

**Department of Computer Science and Engineering**

**PES UNIVERSITY**

**Object Oriented Programming with C++**

**UE22CS221A**

## **UNIT 3**

**Ms. Kusuma K V**

## Friend Function

A friend function of a class

- has access to the private and protected members of the class
- must have its prototype included within the scope of the class prefixed with the keyword friend
- does not have its name qualified with the class scope (because not a member function)
- is not called with an invoking object of the class (because not a member function)
- can be declared friend in more than one classes

A friend function of a class can be a

- global function
- a member function of a another class
- a function template

Friend Function (Basic Notion)

Ordinary function	friend function
<pre>#include&lt;iostream&gt; using namespace std;  class MyClass { int data_; public:     MyClass(int i) : data_(i) {}  }; void display(const MyClass&amp; a) {     cout &lt;&lt; "data = " &lt;&lt; a.data_; // Error 1 } int main(){     MyClass obj(10);      display(obj);      return 0; }</pre>	<pre>#include&lt;iostream&gt; using namespace std;  class MyClass { int data_; public:     MyClass(int i) : data_(i) {}      friend void display(const MyClass&amp; a); }; void display(const MyClass&amp; a) {     cout &lt;&lt; "data = " &lt;&lt; a.data_; // Okay } int main(){     MyClass obj(10);      display(obj);      return 0; }</pre>
<ul style="list-style-type: none"> <li>• <b>display() is a non-member function</b></li> <li>• <b>Error 1: 'MyClass::data_' : cannot access private member declared in class 'MyClass'</b></li> </ul>	<ul style="list-style-type: none"> <li>• <b>display() is a non-member function; but friend to class MyClass</b></li> <li>• <b>Able to access data_ even though it is private in class MyClass</b></li> <li>• <b>Output: data = 10</b></li> </ul>

---

### //friendFn\_Basic.cpp

//Non member function print is able to access private data, because it is a friend to the class

```
#include<iostream>
```

```
using namespace std;
```

```
class MyFriend
```

```
{
```

```
    int m_a;
```

```
public:
```

```
    MyFriend(int a):m_a(a){}
```

```
    friend void print(MyFriend);
```

```
};
```

```
void print(MyFriend o){cout<<"a="<<o.m_a<<endl;}
```

```
int main()
```

```
{
```

```
    MyFriend obj1(10);
```

```
    print(obj1);
```

```
    return 0;
```

```
}
```

/\*Output:

a=10

\*/

### Multiply a Matrix with a Vector

#### //FriendFn\_VecMat.cpp

//A function which multiplies a vector to a matrix needs access to internals of

//both vector and matrix class

//So make prodVectorMatrix a global function and a friend to both the classes

//prodVectorMatrix should be defined after both the classes are defined

```
#include<iostream>
```

```
using namespace std;
```

```
class Matrix;           //Forward Declaration
```

```
class Vector
```

```
{
```

```
    int v[3];
```

```
    int m_n;
```

```
public:
```

```
    Vector(int n):m_n(n)
```

```
    {
```

```
        for(int i=0;i<m_n;i++)
```

```
            v[i]=i;
```

```
    }
```

```
    void clear()
```

```
{
    for(int i=0;i<m_n;i++)
        v[i]=0;
}
void showVector()
{
    for(int i=0;i<m_n;i++)
        cout<<v[i]<<endl;
}
friend Vector prodVectorMatrix(Matrix*,Vector*);
};
class Matrix
{
    int m[3][3];
    int m_r,m_c;
public:
    Matrix(int r,int c):m_r(r),m_c(c)
    {
        for(int i=0;i<m_r;i++)
            for(int j=0;j<m_c;j++)
                m[i][j]=i+j;
    }
    void showMatrix()
    {
        for(int i=0;i<m_r;i++)
        {
            for(int j=0;j<m_c;j++)
            {
                cout<<m[i][j]<<" ";
            }
            cout<<endl;
        }
    }
    friend Vector prodVectorMatrix(Matrix*,Vector*);
};
Vector prodVectorMatrix(Matrix* pM,Vector* pV)
{
    Vector res(pM->m_r);
    res.clear();

    for(int i=0;i<pM->m_r;i++)
        for(int j=0;j<pV->m_n;j++)
            res.v[i]=res.v[i]+pM->m[i][j]*pV->v[j];

    return res;
}
int main()
{
```

```
Matrix m(2,3);
Vector v(3);
Vector prod=prodVectorMatrix(&m,&v);
cout<<"Matrix"<<endl;
m.showMatrix();
cout<<"Vector"<<endl;
v.showVector();
cout<<"Result"<<endl;
prod.showVector();
return 0;
```

```
}
```

/\*Output:

Matrix

0 1 2

1 2 3

Vector

0

1

2

Result

5

8

\*/

### Linked List

**//friendFn\_LinkedList.cpp**

//List class which is built on Node class needs to access internals of node

//So make the required functions of List class as a friend to Node class

//In this type of pgms, it's better to make the class itself as a friend(see FriendClass\_LinkedList.cpp)

```
#include <iostream>
```

```
using namespace std;
```

```
class Node;           // Forward declaration
```

```
class List
```

```
{
```

```
    Node *head;        // Head of the list
```

```
    Node *tail;        // Tail of the list
```

```
public:
```

```
    List():head(NULL),tail(NULL) {}
```

```
    void display();
```

```
    void append(Node *p);
```

```
};
```

```
class Node {
```

```
    int info;          // Data of the node
```

```
    Node *next;        // Ptr to next node
```

```
public:
    Node(int i): info(i), next(NULL) { }
    friend void List::display();
    friend void List::append(Node *);
};

void List::display()
{
    Node *ptr = head;
    while (ptr)
    {
        cout << ptr->info << " ";
        ptr = ptr->next;
    }
    cout<<endl;
}

void List::append(Node *p)
{
    if (!head)
        head = tail = p;
    else
    {
        tail->next = p;
        tail = tail->next;
    }
}

int main()
{
    List l; // Init null list
    Node n1(1), n2(2), n3(3); // Few nodes
    l.append(&n1); // Add nodes to list
    l.append(&n2);
    l.append(&n3);
    l.display(); // Show list

    return 0;
}

/*Output:
1 2 3
*/
```

