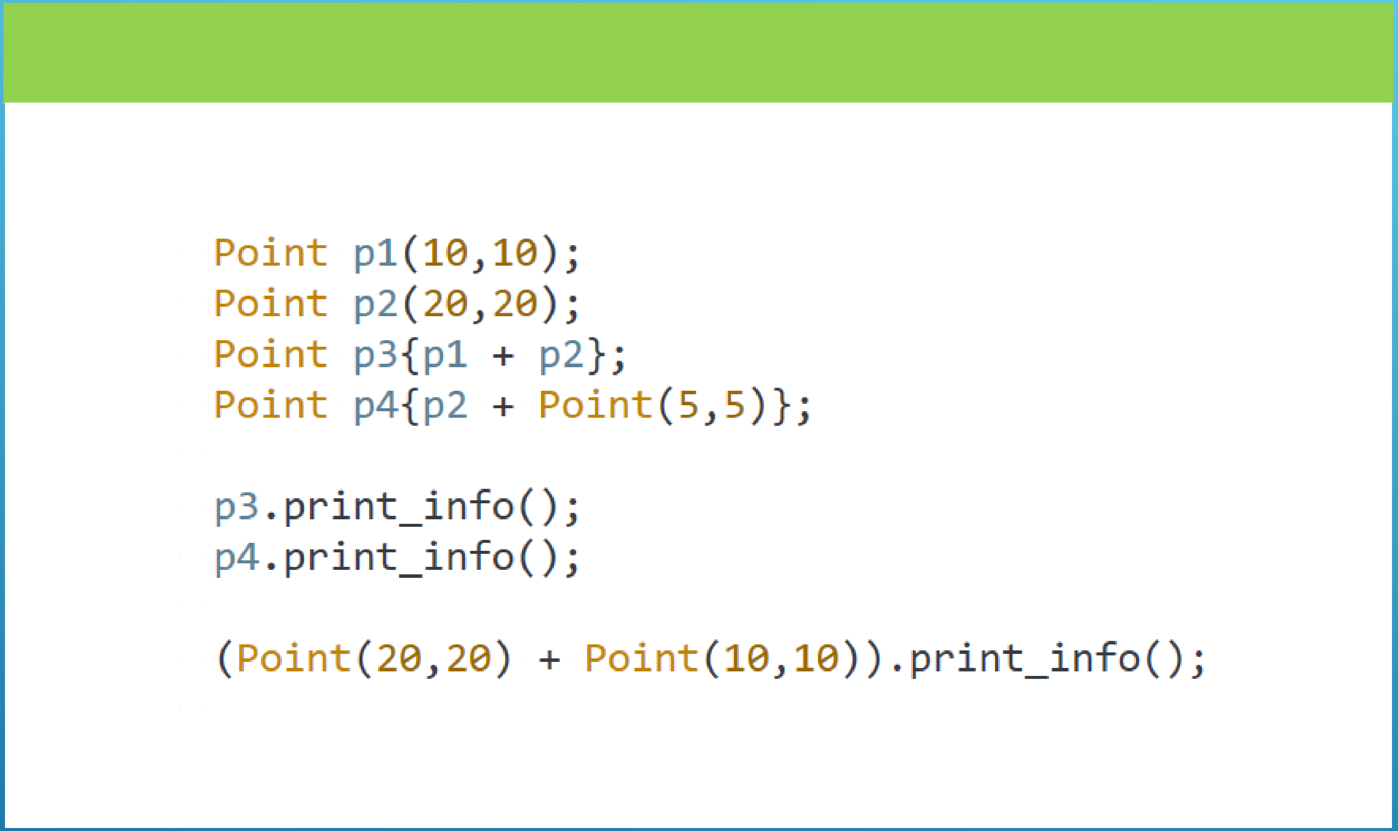
Objects of a class that overloads the () operator



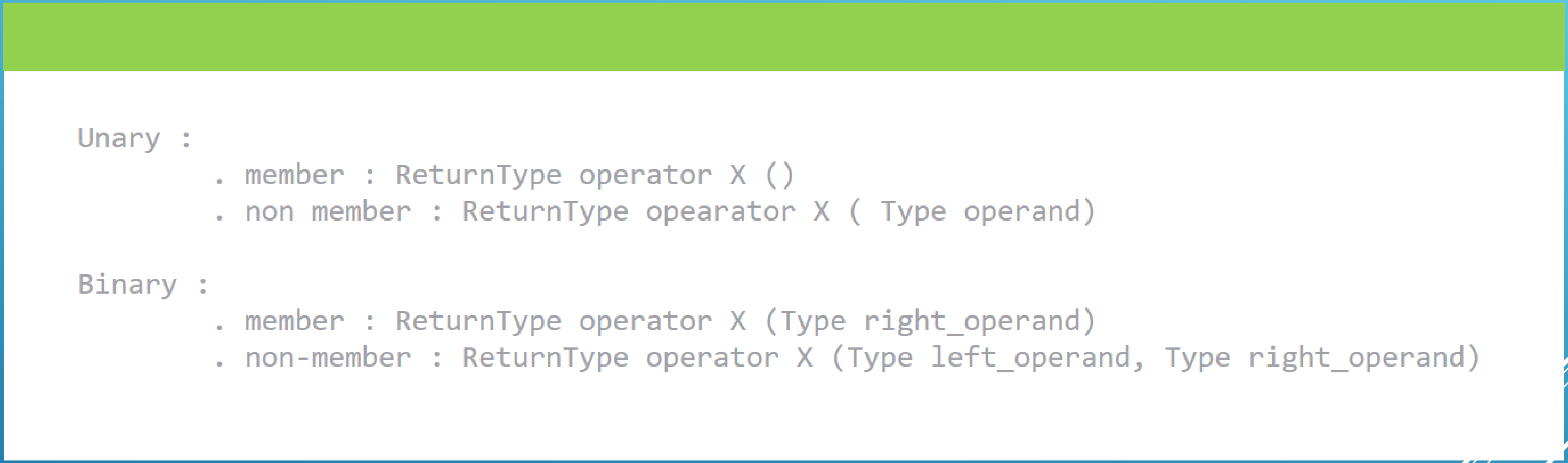
```
class Print{
    public :
    void operator()(std::string name) {
        std::cout << "The name is : " << name << std::endl;
    }
    std::string operator()(std::string last_name, std::string first_name){
       return last_name + " " + first_name;
    }
};

void do_something(Print& printer){
    printer("Johnson");
}

int main(int argc, char **argv)
{
    Print print;
    print("Duncan");
    std::cout << print("John","Snow") << std::endl;
    return 0;
}
```

