MARUDHAR KESARI JAINCOLLEGE FOR WOMEN,VANIYAMBADI

PG AND RESEARCH DEPARTMENT OF COMPUTER SCIENCE

CLASS: I BSC COMPUTER SCIENCE

SUBJECT CODE: 23UCS11

SUBJECT NAME: Object Oriented Programming Concepts using C++

SYLLABUS

UNIT III

Operator Overloading: Overloading unary, binary operators –Overloading Friend functions –type conversion – Inheritance: Types of Inheritance – Single, Multilevel, Multiple, Hierarchal, Hybrid, Multi path inheritance – Virtual base Classes – Abstract Classes.

Operator Overloading

C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. Operator overloading is a compile-time polymorphism. For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +. Other example classes where arithmetic operators may be overloaded are Complex Numbers, Fractional Numbers, Big integers, etc.

What is a Unary Operator in C++?

To conduct mathematical and logical operations on numerical quantities, C++ includes a wide variety of operators. The unary operators are one such extensively used operator. Unary Operators work on the operations carried out on just one operand. Unary operators do not utilize two operands to calculate the result as binary operators do.

A single operand/variable is used with the unary operator to determine the new value of that variable. Unary operators are used with the operand in either the prefix or postfix position. Unary operators come in various forms and have right-to-left associativity and equal precedence.

These are some examples of, unary operators:

1. Increment operator (++),
2. Decrement operator (--),
3. Unary minus operator (-),
4. Logical not operator (!),
5. Address of (&), etc.

Overloading Unary Operators for User-Defined Classes:

Defining a Complex user-defined class in a C++ Program:

C++ Program:

```
#include <iostream>

using namespace std;

class Complex {
   private:
   int real, img;

   public: Complex() {
      real = 0;
      img = 0;
   }

   Complex(int r, int i) {
      real = r;
      img = i;
   }
}
```

There are two ways for unary operation overloading in C++, i.e.,

1. By adding the operator function as a class member function.
2. By using the global friend function created for the operator function.

Let's see both methods below using the Complex class defined above.

B. Overload Unary Minus (-) Operator using class Member function

C++ Program:

```
#include <iostream>

using namespace std;

class Complex {
   private: int real,
   img;

   public: Complex() {
      real = 0;
      img = 0;
   }

   Complex(int r, int i) {
      real = r;
      img = i;
   }

   // Printing the complex number in the output.
   void print() {
      int newImg = img < 0 ? -img : img;
      cout << real << (img < 0 ? " - " : " + ") << "i" << newImg << endl;
   }

   // Overloading unary Minus (-) operator.
   Complex operator - () {
      return Complex(-(this -> real), -(this -> img));
   }
```

```
};

int main() {
    // Instantiating a Complex object c1 with values.
    Complex c1(-3, 4);

    // Printing the c1 complex object in the output.
    cout << "c1 = ";
    c1.print();

    // Invoking the overloaded unary minus (-) on c1 object and
    // storing the returned object in a new c2 Complex object.
    Complex c2 = -c1;

    // Printing the c2 complex object in the output.
    cout << "c2 = ";
    c2.print();

    return 0;
}
```

Try it yourself

Output:

