1. **Write a C++ program to understand the concept of operator overloading by taking the example of the operators ‘+’ and ‘-‘**

Function overloading is a feature in C++ that allows multiple functions to have the same name but different parameters. It is also known as "compile-time polymorphism" or "static polymorphism". The compiler determines which function to call based on the number, types, and order of the function arguments at compile-time, and the appropriate function is automatically selected.

Function overloading provides a way to create multiple functions with the same name but different behavior based on the input arguments. This helps in writing more concise and readable code, as well as providing flexibility in function call syntax.

Rules for Function Overloading:

1. Function Name: Functions that are overloaded must have the same name. The function name is used by the compiler to determine which function to call based on the function arguments.

2. Function Parameters: Overloaded functions must have a different number or type of parameters. This is called "parameter signature" or "function signature". The function signature includes the function name, number of parameters, and their types. If two functions have the same name and the same parameter signature, they are considered the same function, and a compilation error will occur.

3. Return Type: The return type of the function does not play a role in function overloading. Overloaded functions can have the same or different return types. 4. Scope: Function overloading can be done

within the same class or in different classes, including base and derived classes.

5. Access Specifiers: Overloaded functions can have different access specifiers (public, private, protected). The access specifier of the overloaded function does not affect the function overloading mechanism.

6. Default Arguments: Functions with default arguments can be overloaded. However, it's important to note that default arguments must be specified only in the function declaration, not in the function definition. If a default argument is specified in the function declaration, it must be specified in all the overloaded functions that share the same name.

```
// sample program to overload + and - operators
#include <iostream>
using namespace std;

// Define a class called "Number"
class Number {
 private:
        int num;  // Private variable "num"
 public:
  // Constructors
       Number() : num(0) {} // Default constructor initializes num to 0
       Number(int n) : num(n) {} // Constructor that takes an int
argument

  // Overloaded operator functions
  Number operator+(const Number& other) const { // Overload +
operator
```

```
    return Number(num + other.num); // Return new Number object with sum of "num" and "other.num"
  }

  Number operator-(const Number& other) const { // Overload - operator
    return Number(num - other.num); // Return new Number object with difference of "num" and "other.num"
  }

  // Print function to print the value of "num"
  void print() const {
    cout << num << endl;
  }
};

// Main function
int main() {
  // Create two Number objects and add/subtract them using the overloaded operators
  Number n1(5), n2(10);
  Number sum = n1 + n2; // Use overloaded + operator to add n1 and n2
  Number diff = n1 - n2; // Use overloaded - operator to subtract n2 from n1

  // Print the results
  cout << "Sum: ";
  sum.print();

  cout << "Difference: ";
  diff.print();
```

```
    return 0;
}
```

Output:

```
Sum: 15
Difference: -5
```

2. List and explain all the regular operators for which operators can be performed.

. **Arithmetic Operators:** Arithmetic operators are used to perform mathematical operations on numerical values. The common arithmetic operators include:

- Addition (+): Used to add two values.
- Subtraction (-): Used to subtract one value from another.
- Multiplication (*): Used to multiply two values.
- Division (/): Used to divide one value by another.
- Modulo (%): Used to get the remainder of the division operation.

```
//program to demonstrate arithmetic operators overloading
#include <iostream>

class MyNumber {
 private:
  int value;
 public:
  MyNumber(int v): value(v) {}

  // Overloading the + operator
  MyNumber operator+(const MyNumber& other) const {
   return MyNumber(value + other.value);
  }

  // Overloading the - operator
  MyNumber operator-(const MyNumber& other) const {
   return MyNumber(value - other.value);
  }

  // Overloading the * operator
```

```
    MyNumber operator*(const MyNumber& other) const {
      return MyNumber(value * other.value);
    }

    // Overloading the / operator
    MyNumber operator/(const MyNumber& other) const {
      return MyNumber(value / other.value);
    }

    // Overloading the % operator
    MyNumber operator%(const MyNumber& other) const {
      return MyNumber(value % other.value);
    }

    void print() const {
      std::cout << "Value: " << value << std::endl;
    }
};

int main() {
  MyNumber num1(10);
  MyNumber num2(20);

  MyNumber num3 = num1 + num2;
  MyNumber num4 = num1 - num2;
  MyNumber num5 = num1 * num2;
  MyNumber num6 = num2 / num1;
  MyNumber num7 = num2 % num1;

  std::cout << "num1: ";
  num1.print();
  std::cout << "num2: ";
  num2.print();
```

```
    std::cout << "num1 + num2: ";
    num3.print();
    std::cout << "num1 - num2: ";
    num4.print();
    std::cout << "num1 * num2: ";
    num5.print();
    std::cout << "num2 / num1: ";
    num6.print();
    std::cout << "num2 % num1: ";
    num7.print();

    return 0;
}
```