Slide intentionally left empty

Operator overloading : Summary

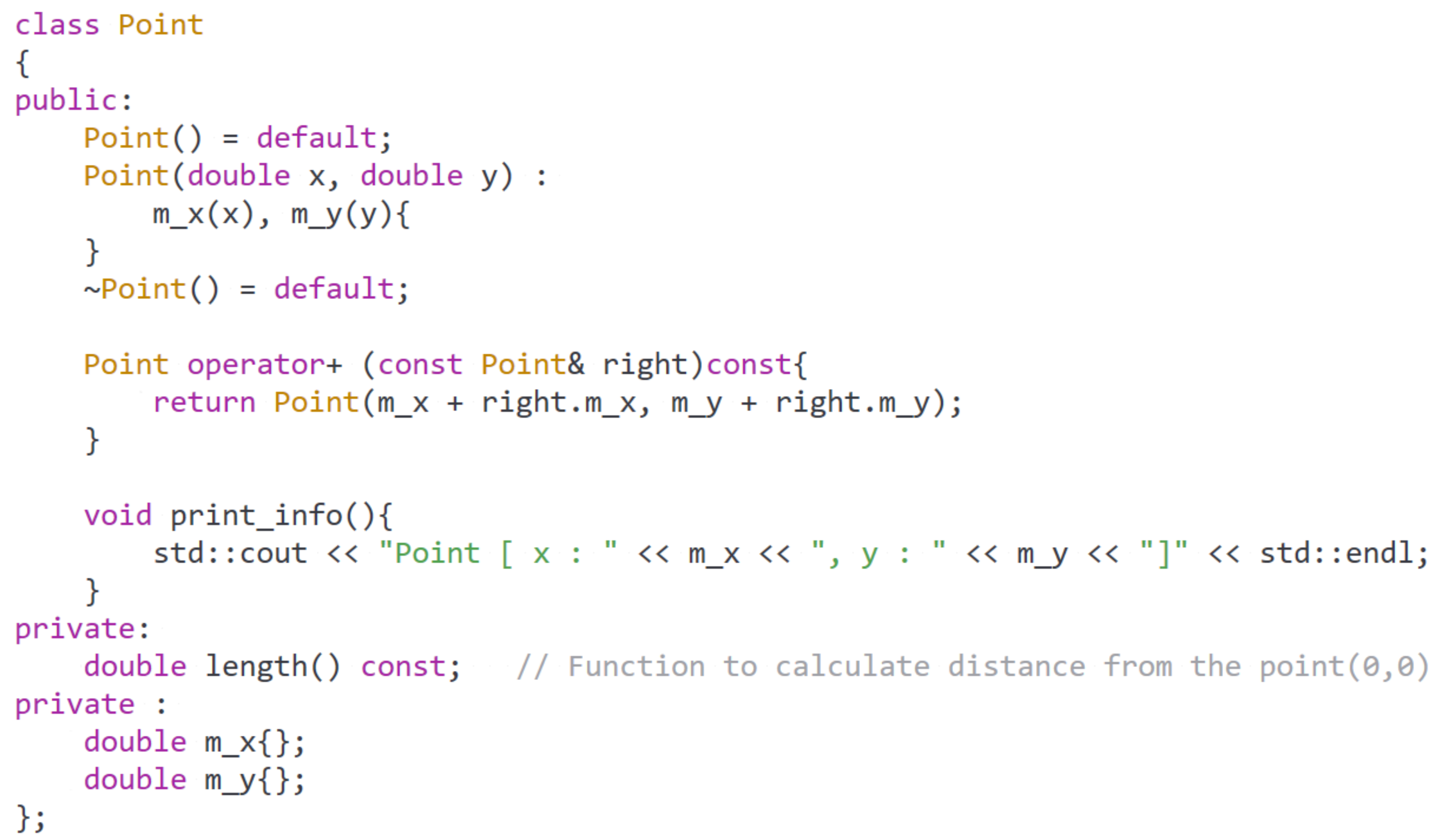


```
Point p1(10,10);
Point p2(20,20);
Point p3{p1 + p2};
Point p4{p2 + Point(5,5)};

p3.print_info();
p4.print_info();

(Point(20,20) + Point(10,10)).print_info();
```



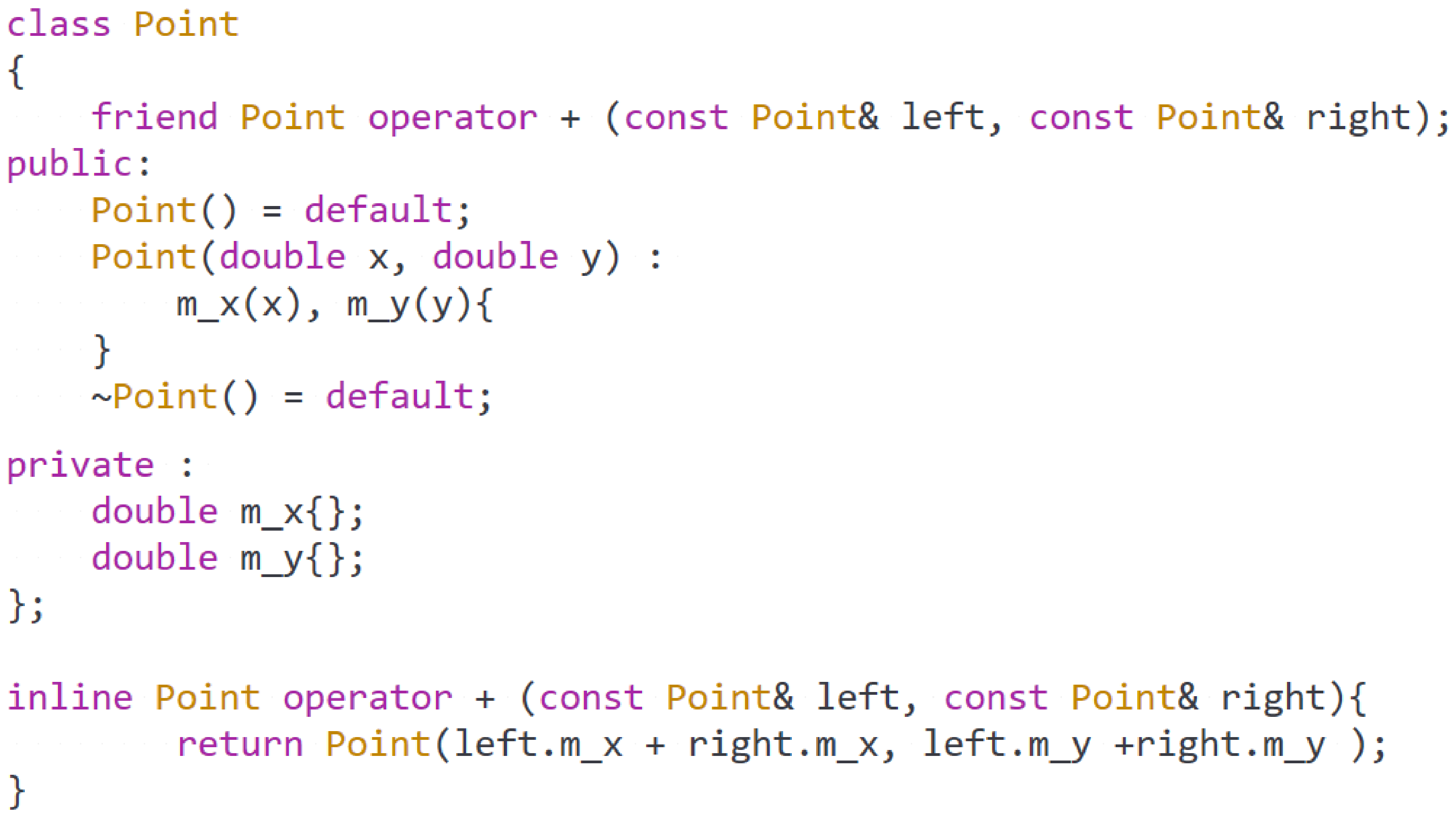
```
Unary :
        . member : ReturnType operator X ()
        . non member : ReturnType opearator X ( Type operand)

Binary :
        . member : ReturnType operator X (Type right_operand)
        . non-member : ReturnType operator X (Type left_operand, Type right_operand)
```