## Friend Class

A friend class of a class

- has access to the private and protected members of the class
- does not have its name qualified with the class scope (not a nested class)

- can be declared friend in more than one class
- A friend class can be a
  - class
  - class template

### Linked List : friend class

#### //FriendClass\_LinkedList.cpp

//List class is now a friend of Node class. Hence it has full visibility into the internals of Node

//When multiple member functions need to be friends, it is better to use friend class

```
#include <iostream>
```

```
using namespace std;
```

```
class Node;           // Forward declaration
```

```
class List
```

```
{
```

```
    Node *head;       // Head of the list
```

```
    Node *tail;       // Tail of the list
```

```
public:
```

```
    List():head(NULL),tail(NULL) {}
```

```
    void display();
```

```
    void append(Node *p);
```

```
};
```

```
class Node {
```

```
    int info; // Data of the node
```

```
    Node *next; // Ptr to next node
```

```
public:
```

```
    Node(int i): info(i), next(NULL) {}
```

```
// friend void List::display();
```

```
// friend void List::append(Node *);
```

```
    friend class List; //List class is made friend to Node class
```

```
};
```

```
void List::display()
```

```
{
```

```
    Node *ptr = head;
```

```
    while (ptr)
```

```
    {
```

```
        cout << ptr->info << " ";
```

```
        ptr = ptr->next;
```

```
    }
```

```
    cout<<endl;
```

```
}
```

```
void List::append(Node *p)
```

```
{
```

```
    if (!head)
```

```
        head = tail = p;
```

```
    else
```

```
{
    tail->next = p;
    tail = tail->next;
}

int main()
{
    List l; // Init null list
    Node n1(1), n2(2), n3(3); // Few nodes
    l.append(&n1); // Add nodes to list
    l.append(&n2);
    l.append(&n3);
    l.display(); // Show list
    return 0;
}

/*Output:
1 2 3
*/
```

#### Note:

- friend-ship is neither commutative nor transitive
- A is a friend of B does not imply that B is a friend of A
- A is a friend of B and B is a friend of C does not imply that A is a friend of C
- Visibility and Encapsulation
  - public: a declaration that is accessible to all
  - protected: a declaration that is accessible only to the class itself and its subclasses
  - private: a declaration that is accessible only to the class itself
  - friend: a declaration that is accessible only to friend's of a class.
  - friend's tend to break data hiding and should be used judiciously.Like:
  - A function needs to access the internals of two (or more) independent classes (Eg: Matrix-Vector Multiplication)
  - A class is built on top of another (Eg: List-Node Access, List Iterator)
  - Certain situations of operator overloading (like streaming operators)

## Operator Functions

Operator overloading lets us define the meaning of an operator when applied to operand(s) of a class type. Judicious use of operator overloading can make our programs easier to write and easier to read.



---

## Why Overload Operators?

The reasons vary for the different operators, but the general guiding principle is to make your classes behave like built-in types.

- The closer your classes are to built-in types, the easier they will be for clients to use. For example, if you want to write a class to represent fractions, it's quite helpful to have the ability to define what + , - , \* , and / mean when applied to objects of that class.
- The second reason to overload operators is to gain greater control over the behavior in your program. For example, you can overload memory allocation and deallocation routines for your classes to specify exactly how memory should be distributed and reclaimed for each new object.

It's important to emphasize that operator overloading doesn't necessarily make things easier for you as the class developer; its main purpose is to make things easier for clients of the class.

## Choices in Operator Overloading

When you overload an operator, you write a function or method with the name `operatorX` , where X is the symbol for some operator, and with optional whitespace between operator and X .

The overloaded operator function you write may be Method or Global Function. First, you must decide whether your operator should be a method of your class or a global function (usually a friend of the class). How do you choose? First, you need to understand the difference between these two choices. When the operator is a method of a class, the left-hand side of the operator expression must always be an object of that class. If you write a global function, the left- hand side can be an object of a different type.

There are three different types of operators:

- Operators that must be methods: The C++ language requires some operators to be methods of a class because they don't make sense outside of a class. For example, `operator=` is tied so closely to the class that it can't exist anywhere else. Most operators do not impose this requirement.
- Operators that must be global functions: Whenever you need to allow the left-hand side of the operator to be a variable of a different type than your class, you must make the operator a global function. This rule applies specifically to `operator<<` and `operator>>` , where the left-hand side is the `iostream` object, not an object of your class. Additionally, commutative operators like binary + and – should allow variables that are not objects of your class on the left-hand side.
- Operators that can be either methods or global functions: There is some disagreement in the C++ community on whether it's better to write methods or global functions to overload operators. However, following rule may be followed: Make every operator a method unless you must make it a global function, as described previously. One major advantage to this rule is that methods can be virtual , but friend functions cannot. Therefore, when you plan to write overloaded operators in an inheritance tree, you should make them methods if possible.

---

Good practice: When you write an overloaded operator as a method, you should mark it `const` if it doesn't change the object. That way, it can be called on `const` objects.

### Choosing Argument Types

You are somewhat limited in your choice of argument types because as stated earlier for most operators you cannot change the number of arguments. For example, `operator/` (binary operator) must always have two arguments if it is a global function; one argument if it's a method. The compiler issues an error if it differs from this standard. In this sense, the operator functions are different from normal functions, which you can overload with any number of parameters.

The real choice arises when you try to determine whether to take parameters by value or by reference, and whether or not to make them `const`.

- a) The choice of value vs. reference is easy: you should take every non-primitive parameter type by reference.
- b) The `const` decision is also trivial: mark every parameter `const` unless you actually modify it.

### Choosing Return Types

C++ doesn't determine overload resolution based on return type. Thus, you can specify any return type you want when you write overloaded operators. However, just because you can do something doesn't mean you should do it. This flexibility implies that you could write confusing code in which comparison operators return pointers, and arithmetic operators return bools. However, you shouldn't do that.

