Reference variable

When a variable is declared as a reference, it becomes an alternative name for an existing variable. A variable can be declared as a reference by putting ‘&’ in the declaration.

```
data_type & ref = variable;
```

Float total = 100;

Float & sum=total ;

cout<< total ; cout<<sum;

Output will be 100

A refences variable must be initialized at the time of declaration.

Example:

```
int x;
Int *p = &x;
Int & m= *p;   (m to refer to x which is pointed by pointer p)
```

```
Void f( int & x){
x=x+10;
}
Int main(){
Int m = 10;
f(m);
….
…
}
```

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

int main() {
    int x = 10;
    int& ref = x;                       // Reference variable 'ref' referring to 'x'

    cout << "x = " << x << endl;        // Output: x = 10
    cout << "ref = " << ref << endl;    // Output: ref = 10

                                        // Modifying 'x' indirectly through reference variable 'ref'
    ref = 20;
    cout << "x = " << x << endl;         // Output: x = 20
    cout << "ref = " << ref << endl;    // Output: ref = 20

    return 0;
}
```

Operators in C++

1. SCOPE RESOLUTION

- The scope resolution operator is used to reference the global variable or member function that is out of scope. Therefore, we use the scope resolution operator to access the hidden variable or function of a program. The operator is represented as the double colon (::) symbol.

- **Uses of the scope resolution Operator**

1.It is used to access the hidden variables or member functions of a program.
2.It defines the member function outside of the class using the scope resolution.
3.It is used to access the static variable and static function of a class.
4.The scope resolution operator is used to override function in the Inheritance.

Static Variable

- When a variable is declared as static, space for **it gets allocated for the lifetime of the program**. Even if the function is called multiple times, space for the static variable is allocated only once and the value of the variable in the previous call gets carried through the next function call.

- **SCOPE RESOLUTION syntax - : : variable-name ;**

```
#include <iostream>
using namespace std;
// declare global variable
int num = 50;
int main ()
{
// declare local variable
int num = 100;
  // print the value of the variables
cout << " The value of the local variable num: " << num;
  // use scope resolution operator (::) to access the global variable
cout << "\n The value of the global variable num: " << ::num;
return 0;
}
```

Output:

The value of the local variable num: 100
The value of the global variable num: 50

What is Memory Management

- Memory management is a process of managing computer memory, assigning the memory space to the programs to improve the overall system performance.
- In C language, we use the **malloc()** or **calloc()** functions to allocate the memory dynamically at run time, and free() function is used to deallocate the dynamically allocated memory. C++ also supports these functions, but C++ also defines unary operators such as **new** and **delete** to perform the same tasks, i.e., allocating and freeing the memory.

New operator

- A **new** operator is used to create the object while a **delete** operator is used to delete the object. When the object is created by using the new operator, then the object will exist until we explicitly use the delete operator to delete the object. Therefore, we can say that the lifetime of the object is not related to the block structure of the program.**(It can only be used with pointer variable)**
- **SYNTAX - pointer_variable = new data-type**

Eg-

```
int *p;
p = new int;
```

Example

```
int *p = new int;
float *q =   new float
```

Delete operator

- When memory is no longer required, then it needs to be deallocated so that the memory can be used for another purpose. This can be achieved by using the delete operator, as shown below:
- Syntax -**delete** pointer_variable;

```
#include<iostream>
using namespace std;
int main()
{
int *arr; int size;
cout<< "enter the size of the integer array:" ;
cin>> size ;
cout<< "the size of integer array is: " <<size ;
arr = new int[size];
cout<<"dynamic allocation is done" ;
return 0;
}
```

Note: If sufficient memory is not present to allocate, new returns a null pointer

STRUCTURE LIMITATIONS

- **Data hiding** - Structure members can be directly accessed by the structure variables by any function anywhere in their scope. Structure members are public members
- In C ++ structure can have both variable and functions as data members
- The only difference between structure and classes in C++ is that members od class are by default private and structure are public.

CLASS

A class definition starts with the keyword **class** followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations. For example, we defined the Box data type using the keyword **class** as follows

```
class Box {
   public:
      double length;   // Length of a box
      double breadth;  // Breadth of a box
      double height;   // Height of a box
};
```

The keyword **public** determines the access attributes of the members of the class that follows it. A public member can be accessed from outside the class anywhere within the scope of the class object. You can also specify the members of a class as **private** or **protected**

OBJECT

- A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types.

Box Box1; // Declare Box1 of type Box

Box Box2; // Declare Box2 of type Box

Both of the objects Box1 and Box2 will have their own copy of data members.

Accessing the Data Members:

The public data members of objects of a class can be accessed using the direct member access operator (.)

Example of class and object