Operator+ as a member



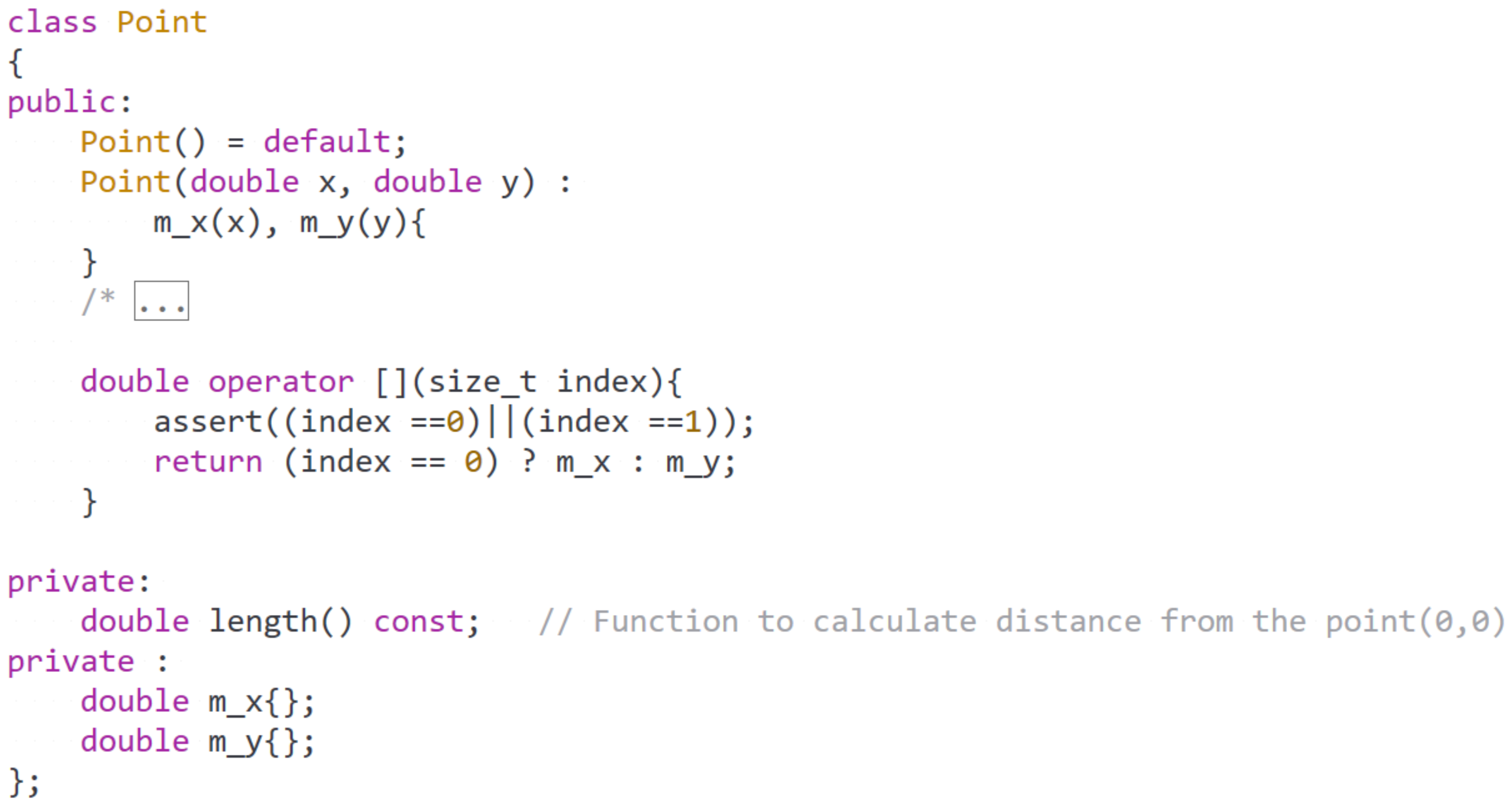
```cpp
class Point
{
public:
    Point() = default;
    Point(double x, double y) :
        m_x(x), m_y(y){
    }
    ~Point() = default;

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

    void print_info(){
        std::cout << "Point [ x : " << m_x << ", y : " << m_y << "]" << std::endl;
    }
private:
    double length() const;    // Function to calculate distance from the point(0,0)
private :
    double m_x{};
    double m_y{};
};
```

Operator+ as a non member



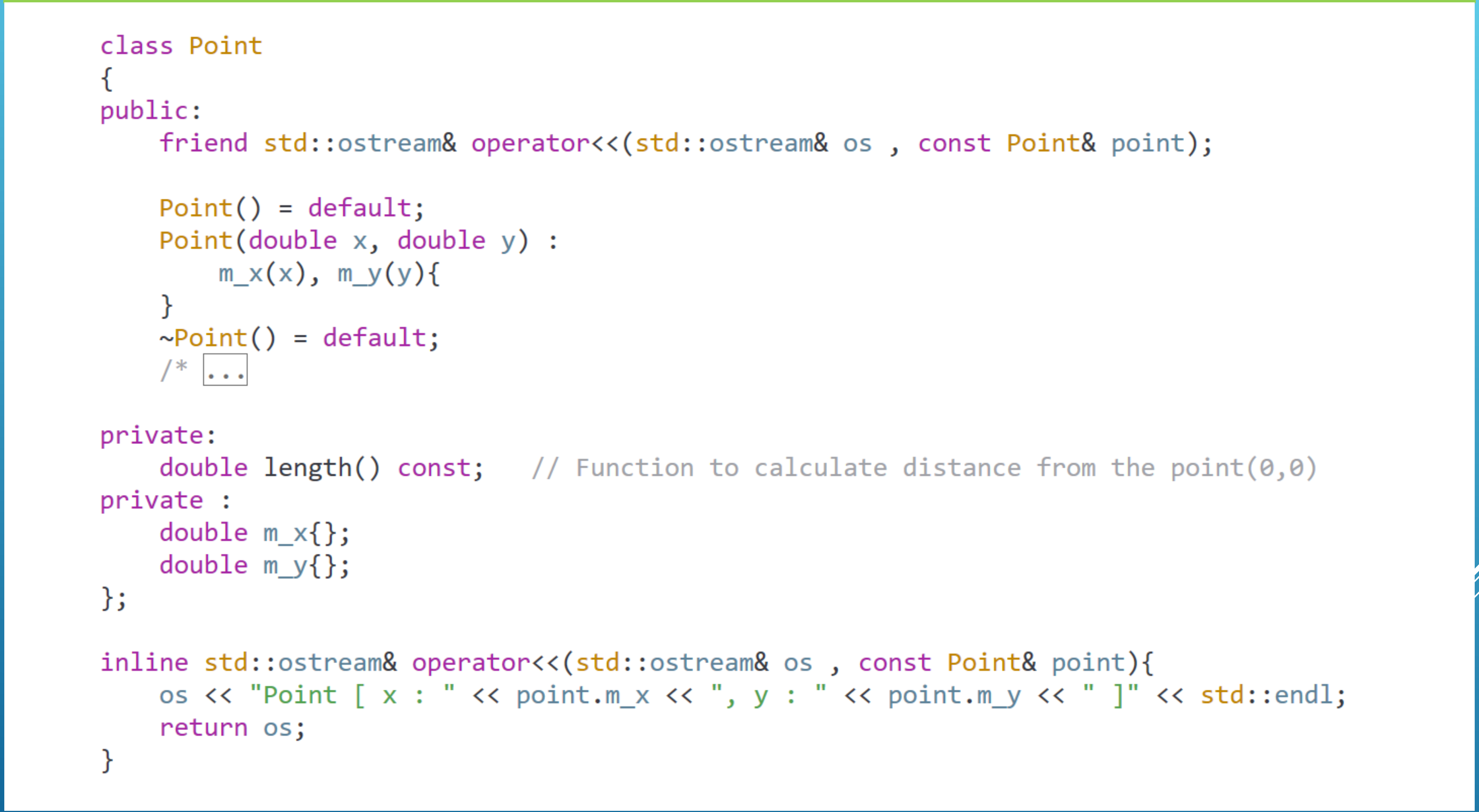
```
class Point
{
    friend Point operator + (const Point& left, const Point& right);
public:
    Point() = default;
    Point(double x, double y) :
        m_x(x), m_y(y){
    }
    ~Point() = default;

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 );
}
```

