UNIT -3

Operator

C++ operator, like a symbol, just like we use operator for mathematical task in real world, similarly we use operator to do the same task through a program.

For example,

```
A + B
```

Where “+” is an operator and a and b can also be called operands or variables.

In the programming world, these operators are divided into different classes, in C++ these operators are divided as follows.

Arithmetic operator -

These are basic operators or default operators. These operators use two variables to perform a task. Generally these are present in all programming languages.

These are described in the table below-

Operator	Name	Example
A=	addition	a+b
-	subtraction	a-b
*	multiplication	a*b

-

/	division	a/b
%	modulus	a%b

like-

```
int a= 3,b=4;

cout<<a+b; // print addition 7

cout<<a-b; // print addition -1

cout<<a*b; // print addition 12
```

Relational Operator

In C++, an operator of a third type is created by combining two operators of different or similar types. Lajsere is called the relational operator. In C++ these operators are used to pass two variables in a single statement compare lucky to do

Operator	Name	Example
<	Less than	a<b
<=	Less than equal to	a<=b
>	Greater than	a>b
>=	Greater than and equal to	a>=b

==	Equal to	a==b
!=	Not equal to	a!=b

relational operators will be used with control statements. like,

For example, a = 3, b = 4,

less than operator

```
if(a<b) // if 3 is greater than 4 than body of if will be execute.
   {
   // body of if
   }
```

“Not Equal To” Operator

```
if(a!= b) if a and b value is not equal than body of if will be execute.
 {
   // body of if
 }
```

“Equal To” operator

```
if(a== b) if a and b value is equal than body of if will be
execute.
{
  // body of if
```

```
}
```

Here is its program-

```
#include <conio.h>

#include<iostream.h>

/

  void main()

    {

      clrscr();

      int a = 3, b = 4              c =4;



      if(a>b) {

      cout<<b<<" is greater than "<<a;

  }



      if(a!=b) {

      cout<<a<<" is Not Equal to "<<b;

  }



      if(b==c) {

      cout<<b<<" is Equal to "<<c;
```

```
  }


    getch();

}
```

OUTPUT

```
4 is greater than 3

4 is Not Equal to 3

4 is Equal to 4
```

This type of operator is used for two or more than two conditions.compare to do

Operator	Name	Example
&&	AND	a > b && a != b
||	OR	a > b && a != ba >b || a != b
.	NOT	!A

Like in some program,

If the value of variable “a” is less than variable “b” and variable “a” is greater than 0 (both condition should be true) then only body of loop will execute.

int a = 5, b = 10 c = 20;

```
if(a < b && a > 0) {
```

```
//body of if

}
```

If the value of variable “a” is greater than variable “b” or the value of variable “b” 0 is equal to variable “b” (any one of the conditon should be true) then body of loop will execute.

```
if(a < b || a != b) {

//body of if

}
```

NOT operator can be used with a single variable,

“NOT-operator” is used with a single variable,

int c = 0;

```
if(!c) // if c value 0, than execute body of if

{

//body of if

}
```

, The operator behaves like an “ON” and “OFF” button where ON is 0 and the other value represents OFF. Therefore it is used to check the execution of the function. melksi statement

```
#include <conio.h>

#include<iostream.h>
```

-


```
void main()

 {

   clrscr();

   int a = 5, b = 10 ;



   if(a < b && a > 0) {

     cout<<a<<" is smaller than "<<b<<" but greater than 0;

 }



   if(a < b || a != b) {

     cout<<a<<" and "<<b<<"are not Equal";

 }



 getch();

}
```

OUTPUT

```
5 is smaller than 10 but greater than 0

5 and 10 are Not Equal
```

Assignment operator -

Assignment operator is used to initialize / assign a variable / operand in the program, like,

```
int a = 5;
float b = 0.5;
```

Here are some assignment operators in C++-

Operator	Name	description
=	assignment	a = 5 assign values to operands
+=	addition and assign	a += b // a = a + b add Left Side variable to the Right Side variable then assign the final result to the Left side variable.
-=		a -= b // a = a – b same as above with different operation
/=		a /= b // a = a / b same as above with different operation
%=		a %= b // a = a % b same as above with different

		operation

Here are the examples given below-

```
#include <conio.h>

#include<iostream.h>



 void main()

  {

      clrscr();

      int count,sum=0,mul=1; //variable initialization using assignment  operator



  for(count = 1; count<=5;count++)

    {

       sum += count; // store result left side

       mul *= count; //

    }

  cout<<"Adition : "<<sum; // print variable sum value

  cout<<"\nMultiply: "<<mul<<endl; // print variable mul value

  getch();

}
```

OUTPUT

Addition: 15

Multiply: 120

C++ special operator

These operators are pre-defined in the library of C++. These types of operators are used for advanced tasks in C++.