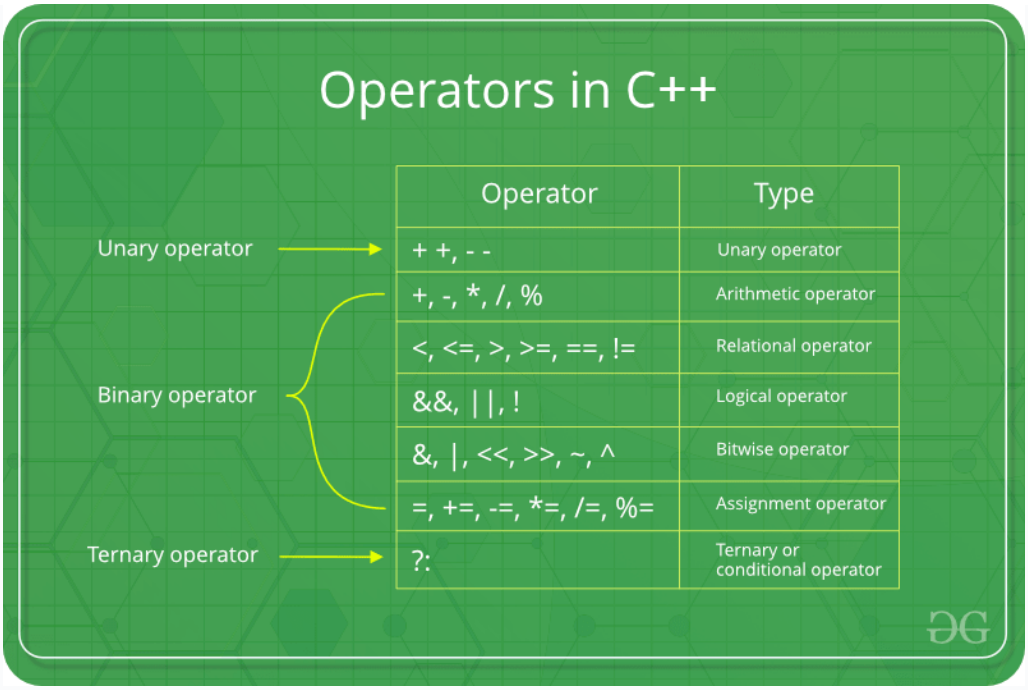
```
c1 = -3 + i4
c2 = 3 - i4
```

Explanation:

- In the above C++ program, we have overloaded a minus (-) unary operator to work with the Complex class objects.
- We can't simply use the minus (-) operator with the Complex class object. It will give a compilation error until we have defined the minus (-) operator w.r.t to the Complex class objects.
- We have provided a minus (-) operator overloaded definition in the Complex class. It is invoked when we have used the unary minus(-) operator in the main() function with the c2 object.

BINARY

- An **operator** is a symbol that operates on a value to perform specific mathematical or logical computations. They form the foundation of any programming language. In C++, we have built-in operators to provide the required functionality.
- An operator operates the **operands**. For example,
- int c = a + b;
- Here, '+' is the addition operator. 'a' and 'b' are the operands that are being 'added'.
- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Ternary or Conditional Operators



1) Arithmetic Operators

These operators are used to perform arithmetic or mathematical operations on the operands. For example, '+' is used for addition, '-' is used for subtraction '*' is used for multiplication, etc.

Arithmetic Operators can be classified into 2 Types:

A) Unary Operators: These operators operate or work with a single operand. For example: Increment(++) and Decrement(–) Operators.

Name	**Symbol**	**Description**	**Example**
Increment Operator	++	Increases the integer value of the variable by one	int a = 5; a++; // returns 6

Name	Symbol	Description	Example
Decrement Operator	—	Decreases the integer value of the variable by one	int a = 5; a–; // returns 4

Example:

the description

Output

a++ is 10

++a is 12

b-- is 15

--b is 13

Time Complexity: O(1)

Auxiliary Space : O(1)

Note: ***++a*** *and* ***a++****, both are increment operators, however, both are slightly different.*

In ***++a****, the value of the variable is incremented first and then It is used in the program. In* ***a++****, the value of the variable is assigned first and then It is incremented. Similarly happens for the decrement operator.*

B) Binary Operators: These operators operate or work with two operands. For example: Addition(+), Subtraction(-), etc.

Name	Symbol	Description	Example
Addition	+	Adds two operands	int a = 3, b = 6; int c = a+b; // c =

Name	Symbol	Description	Example
			9
Subtraction	–	Subtracts second operand from the first	int a = 9, b = 6; int c = a-b; // c = 3
Multiplication	*	Multiplies two operands	int a = 3, b = 6; int c = a*b; // c = 18
Division	/	Divides first operand by the second operand	int a = 12, b = 6; int c = a/b; // c = 2
Modulo Operation	%	Returns the remainder an integer division	int a = 8, b = 6; int c = a%b; // c = 2

Note: *The Modulo operator(%) operator should only be used with integers.*

Example:

- C++

```
// CPP Program to demonstrate the Binary Operators
```

```
#include <iostream>

using namespace std;



int main()

{

    int a = 8, b = 3;



    // Addition operator

    cout << "a + b = " << (a + b) << endl;



    // Subtraction operator

    cout << "a - b = " << (a - b) << endl;



    // Multiplication operator

    cout << "a * b = " << (a * b) << endl;
```

```
    // Division operator
    cout << "a / b = " << (a / b) << endl;

    // Modulo operator
    cout << "a % b = " << (a % b) << endl;

    return 0;
}
```

Output

a + b = 11

a - b = 5

a * b = 24

a / b = 2

a % b = 2

Time Complexity: O(1)

Auxiliary Space : O(1)

2) Relational Operators

These operators are used for the comparison of the values of two operands. For example, '>' checks if one operand is greater than the other operand or not, etc. The result returns a Boolean value, i.e., **true** or **false.**

Name	Symbol	Description	Example

Is Equal To	==	Checks if both operands are equal	int a = 3, b = 6;
			a==b;
			// returns false

Greater Than	>	Checks if first operand is greater than the second operand	int a = 3, b = 6;
			a>b;
			// returns false

Greater Than or Equal To	>=	Checks if first operand is greater than or equal to the second operand	int a = 3, b = 6;
			a>=b;
			// returns false

Name	Symbol	Description	Example

Less Than	<	Checks if first operand is lesser than the second operand	int a = 3, b = 6;
			a<b;
			// returns true

