- . (member access)
- -> (pointer-to-member)
- ? : (ternary conditional)

Overloading Various Operators:

Operators can be overloaded to perform custom operations depending on the context and the type of objects involved. Some common Unary or Binary operators that are frequently overloaded include:

- Arithmetic Operators: `+`, `-`, `*`, `/`, `%`
- Relational Operators: `==`, `!=`, `<`, `>`, `<=`, `>=`
- Logical Operators: `&&`, `||`, `!`
- Assignment Operators: `=`, `+=`, `-=`, `*=`, `/=`
- Increment and Decrement Operators: `++`, `--`
- Subscript Operator: `[ ]`
- Function Call Operator: `( )`
- Stream Operators: `>>`, `<<` (for input/output)

Note: Following operators can't be overloaded :

- Conditional operator: ? : (ternary operator)
- Member access operator: . (dot)
- Pointer to member access operator: -> (arrow)
- Scope resolution operator: : : (double colon)
- Sizeof operator: sizeof

Operators that Cannot Be Overloaded Using Friend Function and Member Function

1. Using Friend Function:
   - = (assignment operator)
   - [ ] (subscript operator)
   - ( ) (function call operator)
   - -> (member access operator)
2. Using Member Function:
   - All operators can be overloaded using member functions, except for those that cannot be overloaded at all (i.e., `::`, `.`, `->`, `?:`).

Example of Overloading a Binary Operator:

Here's how you can overload the binary + operator using both member function and friend function:

Using Member Function:

```
#include <iostream>
using namespace std;

class Complex {
private:
```

```
    double real, imag;

public:
    Complex(double r = 0, double i = 0) : real(r), imag(i) {}

    // Overloading the + operator using a member function
    Complex operator+(const Complex& other) const {
        return Complex(real + other.real, imag + other.imag);
    }

    void display() const {
        cout << real << " + " << imag << "i" << endl;
    }
};

int main() {
    Complex c1(3.0, 4.0), c2(1.0, 2.0);
    Complex c3 = c1 + c2; // Uses the overloaded + operator
    c3.display(); // Output: 4.0 + 6.0i
    return 0;
}
```

Using Friend Function:

```
#include <iostream>
using namespace std;

class Complex {
private:
    double real, imag;

public:
    Complex(double r = 0, double i = 0) : real(r), imag(i) {}

    // Friend function to overload the + operator
    friend Complex operator+(const Complex& c1, const Complex& c2);

    void display() const {
        cout << real << " + " << imag << "i" << endl;
    }
};

// Definition of the friend function
Complex operator+(const Complex& c1, const Complex& c2) {
    return Complex(c1.real + c2.real, c1.imag + c2.imag);
}
```

```
int main() {
    Complex c1(3.0, 4.0), c2(1.0, 2.0);
    Complex c3 = c1 + c2; // Uses the overloaded + operator
    c3.display(); // Output: 4.0 + 6.0i
    return 0;
}
```

2.2 Type Conversion

Type conversion in C++ involves converting values from one data type to another. This can be particularly useful when working with user-defined types (classes) and built-in types. Here, we'll explore different types of conversions involving class types.

A. Basic Type to Class Type Conversion

Basic types, such as integers or floating-point numbers, can be converted to class types using constructors that accept parameters of the corresponding basic types. This allows for initialization of class objects with values from basic types.

Example: Conversion from int to String Class

```
#include <iostream>
#include <string>
using namespace std;

class String {
private:
    string value;

public:
    String(int num) {
        value = to_string(num);
    }

    void display() {
        cout << "String value: " << value << endl;
    }
};

int main() {
    int number = 42;
    String str = number; // Conversion from int to String

    str.display();
    return 0;
}
```

B. Class Type to Basic Type Conversion

Class types can be converted to basic types by overloading conversion operators or by providing explicit conversion functions. This allows for extracting relevant information or converting class objects into values of basic types.

Example: Conversion from String Class to int

```
#include <iostream>
#include <string>
using namespace std;

class String {
private:
    string value;

public:
    String(string val) : value(val) {}

    // Conversion operator for int
    operator int() const {
        return stoi(value); // Convert string to int
    }
};

int main() {
    String str("42");
    int number = str; // Conversion from String to int

    cout << "Integer value: " << number << endl;
    return 0;
}
```

C. Class Type to Another Class Type Conversion

Class types can be converted to other class types through constructors or conversion operators defined in the respective classes. This allows for creating objects of one class type from objects of another class type.

Example: Conversion from Feet Class to Meter Class

```
#include <iostream>
using namespace std;

class Meter {
private:
    double value;

public:
```