**C++ Classes and Objects**

In this tutorial, we will learn about objects and classes and how to use them in C++ with the help of examples.

In previous tutorials, we learned about functions and variables. Sometimes it's desirable to put related functions and data in one place so that it's logical and easier to work with.

Suppose, we need to store the length, breadth, and height of a rectangular room and calculate its area and volume.

To handle this task, we can create three variables, say, length, breadth, and height along with the functions calculateArea() and calculateVolume().

However, in C++, rather than creating separate variables and functions, we can also wrap these related data and functions in a single place (by creating **objects**). This programming paradigm is known as object-oriented programming.  
  
But before we can create **objects** and use them in C++, we first need to learn about **classes**.

**C++ Class**

A class is a blueprint for the object.  
  
We can think of a class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows, etc. Based on these descriptions we build the house. House is the object.

**Create a Class**

A class is defined in C++ using keyword class followed by the name of the class.

The body of the class is defined inside the curly brackets and terminated by a semicolon at the end.

class className {

// some data

// some functions

};

For example,

class Room {

public:

double length;

double breadth;

double height;

double calculateArea(){

return length \* breadth;

}

double calculateVolume(){

return length \* breadth \* height;

}

};

Here, we defined a class named Room.

The variables length, breadth, and height declared inside the class are known as **data members**. And, the functions calculateArea() and calculateVolume() are known as **member functions** of a class.

**C++ Objects**

When a class is defined, only the specification for the object is defined; no memory or storage is allocated.

To use the data and access functions defined in the class, we need to create objects.

**Syntax to Define Object in C++**

className objectVariableName;

We can create objects of Room class (defined in the above example) as follows:

// sample function

void sampleFunction() {

// create objects

Room room1, room2;

}

int main(){

// create objects

Room room3, room4;

}

Here, two objects room1 and room2 of the Room class are created in sampleFunction(). Similarly, the objects room3 and room4 are created in main().

As we can see, we can create objects of a class in any function of the program. We can also create objects of a class within the class itself, or in other classes.

Also, we can create as many objects as we want from a single class.

**C++ Access Data Members and Member Functions**

We can access the data members and member functions of a class by using a . (dot) operator. For example,

room2.calculateArea();

This will call the calculateArea() function inside the Room class for object room2.

Similarly, the data members can be accessed as:

room1.length = 5.5;

In this case, it initializes the length variable of room1 to 5.5.

**Example 1: Object and Class in C++ Programming**

// Program to illustrate the working of

// objects and class in C++ Programming

#include <iostream>

using namespace std;

// create a class

class Room {

public:

double length;

double breadth;

double height;

double calculateArea() {

return length \* breadth;

}

double calculateVolume() {

return length \* breadth \* height;

}

};

int main() {

// create object of Room class

Room room1;

// assign values to data members

room1.length = 42.5;

room1.breadth = 30.8;

room1.height = 19.2;

// calculate and display the area and volume of the room

cout << "Area of Room = " << room1.calculateArea() << endl;

cout << "Volume of Room = " << room1.calculateVolume() << endl;

return 0;

}

**Output**

Area of Room = 1309

Volume of Room = 25132.8

In this program, we have used the Room class and its object room1 to calculate the area and volume of a room.

In main(), we assigned the values of length, breadth, and height with the code:

room1.length = 42.5;

room1.breadth = 30.8;

room1.height = 19.2;

We then called the functions calculateArea() and calculateVolume() to perform the necessary calculations.

Note the use of the keyword public in the program. This means the members are public and can be accessed anywhere from the program.

As per our needs, we can also create private members using the private keyword. The private members of a class can only be accessed from within the class. For example,

class Test {

private:

int a;

void function1() { }

public:

int b;

void function2() { }

}

Here, a and function1() are private and are. Thus they cannot be accessed from outside the class.

On the other hand, b and function2() are accessible from everywhere in the program.

To learn more about public and private keywords, please visit our [C++ Class Access Modifiers](https://www.programiz.com/cpp-programming/access-modifiers) tutorial.