```
#include <iostream>
using namespace std;
class Box {
   public:
      double length;   // Length of a box
      double breadth;  // Breadth of a box
      double height;   // Height of a box
};
int main() {
   Box Box1;        // Declare Box1 of type Box
   Box Box2;        // Declare Box2 of type Box
   double volume = 0.0;     // Store the volume of a box here
 // box 1 specification
   Box1.height = 5.0;
   Box1.length = 6.0;
   Box1.breadth = 7.0;
// box 2 specification
  Box2.height = 10.0;
   Box2.length = 12.0;
   Box2.breadth = 13.0;
    // volume of box 1
   volume = Box1.height * Box1.length * Box1.breadth;
   cout << "Volume of Box1 : " << volume <<endl;

   // volume of box 2
   volume = Box2.height * Box2.length * Box2.breadth;
   cout << "Volume of Box2 : " << volume <<endl;
   return 0;
}
```

C++ Class Example: Initialize and Display data through method

```
#include <iostream>
using namespace std;
class Student {
  public:
     int id;//data member (also instance variable)
     string name;//data member(also instance variable)
     void insert(int i, string n)
      {
         id = i;
         name = n;
      }
     void display()
      {
         cout<<id<<"  "<<name<<endl;
      }
};
int main(void) {
    Student s1; //creating an object of Student
    Student s2; //creating an object of Student
    s1.insert(201, "Sonoo");
    s2.insert(202, "Nakul");
    s1.display();
    s2.display();
    return 0;
}
```

Example:

```
#include <iostream>
using namespace std;
class Employee {
  public:
     int id;//data member (also instance variable)
     string name;//data member(also instance variable)
     float salary;
     void insert(int i, string n, float s)
      {
         id = i;
         name = n;
         salary = s;
      }
     void display()
      {
         cout<<id<<"  "<<name<<"  "<<salary<<endl;
      }
};
int main(void) {
    Employee e1; //creating an object of
 Employee
    Employee e2; //creating an object of
 Employee
    e1.insert(201, "Sonoo",990000);
    e2.insert(202, "Nakul", 29000);
    e1.display();
    e2.display();
    return 0;
}
```

ACCESS MODIFIER:

- Access Modifiers or Access Specifiers in a class are used to assign the accessibility to the class members, i.e., they set some restrictions on the class members so that they can't be directly accessed by the outside functions.
  There are 3 types of access modifiers available in C++:

1. **Public**
2. **Private**
3. **Protected**

Note: If we do not specify any access modifiers for the members inside the class, then by default the access modifier for the members will be Private.

- **1. Public**: All the class members declared under the public specifier will be available to everyone.

  The data members and member functions declared as public can be accessed by other classes and functions too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

Example of public access modifier

```
#include<iostream>
using namespace std;
// class definition
class Circle
{
        public:
                double radius;

                double compute_area()
                {
                        return 3.14*radius*radius;
                }

};
```

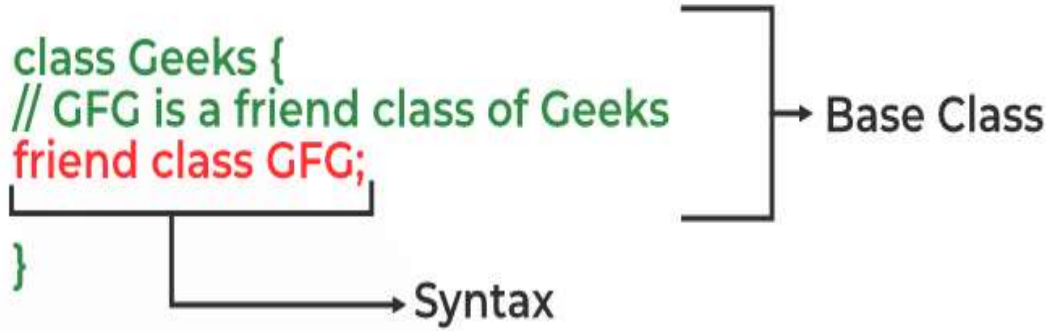
```
int main()
{
        Circle obj;
        // accessing public data member
outside class
        obj.radius = 5.5;
        cout << "Radius is: " << obj.radius <<
"\n";
        cout << "Area is: " <<
obj.compute_area();
        return 0;
}
```

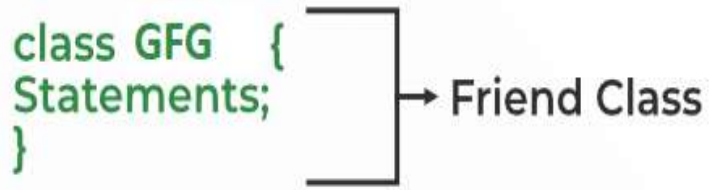
2. Private: The class members declared as *private* can be accessed only by the member functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of the class.

A **friend class** can access private and protected members of other classes in which it is declared as a friend. It is sometimes useful to allow a particular class to access private and protected members of other classes.

We can declare a friend class in C++ by using the **friend** keyword.

Syntax- friend class class_name;





Example of private access modifier:

