# Chapter 4: Part-1
# Operator Overloading

BIBHA STHAPIT
ASST. PROFESSOR
IOE, PULCHOWK CAMPUS

# Operator Overloading- Introduction

- Use operators with objects (operator overloading)
  - Clearer than function calls for certain classes
  - Operator sensitive to context
- Examples
  - **<<**
    - Stream insertion, bitwise left-shift
  - **+**
    - Performs arithmetic on multiple types (integers, floats, etc.)

# Introduction

- Types
  - Built in (**`int`**, **`char`**) or user-defined
  - Can use existing operators with user-defined types
    - Cannot create new operators

- **Definition: Operator overloading** means to redefine the standard operators so that they can also perform operations on classes as in fundamental types.

# Introduction

- Overloading operators
  - Create a function for the class
  - Name function **`operator`** followed by symbol
    - **`Return_type operator op_sign(list of arg/s);`**
    - » **`Operator+`** is the ‘operator function’ for the addition operator **`+`**

- Overloading provides concise notation
  - **`object2 = object1.add(object2);`**
  - **`object2 = object2 + object1;`**

# Restrictions

| Operators that cannot be overloaded | | | | |
|---|---|---|---|---|
| . | .* | :: | ?: | sizeof |

| Operators that can be overloaded | | | | | | | |
|---|---|---|---|---|---|---|---|
| + | - | * | / | % | ^ | & | \| |
| ~ | ! | = | < | > | += | -= | *= |
| /= | %= | ^= | &= | \|= | << | >> | >>= |
| <<= | == | != | <= | >= | && | \|\| | ++ |
| -- | ->* | , | -> | [] | () | new | delete |
| new[] | delete[] | | | | | | |

# Restrictions

- Cannot change
  - How operators act on built-in data types
    - i.e., cannot change integer addition
  - Precedence of operator (order of evaluation)
    - Use parentheses to force order-of-operations
  - Associativity (left-to-right or right-to-left)
  - Number of operands
    - ++ is unitary, only acts on one operand
- Cannot create new operators
- Operators must be overloaded explicitly
  - Overloading + does not overload +=

# Operator functions

- An operator is overloaded by writing a `non-static` member function definition or non-member function definition as you normally would, except that the function name starts with the keyword `operator` followed by the symbol for the operator being overloaded.

- For example, the function name `operator+` would be used to overload the addition operator (+) for use with objects of a particular class.

void operator ++( ) *// ++ obj -> obj.operator++( )*
friend void operator ++ (complex) ) *// ++ obj -> operator++(obj )*
complex operator + (complex) ) *// obj3=obj1+ obj2 -> obj3=ob1.operator+(obj2 )*
friend void operator + (complex, complex) *// obj3=obj1+ obj2 -> obj3=operator+(obj1,obj2 )*

# Operator functions

- Member functions
  - When operators are overloaded as member functions, they must be non-static, because they must be called on an object of the class and operate on that object.
  - Use **`this`** keyword to implicitly get argument
- Non member functions
  - Need parameters for both operands (for binary operators)
  - Must be a **`friend`** to access **`private`** or **`protected`** data
- Arguments can be passed by value or by reference

# Overloading Unary operator ‘-’

```
class abc
{
int m;
public:
   abc(){ }
abc(int n): m(n)
        {  }
void show()
    {   cout<<m;   }
void operator - ()
    {   m=-m;    }
};
```

```
main()
{
abc x(10);
x.show();
-x;
x.show();
}
```

# Overloading Unary operator ‘-’

```
class abc
{
int m;
public:
   abc(){  }
abc(int n): m(n){  }
void show()
    {
    cout<<m;
    }
abc operator - ();
};
```

```
abc  abc::operator - ()
{
        m=-m;
        return *this;
}

main()
{
    abc x(10),y;
    x.show();
    -x;
    x.show();
    y = -x;
    y.show();
}
```

# Overloading Unary operators ++,--

- Increment/decrement operators can be overloaded
  - Add 1 to a **`Date`** object, **`d1`**
  - Prototype (member function)
    - **`Date &operator++();`**
    - **`++d1`** same as **`d1.operator++()`**
  - Prototype (non-member)
    - **`Friend Date &operator++( Date &);`**
    - **`++d1`** same as **`operator++( d1 )`**

# Overloading Unary operators ++,--

- To distinguish pre/post increment
  - Post increment has a dummy parameter
    - **`int`** of **`0`**
  - Prototype (member function)
    - **`Date operator++( int );`**
    - **`d1++`** same as **`d1.operator++( 0 )`**
  - Prototype (non-member)
    - **`friend Date operator++( Data &, int );`**
    - **`d1++`** same as **`operator++( d1, 0 )`**
  - Integer parameter does not have a name
    - Not even in function definition