**Example 2: Using public and private in C++ Class**

// Program to illustrate the working of

// public and private in C++ Class

#include <iostream>

using namespace std;

class Room {

private:

double length;

double breadth;

double height;

public:

// function to initialize private variables

void getData(double len, double brth, double hgt) {

length = len;

breadth = brth;

height = hgt;

}

double calculateArea() {

return length \* breadth;

}

double calculateVolume() {

return length \* breadth \* height;

}

};

int main() {

// create object of Room class

Room room1;

// pass the values of private variables as arguments

room1.getData(42.5, 30.8, 19.2);

cout << "Area of Room = " << room1.calculateArea() << endl;

cout << "Volume of Room = " << room1.calculateVolume() << endl;

return 0;

}

**Output**

Area of Room = 1309

Volume of Room = 25132.8

The above example is nearly identical to the first example, except that the class variables are now private.

Since the variables are now private, we cannot access them directly from main(). Hence, using the following code would be invalid:

// invalid code

obj.length = 42.5;

obj.breadth = 30.8;

obj.height = 19.2;

Instead, we use the public function getData() to initialize the private variables via the function parameters double len, double brth, and double hgt.

To learn more on objects and classes, visit these topics:

* [C++ Constructors](https://www.programiz.com/cpp-programming/constructors)
* [How to pass and return an object from a function?](https://www.programiz.com/cpp-programming/pass-return-object-function)

**C++ Constructors**

In this tutorial, we will learn about the C++ constructor and its type with the help examples.

A constructor is a special type of member function that is called automatically when an object is created.

In C++, a constructor has the same name as that of the class and it does not have a return type. For example,

class Wall {

public:

// create a constructor

Wall() {

// code

}

};

Here, the function Wall() is a constructor of the class Wall. Notice that the constructor

* has the same name as the class,
* does not have a return type, and
* is public

**C++ Default Constructor**

A constructor with no parameters is known as a **default constructor**. In the example above, Wall() is a default constructor.

**Example 1: C++ Default Constructor**

// C++ program to demonstrate the use of default constructor

#include <iostream>

using namespace std;

// declare a class

class Wall {

private:

double length;

public:

// create a constructor

Wall() {

// initialize private variables

length = 5.5;

cout << "Creating a wall." << endl;

cout << "Length = " << length << endl;

}

};

int main() {

// create an object

Wall wall1;

return 0;

}

**Output:**

Creating a Wall

Length = 5.5

Here, when the wall1 object is created, the Wall() constructor is called. This sets the length variable of the object to 5.5.

**Note:** If we have not defined a constructor in our class, then the C++ compiler will automatically create a default constructor with an empty code and no parameters.

**C++ Parameterized Constructor**

In C++, a constructor with parameters is known as a parameterized constructor. This is the preferred method to initialize member data.

**Example 2: C++ Parameterized Constructor**

// C++ program to calculate the area of a wall

#include <iostream>

using namespace std;

// declare a class

class Wall {

private:

double length;

double height;

public:

// create parameterized constructor

Wall(double len, double hgt) {

// initialize private variables

length = len;

height = hgt;

}

double calculateArea() {

return length \* height;

}

};

int main() {

// create object and initialize data members

Wall wall1(10.5, 8.6);

Wall wall2(8.5, 6.3);

cout << "Area of Wall 1: " << wall1.calculateArea() << endl;

cout << "Area of Wall 2: " << wall2.calculateArea() << endl;

return 0;

}

**Output:**

Area of Wall 1: 90.3

Area of Wall 2: 53.55

Here, we have created a parameterized constructor Wall() that has 2 parameters: double len and double hgt. The values contained in these parameters are used to initialize the member variables length and height.

When we create an object of the Room class, we pass the values for the member variables as arguments. The code for this is:

Wall wall1(10.5, 8.6);

Wall wall2(8.5, 6.3);

With the member variables thus initialized, we can now calculate the area of the wall with the calculateArea() function.

**C++ Copy Constructor**

The copy constructor in C++ is used to copy data of one object to another.