Instead, you should write your overloaded operators such that they return the same types as the operators do for the built-in types. If you write a comparison operator, return a `bool`. If you write an arithmetic operator, return an object representing the result of the arithmetic.

Sometimes the return type is not obvious at first. For example, `operator=`, `insertion(<<)`, `extraction operator(>>)` should return a reference to the object on which it's called in order to support nested(chained) assignments.

### Choosing Behavior

You can provide whichever implementation you want in an overloaded operator. For example, you could write an `operator+` that launches a game of Scrabble. However, you should generally constrain your implementations to provide behaviors that clients expect. Write `operator+` so that it performs addition, or something like addition, such as string concatenation. `operator+` in a `Set` class should compute union and NOT intersection.

Here we discuss how you should implement your overloaded operators. In exceptional circumstances, you might want to differ from these recommendations; but, in general, you should follow the standard patterns.

Overloaded operators are functions with special names: the keyword `operator` followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type, a parameter list, and a body.

An operator function has the same number of parameters as the operator has operands. A unary operator has one parameter; a binary operator has two. In a binary operator, the left-hand operand is passed to the first parameter and the right-hand operand to the second. Except for the overloaded function-call operator, operator(), an overloaded operator may not have default arguments.

If an operator function is a member function, the first (left-hand) operand is bound to the implicit this pointer. Because the first operand is implicitly bound to this, a member operator function has one less (explicit) parameter than the operator has operands.

An operator function must either be a member of a class or have at least one parameter of class type:

```
// error: cannot redefine the built-in operator for ints
int operator+(int, int);
```

This restriction means that we cannot change the meaning of an operator when applied to operands of built-in type.

We can overload most, but not all, of the operators. Table 1 shows whether or not an operator may be overloaded.

Table 1: Operators

Operators That May Be Overloaded					
+	-	*	/	%	^
&		~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	
+=	-=	/=	%=	^=	&=
=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []
Operators That Cannot Be Overloaded					
::	.*	.	?:		

- We can overload only existing operators and cannot invent new operator symbols. For example, we cannot define operator\*\* to provide exponentiation.
- Four symbols (+, -, \*, and &) serve as both unary and binary operators. Either or both of these operators can be overloaded. The number of parameters determines which operator is being defined.
- An overloaded operator has the same precedence and associativity as the corresponding built-in operator. Regardless of the operand types.

$x == y + z;$  is always equivalent to  $x == (y + z).$

### Operator Overloading – Summary of Rules:

- No new operator such as \*\*, <>, or &| can be defined for overloading.
- Intrinsic properties of the overloaded operator cannot be change
  - Preserves arity
  - Preserves precedence

- Preserves associativity
- These operators can be overloaded:  
`[] + - * / % & | ~ ! = += -= *= /= %= = &= |= << >> >>= <<= == != < > <= >= && || ++ -- , ->* -> ( ) []`
- The operators `::` (scope resolution), `.` (member access), `.*` (member access through pointer to member), `sizeof`, and `?:` (ternary conditional) cannot be overloaded.
- The overloads of operators `&&`, `||`, and `,` (comma) lose their special properties: short-circuit evaluation and sequencing
- The overload of operator `->` must either return a raw pointer or return an object (by reference or by value), for which operator `->` is in turn overloaded.
- For a member operator function, invoking object is passed implicitly as the left operand but the right operand is passed explicitly
- For a non-member operator function (Global/friend) operands are always passed explicitly.

### Guidelines for Operator Overloading

- a) Use global function when encapsulation is not a concern. For example, using struct `String { char* str; }` to wrap a C-string and overload operator `+` to concatenate strings and build a `String algebra`
- b) Use member function when the left operand is necessarily a class where the operator function is a member and multiple types of operands are not involved
- c) Use friend function, otherwise
- d) While overloading an operator, try to preserve its natural semantics for built-in types as much as possible. For example, operator `+` in a `Set` class should compute union and NOT intersection
- e) Usually stick to the parameter passing conventions (built-in types by value and UDT's by constant reference)
- f) Decide on the return type based on the natural semantics for built-in types. For example, as in pre-increment and post-increment operators
- g) Consider the effect of casting on operands
- h) Only overload the operators that you may need (minimal design)

### Overloading Binary Operators

Complex no program to demonstrate overloading of plus operator which allows `complex+complex`, `complex+integer`, `integer+complex`. If operator `+` is a **member function** of the complex type, then only `complex+integer` would compile, and not `integer+complex`. So make it a **friend**.

`//ovrldPlsFrn.cpp`

`//Operator Plus, is commutative, to allow having a non object on LHS of operator`

`//it is implemented as a friend function`

---

//You may overload other arithmetic operators in the same manner

```
#include<iostream>
using namespace std;
class Complex
{
    double re;
    double im;
public:Complex(double r=0,double i=0){re=r;im=i;}
    void display(){cout<<re<<"+"<<im<<"i"<<endl;}
    friend Complex operator+(const Complex&,const Complex&);
};
Complex operator+(const Complex& c1,const Complex& c2)
{
    Complex sum;
    sum.re=c1.re+c2.re;
    sum.im=c1.im+c2.im;
    return sum;
}
int main()
{
    Complex c1(2,3);
    Complex c2(4,5);
    cout<<"1st Complex Number:";
    c1.display();
    cout<<"2nd Complex Number:";
    c2.display();

    cout<<"sum of c1 and c2:";
    Complex cSum1=c1+c2;           //calls operator+
    cSum1.display();

    //Complex cSum2=6.5+c is an Error, if operator+ is implemented as a member function
    //here works fine, because operator+ is implemented as friend fn
    Complex cSum2=6.5+c1;           //calls operator+
    cout<<"6.5 added to c1, sum:";
    cSum2.display();

    //Complex(0,6.5) is an anonymous object
    Complex cSum3=c1+Complex(0,6.5); //calls operator+
    cout<<"6.5i added to c1, sum:";
    cSum3.display();

    return 0;
}
/*
```