# Overloading Unary operators ++,--

```
class counter{
int cnt;
public:
counter():cnt(10) { }
counter(int c):cnt(c){ }
void show()
    {   cout<<cnt;   }
void operator -- ()
    {   - -cnt;    }
void operator -- (int)
    {   cnt- -;   }
counter operator ++();
friend counter operator ++(counter& ,int);
};
```

```
counter counter:: operator ++()
    {    return counter(++cnt) ;    }

counter operator++ (counter& c,int)
    {   return counter (c.cnt++ );     }

main()
{
    counter x,y;
    x.show();
    - -x;    x.show();
    x- -;    x.show();
    y=++x;  y.show();
    y=x++;   y.show();
}
```

# Overloading Binary operators +,-,*,/,%

```
class comp
{    int real, imag;
   public:
      comp():real(0),imag(0){}
      comp(int r, int i):real(r), imag(i){}
      comp operator + (const comp&);
      friend comp operator -(const comp &,const comp&);
      void display()
         {     cout<<real<<"+"<<imag<<"i"<<endl;    }
};

comp comp :: operator +(const comp &c2)
     {     return comp ( real + c2.real, imag + c2.imag );    }

comp operator -(const comp &c1, const comp &c2)
     { return comp ( c1.real - c2.real,c1.imag - c2.imag );}
```

```
main()
{
   comp n1(10,20),n2(30,40), n3;
   n3= n1 + n2;
   cout<<"First number:";
   n1.display();
   cout<<"Second number:";
   n2.display();
   cout<<"After addition:";
   n3.display();
   n3= n2 - n1;
   cout<<"After subtraction:";
   n3.display();
}
```

# Overloading Binary operators +,-,*,/,%

- In above example, **operator+ () using member function** has only one parameter even though it overloads the binary + operator.
- The reason that operator+ () takes only one parameter is that the operand on the left side of the + is passed implicitly to the function through the **this** pointer. The operand on the right is passed in the parameter c.
- The statement **n3= n1 + n2;** is same as *n3=n1.operator+(n2).*

- Similarly, **operator - () using non-member function** has both parameters passed explicitly.
- The statement **n3= n2 - n1;** is same as *n3=operator-(n2, n1).*

# Overloading Relational operators ==, !=, <, >, >=, <=

```
class length
{    int mtr, cmtr;
   public:
      length():mtr(0),cmtr(0){ }
      length(int m, int cm): mtr(m), cmtr(cm){ }
      int operator > ( length&);
      friend bool operator < (length &,length&);
      void display()
      {
         cout<<mtr<<"m"<<cmtr<<"cm"<<endl;
      }
};
```

```
main()
{
   length L1(10,20),L2(30,40), L3(30,50);

   cout<<"L1=";          L1.display();
   cout<<"L2=";          L2.display();
   cout<<"L3=";          L3.display();

   if(L1>L2)
         cout<<" L1is greater than L2"<<endl;
   else
         cout<<" L1is not greater than L2"<<endl;

   if(L2<L3)
         cout<<" L2is less than L3"<<endl;
   else
         cout<<" L2is less not than L3"<<endl;
}
```

# Overloading Relational operators ==, !=, <, >, >=, <=

```
int length :: operator >( length &c2)
{    if(mtr>c2.mtr)
        return true;
    else if(mtr==c2.mtr && cmtr>c2.cmtr)
        return true;
    else
            return false;
}
```

```
bool operator <( length &c1, length &c2)
{    if(c1.mtr<c2.mtr)
        return true;
    else if(c1.mtr==c2.mtr && c1.cmtr<c2.cmtr)
        return true;
    else
        return false;
}
```

# Chapter 5: Part-2
# Data Conversion

# Introduction

- Data conversion deals with techniques for converting variables from one type to another.
- The type conversion is either implicit or explicit as far as data types involved are built in types.
- We can also use the assignment operator in case of objects to copy values of all data members of right hand object to the object on left hand. The objects in this case are of same data type.
- But of objects are of different data types we must apply conversion rules for assignment.

# Introduction

- Conversion of one type to another is achieved by the use of constructors and type conversion functions.
- Three types of situations might arise for data conversion between different types :
  - (i) Conversion from basic type to class type.
  - (ii) Conversion from class type to basic type.
  - (iii) Conversion from one class type to another class type.

# Conversion from basic to class

- To convert basic data type to class type, we use a constructor which takes single argument whose type is to be converted
- This one-argument constructor is called “**conversion constructor**”

# Conversion from basic to class

```
class time
{    int h,m,s;
public:
   time(){}
   time (int t) //conversion constructor
   {
      h=t/3600;
      m=t%3600/60;
      s=t%3600%60;
   }
   void display()

{
cout<<h<<":"<<m<<":"<<s<<endl;    }
};
```

```
main()
{
   time t1;
   int duration=7285;

   t1=duration; //implicit
   t1.display();

   time t2(7285); //explicit
   t2.display();
}
```