**Example 3: C++ Copy Constructor**

#include <iostream>

using namespace std;

// declare a class

class Wall {

private:

double length;

double height;

public:

// parameterized constructor

Wall(double len, double hgt) {

// initialize private variables

length = len;

height = hgt;

}

// copy constructor with a Wall object as parameter

Wall(Wall &obj) {

// initialize private variables

length = obj.length;

height = obj.height;

}

double calculateArea() {

return length \* height;

}

};

int main() {

// create an object of Wall class

Wall wall1(10.5, 8.6);

// print area of wall1

cout << "Area of Room 1: " << wall1.calculateArea() << endl;

// copy contents of room1 to another object room2

Wall wall2 = wall1;

// print area of wall2

cout << "Area of Room 2: " << wall2.calculateArea() << endl;

return 0;

}

**Output**

Area of Room 1: 90.3

Area of Room 2: 90.3

In this program, we have used a copy constructor to copy the contents of one object of the Wall class to another. The code of the copy constructor is:

Room(Room &obj) {

length = obj.length;

height = obj.height;

}

Notice that the parameter of this constructor has the address of an object of the Wall class.

We then assign the values of the variables of the first object to the corresponding variables of the second object. This is how the contents of the object is copied.

In main(), we then create two objects wall1 and wall2 and then copy the contents of the first object to the second with the code

Wall wall2 = wall1;

**Note**: A constructor is primarily used to initialize objects. They are also used to run a default code when an object is created.

# How to pass and return object from C++ Functions?

#### In this tutorial, we will learn to pass objects to a function and return an object from a function in C++ programming.

In C++ programming, we can pass objects to a function in a similar manner as passing regular arguments.

## Example 1: C++ Pass Objects to Function

// C++ program to calculate the average marks of two students

#include <iostream>

using namespace std;

class Student {

public:

double marks;

// constructor to initialize marks

Student(double m) {

marks = m;

}

};

// function that has objects as parameters

void calculateAverage(Student s1, Student s2) {

// calculate the average of marks of s1 and s2

double average = (s1.marks + s2.marks) / 2;

cout << "Average Marks = " << average << endl;

}

int main() {

Student student1(88.0), student2(56.0);

// pass the objects as arguments

calculateAverage(student1, student2);

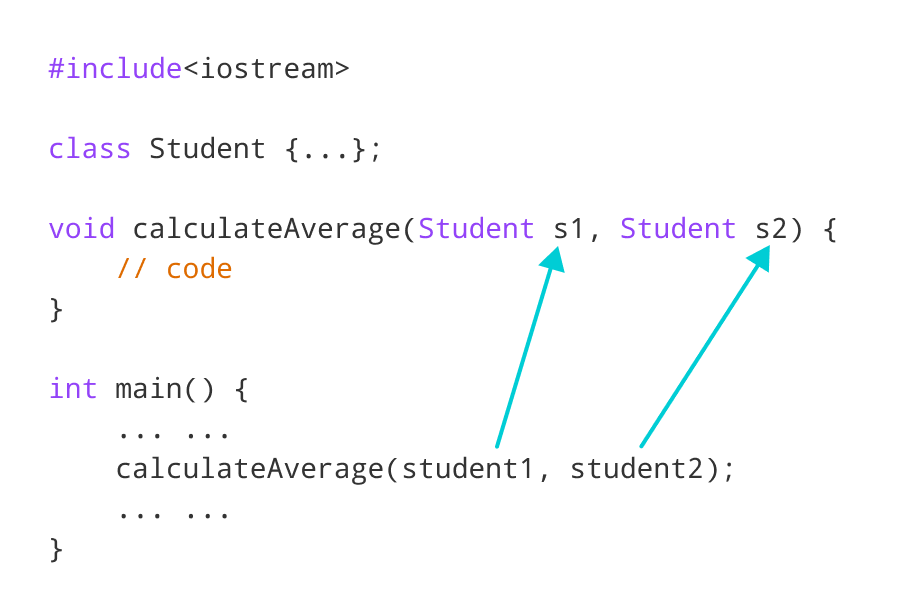
return 0;

}

**Output**

Average Marks = 72

Here, we have passed two Student objects student1 and student2 as arguments to the calculateAverage() function.