Less Than or Equal To	<=	Checks if first operand is lesser than or equal to the second operand	int a = 3, b = 6;
			a<=b;
			// returns true

Not Equal To	!=	Checks if both operands are not equal	int a = 3, b = 6;
			a!=b;
			// returns true

Example:

- C++

```
// CPP Program to demonstrate the Relational Operators
```

```
#include <iostream>

using namespace std;



int main()

{

    int a = 6, b = 4;



    // Equal to operator

    cout << "a == b is " << (a == b) << endl;



    // Greater than operator

    cout << "a > b is " << (a > b) << endl;



    // Greater than or Equal to operator

    cout << "a >= b is " << (a >= b) << endl;
```

```
    //  Lesser than operator
    cout << "a < b is " << (a < b) << endl;


    // Lesser than or Equal to operator
    cout << "a <= b is " << (a <= b) << endl;


    // true
    cout << "a != b is " << (a != b) << endl;


    return 0;
}
```

Output

```
a == b is 0
a > b is 1
a >= b is 1
a < b is 0
```

a <= b is 0

a != b is 1

Time Complexity: O(1)

Auxiliary Space : O(1)

Here, **0** denotes **false** and **1** denotes **true**. To read more about this, please refer to the article – Relational Operators.

3) Logical Operators

These operators are used to combine two or more conditions or constraints or to complement the evaluation of the original condition in consideration. The result returns a Boolean value, i.e., **true** or **false**.

Name	Symbol	Description	Example

Logical AND	&&	Returns true only if all the operands are true or non-zero	int a = 3, b = 6;
			a&&b;
			// returns true

Logical OR	||	Returns true if either of the operands is true or non-zero	int a = 3, b = 6;
			a||b;
			// returns true

Logical NOT	!	Returns true if the operand is false or zero	int a = 3; !a;

Name	Symbol	Description	Example
			// returns false

Example:

- C++

```
// CPP Program to demonstrate the Logical Operators
#include <iostream>
using namespace std;


int main()
{
    int a = 6, b = 4;


    // Logical AND operator
    cout << "a && b is " << (a && b) << endl;
```

```
    // Logical OR operator
    cout << "a ! b is " << (a > b) << endl;

    // Logical NOT operator
    cout << "!b is " << (!b) << endl;

    return 0;
}
```

Output

a && b is 1

a ! b is 1

!b is 0

Time Complexity: O(1)

Auxiliary Space : O(1)

Here, **0** denotes **false** and **1** denotes **true**. To read more about this, please refer to the article – Logical Operators.

4) Bitwise Operators

These operators are used to perform bit-level operations on the operands. The operators are first converted to bit-level and then the calculation is performed on the operands. Mathematical

operations such as addition, subtraction, multiplication, etc. can be performed at the bit level for faster processing.

Name	Symbol	Description	Example
Binary AND	&	Copies a bit to the evaluated result if it exists in both operands	int a = 2, b = 3; (a & b); //returns 2
Binary OR	|	Copies a bit to the evaluated result if it exists in any of the operand	int a = 2, b = 3; (a | b); //returns 3
Binary XOR	^	Copies the bit to the evaluated result if it is present in either of the operands but not both	int a = 2, b = 3; (a ^ b); //returns 1
Left Shift	<<	Shifts the value to left by the number of bits specified by the right operand.	int a = 2, b = 3; (a << 1); //returns 4
Right Shift	>>	Shifts the value to right by the number of bits	int a = 2, b =

Name	Symbol	Description	Example
		specified by the right operand.	3; (a >> 1); //returns 1

One's Complement	~	Changes binary digits 1 to 0 and 0 to 1	int b = 3;
			(~b);
			//returns -4

Note: *Only* ***char*** *and* ***int*** *data types can be used with Bitwise Operators.*

Example:

- C++

```
// CPP Program to demonstrate the Bitwise Operators

#include <iostream>

using namespace std;



int main()

{

    int a = 6, b = 4;
```

```
// Binary AND operator

cout << "a & b is " << (a & b) << endl;



// Binary OR operator

cout << "a | b is " << (a | b) << endl;



// Binary XOR operator

cout << "a ^ b is " << (a ^ b) << endl;



// Left Shift operator

cout << "a<<1 is " << (a << 1) << endl;



// Right Shift operator

cout << "a>>1 is " << (a >> 1) << endl;
```

```
    // One's Complement operator
    cout << "~(a) is " << ~(a) << endl;

    return 0;
}
```

Output

a & b is 4

a | b is 6

a ^ b is 2

a<<1 is 12

a>>1 is 3

~(a) is -7

Time Complexity: O(1)

Auxiliary Space : O(1)

To read more about this, please refer to the article – Bitwise Operators.

5) Assignment Operators

These operators are used to assign value to a variable. The left side operand of the assignment operator is a variable and the right side operand of the assignment operator is a value. The value

on the right side must be of the same data type as the variable on the left side otherwise the compiler will raise an error.

