Ch9: Classes, Operator Overloading

Classes

class and struct

While C++ provides a number of fundamental data types (e.g., char, int, long, float, double, etc.) that are often sufficient for solving relatively simple problems, it can be difficult to solve complex problems using just these types. One of C++'s more useful features is the ability to define your own data types (using enum, struct, and class) that better correspond to the problem being solved.

In C++, the main difference between struct and class is that struct has **public members** by default and class has **private members** by default.

```
struct DateStruct {
    int year {};
    int month {};
    int day {};
};
```

⟺

```
class DateClass {
public:
    int m_year {};
    int m_month {};
    int m_day {};
};
```

- By convention, class names should begin with an upper-case letter.

```
DateStruct today { 2020, 10, 14 };
```

```
DateClass today { 2020, 10, 14 };
```

Best Practice: Use the struct keyword for data-only structures. Use the class keyword for objects that have both data and functions.

Member Functions

In addition to holding data (member variables), classes (and structs) can also contain functions. Functions defined inside of a class are called **member functions** (or sometimes **methods**). Member functions can be defined inside or outside of the class definition. Just like members of a struct, members (member variable and member functions) of a class are accessed using the member selection operator ".".

Note: With normal non-member functions (i.e., global functions), a function cannot call a function that is defined "below" it, without a forward declaration. With member functions, this limitation does not apply.

Best Practice: Using the "m_" prefix for member variables helps distinguish member variables from function parameters or local variables inside member functions.

```
#include <iostream>
class DateClass {
public:
   int m_year {};
   int m_month {};
   int m_day {};
   void print()    {
      std::cout << m_year << '/' << m_month << '/' << m_day;
   }
};
int main() {
   DateClass today { 2023, 04, 01 };
   today.m_day = 16;
   today.print();
}
```

Access Controls Work on a Per-Class Basis

When a function has access to the private members of a class, it can access the private members of any object of that class type that it can see.

This can be particularly useful when we need to copy members from one object of a class to another object of the same class.

```
class DateClass {
public:
  void setDate(int month, int day, int year) {
    m_month = month;
    m_day = day;
    m_year = year;
  }
  void print() {
    std::cout << m_month << '/' << m_day << '/' << m_year;
  }
  void copyFrom(const DateClass& d)   {
    m_month = d.m_month;
    m_day = d.m_day;
    m_year = d.m_year;
  }
private:
  int m_month {};
  int m_day {};
  int m_year {};
};
```

```
#include <iostream>
#include "DateClass.h"

int main() {
  DateClass date;
  date.setDate(10, 14, 2020);

  DateClass copy {};
  copy.copyFrom(date);
  copy.print();
  std::cout << '\n';
}
```

Defining Member Functions Outside Class Definition

C++ provides a way to separate the "declaration" portion of the member functions and constructors from the "definition" portion. To do so, simply define the functions of the class as if they were normal functions outside the class but prefix the class name to the function using the scope resolution operator (::) (same as for a namespace). The prototypes of the functions still exist inside the class definition.

Date.h
```
class Date {
public:
    Date(int year, int month, int day);
    void setDate(int year, int month, int day);
    int getYear() { return m_year; }
    int getMonth() { return m_month; }
    int getDay()  { return m_day; }
private:
    int m_year;
    int m_month;
    int m_day;
};
```

Date.cpp
```
#include "Date.h"
// Date constructor
Date::Date(int year, int month, int day) {
    setDate(year, month, day);
}
// Date member function
void Date::setDate(int year, int month, int day) {
    m_month = month;
    m_day = day;
    m_year = year;
}
```

main.cpp
```
#include<iostream>
#include "Date.h"
int main(){
    Date d{2023, 4, 16};
    std::cout << "Date: " << d.getMonth() << "/" << d.getDay() << "/" << d.getYear();
}
```

Defining Member Functions Outside Class Definition (Another Example)

```cpp
#include<iostream>
class Calc {
public:
   Calc(int value=0): m_value{value} {}

   int add(int value) {
      m_value  += value;
      return m_value;
   }
   int sub(int value) {
      m_value -= value;
      return m_value;
   }
   int mult(int value) {
      m_value *= value;
      return m_value;
   }

   int getValue() { return m_value;  }
private:
   int m_value{};
};
```

constructor with a member initialization list

$\longrightarrow$

Calc.h

```cpp
class Calc {
public:
   Calc(int value=0);
   int add(int value);
   int sub(int value);
   int mult(int value);

   int getValue() {
      return m_value;
   }
private:
   int m_value{};
};
```