Pass objects to function in C++

## Example 2: C++ Return Object from a Function

#include <iostream>

using namespace std;

class Student {

public:

double marks1, marks2;

};

// function that returns object of Student

Student createStudent() {

Student student;

// Initialize member variables of Student

student.marks1 = 96.5;

student.marks2 = 75.0;

// print member variables of Student

cout << "Marks 1 = " << student.marks1 << endl;

cout << "Marks 2 = " << student.marks2 << endl;

return student;

}

int main() {

Student student1;

// Call function

student1 = createStudent();

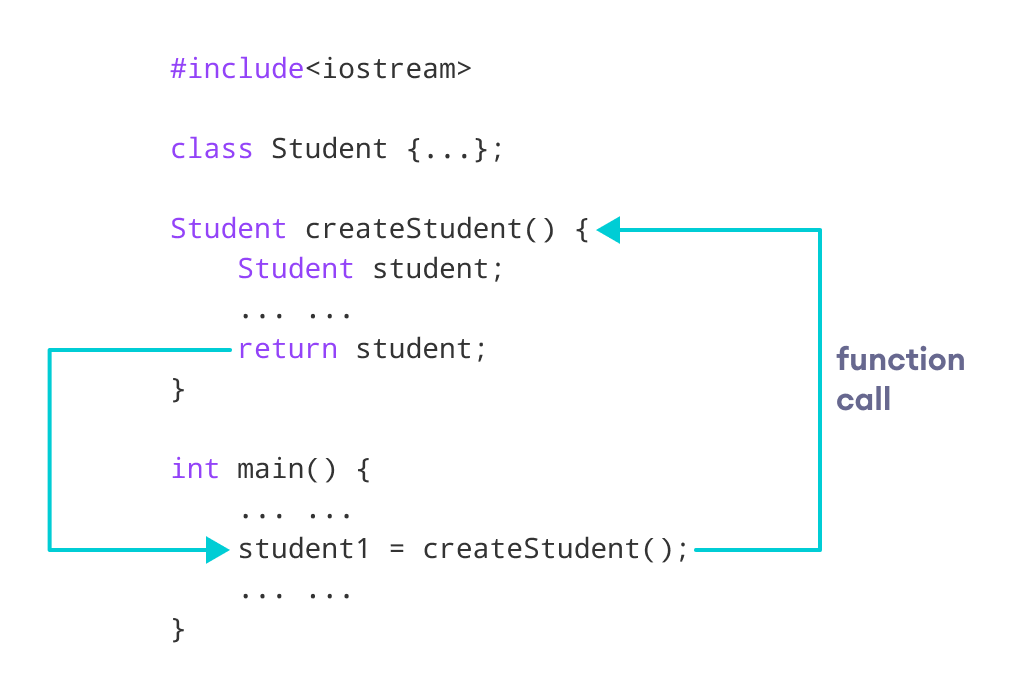
return 0;

}

**Output**

Marks1 = 96.5

Marks2 = 75

Return object from function in C++

In this program, we have created a function createStudent() that returns an object of Student class.

We have called createStudent() from the main() method.

// Call function

student1 = createStudent();

Here, we are storing the object returned by the createStudent() method in the student1.

**C++ Operator Overloading**

In this tutorial, we will learn about operator overloading with the help of examples.

In C++, we can change the way operators work for user-defined types like objects and structures. This is known as **operator overloading**. For example,

Suppose we have created three objects c1, c2 and result from a class named Complex that represents complex numbers.

Since operator overloading allows us to change how operators work, we can redefine how the + operator works and use it to add the complex numbers of c1 and c2 by writing the following code:

result = c1 + c2;

instead of something like

result = c1.addNumbers(c2);

This makes our code intuitive and easy to understand.

**Note:** We cannot use operator overloading for fundamental data types like int, float, char and so on.

**Syntax for C++ Operator Overloading**

To overload an operator, we use a special operator function.

class className {

... .. ...

public

returnType operator symbol (arguments) {

... .. ...

}

... .. ...

};

Here,

* returnType is the return type of the function.
* operator is a keyword.
* symbol is the operator we want to overload. Like: +, <, -, ++, etc.
* arguments is the arguments passed to the function.

**Operator Overloading in Unary Operators**

Unary operators operate on only one operand. The increment operator ++ and decrement operator -- are examples of unary operators.

**Example1: ++ Operator (Unary Operator) Overloading**

// Overload ++ when used as prefix

#include <iostream>

using namespace std;

class Count {

private:

int value;

public:

// Constructor to initialize count to 5

Count() : value(5) {}

// Overload ++ when used as prefix

void operator ++ () {

++value;

}

void display() {

cout << "Count: " << value << endl;

}

};

int main() {

Count count1;

// Call the "void operator ++ ()" function

++count1;

count1.display();

return 0;

}

**Output**

Count: 6

Here, when we use ++count1;, the void operator ++ () is called. This increases the value attribute for the object count1 by 1.