Namemultiply	**Symbol**	**Description**	**Example**
Assignment Operator	=	Assigns the value on the right to the variable on the left	int a = 2; // a = 2
Add and Assignment Operator	+=	First adds the current value of the variable on left to the value on the right and then assigns the result to the variable on the left	int a = 2, b = 4; a+=b; // a = 6
Subtract and Assignment Operator	-=	First subtracts the value on the right from the current value of the variable on left and then assign the result to the variable on the left	int a = 2, b = 4; a-=b; // a = -2
Multiply and Assignment Operator	*=	First multiplies the current value of the variable on left to the value on the right and then assign the result to the variable on the left	int a = 2, b = 4; a*=b; // a = 8
Divide and Assignment	/=	First divides the current value of the variable on left by the value on the right and then	int a = 4, b = 2; a /=b; //

Namemultiply	Symbol	Description	Example
Operator		assign the result to the variable on the left	a = 2

Example:

- C++

```
// CPP Program to demonstrate the Assignment Operators

#include <iostream>

using namespace std;



int main()

{

    int a = 6, b = 4;



    // Assignment Operator

    cout << "a = " << a << endl;
```

```
    //  Add and Assignment Operator

    cout << "a += b is " << (a += b) << endl;



    // Subtract and Assignment Operator

    cout << "a -= b is " << (a -= b) << endl;



    //  Multiply and Assignment Operator

    cout << "a *= b is " << (a *= b) << endl;



    //  Divide and Assignment Operator

    cout << "a /= b is " << (a /= b) << endl;



    return 0;

}
```

Output

```
a = 6
```

a += b is 10

a -= b is 6

a *= b is 24

a /= b is 6

Time Complexity: O(1)

Auxiliary Space : O(1)

6) Ternary or Conditional Operators(?:)

This operator returns the value based on the condition.

Expression1? Expression2: Expression3

The ternary operator **?** determines the answer on the basis of the evaluation of **Expression1**. If it is **true**, then **Expression2** gets evaluated and is used as the answer for the expression. If **Expression1** is **false**, then **Expression3** gets evaluated and is used as the answer for the expression.

This operator takes **three operands**, therefore it is known as a Ternary Operator.

Example:

- C++

```
// CPP Program to demonstrate the Conditional Operators

#include <iostream>

using namespace std;
```

```
int main()

{

    int a = 3, b = 4;



    // Conditional Operator

    int result = (a < b) ? b : a;

    cout << "The greatest number is " << result << endl;



    return 0;

}
```

Output

The greatest number is 4

Time Complexity: O(1)

Auxiliary Space : O(1)

7) There are some other common operators available in C++ besides the operators discussed above. Following is a list of these operators discussed in detail:

A) sizeof Operator: This unary operator is used to compute the size of its operand or variable.

sizeof(char); // returns 1

B) Comma Operator(,): This binary operator (represented by the token) is used to evaluate its first operand and discards the result, it then evaluates the second operand and returns this value (and type). It is used to combine various expressions together.

int a = 6;

int b = (a+1, a-2, a+5); // b = 11

C) -> Operator: This operator is used to access the variables of classes or structures.

cout<<emp->first_name;

D) Cast Operator: This unary operator is used to convert one data type into another.

float a = 11.567;

int c = (int) a; // returns 11

E) Dot Operator(.): This operator is used to access members of structure variables or class objects in C++.

cout<<emp.first_name;

F) & Operator: This is a pointer operator and is used to represent the memory address of an operand.

G) * Operator: This is an Indirection Operator

- C++

```
// CPP Program to demonstrate the & and * Operators

#include <iostream>

using namespace std;
```

```
int main()

{

    int a = 6;

    int* b;

    int c;

    //  & Operator

    b = &a;



    // * Operator

    c = *b;

    cout << " a = " << a << endl;

    cout << " b = " << b << endl;

    cout << " c = " << c << endl;



    return 0;
```

```
}
```

Output

a = 6

b = 0x7ffe8e8681bc

c = 6

H) << Operator: It is called the insertion operator. It is used with cout to print the output.