Output:

1st Complex Number:2+3i  
2nd Complex Number:4+5i  
sum of c1 and c2:6+8i  
6.5 added to c1, sum:8.5+3i  
6.5i added to c1, sum:2+9.5i  
\*/

//ovrldArithFrn.cpp

/\*Operator Plus, is commutative, to allow having a non object on LHS of operator, it is implemented as a friend function. Similarly are overloaded binary minus and multiplication operator below\*/

```
#include<iostream>
```

```
using namespace std;
```

```
class Complex
```

```
{
```

```
    double re;
```

```
    double im;
```

```
public:Complex(double r=0,double i=0){re=r;im=i;}
```

```
    void display(){cout<<re<<"+"<<im<<"i"<<endl;}
```

```
    friend Complex operator+(const Complex&,const Complex&);
```

```
    friend Complex operator-(const Complex&,const Complex&);
```

```
    friend Complex operator*(const Complex&,const Complex&);
```

```
};
```

```
Complex operator+(const Complex& c1,const Complex& c2)
```

```
{
```

```
    Complex sum;
```

```
    sum.re=c1.re+c2.re;
```

```
    sum.im=c1.im+c2.im;
```

```
    return sum;
```

```
}
```

```
Complex operator-(const Complex& c1,const Complex& c2)
```

```
{
```

```
    Complex diff;
```

```
    diff.re=c1.re-c2.re;
```

```
    diff.im=c1.im-c2.im;
```

```
    return diff;
```

```
}
```

```
Complex operator*(const Complex& c1,const Complex& c2)
```

```
{
    Complex prod;
    prod.re=(c1.re*c2.re)-(c1.im*c2.im);
    prod.im=(c1.re*c2.im)+(c1.im*c2.re);
    return prod;
}

int main()
{
    Complex c1(2,3);
    Complex c2(4,5);
    cout<<"1st Complex Number:";
    c1.display();
    cout<<"2nd Complex Number:";
    c2.display();

    cout<<"sum of c1 and c2:";
    Complex cSum1=c1+c2;           //calls operator+
    cSum1.display();

    //Complex cSum2=6.5+c is an Error, if operator+ is implemented as a member function
    //works fine, because operator+ is implemented as friend fn
    Complex cSum2=6.5+c1;           //calls operator+
    cout<<"6.5 added to c1, sum:";
    cSum2.display();

    //Complex(0,6.5) is an anonymous object
    Complex cSum3=c1+Complex(0,6.5); //calls operator+
    cout<<"6.5i added to c1, sum:";
    cSum3.display();

    cout<<"Difference of c1 and c2:";
    Complex cDiff1=c1-c2;           //calls operator-
    cDiff1.display();
    cout<<"Product of c1 and c2:";
    Complex cProd1=c1*c2;          //calls operator*
    cProd1.display();

    return 0;
}
/*
Output:
1st Complex Number:2+3i
2nd Complex Number:4+5i
```

sum of c1 and c2:6+8i

6.5 added to c1, sum:8.5+3i

6.5i added to c1, sum:2+9.5i

Difference of c1 and c2:-2+-2i

Product of c1 and c2:-7+22i

\*/

//equality\_relOpr.cpp

//You may overload other relational operators in the same manner

#include<iostream>

using namespace std;

class Complex

{

double re;

double im;

public:

Complex(double r=0,double i=0)

{

re=r;

im=i;

}

friend bool operator ==(const Complex& c1,const Complex& c2)

{

if(c1.re==c2.re && c1.im==c2.im)

return true;

return false;

}

friend bool operator !=(const Complex& c1,const Complex& c2)

{

return !(c1==c2);

}

friend bool operator <(const Complex& c1,const Complex& c2)

{

if(c1.re<c2.re)

return true;

if(c1.re==c2.re && c1.im<c2.im)

return true;

return false;

}



```
void display()
{
    cout<<re<<"+"<<im<<"i"<<endl;
}

};
int main()
{
    Complex c1(2,3);
    Complex c2(2,3);

    cout<<"c1=";
    c1.display();

    cout<<"c2=";
    c2.display();

    cout<<"Zero indicates false, one indicates true"<<endl;
    cout<<"c1 equals c2:"<<(c1==c2)<<endl;
    cout<<"c1 not equals c2:"<<(c1!=c2)<<endl;
    cout<<"c1 less than c2:"<<(c1<c2)<<endl;
    return 0;
}
/*
Output:
c1=2+3i
c2=2+3i
Zero indicates false, one indicates true
c1 equals c2:1
c1 not equals c2:0
c1 less than c2:0
*/
```

### Overloading Unary Operators

- The pre-operator should first perform the operation (increment / decrement / other) and then return the object. Hence its return type should be **MyClass&** and it should **return \*this**;
- The post-operator should perform the operation (increment / decrement / other) after it returns the original value. Hence it should copy the original object in a **temporary MyClass t**; and then **return t**; Its return type should be **MyClass**.

### Differentiating Prefix and Postfix Operators

There is one problem with defining both the prefix and postfix operators: Normal overloading cannot distinguish between these operators. The prefix and postfix versions use the same symbol, meaning that the overloaded versions of these operators have the same name. They also have the same number and type of operands.

To solve this problem, the postfix versions take an extra (unused) parameter of type int. When we use a postfix operator, the compiler supplies 0 as the argument for this parameter. Although the postfix function can use this extra parameter, it usually should not. That parameter is not needed for the work normally performed by a postfix operator. Its sole purpose is to distinguish a postfix function from the prefix .version.