**Note:** When we overload operators, we can use it to work in any way we like. For example, we could have used ++ to increase value by 100.

However, this makes our code confusing and difficult to understand. It's our job as a programmer to use operator overloading properly and in a consistent and intuitive way.

The above example works only when ++ is used as a prefix. To make ++ work as a postfix we use this syntax.

void operator ++ (int) {

// code

}

Notice the int inside the parentheses. It's the syntax used for using unary operators as postfix; it's not a function parameter.

**Example 2: ++ Operator (Unary Operator) Overloading**

// Overload ++ when used as prefix and postfix

#include <iostream>

using namespace std;

class Count {

private:

int value;

public:

// Constructor to initialize count to 5

Count() : value(5) {}

// Overload ++ when used as prefix

void operator ++ () {

++value;

}

// Overload ++ when used as postfix

void operator ++ (int) {

++value;

}

void display() {

cout << "Count: " << value << endl;

}

};

int main() {

Count count1;

// Call the "void operator ++ (int)" function

count1++;

count1.display();

// Call the "void operator ++ ()" function

++ count1;

count1.display();

return 0;

}

**Output**

Count: 6

Count: 7

The **Example 2** works when ++ is used as both prefix and postfix. However, it doesn't work if we try to do something like this:

Count count1, result;

// Error

result = ++count1;

This is because the return type of our operator function is void. We can solve this problem by making Count as the return type of the operator function.

// return Count when ++ used as prefix

Count operator ++ () {

// code

}

// return Count when ++ used as postfix

Count operator ++ (int) {

// code

}

**Example 3: Return Value from Operator Function (++ Operator)**

#include <iostream>

using namespace std;

class Count {

private:

int value;

public

:

// Constructor to initialize count to 5

Count() : value(5) {}

// Overload ++ when used as prefix

Count operator ++ () {

Count temp;

// Here, value is the value attribute of the calling object

temp.value = ++value;

return temp;

}

// Overload ++ when used as postfix

Count operator ++ (int) {

Count temp;

// Here, value is the value attribute of the calling object

temp.value = ++value;

return temp;

}

void display() {

cout << "Count: " << value << endl;

}

};

int main() {

Count count1, result;

// Call the "Count operator ++ ()" function

result = ++count1;

result.display();

// Call the "Count operator ++ (int)" function

result = count1++;

result.display();

return 0;

}

**Output**

Count: 6

Count: 7

Here, we have used the following code for prefix operator overloading:

// Overload ++ when used as prefix

Count operator ++ () {

Count temp;

// Here, value is the value attribute of the calling object

temp.value = ++value;

return temp;

}

The code for the postfix operator overloading is the same as well. Notice that we have created an object temp and returned its value to the operator function.

Also notice the code

temp.value = ++value;

The variable value belongs to the count1 object in main() because count1 is calling the function, while temp.value belongs to the temp object.

**Operator Overloading in Binary Operators**

Binary operators work on two operands. For example,

result = num + 9;

Here, + is a binary operator that works on the operands num and 9.

When we overload the binary operator for user-defined types by using the code:

obj3 = obj1 + obj2;

The operator function is called using the obj1 object and obj2 is passed as an argument to the function.

**Example 4: C++ Binary Operator Overloading**

// C++ program to overload the binary operator +

// This program adds two complex numbers

#include <iostream>

using namespace std;

class Complex {

private:

float real;

float imag;

public:

// Constructor to initialize real and imag to 0

Complex() : real(0), imag(0) {}

void input() {

cout << "Enter real and imaginary parts respectively: ";

cin >> real;

cin >> imag;

}

// Overload the + operator

Complex operator + (const Complex& obj) {

Complex temp;

temp.real = real + obj.real;

temp.imag = imag + obj.imag;

return temp;

}

void output() {

if (imag < 0)

cout << "Output Complex number: " << real << imag << "i";

else

cout << "Output Complex number: " << real << "+" << imag << "i";

}

};

int main() {

Complex complex1, complex2, result;

cout << "Enter first complex number:\n";

complex1.input();

cout << "Enter second complex number:\n";

complex2.input();