# Conversion from class to basic

- The constructor functions do not support conversion from a class to basic type.
- C++ allows us to define a overloaded casting operator that convert a class type data to basic type.
- The general form of an **overloaded casting operator function**, also referred to as a "**conversion function**", is:

```
operator typename ( )
{
//Program statments `
}
```

- This function converts a class type data to typename. For example, the operator double( ) converts a class object to type double

# Conversion from class to basic

- The casting operator should satisfy the following conditions.
  - It must be a class member.
  - It must not specify a return type.
  - It must not have any arguments. Since it is a member function, it is invoked by the object and therefore, the values used for conversion inside the function belongs to the object that invoked the function. As a result function does not need an argument.
- The casted operator is used as,
  - int x = int(obj); // explicit
  - int x = static_cast<int>(obj);  // explicit
  - int x = obj;  // implicit

# Conversion from class to basic

```
class time
{    int h,m,s;
public:
   time(){ }
   time (int h,int m,int s)
   {       this->h=h;
               this->m=m;
           this->s=s;
   }
   void display()
{
cout<<h<<":"<<m<<":"<<s<<endl;
 }
   operator int(); //conversion function
};
```

```
time::operator int()
{
   int duration;
   duration=h*3600 + m*60 +s;
   return duration;
}
main()
{
   time t1(2, 1, 25);
   int total;
   t1.display();
   total=t1;
   cout<<"Total sec:"<<total;
}
```

# Conversion from one class to other class

- Obj1 = Obj2 ; //Obj1 and Obj2 are objects of different classes.

- Obj1 is an object of class one and Obj2 is an object of class two. The class two type data is converted to class one type data and the converted value is assigned to the Obj1. Since the conversion takes place from class two to class one, two is known as the source and one is known as the destination class.

# Conversion from one class to other class

- Such conversion between objects of different classes can be carried out by either a constructor or a conversion function.
- Which form to use, depends upon where we want the type-conversion function to be located, whether in the source class or in the destination class.

| Conversion | Conversion takes place in | |
|---|---|---|
| | Source class | Destination class |
| Basic to class | Not applicable | Constructor |
| Class to Basic | Casting operator | Not applicable |
| Class to class | Casting operator | Constructor |

# Conversion from one class to other class –

## Casting operator in source class

- Operator typename( ) converts the class object of which it is a member to typename. The type name may be a built-in type or a user defined one(another class type) .
- In the case of conversions between objects, typename refers to the destination class. Therefore, when a class needs to be converted, a casting operator function can be used.
- The conversion takes place in the source class and the result is given to the destination class object.

# Conversion from one class to other class –

## Casting operator in source class

```
class dest
{   int duration;
  public:
      dest(){}
      dest(int d):duration(d){ }
      void display()
      {     cout<<"Total duration:"<<duration<<endl; }
};

class src
{    int h,m,s;
public:
    src(){}
    src (int h,int m,int s)
    {       this->h=h;       this->m=m;       this->s=s;    }
    void display()
    {       cout<<h<<":"<<m<<":"<<s<<endl;   }
    operator dest() //conversion function
    {
    return dest(h*3600 + m*60 +s);
    }
};

main()
{    src s1(2, 1, 25);
    dest d1;
    s1.display();
    d1=s1;
    d1.display();
}
```

# Conversion from one class to other class

## — Conversion constructor in destination class

- Let us consider a single-argument constructor function which serves as an instruction for converting the argument's type to the class type of which it is a member.
- The argument belongs to the source class and is passed to the destination class for conversion. Therefore the conversion constructor must be placed in the destination class.
- When a conversion using a constructor is performed in the destination class, we must be able to access the data members of the object sent (by the source class) as an argument.
- Since data members of the source class are private, we must use special access functions in the source class to facilitate its data flow to the destination class.

# Conversion from one class to other class –

## Conversion constructor in destination class

```
class src
{    int h,m,s;
public:
    src(){ }
    src (int h,int m,int s)
    {        this->h=h;   this->m=m;    this->s=s;    }
    void display()
    {        cout<<h<<":"<<m<<":"<<s<<endl;    }
    int get_h()
    {    return h;    }
    int get_m()
    {    return m;    }
    int get_s()
    {    return s;    }
};
```

```
class dest
{   int duration;
   public:
        dest(){}
        dest(src s)
        {
duration = 3600*s.get_h() + 60*s.get_m() + s.get_s();        }
        void display()
        {          cout<<"Total duration:"<<duration<<endl;   }
};
main()
{    src s1(2, 1, 25);
    dest d1;
     d1=s1;
 s1.display();
d1.display();
}
```

# Explicit Constructors

- Constructor with one argument is used for implicit conversion in which the argument type is converted to object of class in which it is defined.
- If we do not want automatic conversion, we define explicit constructor so that the type conversion is done explicitly.

```
class xyz
{ int a;
public:
explicit xyz(int i)
{ a=i ;}
};
```

```
main()
{
xyz x1(20); // OK
xyz x2 = xyz(30); // OK
// xyz x3 = 10; is not allowed
}
```