Calc.cpp

```cpp
#include "Calc.h"
Calc::Calc(int value): m_value{value} {}
int Calc::add(int value) {
   m_value  += value;
   return m_value;
}
int Calc::sub(int value) {
   m_value -= value;
   return m_value;
}
int Calc::mult(int value) {
   m_value *= value;
   return m_value;
}
```

main.cpp

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

int main(){
   Calc x{3};
   std::cout << x.add(3);
}
```

Anonymous Class Objects

In certain cases, we need an object only temporarily. This is done by creating objects like normal but omitting the variable name.

```
Cents cents1{6};
Cents cents2{8};
```

⇨

```
Cents{6};
Cents{8};
```

```
#include <iostream>
class Cents {
public:
    Cents(int cents) : m_cents {cents} {}
    int getCents() const { return m_cents; }
private:
    int m_cents{};
};
Cents add(const Cents& c1, const Cents& c2) {
    Cents sum{ c1.getCents() + c2.getCents() };
    return sum;
}
int main() {
    Cents cents1{ 6 };
    Cents cents2{ 8 };
    Cents sum{ add(cents1, cents2) };
    std::cout << "I have " << sum.getCents() << " cents.\n";
}
```

⇨

```
#include <iostream>
class Cents {
public:
    Cents(int cents) : m_cents { cents } {}
    int getCents() const { return m_cents; }
private:
    int m_cents{};
};
Cents add(const Cents& c1, const Cents& c2) {
    return { c1.getCents() + c2.getCents() };
}
int main() {
    std::cout << "I have “
      << add(Cents{ 6 }, Cents{ 8 }).getCents()
       << " cents.\n";
}
```

Constructors

Constructors

When all members of a class (or struct) are public, we can initialize the them directly using list-initialization. However, as soon as we make any member variables private, we cannot initialize classes (or structs) in this way, and we must use **Constructors**.

A constructor is a special kind of class member function that is automatically called when an object of that class is created. Unlike normal member functions, constructors must have the same name as the class and no return type (not even void).

C++ allows more than one constructor, as long as they have different signatures.

Default Constructors

A constructor that takes no parameters (or has parameters that all have default values) is called a default constructor. The default constructor is called if no user-provided initialization values are provided.

```
#include <iostream>
class Fraction {
public:
    Fraction() { // default constructor
        m_numerator = 0;
        m_denominator = 1;
    }
    int getNumerator() { return m_numerator; }
    int getDenominator() { return m_denominator; }
    double getValue() { return static_cast<double>(m_numerator) / m_denominator; }
private:
    int m_numerator {};
    int m_denominator {};
};
int main() {
    Fraction frac{}; // calls Fraction() default constructor; or "Fraction frac;"" (not preferred)
    std::cout << frac.getNumerator() << '/' << frac.getDenominator() << '\n';
}
```

Parameterized Constructors

Constructors can also be declared with parameters to initialize member variables. You can define as many constructors as you want, as long as each has a unique signature (due to function overloading)

```
#include <iostream>
class Fraction {
public:
   Fraction() { // default constructor
       m_numerator = 0;
       m_denominator = 1;
   }
   // Constructor with two parameters, one parameter having a default value
   Fraction(int numerator, int denominator=1) {
      m_numerator = numerator;
      m_denominator = denominator;
   }
   int getNumerator() { return m_numerator; }
   int getDenominator() { return m_denominator; }
   double getValue() { return static_cast<double>(m_numerator) / m_denominator; }
private:
   int m_numerator {};
   int m_denominator {};
};
```

```
int main() {
   Fraction fiveThirds{ 5, 3 }; // List initialization, calls Fraction(int, int)
   Fraction six{ 6 }; // calls Fraction(int, int) constructor, second parameter uses default value of 1
}
```

Reducing Constructors

The rules around defining and calling functions that have default parameters apply to constructors too.

```
#include <iostream>
class Fraction {
public:
   // Default constructor
   Fraction(int numerator=0, int denominator=1) {
      m_numerator = numerator;
      m_denominator = denominator;
   }
   int getNumerator() { return m_numerator; }
   int getDenominator() { return m_denominator; }
   double getValue() { return static_cast<double>(m_numerator) / m_denominator; }
private:
   int m_numerator {};
   int m_denominator {};
};
int main() {
   Fraction zero{}; // will call Fraction(0, 1) or "Fraction zero;"
   Fraction six{ 6 }; // will call Fraction(6, 1)
   Fraction fiveThirds{ 5, 3 }; // will call Fraction(5, 3)
}
```

Implicit Constructor

- If your class has no constructors, C++ will automatically generate a public default constructor called an **implicit constructor**.