Subscript operator



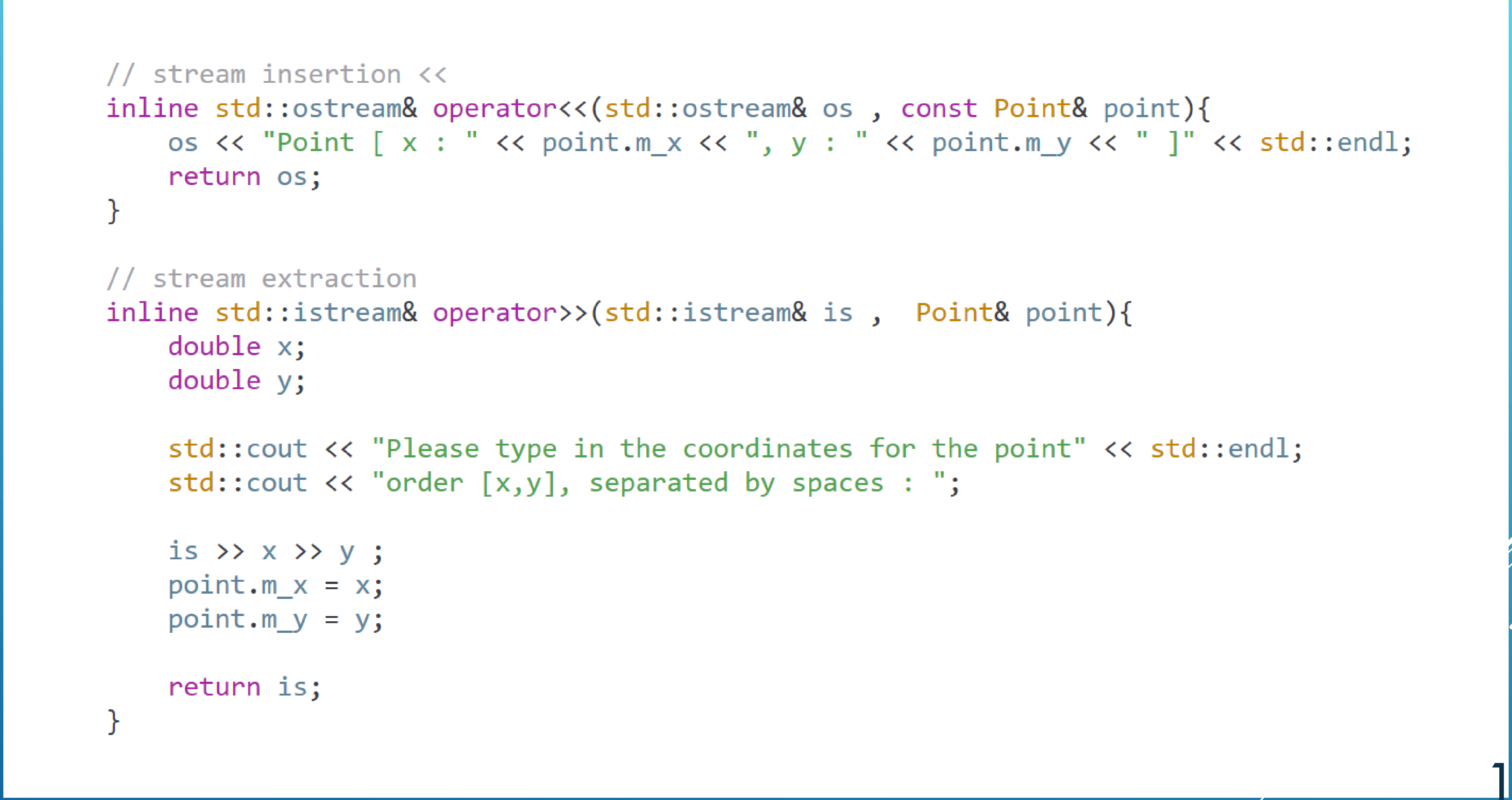
```
class Point
{
public:
    Point() = default;
    Point(double x, double y) :
        m_x(x), m_y(y){
    }
    /* ...

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

private:
    double length() const;   // Function to calculate distance from the point(0,0)
private :
    double m_x{};
    double m_y{};
};
```

Stream insertion operator



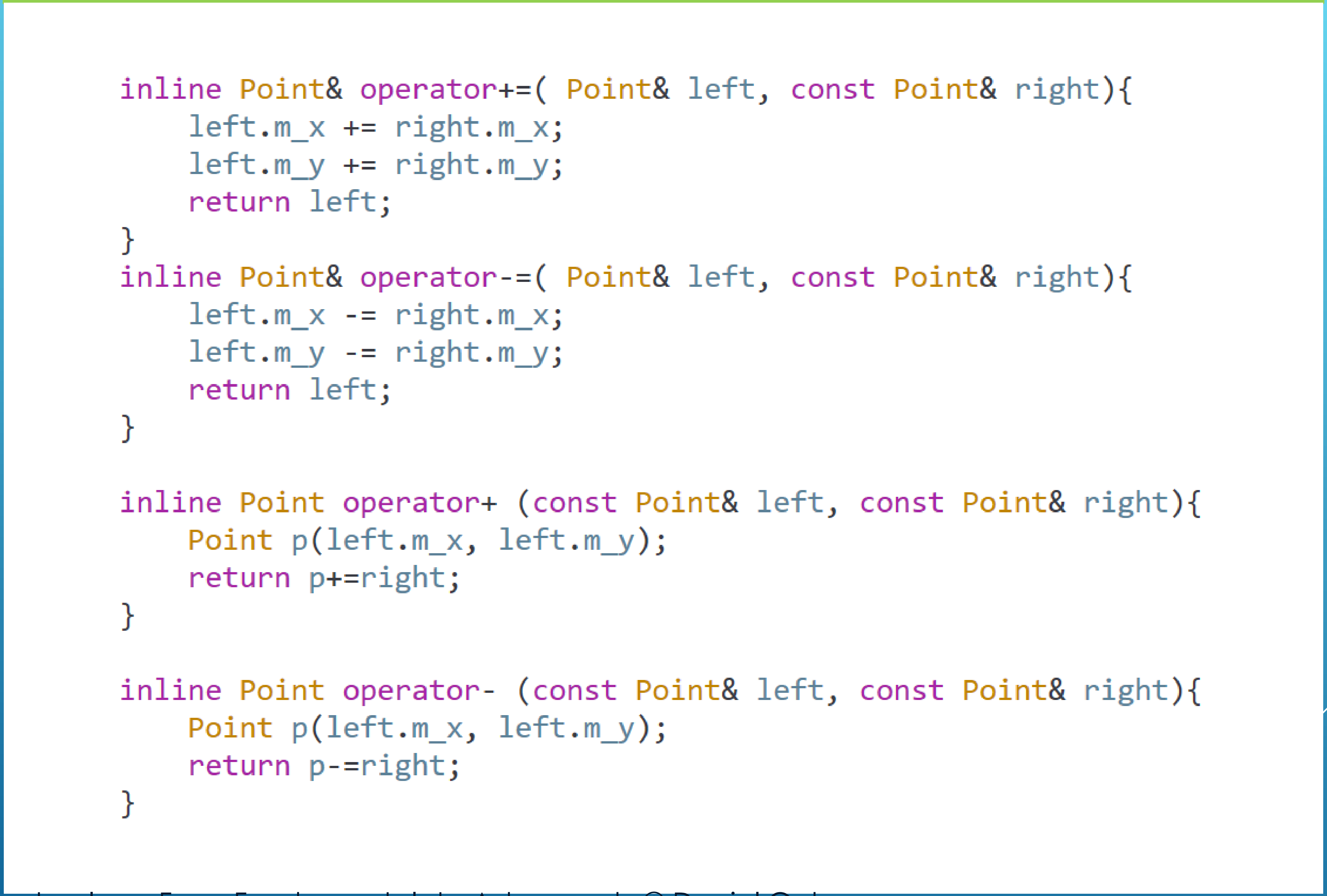
```cpp
class Point
{
public:
    friend std::ostream& operator<<(std::ostream& os , const Point& point);

    Point() = default;
    Point(double x, double y) :
        m_x(x), m_y(y){
    }
    ~Point() = default;
    /* ...

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

inline std::ostream& operator<<(std::ostream& os , const Point& point){
    os << "Point [ x : " << point.m_x << ", y : " << point.m_y << " ]" << std::endl;
    return os;
}
```