// complex1 calls the operator function

// complex2 is passed as an argument to the function

result = complex1 + complex2;

result.output();

return 0;

}

**Output**

Enter first complex number:

Enter real and imaginary parts respectively: 9 5

Enter second complex number:

Enter real and imaginary parts respectively: 7 6

Output Complex number: 16+11i

In this program, the operator function is:

Complex operator + (const Complex& obj) {

// code

}

Instead of this, we also could have written this function like:

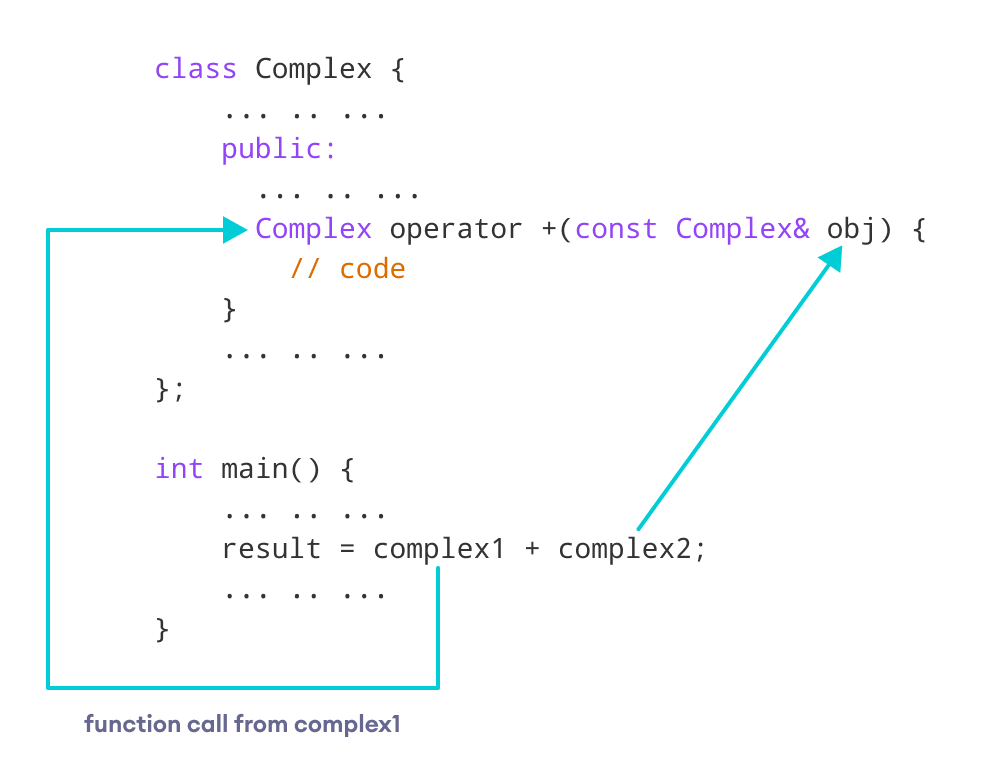
Complex operator + (Complex obj) {

// code

}

However,

* using & makes our code efficient by referencing the complex2 object instead of making a duplicate object inside the operator function.
* using const is considered a good practice because it prevents the operator function from modifying complex2.

Overloading binary operators in C++

**Things to Remember in C++ Operator Overloading**

1. Two operators = and & are already overloaded by default in C++. For example, to [copy objects of the same class](https://www.programiz.com/cpp-programming/constructors#copy-constructor), we can directly use the = operator. We do not need to create an operator function.
2. Operator overloading cannot change the [precedence and associativity of operators](https://www.programiz.com/cpp-programming/operators-precedence-associativity). However, if we want to change the order of evaluation, parentheses should be used.
3. There are 4 operators that cannot be overloaded in C++. They are:
   1. :: (scope resolution)
   2. . (member selection)
   3. .\* (member selection through pointer to function)
   4. ?: (ternary operator)

Visit these pages to learn more on:

* [How to overload increment operator in right way?](https://www.programiz.com/cpp-programming/increment-decrement-operator-overloading)
* [How to overload binary operator - to subtract complex numbers?](https://www.programiz.com/cpp-programming/operator-overloading/binary-operator-overloading)