```
#include<iostream>
using namespace std;
class Circle
{
        // private data member
        private:
                double radius;

        // public member function
        public:
                double compute_area()
                {  // member function can access private
                        // data member radius
                        return 3.14*radius*radius;
                }

};
```

output:

```
In function 'int main()':
11:16: error: 'double Circle::radius' is private
        double radius;
               ^
31:9: error: within this context
    obj.radius = 1.5;
        ^
```

```
int main()
{
        Circle obj;
        obj.radius = 1.5;

        cout << "Area is:" <<
obj.compute_area();
        return 0;
}
```

we can access the private data members of a class indirectly using the public member functions of the class.

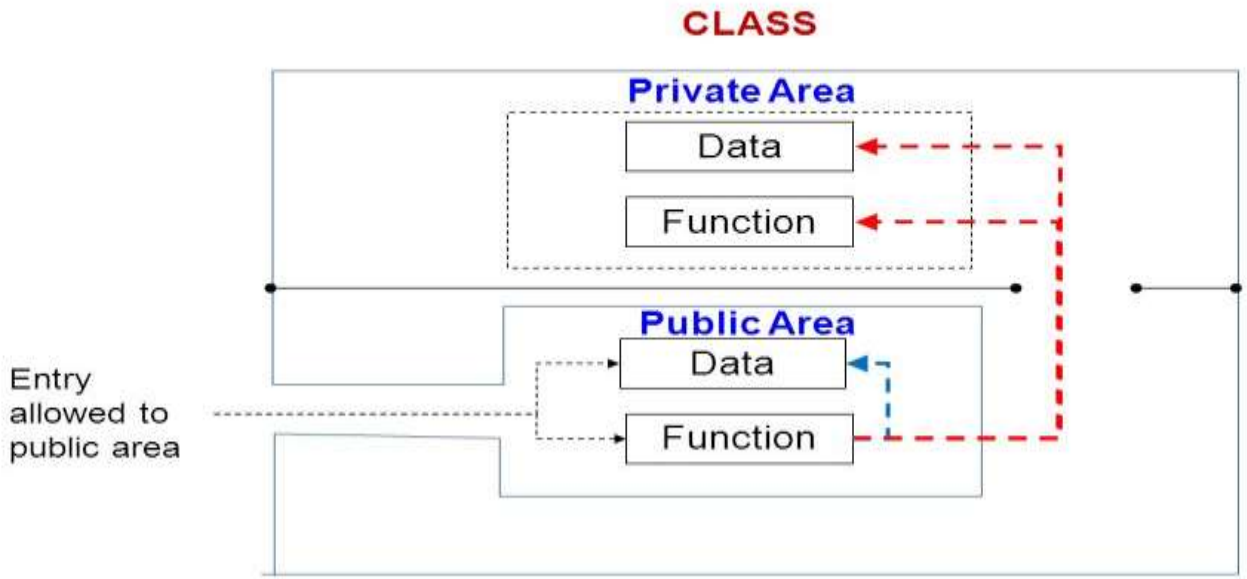
```
#include<iostream>
using namespace std;
class Circle
{
        private:
        double radius;
        // public member function
        public:
                void compute_area(double r)
                {
                        radius = r;
                        double area = 3.14*radius*radius;
                        cout << "Radius is: " << radius << endl;
                        cout << "Area is: " << area;
                }
};
```

OUTPUT:
Radius is: 1.5
Area is: 7.065

```
int main()
{
        Circle obj;
        obj.compute_area(1.5);
        return 0;
}
```

3. Protected: The protected access modifier is similar to the private access modifier in the sense that it can’t be accessed outside of its class unless with the help of a friend class. The difference is that the class members declared as Protected can be accessed by any subclass (derived class) of that class as well.

Data hiding in classes



Example of class

```
Class item
 {
        int number;
        float cost;
   public:
        void getdata(int a, float b);
        void putdata(void);
}; item z,y,z ;
```

Accessing class members:

private data of class can be accessed through the member function of that class. Main() cannot contain statements that can access the number and cost variable . The following is format is used to call the member function:

Object-name. function-name(actual arguments);

Defining member functions:

- Class is a blueprint of an object, which has data members and member functions also known as methods. A method is a procedure or function in the oops concept. A method is a function that belongs to a class.
- There are two ways to define a procedure or function that belongs to a class:
- Inside Class Definition
- Outside Class Definition

1. Inside Class Definition

- The member function is defined inside the class definition it can be defined directly. Similar to accessing a data member in the class we can also access the public member functions through the class object using the dot operator (.).

Syntax:

