Section 2 : Operator Overloading and Type Conversion

2.1 Operator Overloading

Operator overloading is a powerful feature in C++ that allows us to define the behavior of operators such as +, -, *, /, ==, !=, etc., for objects of user-defined classes. It provides a way to to extend the functionality of built-in operators to work with objects of our own classes.

Syntax:

```
return_type operator op (parameters) {
    // Operator implementation
}
```

Where op is the operator being overloaded and parameters represent the operands of the operator.

Conditions where operator overloading is necessary:

1. **When working with user-defined types:** Operator overloading allows user-defined types, such as classes or structures, to behave like built-in types. This can make the code more readable and maintainable by providing a natural syntax for operations involving those types.
2. **When defining mathematical operations for custom types:** For example, when working with complex numbers, matrices, or vectors, overloading operators like +, -, *, etc., can make the code more expressive and concise.
3. **When implementing custom data structures:** Operator overloading can be useful when implementing custom data structures like arrays, lists, or trees. For example, overloading the subscript operator [] can provide array-like access to elements of a custom container class.
4. **When working with streams and IO operations:** Overloading stream insertion (<<) and stream extraction (>>) operators can enable custom formatting and parsing of data types when performing input/output operations.

Rules for Overloading Operators:

1. Only Existing Operators Can Be Overloaded:
    - You cannot create new operators; you can only overload the existing ones.
2. Preserve the Operator's Arity:
    - The number of operands an operator works with cannot be changed. For instance, a binary operator (e.g., +) must remain binary.
3. Operator Overloading Cannot Change Precedence or Associativity:
    - The precedence and associativity of operators remain the same regardless of overloading.
4. Overloaded Operators Must Have at Least One User-Defined Type as Operand:
    - At least one operand must be a user-defined type (class or struct).
5. Certain Operators Cannot Be Overloaded:
    - There are a few operators that cannot be overloaded, including:
        - : : (scope resolution)

- . (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:
```