#### //unaryOpr.cpp

```
//Vector can be represented using rectangular coordinates(x,y)
//Vector can be represented using polar coordinates(m,theta)
//In this pgm, only rectangular coordinate is considered
//unary operator functions are overloaded as methods of the class here
//unary operators take only one parameter: When implemented as method, the
//parameter is passed as an implicit this ptr
#include<iostream>
using namespace std;
class Vector
{
    double x;
    double y;
public:
    Vector(double x1=0,double y1=0);           //ctor taking default parameters
    void display();                             //displays vector
    Vector& operator-();                        //unary minus
    Vector& operator++();                      //pre increment
    Vector operator++(int);                    //post increment
};
Vector::Vector(double x1,double y1)
{
    x=x1;
    y=y1;
}
void Vector::display()
{
    cout<<"("<<x<<" "<<y<<"")<<endl;
}

Vector& Vector::operator-()
{
    x=-x;
    y=-y;
    return *this;
}

Vector& Vector::operator++()
```

```
{
    x++;
    y++;
    return *this;
}

Vector Vector:: operator++(int)
{
    Vector temp=*this;
    x++;
    y++;
    return temp;
}

int main()
{
    Vector v1(2,3);
    Vector v2(4,5);
    Vector v3(6,7);

    cout<<"Vector1: ";
    v1.display();
    Vector res1=++v1;           //calls operator++()
    cout<<"res1:pre incrementing vector1 ";
    res1.display();
    cout<<"vector1 after pre incrementing vector1 ";
    v1.display();
    cout<<endl;

    cout<<"Vector2: ";
    v2.display();
    Vector res2=v2++;           //calls operator++(int)
    cout<<"res2:post incrementing vector2 ";
    res2.display();
    cout<<"vector2 after post incrementing vector2 ";
    v2.display();
    cout<<endl;

    cout<<"Vector3: ";
    v3.display();
    Vector res3=-v3;           //calls operator-()
    cout<<"res3:applying unary minus on vector3 ";
    res3.display();
    cout<<"vector3 after applying unary minus ";
}
```

```
v3.display();

return 0;
}
/*
Output:
Vector1: (2,3)
res1:pre incrementing vector1 (3,4)
vector1 after pre incrementing vector1 (3,4)

Vector2: (4,5)
res2:post incrementing vector2 (4,5)
vector2 after post incrementing vector2 (5,6)

Vector3: (6,7)
res3:applying unary minus on vector3 (-6,-7)
vector3 after applying unary minus (-6,-7)
*/
```

### Overloading Index Operator

Classes that represent containers from which elements can be retrieved by position often define the subscript operator, `operator[]`.

To be compatible with the ordinary meaning of subscript, the subscript operator usually returns a reference to the element that is fetched. By returning a reference, subscript can be used on either side of an assignment. Consequently, it is also usually a good idea to define both const and nonconst versions of this operator. When applied to a const object, subscript should return a reference to const so that it is not possible to assign to the returned object.

#### //indexOpr.cpp

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

class IntList
{
private:
    int m_list[10];

public:
    IntList()                //ctor
    {
        for(int i=0;i<10;i++)
            m_list[i]=0;
    }
}
```

```
int& operator[] (int index);           //overload []
};
int& IntList::operator[] (int index)   //Fn defn for overload[]
{
    static int iErr = -1;
    if(index>0 && index <10)
        return m_list[index];
    cout<<"Index out of bound"<<endl;
    return iErr;
}

int main()
{
    IntList list;
    list[2] = 3;                       // set a value, calls operator[]
    cout <<"list[2]="<<list[2]<<endl;   // get a value, calls operator[]
    cout<<"list[9]="<<list[9]<<endl; // get a value, calls operator[]

    return 0;
}
/*
Output:
list[2]=3
list[9]=0
*/
```

### Conversion Function

Nonexplicit constructor that can be called with one argument defines an implicit conversion. Such constructors convert an object from the argument's type to the class type.

We can also define conversions from the class type. We define a conversion from a class type by defining a conversion operator. Converting constructors and conversion operators define **class-type conversions**. Such conversions are also referred to as **user-defined conversions**.

A conversion operator is a special kind of member function that converts a value of a class type to a value of some other type. A conversion function typically has the general form

**operator type() const;**

where type represents a type.

Conversion operators can be defined for any type (other than void) that can be a function return type. Conversions to an array or a function type are not permitted. Conversions to pointer types—both data and function pointers—and to reference types are allowed.

Conversion operators have **no explicitly stated return type** and **no parameters** and must have an **empty parameter list**, and they must be defined as **member functions**. Conversion operations

ordinarily should not change the object they are converting. As a result, conversion operators usually should be defined as **const** members.

#### [//conversionOpr.cpp](#)

```
#include <iostream>
using namespace std;
class Fraction
{
    int num, den;
public:
    Fraction(int n, int d)           //ctor
    {
        num = n;
        den = d;
    }
    // conversion operator: return float value of fraction
    operator float() const {
        return float(num) / float(den);
    }
};
int main()
{
    Fraction f(2, 5);
    float val = f;                  //calls operator float()
    cout << val<<endl;

    return 0;
}
/*
Output:
0.4
*/
```

### Overloading Insertion (Output) and Extraction (Input) Operators

The object on the left of an extraction or insertion operator is the istream or ostream (such as cin or cout ), not an object of userdefined class. Because you can't add a method to the istream or ostream classes, you must write the extraction and insertion operators as global **friend function** of the required class.

#### IO Operators Must Be Nonmember Functions

Input and output operators that conform to the conventions of the iostream library must be ordinary nonmember functions. These operators cannot be members of our own class. If they were, then the left-hand operand would have to be an object of our class type:

Complex c1;

---

```
c1<< cout;           // if operator<< is a member of Complex
```

If these operators are members of any class, they would have to be members of `istream` or `ostream`. However, those classes are part of the standard library, and we cannot add members to a class in the library.

### Overloading the Output Operator <<

Ordinarily, the first parameter of an output operator is a **reference to a nonconst ostream object**. The `ostream` is nonconst because writing to the stream changes its state. The parameter is a reference because we cannot copy an `ostream` object.

The second parameter ordinarily should be a **reference to const of the class type** we want to print. The parameter is a reference to avoid copying the argument. It can be const because (ordinarily) printing an object does not change that object.

To be consistent with other output operators (i.e., to support chaining), `operator<<` normally **returns its ostream parameter**.

### Overloading the Input Operator >>

Ordinarily the first parameter of an input operator is a **reference to the stream** from which it is to read, and the second parameter is a **reference to the (nonconst) object** into which to read. The operator usually **returns a reference to its given stream**. The second parameter must be nonconst because the purpose of an input operator is to read data into this object.

[//ioOprOverld.cpp](#)