```
class class_name{
public:
return_type Method_name()  // method inside class definition
{
  // body of member function
}
}
```

EXAMPLE OF DEFINING METHOD INSIDE CLASS

```
#include <iostream>
using namespace std;
class rectangle {
private:
        int length;
        int breadth;
public:
        // Constructor
        rectangle(int length, int breadth)
        {
                this->length = length; // this pointer
                 this->breadth = breadth;
        }
        // area() function inside class
        int area() {
        return (length * breadth); }
        // perimeter() function inside class
        int perimeter() { return 2 * (length + breadth); }
};
```

```
int main()
{
        rectangle r(2, 3);
        cout << "perimeter: " << r.perimeter() << endl;
        cout << "area: " << r.area() << endl;
        return 0;
}
```

• 2. Outside Class Definition

The member function is defined outside the class definition it can be defined using the scope resolution operator. Similar to accessing a data member in the class we can also access the public member functions through the class object using the dot operator (.)

Syntax:

```
class Class_name{
public:
return_type Method_name(); // method outside class definition
};
// Outside the Class using scope resolution operator
return_type Class_name :: Method_name() {
// body of member function
}
```

EXAMPLE OF DEFINING METHOD OUTSIDE CLASS

```
#include <iostream>
using namespace std;
class rectangle {
private:
        int length;
        int breadth;
public:
        rectangle(int length, int breadth)
        {
        this->length = length; // this pointer
        this->breadth = breadth;
        }
        int area();
        int perimeter();
};
int rectangle::area() {
 return (length * breadth); }
int rectangle::perimeter()
{
        return 2 * (length + breadth);
}

int main()
{
        // Creating object
        rectangle r(2, 3);
        cout << "perimeter: " << r.perimeter() << endl;
        cout << "area: " << r.area() << endl;
        return 0;
}
```

Use of scope resolution operator for defining function outside the class

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

class A {
public:
    // Only declaration
    void fun();
};

// Definition outside class using ::
void A::fun()
{
 cout << "fun() called"; }

int main()
{
    A a;
    a.fun();
    return 0;
}
```

Write C++ program to print number and cost by creating class and object to show class implementation take number and cost as private variables

```
#include<iostream>
using namespace std;
class item {
   int number;
   float cost;
public:
   void getdata(int a, float b);
   void putdata(void) {
      cout << "number: " << number << "\n";
      cout << "cost: " << cost << "\n";
   }
};
void item::getdata(int a, float b) {
   number = a;
   cost = b;
```

```
int main() {
    item x;
    cout << "\n object x" << "\n";
    x.getdata(100, 299.95);
    x.putdata();
 item y;
    cout << "\n object y" << "\n";
    y.getdata(200, 599.98);
    y.putdata();

    return 0;
}
```

- **Private member function:** private member function can only be called by another function that is member of that same class. Even object cannot invoke a private function using dot operator.

```
Class sample
{
Int m ;
void read(void);
public:
Void update(void)
Void write (void)
};
```

If s1 is an object of sample then,

S1.read(); **//cannot access private members**

However, function read() can be called by the function update to update the value of m