Output:

```
num1: Value: 10
num2: Value: 20
num1 + num2: Value: 30
num1 - num2: Value: -10
num1 * num2: Value: 200
num2 / num1: Value: 2
num2 % num1: Value: 0
```

Relational Operators: Relational operators are used to compare values and return a Boolean value (True or False) as the result. The common comparison operators include:

- Equal to (==): Used to check if two values are equal.
- Not equal to (!=): Used to check if two values are not equal.
- Greater than (>): Used to check if the left value is greater than the right value.

- Greater than or equal to (>=): Used to check if the left value is greater than or equal to the right value.
- Less than (<): Used to check if the left value is less than the right value.
- Less than or equal to (<=): Used to check if the left value is less than or equal to the right value.

```
// program to demonstrate relational operators overloading

#include <iostream>


class MyNumber {
  private:
    int value;
  public:
    MyNumber(int v): value(v) {}


    // Overloading the == operator
    bool operator==(const MyNumber& other) const {
      return value == other.value;
    }


    // Overloading the != operator
    bool operator!=(const MyNumber& other) const {
```

```
  return value != other.value;
}

// Overloading the < operator
bool operator<(const MyNumber& other) const {
  return value < other.value;
}

// Overloading the > operator
bool operator>(const MyNumber& other) const {
  return value > other.value;
}

// Overloading the <= operator
bool operator<=(const MyNumber& other) const {
  return value <= other.value;
}

// Overloading the >= operator
bool operator>=(const MyNumber& other) const {
  return value >= other.value;
}
```

```
    void print() const {
      std::cout << "Value: " << value << std::endl;
    }
};

int main() {
  MyNumber num1(10);
  MyNumber num2(20);

  std::cout << "num1: ";
  num1.print();
  std::cout << "num2: ";
  num2.print();

  if (num1 == num2) {
    std::cout << "num1 is equal to num2" << std::endl;
  } else {
    std::cout << "num1 is not equal to num2" << std::endl;
  }

  if (num1 != num2) {
```

```
  std::cout << "num1 is not equal to num2" << std::endl;
} else {
  std::cout << "num1 is equal to num2" << std::endl;
}

if (num1 < num2) {
  std::cout << "num1 is less than num2" << std::endl;
} else {
  std::cout << "num1 is not less than num2" << std::endl;
}

if (num1 > num2) {
  std::cout << "num1 is greater than num2" << std::endl;
} else {
  std::cout << "num1 is not greater than num2" << std::endl;
}

if (num1 <= num2) {
  std::cout << "num1 is less than or equal to num2" << std::endl;
} else {
  std::cout << "num1 is not less than or equal to num2" << std::endl;
}
```

```
  if (num1 >= num2) {
    std::cout << "num1 is greater than or equal to num2" << std::endl;
  } else {
    std::cout << "num1 is not greater than or equal to num2" << std::endl;
  }

  return 0;
}
```

Output:

```
num1: Value: 10
num2: Value: 20
num1 is not equal to num2
num1 is not equal to num2
num1 is less than num2
num1 is not greater than num2
num1 is less than or equal to num2
num1 is not greater than or equal to num2
```

Logical Operators: Logical operators are used to combine Boolean values or expressions and return a Boolean value as the result. The common logical operators include:

