QUESTION: REVISIT

Which of the following overloaded functions are NOT allowed in C++?

1. Function declarations that differ only in the return type

```
int fun(int x, int y);
void fun(int x, int y);
```

2. Functions that differ only by static keyword in return type

```
int fun(int x, int y);
static int fun(int x, int y);
```

3. Parameter declarations that differ only in a pointer * versus an array []

```
int fun(int *ptr, int n);
int fun(int ptr[], int n);
```

4. Two parameter declarations that differ only in their default arguments

```
int fun( int x, int y);
int fun( int x, int y = 10);
```

EXERCISE 1

```cpp
1   #include <iostream>
2   using namespace std;
3
4   void display(char c) {
5       cout << "Character: " << c << endl;
6   }
7
8   void display(float a) {
9       cout << "Float: " << a << endl;
10  }
11
12  void display(char c, int a) {
13      cout << "Character and Integer: " << c << ", " << a << endl;
14  }
15
16  void display(double a, int b) {
17      cout << "Double and Integer: " << a << ", " << b << endl;
18  }
```

```cpp
20  int main() {
21      display(50);
22      display('A');
23      display(3.14);
24      display('B', 10);
25      display(20.5, 5);
26      return 0;
27  }
```

EXERCISE 2

```cpp
1   #include<iostream>
2   using namespace std;
3
4   void test(int s, int t, int x=0)
5   {
6       cout << "Function with three para called ";
7       cout<<s<<" "<<t<<" "<<x<<endl;
8   }
9
10  void test(int s, int t)
11  {
12      cout << "Function with two para called \n";
13      cout<<s<<" "<<t<<endl;
14  }
15
16
17  int main()
18  {
19      test(4,5);
20      test(4,5,67);
21  }
```

HOME WORK

Write a program that calculates the area of different geometric shapes (circle, rectangle, and triangle) using function overloading.

```
Select a shape to calculate the area:
1. Circle
2. Rectangle
3. Triangle
Enter your choice: 1
Enter the radius of the circle: 5
Area of the circle: 78.54

Select a shape to calculate the area:
1. Circle
2. Rectangle
3. Triangle
Enter your choice: 2
Enter the length and breadth of the rectangle: 4 6
Area of the rectangle: 24.00

Select a shape to calculate the area:
1. Circle
2. Rectangle
3. Triangle
Enter your choice: 3
Enter the base and height of the triangle: 5 7
Area of the triangle: 17.50
```
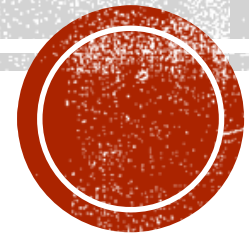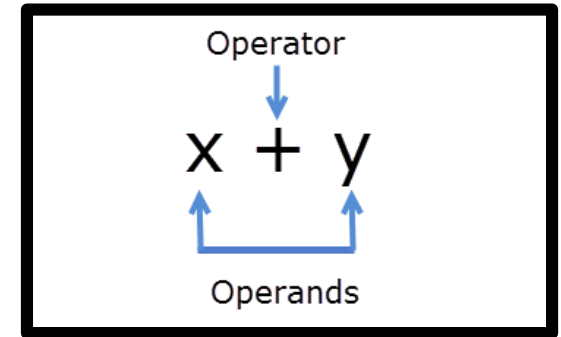
OPERATOR OVERLOADING

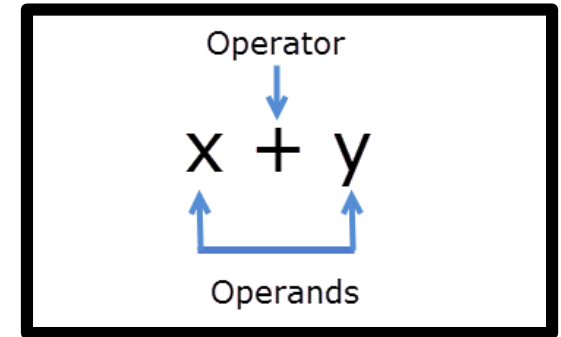What is meant by Operator Overloading?

WHY—OPERATOR OVERLOADING

Operator
x + y
Operands

- Operators work with operands.