```
#include<iostream>
```

```
using namespace std;
```

```
class Complex
```

```
{
```

```
    double re;
```

```
    double im;
```

```
public:
```

```
    Complex()           //default or zero arg ctor
```

```
    {
```

```
        re=0;
```

```
        im=0;
```

```
    }
```

```
friend ostream& operator<<(ostream&,const Complex&);
```

```
friend istream& operator>>(istream&,Complex&);
```

```
};
```

```
ostream& operator<<(ostream &os,const Complex &c)
```

```
{
    os<<c.re<<"+"<<c.im<<"i"<<endl;
    return os;
}

istream& operator>>(istream &is,Complex &c)
{
    is>>c.re>>c.im;
    return is;
}

int main()
{
    Complex obj1,obj2;

    cout<<"Enter 2 complex no's"<<endl;
    cin>>obj1>>obj2;                                //calls operator>>

    cout<<"Complex no's are:"<<endl;
    cout<<obj1<<obj2;                                //calls operator<<

    return 0;
}

/*
Output:
Enter 2 complex no's
2 3
4 5
Complex no's are:
2+3i
4+5i
*/
```



**Inheritance:** Programming technique for defining a new class (known as a **derived class**, **super class**, **parent class**) in terms of an existing class (known as the **base class**, **sub class**, **child class**). The derived class inherits the members of the base class.

- We often find that one object is a specialization / generalization of another.
- Object Oriented Analysis and Design (OOAD) models this using **ISA** relationship.
- C++ models **ISA** relationship by **Inheritance of classes**.

Classes related by inheritance form a hierarchy. Typically there is a base class at the root of the hierarchy, from which the other classes inherit, directly or indirectly. These inheriting classes are known as derived classes. The base class defines those members that are common to the types in the hierarchy. Each derived class defines those members that are specific to the derived class itself.

### Inheritance in C++: Semantics

- Derived ISA Base
- **Data Members**
  - Derived class inherits all data members of Base class
  - Derived class may add data members of its own
- **Member Functions**
  - Derived class inherits all member functions of Base class
  - Derived class may **override** a member function of Base class by redefining it with the same signature
  - Derived class may **overload** a member function of Base class by redefining it with the same name; but different signature
- **Construction-Destruction**
  - A constructor of the Derived class must first call a constructor of the Base class to construct the Base class instance of the Derived class
  - The destructor of the Derived class must call the destructor of the Base class to destruct the Base class instance of the Derived class
- **Access Specification**
  - Derived class cannot access private members of Base class
  - Derived class can access protected, and public members of Base class

The syntax for derivation is as follows:

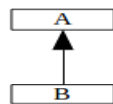
```
class <name of derived class> : <access specifier><name of base class>
{
    /*definition of base class*/
}
```

Eg: Suppose a class **A** already exists. Then, a new class (specialized) **B** can be derived from class (generalized) **A** as follows:

```
class B:public A
{
    /*new features of class B*/
}
```

A derived class must specify from which class(es) it inherits. It does so in its class derivation list, which is a colon followed by a comma-separated list of names of previously defined classes. Each base class name may be preceded by an optional access specifier, which is one of public, protected, or private. During derivation, if access specifier is not specified in class derivation list, the default derivation is private (for class and public for struct).

Diagrammatic representation of inheritance (B is a A) (A is Base class, B is derived class).



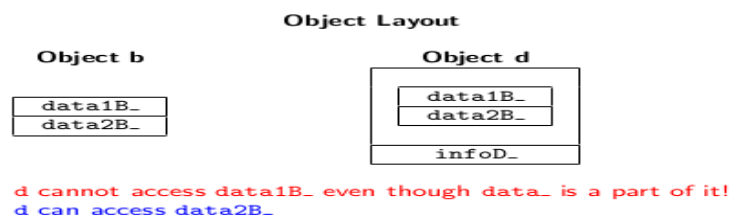
## Inheritance in C++: Data Members and Object Layout

- Derived ISA Base
- Data Members
  - Derived class inherits all data members of Base class
  - Derived class may add data members of its own
- Object Layout
  - Derived class layout contains an instance of the Base class
  - Further, Derived class layout will have data members of its own
  - C++ does not guarantee the relative position of the Baseclass instance and Derived class members

```
class B { // Base Class
    int data1B_;
public:
    int data2B_;
    // ...
};

class D: public B { // Derived Class
    // Inherits B::data1B_
    // Inherits B::data2B_
    int infoD_; // Adds D::infoD_
public:
    / ...
};

B b;
D d;
```



---

## //datamembers.cpp

//This program demonstrates the accessibility of base class and derived class data members

```
#include<iostream>
using namespace std;
class B
{
    int mb1;
public :
    int mb2;
    B(int b1=0,int b2=0)    //ctor of B with default parameters
    {
        mb1=b1;
        mb2=b2;
    }
};

class D: public B    //D derives from B in public mode
{
    int md1;
public:
    int md2;
    //ctor of D with default parameters, explicitly calls base class ctor: B(b1,b2)
    D(int b1=0,int b2=0,int d1=0,int d2=0):B(b1,b2)
    {
        md1=d1;
        md2=d2;
    }
};

int main()
{
    B b;
    D d;

    // cout<<b.mb1<<endl; //B:mb1 is not accessible to Derived class D, as well as main because mb1 is
    private
    cout<<b.mb2<<endl;
    // cout<<d.md1<<endl; //D:md1 is not accessible to main, because md1 is private
    cout<<d.md2<<endl;
    // cout<<d.mb1<<endl; //B:mb1 is not accessible to Derived class D, as well as main because mb1 is
    private
    cout<<d.mb2<<endl;
    return 0;
}
```

---

/\* Output:

0

0

0

\*/

## Inheritance in C++: Member Functions – Overrides and Overloads

- Derived ISA Base
- Member Functions
  - Derived class inherits all member functions of Base class
  - Derived class may **override** a member function of Base class by redefining it with the same signature
  - Derived class may **overload** a member function of Base class by redefining it with the same name; but different signature
  - Derived class may add new member functions
- Static Member Functions
  - Derived class does not inherit the static member functions of Base class
- Friend Functions
  - Derived class does not inherit the friend functions of Base

**//memberFn.cpp**

//This program demonstrates that all member functions of base class are inherited in derived class