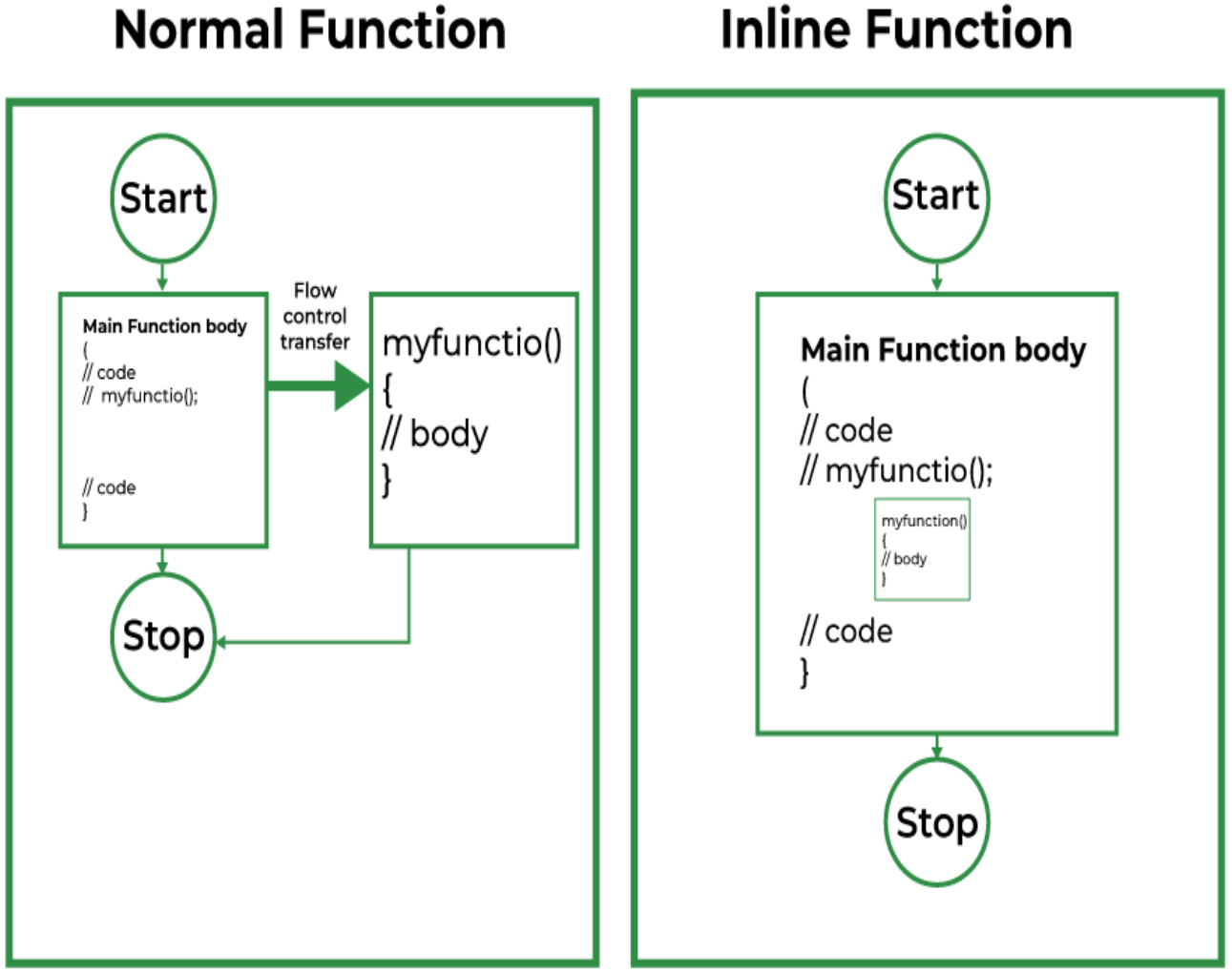
```
Void sample:: update(void)
{   read() ;
}
```

Inline functions:

- An inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of the inline function call. This substitution is performed by the C++ compiler at compile time. An inline function may increase efficiency if it is small.

SYNTAX- inline return_type function_name(parameters) {

```
    // Function body
}
```



Example:

```
#include <iostream>

// Inline function to calculate the square of a number
inline int square(int x) {
    return x * x;
}

int main() {
    int num = 5;
    std::cout << "Square of " << num << " is: " << square(num) << std::endl;
    return 0;
}
```

Constructors : introduction


```
putdata();
getdata();
setvalue();
a.input();
x.getdata(100,299,.95)  //
```

Where the values are assigned to the private variables of object x.
In C++ we use user defined data type “class” , this means that we should be able to initialize the object(variable) at the time of declaration much similar to built in data types: for example: int x=10; float a= 5.57;

In C++, constructor is a special member function which is invoked automatically at the time of object creation. It is used to initialize the data members of new object generally. The constructor in C++ has the same name as class or structure.
Also known as automatic initialization of objects. Destructor which deletes the object when they are no longer required

- The constructor is invoked when the object of that class is created. It is called as "constructor" because it constructs values for the data members of the class.

A constructor is declared and defined as follows:

```
class integer {
int m ,n ;
public:
integer(void);                    //constructor declared
};
integer::integer(void)       // constructor defined
{
m=0; n=0;
}
```

- Above constructor guarantees that the object created by the class will automatically be initialized. for eg- integer int1; will automatically get values m and n = 0.
- There is no need to invoke constructor as we do in normal functions
- a constructor which does not accepts parameters is called as the default constructor.
- Constructor can only be declared in the public section.
- Constructor are invoked automatically when the objects are created.
- They cannot return values they do not have return types.
- When a constructor is declared for a class , initialization for the class objects becomes mandatory.