- AND (&&): Returns True if both values or expressions are True.
- OR (||): Returns True if at least one of the values or expressions is True.
- NOT (!): Reverses the logical state of its operand. If a condition is True, then the NOT operator will make it False.

```
// program to demonstrate logical operators overloading

#include <iostream>


class MyBoolean {

 private:

  bool value;

 public:

  MyBoolean(bool v): value(v) {}


  // Overloading the ! operator

  bool operator!() const {

   return !value;

  }


  // Overloading the && operator
```

```
    bool operator&&(const MyBoolean& other) const {
      return value && other.value;
    }

    // Overloading the || operator
    bool operator||(const MyBoolean& other) const {
      return value || other.value;
    }

    void print() const {
      std::cout << "Value: " << value << std::endl;
    }
};

int main() {
  MyBoolean b1(true);
  MyBoolean b2(false);

  std::cout << "b1: ";
  b1.print();
  std::cout << "b2: ";
  b2.print();
```

```
  if (!b1) {
    std::cout << "b1 is false" << std::endl;
  } else {
    std::cout << "b1 is true" << std::endl;
  }

  if (b1 && b2) {
    std::cout << "b1 and b2 are both true" << std::endl;
  } else {
    std::cout << "b1 and/or b2 is false" << std::endl;
  }

  if (b1 || b2) {
    std::cout << "at least one of b1 and b2 is true" << std::endl;
  } else {
    std::cout << "neither b1 nor b2 is true" << std::endl;
  }

  return 0;
}
```

Output:

b1: Value: 1

b2: Value: 0

b1 is true

b1 and/or b2 is false

at least one of b1 and b2 is true

Bitwise Operators: Bitwise operators are used to perform operations on the individual bits of a binary number. The common bitwise operators include:

- AND (&): Returns 1 if both bits are 1.
- OR (|): Returns 1 if at least one of the bits is 1.
- XOR (^): Returns 1 if only one of the bits is 1.
- NOT (~): Flips all the bits of the number.

```
// program to demonstrate bitwise operators overloading
#include <iostream>

class MyNumber {
  private:
    int value;
  public:
    MyNumber(int v): value(v) {}
```

```
// Overloading the bitwise NOT operator
MyNumber operator~() const {
  return MyNumber(~value);
}

// Overloading the bitwise AND operator
MyNumber operator&(const MyNumber& other) const {
  return MyNumber(value & other.value);
}

// Overloading the bitwise OR operator
MyNumber operator|(const MyNumber& other) const {
  return MyNumber(value | other.value);
}

// Overloading the bitwise XOR operator
MyNumber operator^(const MyNumber& other) const {
  return MyNumber(value ^ other.value);
}

// Overloading the left shift operator
```

```
    MyNumber operator<<(int shift) const {
      return MyNumber(value << shift);
    }

    // Overloading the right shift operator
    MyNumber operator>>(int shift) const {
      return MyNumber(value >> shift);
    }

    void print() const {
      std::cout << "Value: " << value << std::endl;
    }
};

int main() {
  MyNumber n1(10);
  MyNumber n2(20);

  std::cout << "n1: ";
  n1.print();
  std::cout << "n2: ";
  n2.print();
```

```
MyNumber n3 = ~n1;
std::cout << "Bitwise NOT of n1: ";
n3.print();

MyNumber n4 = n1 & n2;
std::cout << "Bitwise AND of n1 and n2: ";
n4.print();

MyNumber n5 = n1 | n2;
std::cout << "Bitwise OR of n1 and n2: ";
n5.print();

MyNumber n6 = n1 ^ n2;
std::cout << "Bitwise XOR of n1 and n2: ";
n6.print();

MyNumber n7 = n1 << 2;
std::cout << "Left shift of n1 by 2 bits: ";
n7.print();

MyNumber n8 = n2 >> 1;
```

```
  std::cout << "Right shift of n2 by 1 bit: ";
  n8.print();

  return 0;
}
```

Output:

n1: Value: 10

n2: Value: 20

Bitwise NOT of n1: Value: -11

Bitwise AND of n1 and n2: Value: 0

Bitwise OR of n1 and n2: Value: 30

Bitwise XOR of n1 and n2: Value: 30

Left shift of n1 by 2 bits: Value: 40

Right shift of n2 by 1 bit: Value: 10

Assignment Operators: Assignment operators are used to assign values to variables. The common assignment operators include:

- Equals (=): Assigns the value on the right to the variable on the left.
- Plus equals (+=): Adds the value on the right to the variable on the left and assigns the result to the variable.
- Minus equals (-=): Subtracts the value on the right from the variable on the left and assigns the result to the variable.