- Increment operator can be used with int values or int type variables.
  Example: count ++.

```
1   #include<iostream>
2   using namespace std;
3
4   int main(){
5
6   int i, j;
7
8       j = 10;
9       i = (j++, j+100, j+10);
10
11      cout << i;
12
13      return 0;
14  }
```

```
21
------------------------------
Process exited after 0.07273 seconds with return value 0
Press any key to continue . . .
```

WHY—OPERATOR OVERLOADING

Operator
x + y
Operands

- Operators work with operands.

- Increment operator can be used with int values or int type variables.
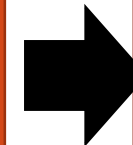 Example: count ++.

Question: Can we use the above operator with objects directly?

Example: Create an object (N1) from Numbers class and increment all the variables inside the class?

We can't use operators on objects as they are NOT applied for classes.

However, these operators can be overloaded to get special functions done.

```
class Numbers{
    int x, y;

    public:
    Numbers():x(0),y(0){
    }
};
```

```
int main(){
    Numbers N1;
    ++N1;
}
```

DEFINING OPERATOR OVERLOADING

Syntax: operator function

```
returnType operator operatorSymbol(parameter)
 {
   // ...
 }
```

EXERCISE

```cpp
1   #include<iostream>
2   using namespace std;
3
4   class Number{
5       int x,y;
6
7       public:
8           Number():x(0),y(0){}
9
10          void operator++(){
11              x=x+100;
12              y=y+100;
13          }
14
15          void display(){
16              cout<<"X is "<<x;
17              cout<<"\nY is "<<y<<endl;
18          }
19  };
```

```cpp
21  int main(){
22
23      Number N1,N2;
24      N1.display();
25
26      ++N1;
27      N1.display();
28      N2.display();
29  }
30
```

```
X is 0
Y is 0
X is 100
Y is 100
X is 0
Y is 0

------------------------------------
Process exited after 1.042 seconds with return value 0
Press any key to continue . . .
```
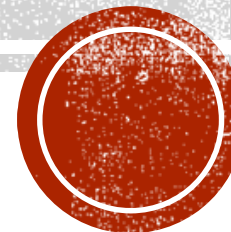
OVERLOADING BINARY OPERATOR

- Overloading of an operator operating on two operands.
- One argument should be passed to the operator function.

PRACTICE EXERCISE

Complex Number Addition & Subtraction

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

class Complex{
    int real;
    int imagi;

    public:
    Complex():real(0),imagi(0){

    cout<<"Inside the constructor"<<endl;}

    Complex(int x, int y):real(x),imagi(y){}

    Complex(Complex& c){

    cout<<"User Defined copy constructor"<<endl;
    real=c.real;
    imagi=c.imagi;
    }

    Complex operator +(Complex& c){
        Complex tempc;
        tempc.real=real+c.real;
        tempc.imagi=imagi+c.imagi;
            return (tempc);

    }

    Complex operator -(Complex& c){
        Complex tempc;
        tempc.real=real-c.real;
        tempc.imagi=imagi-c.imagi;
            cout<<c.real<<" "<<c.imagi<<endl;
        return (tempc);

    }


    void display(){
    if(imagi < 0)
        cout << "Complex No: "<< real << imagi << "i"<<endl;
    else
        cout << "Complex No: " << real << "+" << imagi << "i"<<endl;   }

    ~Complex()
    {
            cout<<"Deallocate Memory"<<endl;
    }

};

int main(){
    Complex C1(3,4);
        C1.display();
    Complex C2(2,3);
        C2.display();
            C1=C2-C1;
            C1.display();
            Complex C3=C1;
            C3.display();
}
```

ASSIGNMENT OPERATOR

- Compiler by default automatically defines an operator overloading for the assignment operator.

- Used to copy values from one object to another *already existing object*.

    Example: obj1 and obj2

        obj1 = obj2   (copy all the data values of obj2 into obj1)

- It works the same as copy constructor

- only shallow copy (makes a bitwise copy of the class instance) is done hence may cause problem while working with pointers.

OVERLOADING THE ASSIGNMENT OPERATOR

Assignment vs Copy constructor

- The purpose is almost equivalent
  - both copy one object to another.
  - The copy constructor initializes new objects, whereas the assignment operator replaces the contents of existing objects.

- A new object needs to be created before the copying - Copy constructor is called.

- No need to create a new object before the copying can occur - Assignment operator is called.

- The copy constructor creates a separate memory block for the new object.

- The assignment operator does not make new memory space (uses the reference variable to point to the previous memory block).

OVERLOADED ASSIGNMENT OPERATOR