Stream extraction operator



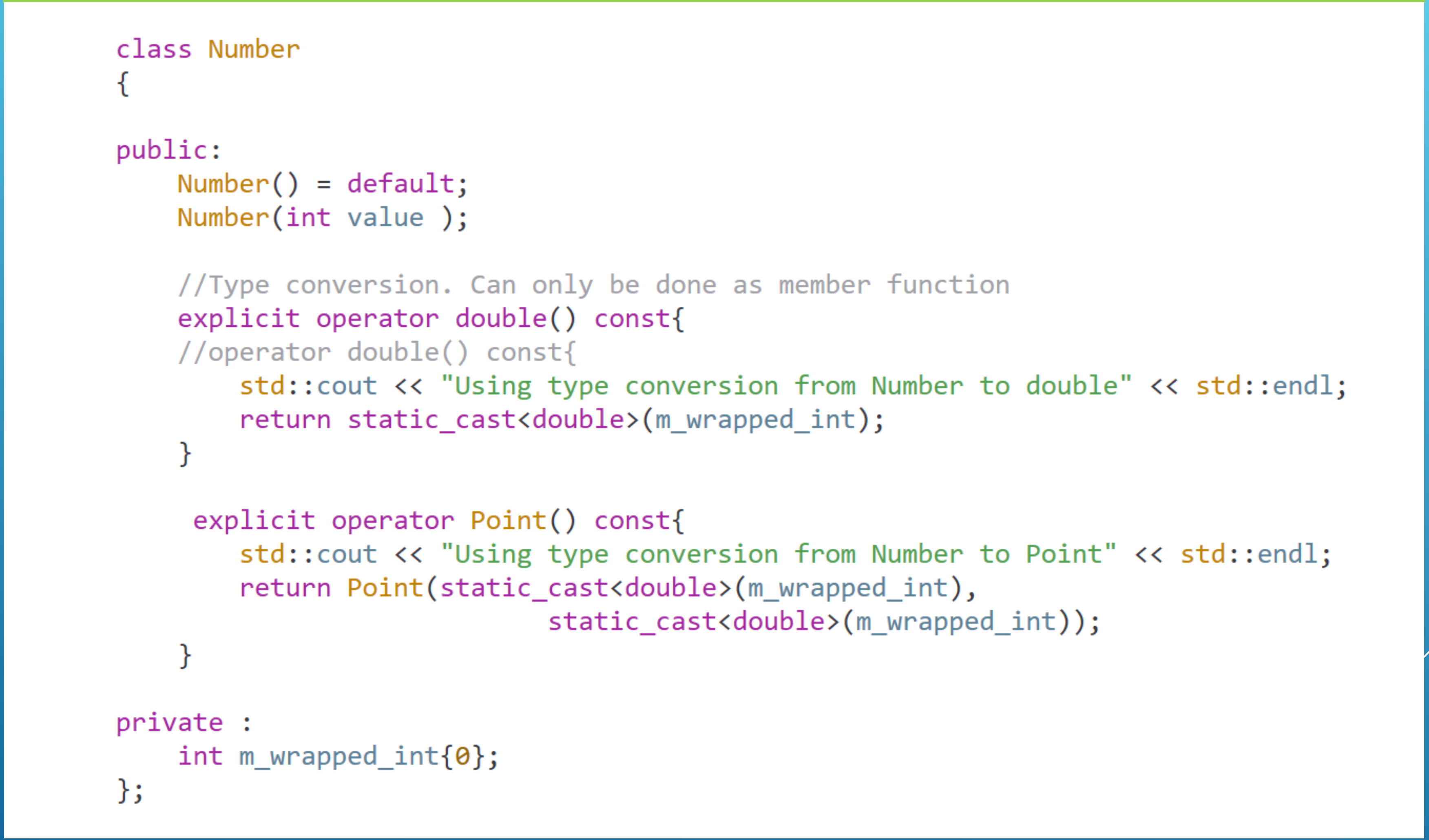
```cpp
// stream insertion <<
inline std::ostream& operator<<(std::ostream& os , const Point& point){
    os << "Point [ x : " << point.m_x << ", y : " << point.m_y << " ]" << std::endl;
    return os;
}

// stream extraction
inline std::istream& operator>>(std::istream& is ,  Point& point){
    double x;
    double y;

    std::cout << "Please type in the coordinates for the point" << std::endl;
    std::cout << "order [x,y], separated by spaces : ";

    is >> x >> y ;
    point.m_x = x;
    point.m_y = y;

    return is;
}
```