```
#include<iostream>
```

```
using namespace std;
```

```
class B
```

```
{
```

```
    public: void x(){cout<<"I'm x of B"<<endl;}
```

```
        void y(){cout<<"I'm y of B"<<endl;}
```

```
};
```

```
class D: public B
```

```
{
```

```
    //inherits x() and y() of B
```

```
    //adds new z() in D
```

```
    public: void z(){cout<<"I'm new to D, not inherited from B"<<endl;}
```

```
};
```

```
int main()
```

```
{
```

```
B b;  
D d;  
  
b.x();  
b.y();  
  
d.x();    //calls the inherited x  
d.y();    //calls the inherited y  
d.z();    //calls z  
  
return 0;  
}  
/* Output:  
I'm x of B  
I'm y of B  
I'm x of B  
I'm y of B  
I'm new to D, not inherited from B  
*/
```

Derived classes inherit the members of their base class. However, a derived class needs to be able to provide its own definition for operations. In such cases, the derived class needs to override the definition it inherits from the base class, by providing its own definition. Also it may overload the methods of base class.

### //ovrldOverrrd.cpp

//This program demonstrates overloading and overriding of methods in derived class

```
#include<iostream>  
using namespace std;  
  
class B  
{  
    public: void x(){cout<<"I'm x of B"<<endl;}  
           void y(){cout<<"I'm y of B"<<endl;}  
  
};  
  
class D: public B  
{  
    public: void x(){cout<<"I'm Overridden x of D"<<endl;} //Overrides B::x()  
           void y(int a){cout<<"I'm Overloaded y of D"<<" , a="<<a<<endl;} //Overloads B::y()  
           void z(){cout<<"I'm new to D, not inherited from B"<<endl;}  
}
```

```
};

int main()
{
    B b;
    D d;

    b.x();
    b.y();

    d.x();           //static polymorphism
    d.y(2);          //static polymorphism
    d.z();

    return 0;
}
/* Output:
I'm x of B
I'm y of B
I'm Overridden x of D
I'm Overloaded y of D, a=2
I'm new to D, not inherited from B
*/
```

Note: It is a good practice to make the methods to be overridden in derived class as virtual in base class. So that it supports dynamic polymorphism. Discussed in virtual functions further in this notes.

### Inheritance in C++: Constructor & Destructor

- Derived ISA Base
- Constructor-Destructor
  - Derived class inherits the Constructors and Destructor of Base class (but in a different semantics)
  - Derived class cannot override or overload a Constructor or the Destructor of Base class
- Construction-Destruction
  - A constructor of the Derived class must first call a constructor of the Base class to construct the Base class instance of the Derived class
  - The destructor of the Derived class must call the destructor of the Base class to destruct the Base class instance of the Derived class

[//ctorDtor1.cpp](#)

[//This program demonstrates the order in which ctor and dtor are called](#)

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

class B
{
    int m_b;
public:
    B(int);
    void show();
    ~B();
};

B::B(int b)
{
    cout<<"Base class ctor"<<this<<endl;
    m_b=b;
}
void B::show()
{
    cout<<"Base class data="<<m_b<<endl;
}

B::~~B()
{
    cout<<"Base class dtor"<<this<<endl;
}

class D:public B
{
    int m_d;
public:
    D(int,int);
    void show();
    ~D();
};

D::D(int b,int d):B(b)    //Note the call to Base class ctor in initialization list of derived class
                        ctor
{
    cout<<"Derived class ctor"<<this<<endl;
    m_d=d;
}
```

```
}

void D::show()
{
    B::show();
    cout<<"Derived class data="<<m_d<<endl;
}

D::~~D()
{
    cout<<"Derived class dtor"<<this<<endl;
}

int main()
{
    B b(10);
    b.show();
    D d(20,30);
    d.show();
    return 0;
}
```

/\*Output:

```
Base class ctor0x7ffc6379df10
Base class data=10
Base class ctor0x7ffc6379df20
Derived class ctor0x7ffc6379df20
Base class data=20
Derived class data=30
Derived class dtor0x7ffc6379df20
Base class dtor0x7ffc6379df20
Base class dtor0x7ffc6379df10
*/
```

Note: Constructors are invoked in the order of inheritance. i.e., First, base class constructor then member objects constructor, then derived class constructor is invoked. See ctorDtor2.cpp.

**//ctorDtor2.cpp**

//This program demonstrates the order in which ctor and dtor are called

//This program also demonstrates multi level inheritance

```
#include<iostream>
```

```
using namespace std;
```



```
class A
{
public:
    A(){ cout<<"A class ctor"<<endl; }
    ~A(){ cout<<"A class dtor"<<endl; }
};

class B:public A
{
public:
    B(){ cout<<"B class ctor"<<endl; }
    ~B(){ cout<<"B class dtor"<<endl; }
};

class C:public B
{
    B bObj; //bObj is an object of class B, so first A class ctor is invoked then B class ctor

public:
    C(){ cout<<"C class ctor"<<endl; }
    ~C(){ cout<<"C class dtor"<<endl; }
};

int main()
{
    C c;
    return 0;
}

/*Output:
A class ctor
B class ctor
A class ctor
B class ctor
C class ctor
C class dtor
B class dtor
A class dtor
B class dtor
A class dtor
*/
```

## Copy Constructors and Assignment Operator in Derived Classes

When defining a derived class, you need to be careful about copy constructors and operator=. We saw that the initialization phase of a derived-class constructor initializes the base-class part(s) of a derived object as well as initializes its own members. As a result, the copy and move constructors for a derived class must copy/move the members of its base part as well as the members in the derived. Similarly, a derived-class assignment operator must assign the members in the base part of the derived object. i.e., When a derived class defines a copy or move operation, that operation is responsible for copying or moving the entire object, including base-class members.