- Multiply equals (*=): Multiplies the value on the right with the variable on the left and assigns the result to the variable.
- Divide equals (/=): Divides the variable on the left by the value on the right and assigns the result to the variable.

```
// program to demonstrate assignment operators overloading

#include <iostream>


class MyNumber {

  private:

    int value;

  public:

    MyNumber(int v): value(v) {}


    // Overloading the += operator

    MyNumber& operator+=(const MyNumber& other) {

      value += other.value;

      return *this;

    }


    // Overloading the -= operator

    MyNumber& operator-=(const MyNumber& other) {
```

```
  value -= other.value;
  return *this;
}

// Overloading the *= operator
MyNumber& operator*=(const MyNumber& other) {
  value *= other.value;
  return *this;
}

// Overloading the /= operator
MyNumber& operator/=(const MyNumber& other) {
  value /= other.value;
  return *this;
}

// Overloading the %= operator
MyNumber& operator%=(const MyNumber& other) {
  value %= other.value;
  return *this;
}
```

```
// Overloading the &= operator
MyNumber& operator&=(const MyNumber& other) {
  value &= other.value;
  return *this;
}

// Overloading the |= operator
MyNumber& operator|=(const MyNumber& other) {
  value |= other.value;
  return *this;
}

// Overloading the ^= operator
MyNumber& operator^=(const MyNumber& other) {
  value ^= other.value;
  return *this;
}

// Overloading the <<= operator
MyNumber& operator<<=(int shift) {
  value <<= shift;
  return *this;
```

```
}

// Overloading the >>= operator
MyNumber& operator>>=(int shift) {
  value >>= shift;
  return *this;
}

// Overloading the ++ (prefix) operator
MyNumber& operator++() {
  ++value;
  return *this;
}

// Overloading the ++ (postfix) operator
MyNumber operator++(int) {
  MyNumber temp(value);
  ++value;
  return temp;
}

// Overloading the -- (prefix) operator
```

```
    MyNumber& operator--() {
      --value;
      return *this;
    }

    // Overloading the -- (postfix) operator
    MyNumber operator--(int) {
      MyNumber temp(value);
      --value;
      return temp;
    }

    void print() const {
      std::cout << "Value: " << value << std::endl;
    }
};

int main() {
  MyNumber n1(10);
  MyNumber n2(20);

  std::cout << "n1: ";
```

```
n1.print();
std::cout << "n2: ";
n2.print();

n1 += n2;
std::cout << "n1 after n1 += n2: ";
n1.print();

n1 -= n2;
std::cout << "n1 after n1 -= n2: ";
n1.print();

n1 *= n2;
std::cout << "n1 after n1 *= n2: ";
n1.print();

n1 /= n2;
std::cout << "n1 after n1 /= n2: ";
n1.print();

n1 %= n2;
std::cout << "n1 after n1 %= n2: ";
```

```
  n1.print();


  n1 &= n2;
  std::cout << "n1 after n1 &= n2
}
```

Output:

```
n1: Value: 10
n2: Value: 20
n1 after n1 += n2: Value: 30
n1 after n1 -= n2: Value: 10
n1 after n1 *= n2: Value: 200
n1 after n1 /= n2: Value: 10
n1 after n1 %= n2: Value: 10
n1 after n1 &= n2: Value: 0
```

Increment and Decrement operators: The increment (++) and decrement (--) operators are used to increase or decrease the value of a variable by 1. In addition to their standard behavior, C++ allows you to overload these operators, which means you can define custom behavior for these operators when applied to objects of your own class.

```
// program to demonstrate increment/decrement operators overloading
#include <iostream>

class MyNumber {
  private:
    int value;
  public:
    MyNumber(int v): value(v) {}

    // Overloading the ++ (prefix) operator
    MyNumber& operator++() {
      ++value;
      return *this;
    }

    // Overloading the ++ (postfix) operator
    MyNumber operator++(int) {
      MyNumber temp(value);
      ++value;
      return temp;
    }
```

```
    // Overloading the -- (prefix) operator
    MyNumber& operator--() {
      --value;
      return *this;
    }

    // Overloading the -- (postfix) operator
    MyNumber operator--(int) {
      MyNumber temp(value);
      --value;
      return temp;
    }

    void print() const {
      std::cout << "Value: " << value << std::endl;
    }
};

int main() {
  MyNumber n(10);

  std::cout << "n: ";
```

```
  n.print();

  ++n;
  std::cout << "n after ++n: ";
  n.print();

  n++;
  std::cout << "n after n++: ";
  n.print();

  --n;
  std::cout << "n after --n: ";
  n.print();

  n--;
  std::cout << "n after n--: ";
  n.print();
}
```