Operator	Name	Description
#	Pound sign	program ÿÿÿheader file include in doing,
?:	Conditional or ternary Operator	In conditional statement,
++	increment	increase value one by one
--	decrement	decrease value one by one
&	reference operator	reference variable ÿÿÿ,
*	de-reference operator	To store the address of a variables

sizeof()	sizeof operator	To find the size of a data-type,
,	comma operator	multiple variable, declaration ÿÿ initialization ÿÿÿ,
::	scope resolution operator	To change the scope of variable and in class and structure, for outline definition
.	dot operator	To access member in structure, union and class
->	arrow operator	To refer to a member of a class or structure from a pointer, access
New	new operator	dynamic memory allocation ÿÿÿ
Delete	delete operator	dynamic memory, to de-locate

Endl	new line feed operator	To insert a new line like “enter” in a keyboard switch.
		To change the scope of a variable and for the outline definition of their function members in class and structure.

Here are some examples of special operators –

```
#include <conio.h>

#include<iostream.h>

 void main()

  {

     clrscr();

     int a =4; // assignment operator

     cout<<"Address of count variable : "<<&a<<"\n"; // reference operator

     cout<<"size of int data type(in byte): "<<sizeof(a); // sizeof  operator


     getch();
```

OUTPUT

```
Address of count variable : 0x8f87fff4

Size of int data type(int byte): 2
```

Operator Overloading – What is operator overloading?

Operator Overloading

In C++, we can specify more than one definition for an operator in the same scope. This is called operator overloading.

In other words, “Operator overloading is a compile-time polymorphism in which operators are overloaded to give a special meaning to user-defined data types.”

In C++, operator overrolling is used for user-defined data type In order to perform the operation. Its main advantage is that through this we can perform different operations in the same operand. **For example** – we use '+' operator to add integer. And '+' is also used to concatenate strings.

Most of the operators in C++ can be overloaded but there are some operators which cannot be overloaded. Operators which cannot be overloaded are the following:-

- scope operator – ::
- sizeof • member selector -. • member pointer selector – • ternary operator -?:

-

Operator overloading ÿÿ syntax

To overload operator we use a special **operator** function:-

```
class className {
... .. ...
public
returnType operator symbol (arguments) {
... .. ...
}
... .. ...
,
```

here,

returnType is the return type of the function.

- **operator** is a keyword.
- **symbol** is an operator which we want to overload.
- **arguments** to the function is the arguments to be passed.

Rules of Operator Overloading – There are some important rules regarding operator overloading which we have to keep in mind.

- Only built-in operators can be overloaded. If some operators are not present in C++ then we cannot overload them.
- The precedence and associativity of operators cannot be changed.
- We cannot use friend function to overload any particular operators. But, we can use member function to overload these operators.

• It is necessary to define assignment “=”, subscript “[]”, function call “()” and arrow operator “->” as member function. • The overloaded operator must contain at least one operand of user-defined data type. • Overloaded operator cannot hold default parameters. • Some operators like – assignment “=”, address “&”, and comma “,” are already overloaded. • The arity (qualification) of operators cannot be changed. That is, unary which is

will remain unary. Binary will remain binary.

Operator overloading

Below you have been given an example of this: -

```
#include <iostream>
using namespace std;
class Demo { private:
int y;

public:
Demo() : y(19) {}
void operator ++() { y
= y + 2; } void Print()
{ cout << "The Count
is: "  << y; } }; int
main() { Demo dd; ++dd;
dd.Print(); return 0; }
```

Its output:– The Count is: 21

Implementing Operator Overloading –

In C++, operator overloading can be performed by implementing the following functions:-

1. Member function
2. Non-member function 3. Friend function

The function of operator overloading can be a member function if the left operand is an object of that class. But if the left operand is different then the function of operator overloading must be a non-member function.