I) >> Operator: It is called the extraction operator. It is used with cin to get the input.

```
int a;

cin>>a;

cout<<a;
```

Time Complexity: O(1)

Auxiliary Space : O(1)

Operator Precedence Chart

Precedence	Operator	Description	Associativity
1.	()	Parentheses (function call)	left-to-right
	[]	Brackets (array subscript)	
	.	Member selection via object name	

Precedence	Operator	Description	Associativity
	->	Member selection via a pointer	
	++/–	Postfix increment/decrement	
	++/–	Prefix increment/decrement	right-to-left
	+/-	Unary plus/minus	
	!~	Logical negation/bitwise complement	
2.	(type)	Cast (convert value to temporary value of type)	
	*	Dereference	
	&	Address (of operand)	
	sizeof	Determine size in bytes on this implementation	
3.	*,/,%	Multiplication/division/modulus	left-to-right
4.	+/-	Addition/subtraction	left-to-right

Precedence	Operator	Description	Associativity
5.	<< , >>	Bitwise shift left, Bitwise shift right	left-to-right
6.	< , <=	Relational less than/less than or equal to	left-to-right
	> , >=	Relational greater than/greater than or equal to	left-to-right
7.	== , !=	Relational is equal to/is not equal to	left-to-right
8.	&	Bitwise AND	left-to-right
9.	^	Bitwise exclusive OR	left-to-right
10.	|	Bitwise inclusive OR	left-to-right
11.	&&	Logical AND	left-to-right
12.	||	Logical OR	left-to-right
13.	?:	Ternary conditional	right-to-left
14.	=	Assignment	right-to-left

Precedence	Operator	Description	Associativity
	+= , -=	Addition/subtraction assignment	
	*= , /=	Multiplication/division assignment	
	%= , &=	Modulus/bitwise AND assignment	
	^= , |=	Bitwise exclusive/inclusive OR assignment	
	<>=	Bitwise shift left/right assignment	
15.	,	expression separator	left-to-right

Last Updated : 24 Aug, 2023

Next

Basic Input / Output in C++

Similar Reads

Operators in C | Set 2 (Relational and Logical Operators)

Increment (Decrement) operators require L-value Expression

Order of operands for logical operators

Conversion Operators in C++

const_cast in C++ | Type Casting operators

Overloading stream insertion (<>) operators in C++

Unary operators in C/C++

reinterpret_cast in C++ | Type Casting operators

Left Shift and Right Shift Operators in C/C++

Operator Overloading using Friend Function in C++ with Examples:

In this article, I am going to discuss **Operator Overloading using Friend Function in C++** with Examples. Please read our previous article where we discussed **Operator Overloading in C++** with Examples. C++ Programming Language provides a special mechanism to change the current functionality of some operators within its class which is often called operator overloading. Operator Overloading is the method by which we can change the function of some specific operators to do some different tasks.

Friend Function Operator Overloading in C++:

In our previous article, we have already seen how to overlord unary (++, –) and binary (+) operators in C++ with Examples. There is one more method for overloading an operator in C++ that is using the friend function. Let us learn it through the same example that is using the same Complex class. The following is the sample code that we have created in our previous article.

```
class Complex

{

private:

int real;

int img;

public:

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

{

real = r;

img = i;

}

Complex add (Complex x)
```

```
{
Complex temp;
temp.real = real + x.real;
temp.img = img + x.img;
return temp;
}
void Display()
{
cout << real << "+i" << img <<endl;
}
};
int main()
{
Complex C1 (3, 7);
C1.Display();
Complex C2 (5, 2);
C2.Display();
Complex C3;
C3 = C1.add (C2); // C2.add(C1);
C3.Display();
}
```

Suppose we want to add two complex numbers i.e. C1 and C2,

C3 = C1 + C2;

Type Conversion in C++

A type cast is basically a conversion from one type to another. There are two types of type conversion:

1. **Implicit Type Conversion** Also known as 'automatic type conversion'.
   - Done by the compiler on its own, without any external trigger from the user.
   - Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid lose of data.
   - All the data types of the variables are upgraded to the data type of the variable with largest data type.
   - bool -> char -> short int -> int ->
   -
   - unsigned int -> long -> unsigned ->
   -
   - long long -> float -> double -> long double
   - It is possible for implicit conversions to lose information, signs can be lost (when signed is implicitly converted to unsigned), and overflow can occur (when long long is implicitly converted to float).

Explicit Type Conversion: This process is also called type casting and it is user-defined. Here the user can typecast the result to make it of a particular data type.

In C++, it can be done by two ways:

- **Converting by assignment:** This is done by explicitly defining the required type in front of the expression in parenthesis. This can be also considered as forceful casting.

Conversion using Cast operator: A Cast operator is an **unary operator** which forces one data type to be converted into another data type.

C++ supports four types of casting:

1. Static Cast
2. Dynamic Cast
3. Const Cast
4. Reinterpret Cast

Advantages of Type Conversion:

5. This is done to take advantage of certain features of type hierarchies or type representations.
6. It helps to compute expressions containing variables of different data types.