Default constructor

OUTPUT ????????????

```
#include <iostream>
using namespace std;
class Employee
 {
   public:
        Employee()
        {
            cout<<"Default Constructor Invoked"<<endl;
        }
};
int main(void)
{
    Employee e1;                    //creating an object of Employee
    Employee e2;
    return 0;
}
```

Parametrized constructors:

- A constructor which has parameters is called parameterized constructor. It is used to provide different values to distinct objects.

```
#include <iostream>
using namespace std;
class Employee {
  public:
    int id;//data member (also instance variable)
    string name;//data member(also instance variable)
    float salary;
    Employee(int i, string n, float s)
     {
        id = i;
        name = n;
        salary = s;
     }
    };
void display()
      {
        cout<<id<<"  "<<name<<"  "<<salary<<endl;

      }

int main(void) {
   Employee e1 =Employee(101, "Sonoo", 890000
);
    Employee e2=Employee(102, "Nakul", 59000);

   e1.display();
   e2.display();
   return 0;
}
```

Example of parametrized constructor

```
#include<iostream>
using namespace std;
class boot{
   int m , n;
   public:
   boot(int x,int y)
   {
      m=x;
      n=y;
   }
void putdata(){
   cout<<"m="<<m<<"n="<<n;
}
};
int main(){
   boot aa(5,10);
   aa.putdata();
   return 0;
   }
```

```
#include<iostream>
using namespace std;
class boot{
   int m , n;
   public:
   boot(int x,int y)
   {
      m=x;
      n=y;
   }
void putdata(){
   cout<<"m="<<m<<"\n"<<"n="<<n;
}
};
int main()
{
   int f,g;
   cout<<"enter two numbers:";
   cin>>f;
   cin>>g;
   boot aa(f,g);
   aa.putdata();
   return 0;
   }
```

Defining Parameterized Constructor Outside the Class.

```
#include <iostream>
#include <string.h>
using namespace std;
class student {
        int rno;
        char name[50];
        double fee;
public:
        student(int, char[], double);
        void display();
};
// parameterized constructor outside class
student::student(int no, char n[], double f)
{
        rno = no;
        strcpy(name, n);
        fee = f;
}
void student::display()
{
cout << endl << rno << "\t" << name << "\t" << fee;
}

int main()
{
        student s(1001, "Ram", 10000);
        s.display();
        return 0;
}
```

QUIZ

Q1 . **Which of the followings is/are automatically added to every class, if we do not write our own.**

a) Copy Constructor
b) Assignment Operator
c) A constructor without any parameter
d) All of the above

Q2. What will be the output??

```
#include <iostream>
using namespace std;
 class Point {
   Point() { cout << "Constructor called"; }
};
 int main()
{
   Point t1;
   return 0;
```

A) Runtime Error
(B) None of these
(C) Constructor called
(D) Compiler Error

Q3. What will be the output?

```
#include<iostream>
using namespace std;
class Point {
public:
    Point() { cout << "Constructor called"; }
};
  int main()
{
   Point t1, *t2;
   return 0;
}
```

(A) Compiler Error

(B) Constructor called
Constructor called

(C) Constructor called

Q4. What will be the output?

```
#include<iostream>
using namespace std;
  class Point {
public:
    Point() { cout << "Normal Constructor called\n"; }
    Point(const Point &t) { cout << "Copy constructor called\n"; }
};

int main()
{
   Point *t1, *t2;
   t1 = new Point();
   t2 = new Point(*t1);
   Point t3 = *t1;
   Point t4;
   t4 = t3;
   return 0;
}
```

(A) Normal Constructor called
Normal Constructor called

Normal Constructor called

Copy Constructor called

Copy Constructor called

Normal Constructor called

Copy Constructor called

(B) Normal Constructor called
Copy Constructor called

Copy Constructor called

Normal Constructor called

Copy Constructor called

(C) Normal Constructor called
Copy Constructor called

- Point *t1, *t2; // No constructor call
- t1 = new Point(10, 15); // Normal constructor call
- t2 = new Point(*t1); // Copy constructor call
- Point t3 = *t1; // Copy Constructor call
- Point t4; // Normal Constructor call
- t4 = t3; // Assignment operator call

Initialization with an Existing Object:

If an object is being initialized with another object of the same class, the copy constructor is called.

For example:

Point t3 = *t1;

- **Dynamic Memory Allocation with an Existing Object:**

If an object is being dynamically allocated with the values of another object, the copy constructor is called.

For example: t2 = new Point(*t1);

Assignment of Objects:

When an object is being assigned the values of another object, if the object being assigned to already exists, the copy constructor is called.

- For example: t4 = t3;

Q. What will be the output?????

```
#include<iostream>
using namespace std;
class Test
{
static int i;
int j;
};
int Test::i;
int main()
{ cout << sizeof(Test);
        return 0;
}
```

Solution:

- The size of an object of a class in C++ is determined by the sum of the sizes of its non-static data members. Static data members do not contribute to the size of an object because they are shared among all objects of the class.