Compound operators and reusing others



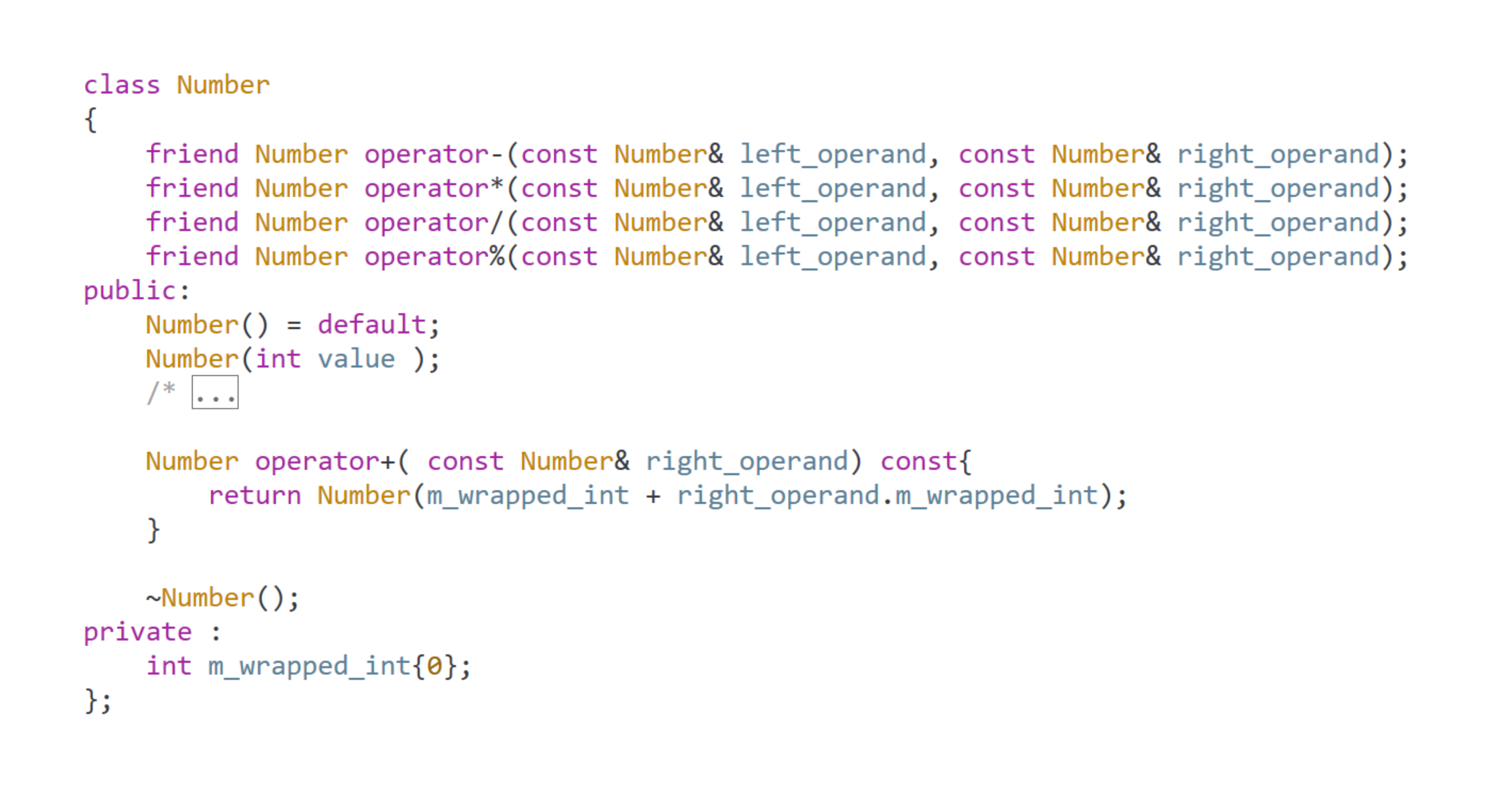
```
inline Point& operator+=( Point& left, const Point& right){
    left.m_x += right.m_x;
    left.m_y += right.m_y;
    return left;
}
inline Point& operator-=( Point& left, const Point& right){
    left.m_x -= right.m_x;
    left.m_y -= right.m_y;
    return left;
}

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

inline Point operator- (const Point& left, const Point& right){
    Point p(left.m_x, left.m_y);
    return p-=right;
}
```

Custom Type Conversions



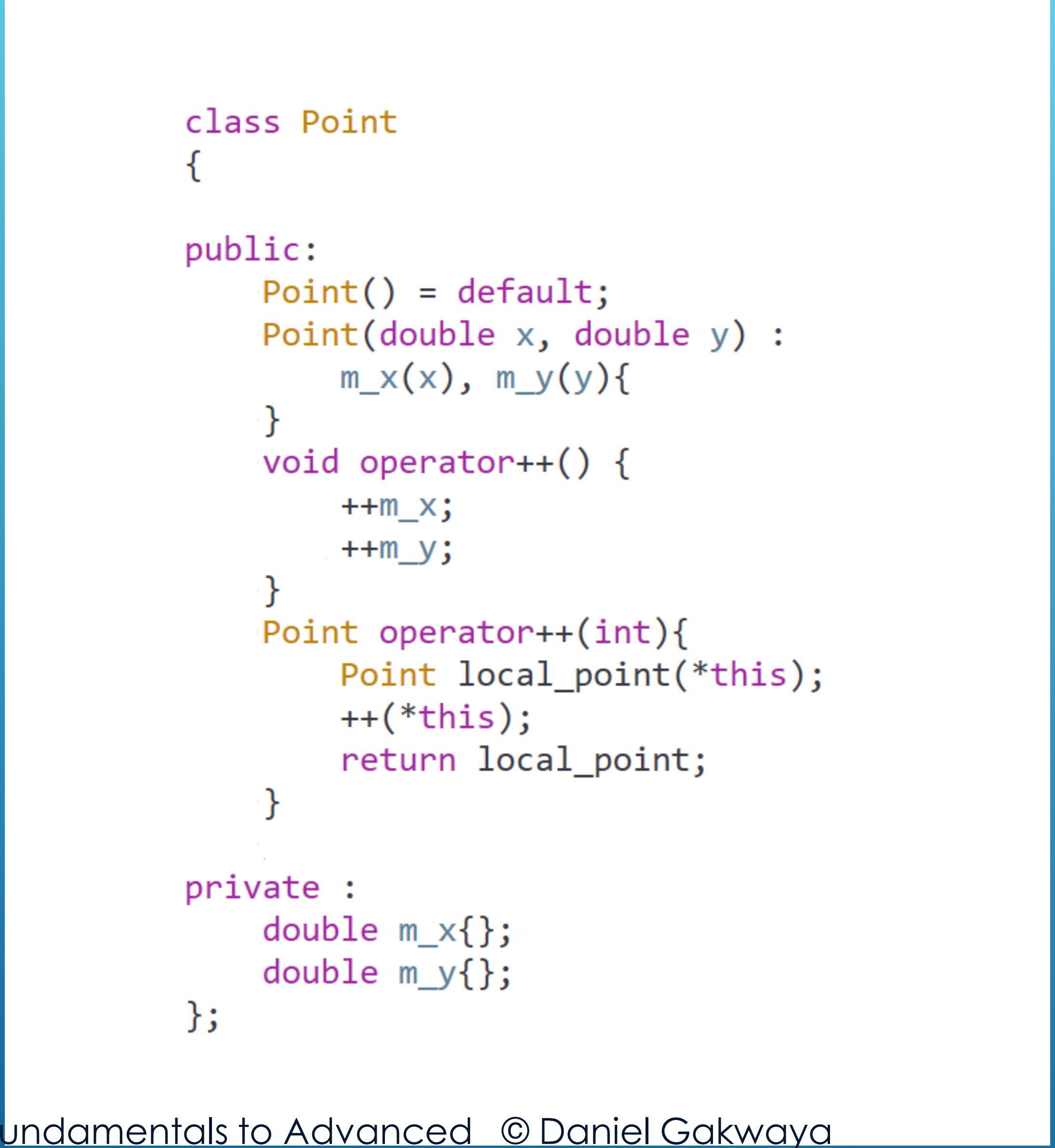
```cpp
class Number
{

public:
    Number() = default;
    Number(int value );

    //Type conversion. Can only be done as member function
    explicit operator double() const{
    //operator double() const{
        std::cout << "Using type conversion from Number to double" << std::endl;
        return static_cast<double>(m_wrapped_int);
    }

     explicit operator Point() const{
        std::cout << "Using type conversion from Number to Point" << std::endl;
        return Point(static_cast<double>(m_wrapped_int),
                            static_cast<double>(m_wrapped_int));
    }

private :
    int m_wrapped_int{0};
};
```