```
class Date {
private:
    int m_year{};
    int m_month{};
    int m_day{};
    // No user-provided constructors,
    // the compiler generates a default constructor.
};
int main() {
    Date date{};
}
```

- If your class has any other constructors, the implicitly generated constructor will not be provided.

```
class Date {
public:
    Date(int year, int month, int day) {
        m_year = year;
        m_month = month;
        m_day = day;
    }
    // No implicit constructor provided because we already defined our own constructor
private:
    int m_year{};
    int m_month{};
    int m_day{};
};
int main() {
    Date date{}; // error: default constructor doesn't exist, and the compiler won't generate one
    Date today{ 2023, 1, 1 }; // today is initialized to Jan 1, 2023
}
```

Implicit Constructor (cont.)

If your class has another constructor and you want to allow default construction, you can (1) explicitly define a default constructor, (2) add default arguments to every parameter of a constructor with parameters, or (3) use the **default** keyword to tell the compiler to create a default constructor.

```
class Date {
public:
    // Tell the compiler to create a default constructor, even if there are other constructors.
    Date() = default;
    Date(int year, int month, int day) {
        m_year = year;
        m_month = month;
        m_day = day;
    }
private:
    int m_year;
    int m_month;
    int m_day;
};
int main() {
    Date date{}; // date is zero-initialized
    Date today{ 2020, 10, 14 }; // today is initialized to Oct 14th, 2020
}
```

Classes Containing Class Members

A class may contain other class objects as member variables. By default, when the outer class is constructed, the member variables will have their default constructors called. This happens before the body of the constructor executes.

```
#include <iostream>
class A {
public:
    A() {
        std::cout << "A\n";
        }
};
class B {
public:
    B() {
        std::cout << "B\n";
        }
private:
    A m_a; // B contains A as a member variable
};
int main() {
    B b;
}
```

Initializing const Member Variables

We can initialize class member variables in the constructor using the assignment operator. However, some types of data (e.g., const and reference variables) must be initialized on the line they are declared.

```
class Something {
public:
    Something() {
        m_value = 1; // error: const vars can not be assigned to
    }
private:
    const int m_value;
};
```

To solve this problem, C++ provides a method for initializing member variables (rather than assigning values to them after they are created) via a member initializer list.

Initializing const Member Variables

- **Note**: You can use default parameters to provide a default value in case the user did not pass one in.

```
#include <iostream>
class MyClass {
public:
   MyClass(int a, double b, char c='c'):m_a{a}, m_b{b}, m_c{c} {
   // No need for assignment here
   }
   void print() {
       std::cout << "MyClass(" << m_a << ", " << m_b << ", " << m_c << ")\n";
   }
private:
   const int m_a {};
   double m_b {};
   char m_c {};
};
int main() {
   std::cout << "Enter an integer: ";
   int x{};
   std::cin >> x;
   MyClass myObject{ x, 2.2 }; // a = 1, b=2.2, c gets default value 'c'
   myObject.print();
}
```

Initializing Member Variable That Are Classes

Using member initializer lists, we can initialize member variable that are classes.

```
#include <iostream>

class A {
public:
   A(int x = 0) { std::cout << "A: " << x << '\n'; }
};

class B {
private:
   A m_a {};
public:
   B(int y):m_a{ y - 1 } { // call A(int) constructor to initialize member m_a
      std::cout << "B: " << y << '\n';
   }
};

int main() {
   B b{ 5 };
}
```

Best practice: In general, use member initializer lists to initialize your member variables instead of assignment.

Member Initialization

When writing a class that has multiple constructors, having to specify default values for all members in each constructor results in redundant code. It is possible to give class member variables (those that don’t use the static keyword) a default initialization value directly.

```
#include <iostream>
class Rectangle {
public:
    void print() {
        std::cout << "length: " << m_length << ", width: " << m_width << '\n';
    }
private:
    double m_length{ 1.0 }; // m_length has a default value of 1.0
    double m_width{ 1.0 }; // m_width has a default value of 1.0
};

int main() {
    Rectangle x{}; // x.m_length = 1.0, x.m_width = 1.0
    x.print();
}
```

Member Initialization (cont.)