Q. What will be the output?????

```
#include<iostream>
using namespace std;
int x = 10;
void fun()
{
int x = 2;
        {
                int x = 1;
                cout << ::x << endl;
        }
}
int main()
{
        fun();
        return 0;
}
```

Q. What will be the output????

```
#include<iostream>
using namespace std;
int &fun() {
static int a = 10;
return a;
}

int main() {
int &y = fun();
y = y +30;
cout<<fun();
return 0;
}
```

- The static variable “a” is modified inside the main() function using the reference y. Since y is a reference to the static variable “a”, any modification made to y will also affect the value of “a”.

- The static variable “a” will retain its modified value (40) throughout the program execution because it is a static variable. Static variables persist their values across function calls and throughout the program's lifetime. Therefore, the value of “a” will not be constant throughout the program; it will be modified by the assignment y = y + 30 in the main() function and will retain this modified value (40) thereafter.

Your Task:

- In the function printInNewLine(), output each word of Geeks for Geeks in a separate line.
- Your task is to complete the provided function isPrime() which should return a string "Yes" if n is prime and "No" if not.

Constructor overloading

```
#include<iostream>
using namespace std;
class boot{
    int m , n;
    public:
    boot(int x , int y)
    {
        m=x;
        n=y;
    }
    boot(int x){
        m=x;
        n=0;
    }
    void printnumber(){
        cout<<"yournumber
is:"<<"m="<<m<<"n="<<n;
}
};

int main(){
    boot hello(5,10);
hello.printnumber();
boot hello2(5);
hello2.printnumber();
    return 0;
    }
```

Constructor with default argument

```
#include<iostream>
using namespace std;
class boot{
    int m , n;
    public:
    boot(int x,int y=9)
    {
        m=x;
        n=y;

    }
   void printnumber(){
    cout<<"your number is:"<<"m="<<m<<"\n"<<"and"<<" "<<"n="<<n
<<“\n”;
}
};
```

```
your number is:m=5
and n=9your number is:m=7
and n=10
PS C:\Users\Lenovo\c++>
```

```
int main(){
    boot hello(5);
hello.printnumber();
boot hi(7,10);
hi.printnumber();
    return 0;
    }
```

Dynamic initialization of object in C++

- Dynamic initialization of object refers to initializing the objects at a run time i.e., the initial value of an object is provided during run time.
- **Dynamic Constructor:**
- The constructor used for allocating the memory at runtime is known as the **dynamic constructor**.
- The memory is allocated at runtime using a new operator and similarly, memory is deallocated at runtime using the delete operator.

Dynamic Allocation: Approach

- In the below example, new is used to dynamically initialize the variable in default constructor and memory is allocated on the heap.
- The objects of the class geek calls the function and it displays the value of dynamically allocated variable i.e ptr.

Memory in a C/C++/Java program can either be allocated on a stack or a heap.

Stack Allocation:

The allocation happens on contiguous blocks of memory. We call it a stack memory allocation because the allocation happens in the function call stack. The size of memory to be allocated is known to the compiler and whenever a function is called, its variables get memory allocated on the stack. And whenever the function call is over, the memory for the variables is de-allocated.

- Stack memory allocation is considered safer as compared to heap memory allocation because the data stored can only be accessed by the owner thread.
- Memory allocation and de-allocation are faster as compared to Heap-memory allocation.

```
int main()
{
  // All these variables get memory
  // allocated on stack
  int a;
  int b[10];
  int n = 20;
  int c[n];
}
```

- **Heap memory allocation** isn't as safe as Stack memory allocation because the data stored in this space is accessible or visible to all threads.
- The memory is allocated during the execution of instructions written by programmers

```
#include <iostream>
using namespace std;
class geeks {
        int* ptr;
public:
        geeks()
        {
                ptr = new int;
                *ptr = 10;
        }
        void display()
        {
                cout << *ptr << endl;
        }
};

int main()
{
        geeks obj1;

        // Function Call
        obj1.display();

        return 0;
}
```

Stack Variables:

- Local variables declared inside functions have their memory automatically allocated on the stack when the function is called, and deallocated when the function exits.
- Stack variables have a limited lifetime, tied to the scope of the block in which they are declared.
- Examples include function parameters, local variables inside functions, and variables declared within loops or conditional blocks.

Heap Variables:

Variables allocated dynamically using new operator have their memory allocated on the heap.

- Heap variables have a lifetime until explicitly deallocated using the delete operator.
- Heap variables are typically used for objects that need to persist beyond the scope of the function in which they are allocated.
- Examples include objects created using new, arrays allocated using new[], and objects created by library functions like malloc() (in C).