Implicit conversions with member binary operators



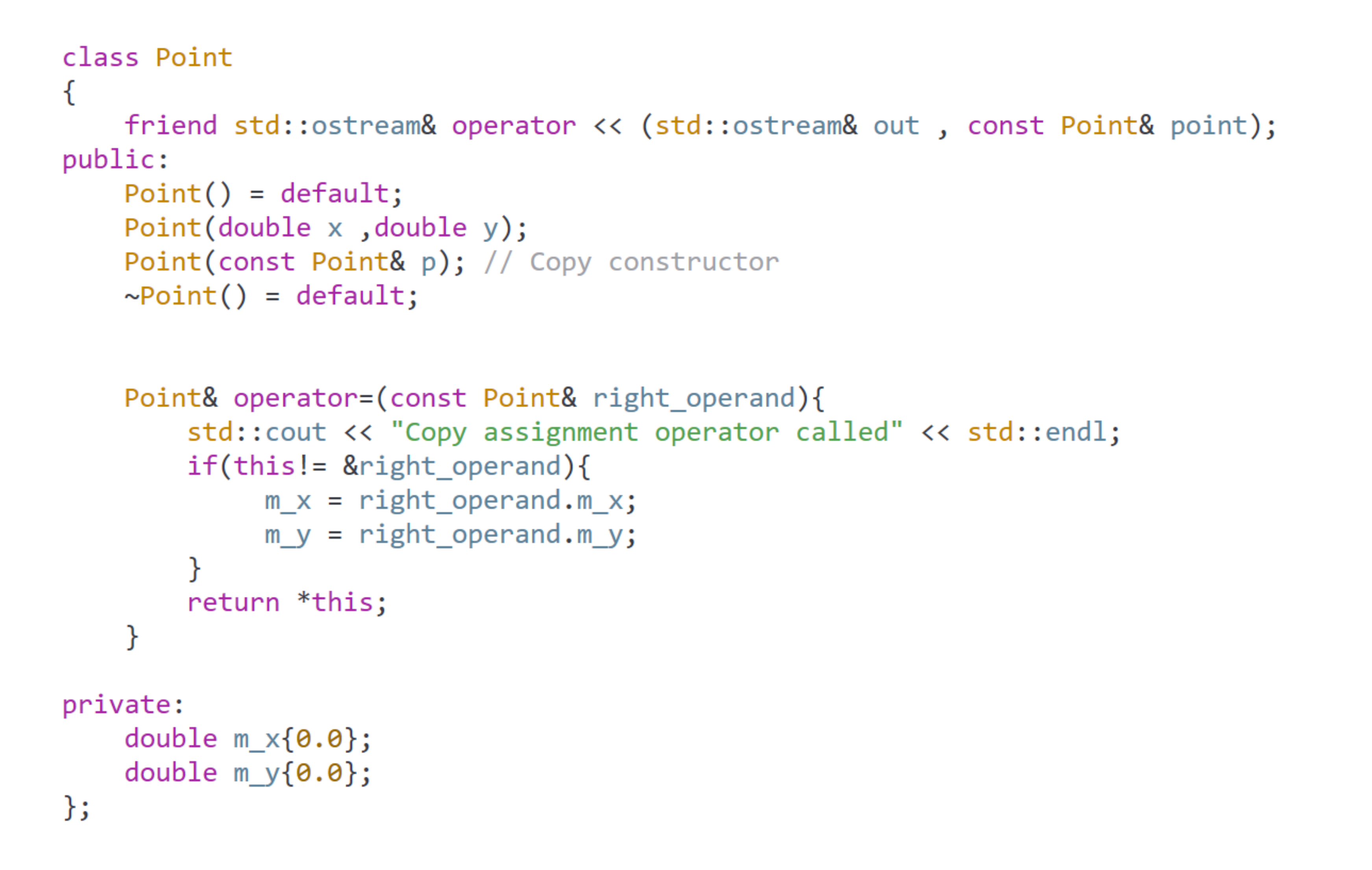
```cpp
class Number
{
    friend Number operator-(const Number& left_operand, const Number& right_operand);
    friend Number operator*(const Number& left_operand, const Number& right_operand);
    friend Number operator/(const Number& left_operand, const Number& right_operand);
    friend Number operator%(const Number& left_operand, const Number& right_operand);
public:
    Number() = default;
    Number(int value );
    /* ...

    Number operator+( const Number& right_operand) const{
        return Number(m_wrapped_int + right_operand.m_wrapped_int);
    }

    ~Number();
private :
    int m_wrapped_int{0};
};
```

Unary Increment operator



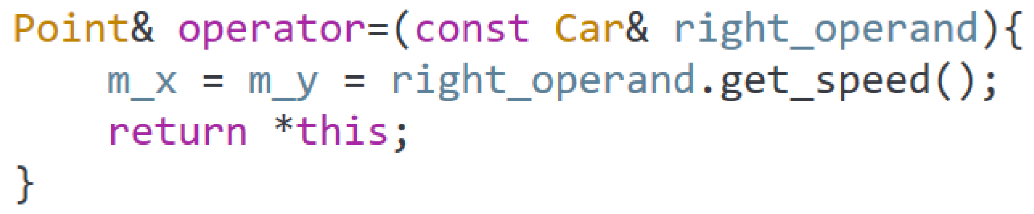
```cpp
class Point
{

public:
    Point() = default;
    Point(double x, double y) :
        m_x(x), m_y(y){
    }
    void operator++() {
        ++m_x;
        ++m_y;
    }
    Point operator++(int){
        Point local_point(*this);
        ++(*this);
        return local_point;
    }

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

Copy assignment operator



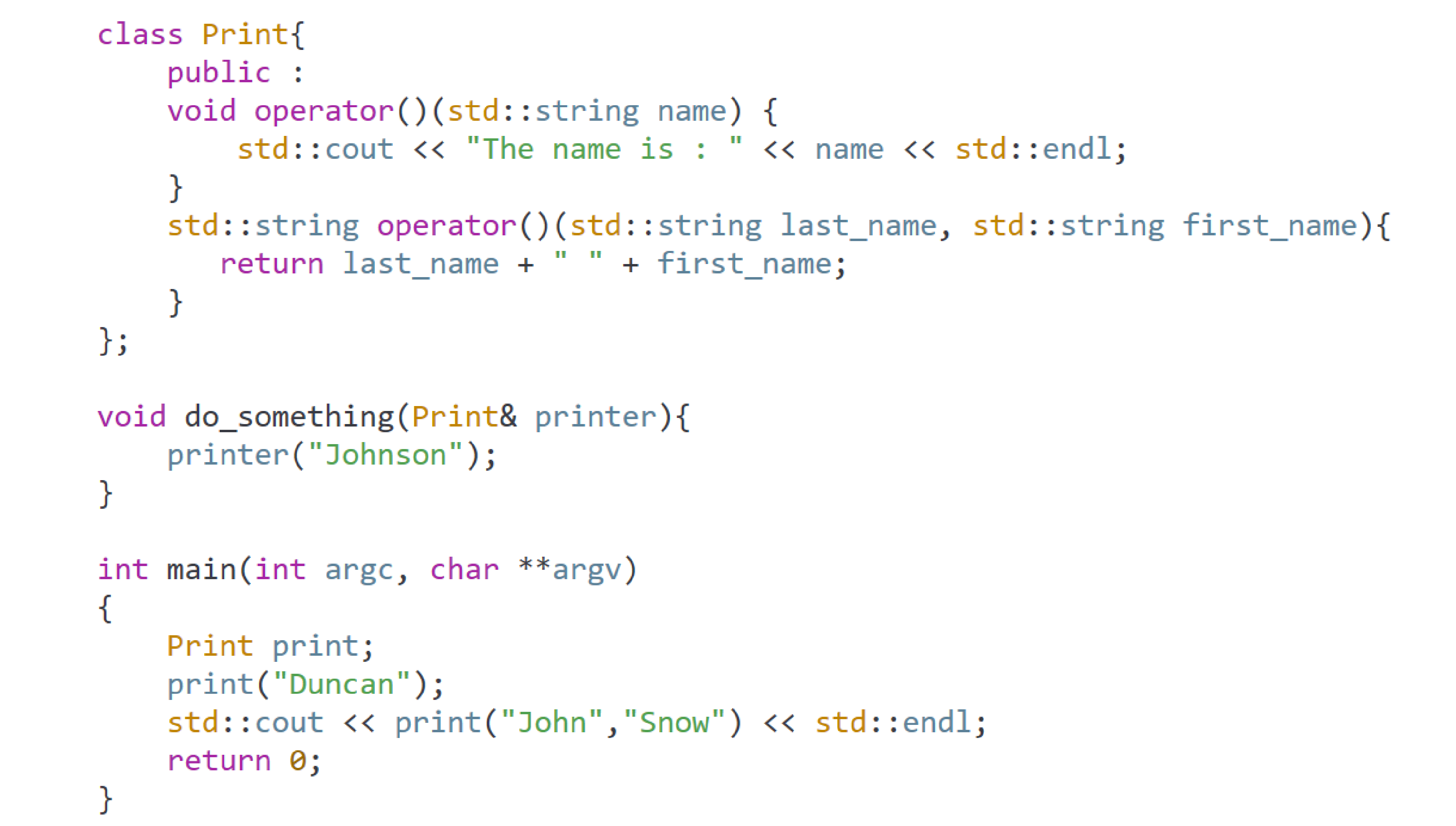
```
class Point
{
    friend std::ostream& operator << (std::ostream& out , const Point& point);
public:
    Point() = default;
    Point(double x ,double y);
    Point(const Point& p); // Copy constructor
    ~Point() = default;