```
#include <iostream>
class Rectangle {
public:
   Rectangle(double length, double width):m_length{length}, m_width{width} {
      // m_length and m_width are initialized by the constructor
   }
   Rectangle(double length):m_length{length} {
      // m_length is initialized by the constructor.
      // m_width's default value (1.0) is used.
   }
   Rectangle() = default;
   void print() {
      std::cout << "length: " << m_length << ", width: " << m_width << '\n';
   }
private:
   double m_length{1.0};
   double m_width{1.0};
};
```

Best Practice: Initialize all member variables on creation of the object via either a constructor or default member initialization.

```
int main() {
   Rectangle x{2.0, 3.0};
   x.print();
   Rectangle y{4.0};
   y.print();
   Rectangle z{}; // There will be an error without default constructor "Rectangle() = default;"
   z.print();
}
```

Overlapping and Delegating Constructors

Assume that you have a class with multiple constructors that have overlapping functionality. However, having duplicate code is something to be avoided as much as possible

```
class Foo {
public:
   Foo() {
      // code to do A
   }
   Foo(int value)    {
      // code to do A
      // code to do B
   }
};
```

Two method for the issue:

1) Delegating constructors
2) Using a member function for setup

1) Delegating Constructors

Constructors are allowed to call other constructors from the same class from the member initializer list. This process is called **Delegating Constructors**.

```
class Foo {
public:
    Foo() {
        // code to do A
    }

    Foo(int value): Foo{} {
        // code to do B
    }

};
```

```
#include <iostream>
#include <string>
class Employee {
public:
    Employee(int id=0, std::string name=""):m_id{id}, m_name{name} {
        std::cout << "Employee " << m_name << " created.\n";
    }
    // Use a delegating constructor to minimize redundant code
    Employee(std::string name):Employee{0, name} {
    }
private:
    int m_id{};
    std::string m_name{};
};
```

Note: A constructor that delegates to another constructor is not allowed to do any member initialization itself. Thus, your constructors can delegate or initialize, but not both.

2) Using a Member Function for Setup

In this method, we create a private setup() member function to handle various setup tasks that we need, and both of our constructors call setup().

```
#include <iostream>
class Foo {
public:
    Foo() {
        setup();
    }
    Foo(int value) : m_value { value } { // we must initialize m_value since it's const
        setup();
    }
private:
    const int m_value {0};
    void setup() { // setup is private so it can only be used by our constructors
        // code to do some common setup tasks (e.g., open a file or database)
        std::cout << "Setting things up...\n";
    }
};
int main() {
    Foo a;
    Foo b{ 5 };
}
```

Destructors

A destructor is another special kind of class member function that is executed when an object of that class is destroyed. Whereas constructors are designed to initialize a class, destructors are designed to help clean up.

- The destructor must have the same name as the class, preceded by a tilde (~).
- The destructor can not take arguments.
- The destructor has no return type.
- A class can only have a single destructor.

```
#include<iostream>
class Test {
public:
  Test() {
    std::cout<<"\n Constructor executed";
  }
  ~Test() {
    std::cout<<"\n Destructor executed";
  }
};
int main() {
  Test t;
}
```

Const Class Objects & Member Functions

Const Class Objects

Instantiated class objects can be made const by using the const keyword. Initialization is done via class constructors (default or parameterized). Once a const class object has been initialized via constructor, any attempt to modify the member variables of the object is disallowed.

```
class Something {
public:
    Something(): m_value{0} { }
    void setValue(int value) { m_value = value; }
    int getValue() { return m_value ; }

private:
    int m_value {};
};

int main() {
    const Something something{}; // calls default constructor
    something.setValue(5); // compiler error: violates const
}
```

Const Member Functions

A const member function is a member function that guarantees it will not modify the object or call any non-const member functions (as they may modify the object). To make a const member function, we simply append the const keyword to the function prototype, after the parameter list, but before the function body.

```
class Something {
public:
   Something(): m_value{0} { }
   void resetValue() { m_value = 0; }
   void setValue(int value) { m_value = value; }
   int getValue() const { return m_value; }
private:
   int m_value {};
};
```

Note: Constructors cannot be marked as const. This is because constructors need to be able to initialize their member variables.