Inheritance in C++

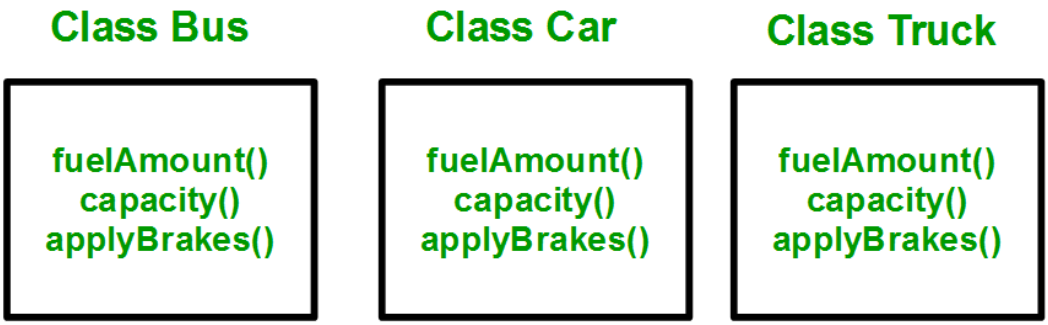
Inheritance is a feature or a process in which, new classes are created from the existing classes. The new class created is called “derived class” or “child class” and the existing class is known as the “base class” or “parent class”. The derived class now is said to be inherited from the base class.

When we say derived class inherits the base class, it means, the derived class inherits all the properties of the base class, without changing the properties of base class and may add new features to its own. These new features in the derived class will not affect the base class. The derived class is the specialized class for the base class.

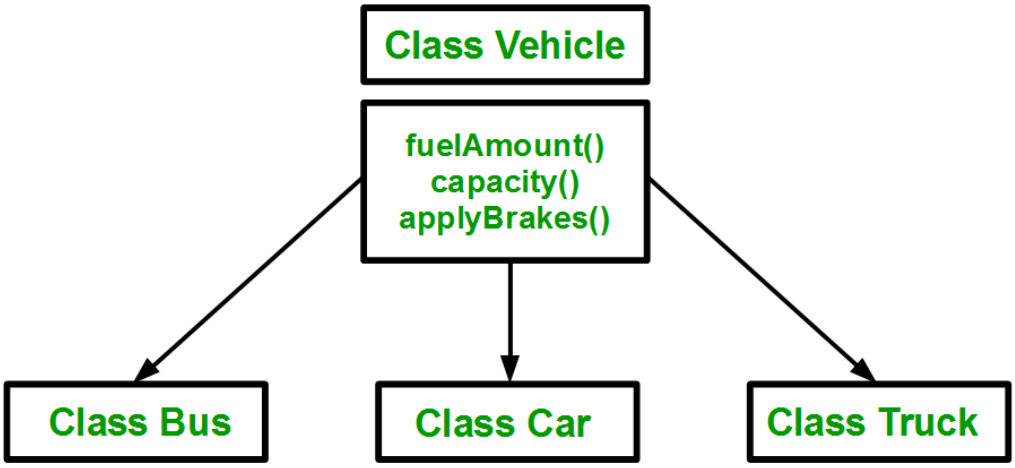
- **Sub Class:** The class that inherits properties from another class is called Subclass or Derived Class.
- **Super Class:** The class whose properties are inherited by a subclass is called Base Class or Superclass.

Why and when to use inheritance?

Consider a group of vehicles. You need to create classes for Bus, Car, and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be the same for all three classes. If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown below figure:



You can clearly see that the above process results in duplication of the same code 3 times. This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used. If we create a class Vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase re-usability. Look at the below diagram in which the three classes are inherited from vehicle class:



Using inheritance, we have to write the functions only one time instead of three times as we have inherited the rest of the three classes from the base class (Vehicle). **Implementing inheritance in C++**: For creating a sub-class that is inherited from the base class we have to follow the below syntax.

Derived Classes: A Derived class is defined as the class derived from the base class.

Syntax:

class <derived_class_name> : <access-specifier> <base_class_name>

{

//body

}

Example:

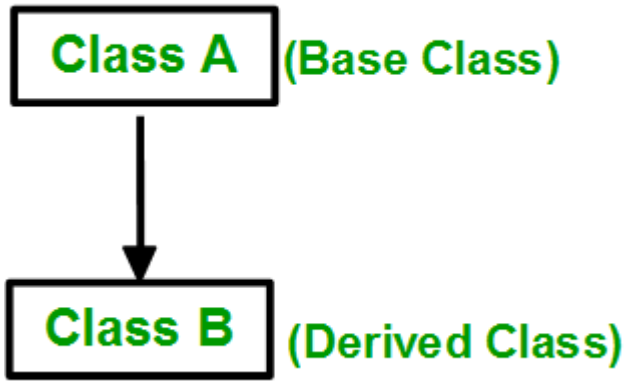
1. class ABC : private XYZ //private derivation
{ }
2. class ABC : public XYZ //public derivation
{ }
3. class ABC : protected XYZ //protected derivation
{ }
4. class ABC: XYZ //private derivation by default
{ }

Types Of Inheritance:-

1. Single inheritance
2. Multilevel inheritance
3. Multiple inheritance
4. Hierarchical inheritance
5. Hybrid inheritance

Types of Inheritance in C++

1. Single Inheritance: In single inheritance, a class is allowed to inherit from only one class. i.e. one subclass is inherited by one base class only.