    Point& operator=(const Point& right_operand){
        std::cout << "Copy assignment operator called" << std::endl;
        if(this!= &right_operand){
             m_x = right_operand.m_x;
             m_y = right_operand.m_y;
        }
        return *this;
    }

private:
    double m_x{0.0};
    double m_y{0.0};
};
```

Copy assignment for other types



```
Point& operator=(const Car& right_operand){
    m_x = m_y = right_operand.get_speed();
    return *this;
}
```

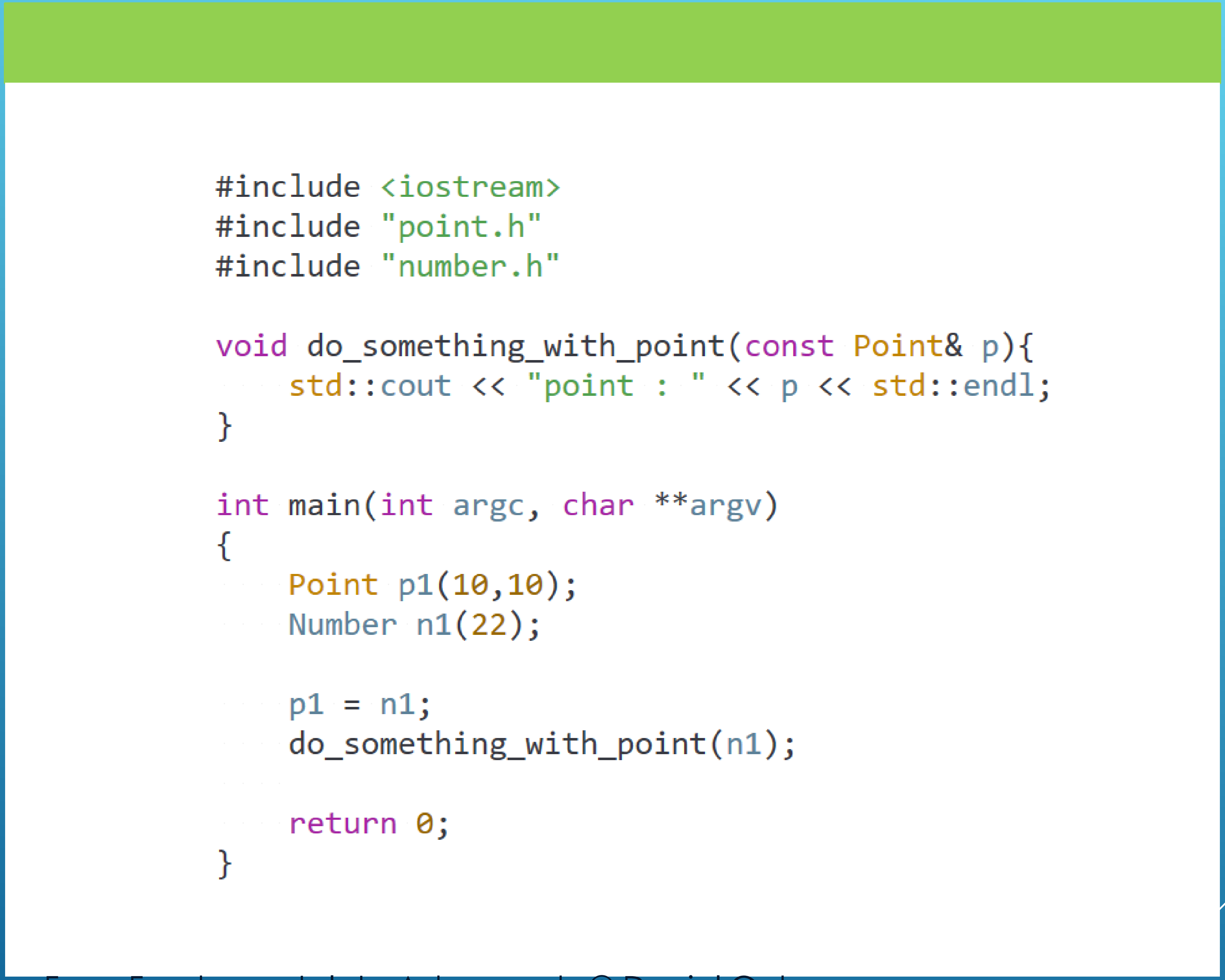
Functors



```
class Print{
    public :
    void operator()(std::string name) {
        std::cout << "The name is : " << name << std::endl;
    }
    std::string operator()(std::string last_name, std::string first_name){
       return last_name + " " + first_name;
    }
};

void do_something(Print& printer){
    printer("Johnson");
}

int main(int argc, char **argv)
{
    Print print;
    print("Duncan");
    std::cout << print("John","Snow") << std::endl;
    return 0;
}
```

Type conversions

- Constructors
- Custom Type Conversion Operators
- Copy assignment operators for different types



```cpp
#include <iostream>
#include "point.h"
#include "number.h"

void do_something_with_point(const Point& p){
    std::cout << "point : " << p << std::endl;
}

int main(int argc, char **argv)
{
    Point p1(10,10);
    Number n1(22);

    p1 = n1;
    do_something_with_point(n1);

    return 0;
}
```