```
ClassName& ClassName::operator=(const ClassName& op)
{
   //………………..
   return *this;
}
```

OPERATOR FUNCTIONS: MEMBER/NON-MEMBER

- Operator functions can be either members or non-members of the class they operate on.

- Non-member operator functions:
  - Should be defined as friend functions
  - Need access to the class's private data members.

- The way an operator function is written differs depending on whether it is a member function or a friend function.

FRIEND FUNCTIONS

- Data hiding - A fundamental concept of object-oriented programming.
  - Restricts access to private members from outside the class.
  - Protected members can only be accessed by derived classes and are inaccessible from outside.

FRIEND FUNCTIONS

- What is a friend function?
  - A friend function is a function that is not a member of a class but is granted access to its private and protected members.

- Why use a friend function?
  - Allow an external function (or another class) to access private/protected data of a class without making the data public.
  - Useful for operator overloading and functions that work on multiple classes.

- How do you define and declare a friend function?
  - Declare the function inside the class with the friend keyword.
  - Define it outside the class, similar to a regular function.

FRIEND OPERATOR FUNCTIONS

Binary operator overloaded using a friend function:

- The **left operand is passed in as the first parameter** and the **right operand is passed in as the second parameter**.

friend Complex& operator+(const Complex& ob1, const Complex& ob2);

Unary operator overloaded using a friend function:

- the operand is passed to the function as a parameter.

friend Counter& operator++(Counter& ob);

It is common for an overloaded operator function's return type to be the same as the class it operates on, so that several operations may be chained together.

An operator function that changes the value of its invoking operand normally returns a reference to that operand.

MEMBER OPERATOR FUNCTIONS

Binary operator overloaded using a member function:

▪ The left operand generates the function call, and the right operand is passed to the function as a parameter.

Unary operator overloaded using a member function:

▪ The operand generates the function call, parameter is not required.

THE ASSIGNMENT OPERATOR

- The assignment operator must be overloaded as a member function, not a friend function [Justify (H/W)]

- The left operand invokes the **operator=()** function and accepts a constant reference to the right operand as an argument.

THE OUTPUT OPERATOR

- The output operator must be overloaded as a friend function because its left operand is an output stream such as **cout.**

- The general form of the **operator<<()** function is

  ostream& operator<<(ostream& out, const ClassName& op)

  {

  ** // ...**

  }

- The **operator<<()** function returns a reference to an **ostream** object so that output operations may be chained together.

```cpp
// Output operator
ostream& operator<<(ostream& out, const Money& amount)
{
    out << '$' << amount.dollars << '.';
    if (amount.cents < 10)
        out << '0';
    out << amount.cents;
    return out;
}
```

THE INPUT OPERATOR

- The input operator must be overloaded as a friend function because its left operand is an input stream such as **cin.**

- The general form of the **operator>>()** function is

  istream& operator>>(istream& in, ClassName& op)

  {

    // ...

  }

- The **operator>>()** function returns a reference to an **istream** object so that input operations may be chained together.

```cpp
// Input operator

istream& operator>>(istream& in, Money& amount)

{

  unsigned int d;    // Number of dollars to be read

  unsigned int c;    // Number of cents to be read

  char dollarSign;   // Dollar sign character preceding d

  char decimalPoint; // Decimal point character between

                     // d and c

  in >> dollarSign >> d >> decimalPoint >> c;

  amount = Money(d, c);

  return in;

}
```

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

class Fraction {

private:
    int numerator;
    int denominator;

public:
    // constructor
    Fraction(int x = 0, int y = 1) {
        numerator = x;
        denominator = y;
    }

    friend istream& operator>>(istream& cin, Fraction& c){
        cin >> c.numerator >> c.denominator;
        return cin;
    }

    friend ostream& operator<<(ostream&, Fraction& c)    {
        cout << c.numerator << "/" << c.denominator;
        return cout;
    }
};

int main()
{
    Fraction x;
    cout << "Enter a numerator and denominator of "
                "Fraction: ";
    cin >> x;
    cout << "Fraction is: ";
    cout << x;
    return 0;
}
```

27 DATA CONVERSION

DATA CONVERSION

- Assignments between the same data types are handled by the compiler.
  - Basic types / user-defined types

> **Example:**
> **a=b;**
> **myObj1=myObj2;**

but….