Output:

n: Value: 10

n after ++n: Value: 11

n after n++: Value: 12

n after --n: Value: 11

n after n--: Value: 10

Other operators: Subscript operator: [] (array subscript).Function call operator: () (function call).Member selection operator: -> (member selection via pointer). Comma operator: , (comma).

```
// program to demonstrate other operators overloading

#include <iostream>

class MyClass {

 private:

  int arr[5];

 public:

  MyClass() {

    for (int i = 0; i < 5; ++i) {

      arr[i] = i;

    }

  }
```

```
// Overloading the subscript operator
int& operator[](int index) {
  if (index >= 0 && index < 5) {
    return arr[index];
  }
  std::cerr << "Index out of range!" << std::endl;
  exit(1);
}

// Overloading the function call operator
int operator()(int arg1, int arg2) {
  return arg1 + arg2;
}

// Overloading the arrow operator
MyClass* operator->() {
  return this;
}

// Overloading the comma operator
MyClass operator,(int val) {
  arr[0] = val;
```

```
    return *this;
  }

  void print() {
    for (int i = 0; i < 5; ++i) {
      std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
  }
};

int main() {
  MyClass obj;

  std::cout << "Before overloading [], obj.arr[2]: " << obj.arr[2] << std::endl;
  obj[2] = 20;
  std::cout << "After overloading [], obj[2]: " << obj[2] << std::endl;

  std::cout << "Before overloading (), obj(10, 20): " << obj(10, 20) << std::endl;

  std::cout << "Before overloading ->, obj.print(): ";
  obj.print();
```

```
  obj->print();


  std::cout << "Before overloading comma operator, obj.print(): ";
  obj.print();
  obj = 100, 2, 3, 4, 5;
  std::cout << "After overloading comma operator, obj.print(): ";
  obj.print();
}
```

Output:

```
Before overloading [], obj.arr[2]: 2
After overloading [], obj[2]: 20
Before overloading (), obj(10, 20): 30
Before overloading ->, obj.print(): 0 1 2 3 4
0 1 20 3 4
Before overloading comma operator, obj.print(): 0 1 20 3 4
After overloading comma operator, obj.print(): 2 1 20 3 4
```

3. **List and explain the special operators for which operator overloading can be performed also list and explain the set of operators for which operator overloading cannot be performed.**

In C++, operator overloading allows you to define the behavior of operators for your own custom classes and data types. The following are the special operators for which operator overloading can be performed:

1. Arithmetic operators: +, -, *, /, %

These operators can be overloaded to perform arithmetic operations on user-defined data types such as complex numbers, fractions, and matrices.

2. Comparison operators: ==, !=, <, >, <=, >=

These operators can be overloaded to compare user-defined data types such as strings, dates, and custom classes.

3. Logical operators: !, &&, ||

These operators can be overloaded to perform logical operations on user-defined data types such as Boolean objects or sets.

4. Assignment operators: =, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=

These operators can be overloaded to perform custom operations when an object is assigned to another object.

5. Increment and decrement operators: ++, --

These operators can be overloaded to perform custom operations when an object is incremented or decremented.

6. Member access operators: ., ->

These operators can be overloaded to define the behavior of accessing members of objects and pointers to objects.

7. Subscript operator: []

This operator can be overloaded to define the behavior of accessing elements of arrays and custom container classes.

8. Function call operator: ()

This operator can be overloaded to define the behavior of calling a function on an object or class.

9. Type conversion operators: (type)

These operators can be overloaded to define the behavior of converting one data type to another.

10. Comma operator: ,

This operator can be overloaded to define the behavior of the comma operator in user-defined data types.

In C++, some operators cannot be overloaded. These are known as non-overloadable operators. The following operators cannot be overloaded in C++:

:: (scope resolution operator): It is used to access members of a class or namespace, and it cannot be overloaded.

. (dot operator): It is used to access members of an object or structure, and it cannot be overloaded.

.* (pointer-to-member operator): It is used to access members of an object through a pointer-to-member, and it cannot be overloaded.

?: (conditional operator): It is used to create a ternary conditional expression, and it cannot be overloaded.

sizeof (sizeof operator): It is used to determine the size of a type or an object, and it cannot be overloaded.

typeid (typeid operator): It is used to get the type information of an object, and it cannot be overloaded.

const_cast (const_cast operator): It is used to remove the const qualifier from a variable, and it cannot be overloaded.

dynamic_cast (dynamic_cast operator): It is used for dynamic type casting, and it cannot be overloaded.

typeid (typeid operator): It is used to get the type information of an object, and it cannot be overloaded.