```
class Something {
public:
   Something(): m_value{0} { }
   void resetValue() { m_value = 0; }
   void setValue(int value) { m_value = value; }
   int getValue() const;
private:
   int m_value {};
};

int Something::getValue() const {
   return m_value;
}
```

For member functions defined outside of the class definition, the const keyword must be used on both the function prototype in the class definition and on the function definition.

Const Objects via Pass-by-Const-Reference

```
#include<iostream>
class Date {
public:
   Date(int year, int month, int day) {
      setDate(year, month, day);
   }
   void setDate(int year, int month, int day) {
      m_year = year;
      m_month = month;
      m_day = day;
   }
   int getYear() const { return m_year; }
   int getMonth() const { return m_month; }
   int getDay() const { return m_day; }
private:
   int m_year {};
   int m_month {};
   int m_day {};
};
```

```
void printDate(const Date& date) {
   std::cout << date.getYear() << '/'
    << date.getMonth() << '/'
    << date.getDay() << '\n';
}

int main() {
   Date date{2024, 4, 23};
   printDate(date);
}
```

Note: We cannot call non-const member functions on const objects. Thus, in this example, since date is treated as a const object, we must make getYear(), getMonth(), and getDay() const.

Operator Overloading

Friend Functions

A **friend function** is a function that can access the private members of a class as though it was a member of that class. In all other regards, the friend function is just like a normal function. To declare a friend function, simply use the friend keyword in front of the prototype of the function you wish to be a friend of the class.

- Declaration of the friend function can be done in the private or public section of the class.

- A friend function may be either a normal function, or a member function of another class.

```
#include <iostream>
class Value {
public:
   Value(int value) : m_value{ value } {}
   friend bool isEqual(const Value& value1, const Value& value2);
private:
   int m_value{};
};
bool isEqual(const Value& value1, const Value& value2) {
   return (value1.m_value == value2.m_value);
}
int main() {
   Value v1{ 5 };
   Value v2{ 6 };
   std::cout << std::boolalpha << isEqual(v1, v2);
}
```

Friend Functions: Multiple Friends

A function can be a friend of more than one class at the same time.

This is a **class prototype** that tells the compiler that we are going to define a class called Humidity in the future.

```
#include <iostream>
class Humidity;
class Temperature {
public:
    Temperature(int temp=0) : m_temp {temp} { }
    friend void printWeather(const Temperature& temperature, const Humidity& humidity);
private:
    int m_temp {};
};
class Humidity {
public:
    Humidity(int humidity=0) : m_humidity {humidity} { }
    friend void printWeather(const Temperature& temperature, const Humidity& humidity);
private:
    int m_humidity {};
};
void printWeather(const Temperature& temperature, const Humidity& humidity) {
    std::cout << "The temperature is " << temperature.m_temp <<
       " and the humidity is " << humidity.m_humidity << '\n';
}
int main() {
    Humidity hum{10};
    Temperature temp{12};
    printWeather(temp, hum);
}
```

Friend Classes

It is also possible to make an entire class a friend of another class. This gives all of the member functions of the friend class access to the private members of the other class.

- If class A is a friend of B, that does not mean B is also a friend of A.

- If class A is a friend of B, and B is a friend of C, that does not mean A is a friend of C.

```
#include <iostream>
class Storage {
public:
    Storage(int nValue, double dValue) : m_nValue{nValue}, m_dValue{dValue} {}
    friend class Display; // Make the Display class a friend of Storage
private:
    int m_nValue {};
    double m_dValue {};
};
class Display {
public:
    Display(bool displayIntFirst) : m_displayIntFirst{displayIntFirst} {}
    void displayItem(const Storage& storage) {
        if (m_displayIntFirst)
            std::cout << storage.m_nValue << ' ' << storage.m_dValue << '\n';
        else // display double first
            std::cout << storage.m_dValue << ' ' << storage.m_nValue << '\n';
    }
private:
    bool m_displayIntFirst;
};
int main() {
    Storage storage{5, 6.7};
    Display display{false};
    display.displayItem(storage);
}
```

Operator Overloading

In C++, operators are implemented as functions. By using function overloading on the operator functions, you can define your own versions of the operators that work with different data types (including classes that you have written).

x + y ≡ operator+(x, y)
x - y ≡ operator-(x, y)
x * y ≡ operator*(x, y)
x / y ≡ operator/(x, y)
std::cout << x ≡ operator<<(std::ostream& out, x)
std::cin >> x ≡ operator>>(std::istream& in, x)

Overloading Operators Using Friend Functions