If your derived class does not have any special data (pointers, usually) that require a nondefault copy constructor or operator=, you don't need to have one, regardless of whether or not the base class has one. If your derived class omits the copy constructor or operator=, a default copy constructor or operator= will be provided for the data members specified in the derived class and the base class copy constructor or operator= will be used for the data members specified in the base class.

On the other hand, if you do specify a copy constructor in the derived class, you need to explicitly chain to the parent copy constructor, as shown in the following code. If you do not do this, the **default constructor (not the copy constructor!)** will be used for the parent portion of the object.

### Defining a derived copy constructor

When we define a copy or move constructor for a derived class, we ordinarily use the corresponding base-class constructor to initialize the base part of the object. The initializer **Base(d)** passes a Derived object to a base-class constructor. **Base(d)** will (ordinarily) match the Base copy constructor. The Derived object, d, will be bound to the Base& parameter in that constructor. The Base copy constructor will copy the base part of d into the object that is being created.

```
class Base
{
public:
    Base();                //Base Class ctor prototype
    Base(const Base& b);    //Base class copy ctor prototype
};
class Derived : public Base
{
public:
    Derived();              //Derived Class ctor prototype
    Derived(const Derived& d); //Derived class copy ctor prototype
};
Derived::Derived(const Derived& d) : Base(d)//Copy the base members
    /*Initializers for members of Derived*/ { /*.....*/ }
```

Had the initializer for the base class been omitted, the Base default constructor would be used to initialize the base part of a D object. Assuming D's constructor copies the derived members from d, this newly constructed object would be oddly configured: Its Base members would hold default values, while its D members would be copies of the data from another object.

// probably incorrect definition of the D copy constructor

// base-class part is default initialized, not copied

```
D(const D& d) /* member initializers, but no base-class initializer */  
{ /* ... */ }
```

### Defining a derived assignment operator

Like the copy and move constructors, a derived-class assignment operator, must assign its base part explicitly. The following code shows how to call the parent's assignment operator from the derived class:

```
Derived& Derived::operator=(const Derived& rhs)  
{  
    if (&rhs == this)                                //handle self assignment  
    {  
        return *this;  
    }  
  
    Base::operator=(rhs);                             // Calls parent's operator=  
    // Do necessary assignments for derived class  
    return *this;  
}
```

This operator starts by checking for self assignment. If self assignment then returns the current object itself. If not self assignment then explicitly calls the base-class assignment operator to assign the members of the base part of the derived object. The base-class operator will (presumably) correctly handle self-assignment and, if appropriate, will free the old value in the base part of the left-hand operand and assign the new values from rhs. Once that operator finishes, we continue doing whatever is needed to assign the members in the derived class.

### Access Specifiers

A derived class inherits the members defined in its base class. However, the member functions in a derived class may not necessarily access the members that are inherited from the base class. Like any other code that uses the base class, a derived class may access the public members of its base class but may not access the private members. However, sometimes a base class has members that it wants to let its derived classes use while still prohibiting access to those same members by other users. We specify such members after a protected access specifier.

- Derived Is A Base
- Access Specification
  - Derived class cannot access private members of Base class
  - Derived class can access public members of Base class
- Protected Access Specification
  - A new protected access specification is introduced for Base class
  - Derived class can access protected members of Base class
  - No other class or global function can access protected members of Base class

### //protected.cpp

//Demonstrates the accessibility of different data members in derived class when publicly derived

//Protected members are accessible only in its own class and its child class

```
#include<iostream>
using namespace std;
class B
{
    int b1;
    public : int b2;
    B(int=0,int=0,int=0);
    void showB(){cout<<"B::"<<b1<<" "<<b2<<" "<<b3<<endl;}
    protected: int b3;
};
B::B(int p,int q,int r)
{
    b1=p;
    b2=q;
    b3=r;
}
class D: public B
{
    //b1(inherited) is not accessible directly in D. Because it is private in B.
    //b2(inherited) accessible in D. Because it is public in B.
    //b3(inherited) accessible in D. Because it is protected in B.
    int d1;
    public: int d2;
    D(int p,int q,int r,int s,int t):B(p,q,r){d1=s; d2=t;}    //ctor
    void showD() {
        cout<<"Base class instance of D is ";
        B::showB();
        cout<<"Specific members of D are ";
```

---

```
    cout<<"D::"<<d1<<" "<<d2<<endl;
}
void showProtectedB() {
    cout<<"To show, protected members of base class are accessible in derived class"<<endl;
    cout<<"I'm protected member of base class B::b3, accessed in derived class D:"<<b3<<endl;
}
};
int main()
{
    B b(1,2,3);
    D d(10,20,30,40,50);
    //b1 and d1 are not accessible because they are private members
    //b3 is not accessible because it is protected member
    // cout<<b.b1<<endl;
    // cout<<b.b2<<endl; //accessible, because public in B
    // cout<<b.b3<<endl;
    // cout<<d.d1<<endl;
    // cout<<d.d2<<endl; //accessible, because public in D
    // cout<<d.b1<<endl;
    // cout<<d.b2<<endl; //accessible, because public in D
    // cout<<d.b3<<endl;
    cout<<"I'm object B"<<endl;
    b.showB();

    cout<<endl<<"I'm object D"<<endl;
    d.showD();

    cout<<endl;
    d.showProtectedB();

    return 0;
}
/* Output:
I'm object B
B::1 2 3

I'm object D
Base class instance of D is B::10 20 30
Specific members of D are D::40 50
```

To show, protected members of base class are accessible in derived class

I'm protected member of base class B::b3, accessed in derived class D:30

\*/

Visibility across Access and Inheritance

## Visibility Matrix

		Inheritance		
Visibility		public	protected	private
	public	public	protected	private
	protected	protected	protected	private
	private	private	private	private