The function of operator overrolling class is made a friend function when there is a need to access the private and protected members of the class.

Function Overloading

Introduction

C++ language has the facility of specifying the same name for more than one function according to a condition. By specifying the same name of the function, the number of parameters is kept different in each function. This feature of C++ language is called Function Overloading. Function overloading is used when multiple functions with the same name take different types of parameters.

Definition

Function overloading is a logical method of calling multiple functions with different arguments and data types.

Declaration

The following two functions are different in C++-

Float area (float radius);

Float area (float len, float wid);

Here both the function have same name but number of argument is different. At the time of taking the function call, the compiler of C++ language checks the number of arguments.

Program

```
#include<iostream.h>
#include <conio.h>
floaty area (float radius);
float area (float len, float wid);
void main()
{
  char ch;
float radius, len,wid;
clrscr();
cout<<"\n enter C for circle and R for rectangle";
cin>>ch;
if ((ch=="C" :: (ch="C"))
{
cout<<"Enter radius"
cin>>radius;
cout<<"The area of the circle is : <<area (radius);
}
  else
if (ch=="R") :: (ch=="R"))
{
  cout<<"enter length :" cin>>len;
```

```
cout<<"enter width :" cin>>wid;
cout<<"The area of the rectangle :"<<area(len,wid);
}}                                                    // End of main function
float area pi=3.14159
return(pi*radius*radius);  float area(float
len;float wid)
{ return(len*wid);
}
```

Advantage 1.

Helps to understand, debug and use the function easily. 2. Easy Management of Code. 3. Prevents the use of different function names for the same operation.

Function Overloading in C++ In C++, Function overloading is an important feature in which two or more functions have the same name but their parameters are different.

In other words, “In C++, many functions can have the same name but have different parameters. This is called function overrolling.

compile-time polymorphism to function overrolling It is also said

The main advantage of Function overloading is that it increases the readability of the program as we do not need different names for the same function.

For example –

```
// same name different arguments
int demo() { } int demo(int a) { }
float demo(double a) { } int
demo(int a, double b) { }
```

In the above example all the 4 functions are overloaded. Their names are same but their arguments (parameters) are different.

How to do Function Overloading?

Functions are overloaded in two ways.

1. By changing the number of Arguments.

By changing the type of 2. Arguments.

By changing the number of Arguments – In this type of function overrolling, the number of arguments of the function is different.

Example of this –

```
#include <iostream>
using namespace
std; int add (int a, int
b) { cout << a + b <<
endl; return 0; } int add
(int a, int b, int c)
{ cout << a + b + c <<
endl; return 0; } int main
() { add (10, 30); add (50,
20, 30); } ÿÿÿÿ ÿÿÿÿÿÿ–
40
```

100

In the above example, we have overloaded the add() function by changing the number of arguments. First we have defined add() with two parameters and then we have defined add() with three parameters.

By changing the type of Argument – In this method, the data types of the parameters of the function are different.

Example of this –

```
#include <iostream>
using namespace std;
int add(int x, int y) // first definition
{ cout<< x+y << endl; return 0; } float
add(float a, float b) { cout << a+b <<
endl; return 0; }



double add(double x, double y)
{ cout << x+y << endl; return 0;



} int main()
{ add(30,
10);
add(21.45f, 38.4f);
add(41.24, 21.234); }
```

Its output is - 40
59.85 62.474

In the above example, we have defined the add() function thrice. In first we have used int, in second float and in third double parameters.

Method Overloading in Inheritance

If you want to achieve method overloading in inheritance, then it should not be Will get there, because of every inheriting, the **scope** of the method changes . So every time overloaded method of Base / Parent class **will** be called if

You also define the method in its child / derived class.

For Example :

```
#include <iostream>

using namespace std;


// define class.

class A{

   // define a public method to print string.

   public :

   void print(string s){

      cout << "String :          "   << s << endl;

   }

};


// define another class B that inherits class A.

class B : public A{

   // define same name method but different definition.

   public :

   void print(double d){

      cout << "Double Number :          "   << d;

   }

};

int main() {

            B bObj;

            bObj.print("learnhindituts");
```

-


```
        bObj.print(23.23);

        return 0;

}
```