- What happens when the variables on different sides of the = are of different types?

- Basic types – Compiler handles automatically

- Compiler does not handle automatically
  - Conversions between basic types and user-defined types, and
  - Conversions between different user-defined types.

CONVERSIONS BETWEEN BASIC TYPES

- Basic Data Types
  - Float to int
  - Float to double
  - Char to float
  - int to Char etc….

- Convert the value of floatvar (expressed in floating-point format) to an integer format and assigned to intvar.

intvar = floatvar;

DATA TYPE CONVERSION

- A type conversion occurs when the compiler changes the type of an object.

- An explicit type conversion occurs when the programmer uses a cast in the code to change the type of an object.

```
double x = double(3) / 2;    // The value of x is 1.5
```

- An implicit type conversion occurs when the compiler automatically changes the type of an object.

```
int  i = 32.4;        // The value of i is 32
```

EXERCISE

```
1    #include<iostream>
2    using namespace std;
3
4    int main()
5    {
6        char a;
7        float x=100.9;
8
9        a = x;
10
11       cout<<"a: "<<a<<" "<<"x:"<<x<<" "<<(int)a<<endl;
12   }
```

```
a: d x:100.9 100

_____
Process exited after 0.1547 seconds with return value 0
Press any key to continue . . .
```

TYPE CASTING

- Converting a variable from one data type to another.

- **Implicit conversion**
  - Do not require any operator.
  - Automatically performed when a value is copied to a compatible type.

- **Explicit conversion**
  - There are many conversions
  - Imply a different interpretation of the value, require an explicit conversion.

- Casting operators:
  - dynamic_cast,
  - reinterpret_cast,
  - static_cast and
  - const_cast.

33 DATA CONVERSION

Conversion of primitive data type to user-defined

```cpp
// Time Class
class Time {
    int hour;
    int mins;

public:
    // Default Constructor
    Time() : hour(0),mins(0){  }

    // Parameterized Constructor
    Time(int t)
    {
        hour = t / 60;
        mins = t % 60;
    }

    void Display()
    {
        cout << "Time = " << hour
            << " hrs and "
            << mins << " mins\n";
    }
};
```

```cpp
int main()
{
    // Object of Time class
    Time T1;
    int dur = 95;

    // Conversion of int type to class type
    T1 = dur;
    T1.Display();
}
```

35 DATA CONVERSION

Conversion of class object to primitive data type

CONVERSION OF CLASS OBJECT TO PRIMITIVE DATA TYPE

- Convert from class object and a primitive data type.

- Need to use an overloaded casting operator function (conversion function)

The syntax:

```
operator typename()
{
    // Code
}
```

Example:

The **operator double()** converts a class object to type double,

The **operator int()** converts a class type object to type int etc..

```cpp
class Time {
    int hrs, mins;

public:
    // Constructor
    Time(int, int);

    // Casting operator
    operator int();

};

Time::Time(int a, int b)
{

    hrs = a;
    mins = b;

}
```

```cpp
// Used for Data conversion of class to primitive
Time::operator int()
{

    cout << "Conversion of Class"
        << " Type to Primitive Type"
        << endl;

    return (hrs * 60 + mins);

}

int main()
{

    int hour, mins;
    hour = 3;
    mins = 20;

    Time t(hour,mins);

    int duration = t;

    cout << "Total Minutes are "
        << duration << endl;

}
```

38 DATA CONVERSION

Conversion of one class type to another class type

```cpp
// Time Class
class Time {
    int hr, mins;
public:
    // Constructors
    Time(int h, int m) : hr(h),mins(m){ }
    Time()
    {
        cout << "\nTime's Object Created";
    }

    int Minute()
    {
        int totalmin = (hr * 60) + mins;
        return totalmin;
    }

    void show()
    {
        cout << "Hour: " << hr << endl;
        cout << "Minute : " << mins << endl;
    }
};
```

```cpp
// Minutes Class
class Minute {
    int mins;

public:
    // Constructors
    Minute()
    {
        mins = 0;
    }
    void operator=(Time T)
    {
        mins = T.Minute();
    }

    void show()
    {
        cout << "\nTotal Minute : "
<< mins << endl;
    }
};
```

```cpp
int main()
{
    int hour, mins;
    hour = 3, mins = 40;

    Time T1(hour,mins);
    T1.show();

    Minute M1;
        M1.show();
    M1=T1;
        M1.show();
}
```