Syntax:

class subclass_name : access_mode base_class

{

// body of subclass

};

OR

class A

{

... .

};

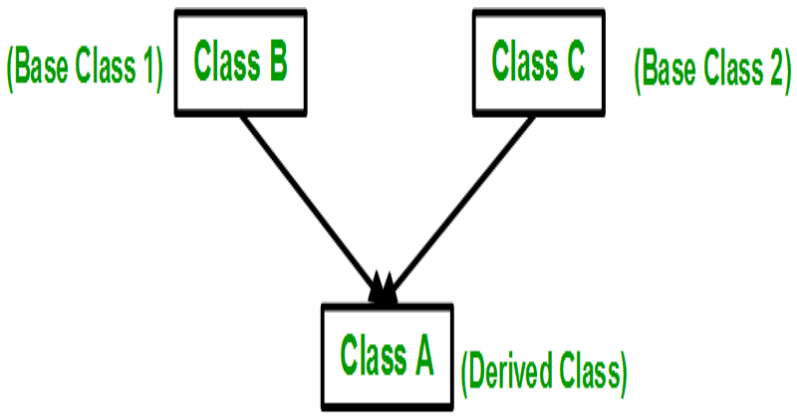
class B: public A

{

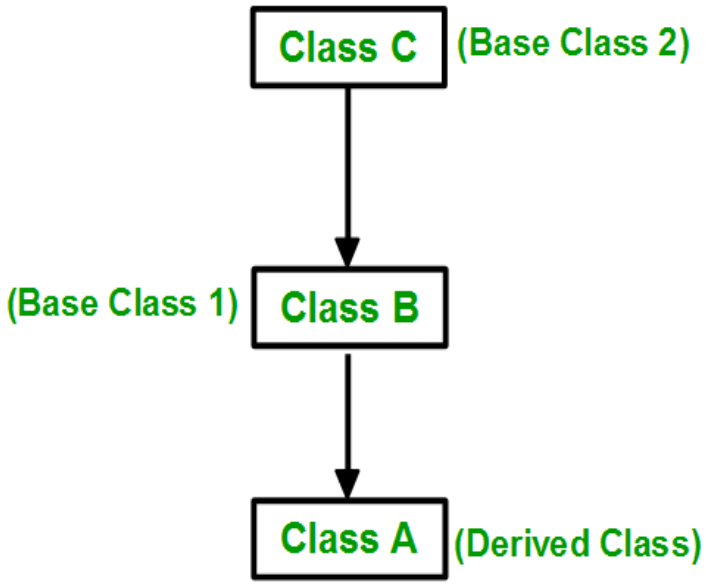
... .

};

Multiple Inheritance: Multiple Inheritance is a feature of C++ where a class can inherit from more than one class. i.e one **subclass** is inherited from more than one **base class**.



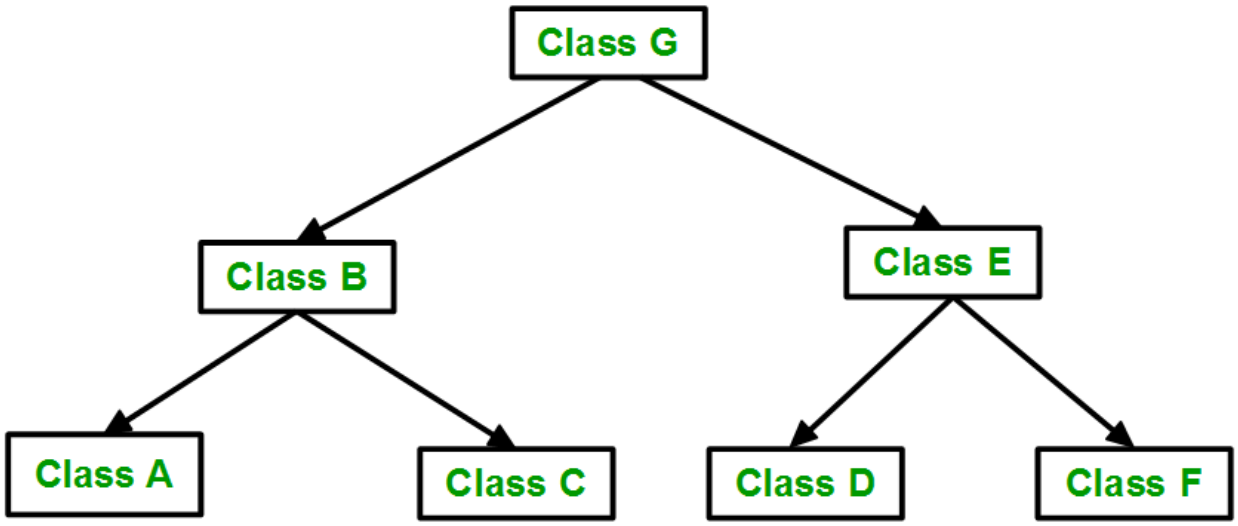
3. Multilevel Inheritance: In this type of inheritance, a derived class is created from another derived class.