```
#include <iostream>
class Cents {
public:
  Cents(int cents) : m_cents{ cents } { }
  friend Cents operator+(const Cents& c1, const Cents& c2);
  friend Cents operator-(const Cents& c1, const Cents& c2);
  int getCents() const { return m_cents; }
private:
  int m_cents {};
};
Cents operator+(const Cents& c1, const Cents& c2) {
  return c1.m_cents + c2.m_cents;
}
Cents operator-(const Cents& c1, const Cents& c2) {
  return c1.m_cents - c2.m_cents;
}
int main() {
  Cents cents1{ 6 };
  Cents cents2{ 2 };
  Cents centsSum{ cents1 + cents2 };
  std::cout << "I have " << centsSum.getCents() << " cents.\n";
}
```

Overloading the multiplication operator (*) and the division operator (/) is as easy as defining functions for operator* and operator/, respectively.

Overloading Operators for Operands of Different Types

Whenever we overload binary operators for operands of different types, we actually need to write two functions, one for each case.

```
#include <iostream>
class Cents {
public:
  Cents(int cents) : m_cents{ cents } { }
  friend Cents operator+(const Cents& c1, int value);
  friend Cents operator+(int value, const Cents& c1);
  int getCents() const { return m_cents; }
private:
  int m_cents {};
};
Cents operator+(const Cents& c1, int value) {
  return c1.m_cents + value;
}
Cents operator+(int value, const Cents& c1) {
  return c1.m_cents + value; // or "return c1 + value;" which calls
                             // operator+(Cents, int), the other overloaded operator
}
int main() {
  Cents c1{ Cents{ 4 } + 6 };
  Cents c2{ 6 + Cents{ 4 } };
  std::cout << "I have " << c1.getCents() << " cents.\n";
  std::cout << "I have " << c2.getCents() << " cents.\n";
}
```

Overloading Operators Using Normal Functions

```
#include <iostream>

class Cents {
public:
   Cents(int cents) : m_cents{ cents } { }
   int getCents() const { return m_cents; }
private:
   int m_cents {};
};

Cents operator+(const Cents& c1, const Cents& c2) {
  // we don't need direct access to private members here
  return Cents{ c1.getCents() + c2.getCents() };
}

int main() {
  Cents cents1{ 6 };
  Cents cents2{ 8 };
  Cents centsSum{ cents1 + cents2 };
  std::cout << "I have " << centsSum.getCents() << " cents.\n";
}
```

Similarly, we can overload operators (-), (*), and (/).

Overloading Operator <<

```
#include <iostream>

class Point {
public:
   Point(double x=0.0, double y=0.0, double z=0.0) : m_x{x}, m_y{y}, m_z{z} {}
   friend std::ostream& operator<< (std::ostream& out, const Point& point);
private:
   double m_x{};
   double m_y{};
   double m_z{};
};

// std::ostream is the type for object std::cout
std::ostream& operator<< (std::ostream& out, const Point& point) {
   out << "Point(" << point.m_x << ", " << point.m_y << ", " << point.m_z << ')';
   return out; // return std::ostream so we can chain calls to operator<<
}

int main() {
   Point point1{2.0, 3.5, 4.0};
   Point point2{6.0, 7.5, 8.0};
   std::cout << point1 << '\n' << point2 << '\n';
}
```

Any time we want our overloaded binary operators to be chainable, the left operand should be returned (by reference).

Overloading Operator >>

```
#include <iostream>
class Point {
public:
    Point(double x=0.0, double y=0.0, double z=0.0) : m_x{x}, m_y{y}, m_z{z} {}
    friend std::ostream& operator<< (std::ostream& out, const Point& point);
    friend std::istream& operator>> (std::istream& in, Point& point);
private:
    double m_x{};
    double m_y{};
    double m_z{};
};
// std::ostream is the type for object std::cout
std::ostream& operator<< (std::ostream& out, const Point& point) {
    out << "Point(" << point.m_x << ", " << point.m_y << ", " << point.m_z << ')';
    return out;
}
// std::istream is the type for object std::cin
std::istream& operator>> (std::istream& in, Point& point) {
    in >> point.m_x;
    in >> point.m_y;
    in >> point.m_z;
    return in;
}
```

```
int main() {
    std::cout << "Enter a point: ";
    Point point;
    std::cin >> point;
    std::cout << "You entered: " << point << '\n';
}
```