Syntax:-

```
class C
{
... .. ...
};
class B:public C
{
... .. ...
};
class A: public B
{
... ... ...
};
```

4. Hierarchical Inheritance: In this type of inheritance, more than one subclass is inherited from a single base class. i.e. more than one derived class is created from a single base class.



Syntax:-

```
class A
{
   // body of the class A.
}
class B : public A
{
   // body of class B.
}
class C : public A
{
   // body of class C.
}
class D : public A
```

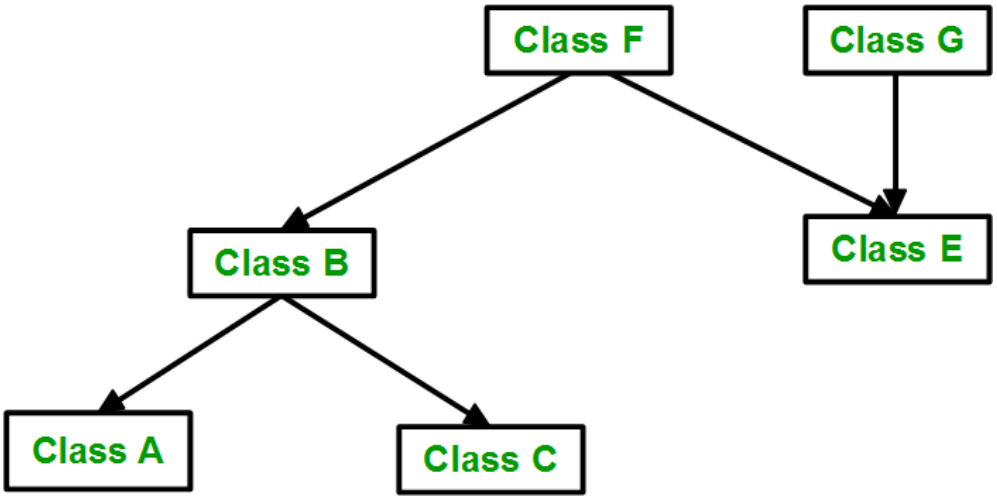
{

// body of class D.

}

5. Hybrid (Virtual) Inheritance: Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.

Below image shows the combination of hierarchical and multiple inheritances:



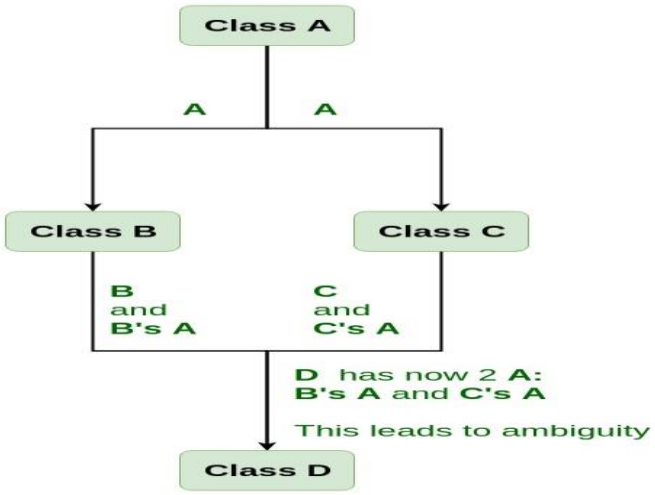
6. **A special case of hybrid inheritance: Multipath inheritance**: A derived class with two base classes and these two base classes have one common base class is called multipath inheritance. Ambiguity can arise in this type of inheritance.

Virtual base class in C++

Virtual base classes are used in virtual inheritance in a way of preventing multiple "instances" of a given class appearing in an inheritance hierarchy when using multiple inheritances.

Need for Virtual Base Classes: Consider the situation where we have one class **A** . This class **A** is inherited by two other classes **B** and **C**. Both these class are inherited into another in a new class **D** as shown in figure below.

As we can see from the figure that data members/function of class **A** are inherited twice to class **D**. One through class **B** and second through class **C**. When any data / function member of class **A** is accessed by an object of class **D**, ambiguity arises as to which data/function member would be called? One inherited through **B** or the other inherited through **C**. This confuses compiler and it displays error.



Abstract classes

An *abstract class* is a class that is designed to be specifically used as a base class. An abstract class contains at least one *pure virtual function*. You declare a pure virtual function by using a *pure specifier* (= 0) in the declaration of a virtual member function in the class declaration.