COPY CONSTRUCTOR

- A copy constructor is a member function that initializes an object using another object of the same class.
- A constructor which creates an object by initializing it with an object of the same class, which has been created previously is known as a **copy constructor**.
- Copy constructor is used to initialize the members of a newly created object by copying the members of an already existing object.


```
ClassName (const ClassName &old_obj);
```

```
#include <iostream>
#include <string.h>
using namespace std;
class student {
        int rno;
        char name[50];
        double fee;
public:
        student(int, char[], double);
        student(student& t) // copy constructor
        {       rno = t.rno;
                strcpy(name, t.name);
                fee = t.fee;
        }
        void display();
};

student::student(int no, char n[], double f)
{
        rno = no;
        strcpy(name, n);
        fee = f;
}

void student::display()
{
        cout << endl << rno << "\t" << name << "\t" << fee;
}
int main()
{
        student s(1001, "Manjeet", 10000);
        s.display();

        student hello(s); // copy constructor called
        hello.display();

        return 0;
}
```

Destructor

1. Destructor is an instance member function that is invoked automatically whenever an object is going to be destroyed. Meaning, a destructor is the last function that is going to be called before an object is destroyed.
2. Destructor has the same name as their class name preceded by a tilde (~) symbol.
3. The syntax for **defining the destructor within the class**:

```
~ <class-name>() {
   // some instructions
}
```

The syntax for defining the destructor outside the class:

```
<class-name> :: ~<class-name>() {
   // some instructions
}
```

Array of objects:

Syntax:

ClassName ObjectName[number of objects];

The Array of Objects stores objects. An array of a class type is also known as an array of objects.

Example:

Array of objects:

```
#include<iostream>
using namespace std;
class Employee
{
int id;
char name[30];
public:
void getdata();
void putdata();
};
void Employee::getdata()
{
cout << "Enter Id : ";
cin >> id;
cout << "Enter Name : ";
cin >> name;
}
void Employee::putdata()
{
cout << id << " ";
cout << name << " ";
cout << endl;
}
int main()
{
Employee emp[30];
int n, i;
cout << "Enter Number of Employees - ";
cin >> n;
// Accessing the function
for(i = 0; i < n; i++)
        emp[i].getdata();

cout << "Employee Data - " << endl;

// Accessing the function
for(i = 0; i < n; i++)
        emp[i].putdata();
}
```

Program to add two complex numbers using object as function arguments

```
#include<iostream>
using namespace std;
class complex{
   int a;
   int b;
     public:
      void setData(int v1, int v2){
         a = v1;
         b = v2;
      }
            void setDataBySum(complex o1, complex o2){
         a = o1.a + o2.a;
         b = o1.b + o2.b;
      }     void printNumber(){
         cout<<"Your complex number is "<<a<<" + "<<b<<"i"<<endl;
      }
};
int main(){
   complex c1, c2, c3;
   c1.setData(1, 2);
   c1.printNumber();

   c2.setData(3, 4);
   c2.printNumber();

   c3.setDataBySum(c1, c2);
   c3.printNumber();
   return 0;
}
```

Returning Object from functions:

```
#include <bits/stdc++.h>
using namespace std;
class Example {
public:
        int a;
        Example add(Example Ea, Example Eb)
        {
                Example Ec;
                Ec.a = Ea.a + Eb.a;
                return Ec;
        }
};
int main()
{       Example E1, E2, E3;
        E1.a = 50;
        E2.a = 100;
        E3.a = 0;

cout << "Initial Values \n";
        cout << "Value of object 1: " << E1.a
                << ", \object 2: " << E2.a
                << ", \object 3: " << E3.a
                << "\n";
        E3 = E3.add(E1, E2);

// Changed values after passing object as an argument

        cout << "New values \n";
        cout << "Value of object 1: " << E1.a
                << ", \nobject 2: " << E2.a
                << ", \nobject 3: " << E3.a
                << "\n";

        return 0;
}
```

OUTPUT:

Initial Values

Value of object 1: 50,

object 2: 100,

object 3: 0

New values

Value of object 1: 50,

object 2: 100,

object 3: 150

#include <bits/stdc++.h>

This line includes the standard C++ library header <bits/stdc++.h>, which includes most of the standard headers such as <iostream>, <vector>, <algorithm>, etc., in a single header. It's commonly used in competitive programming or when you want to include all standard headers in one go.

Operators in C

- All operators available in C are also available in C++. Some extra operators are listed below:

Operators in C

1. **setw() function:** set width for input and output stream:The setw() method of iomanip library in C++ is used to set the ios library field width based on the width specified as the parameter to this method. The setw() stands for set width and it works for both the input and the output streams.

Syntax:

std::setw(int n);

This method does not return anything. It only acts as a stream manipulator.

EXAMPLE: setw() to add padding to the integer output

```
#include <iomanip>
#include <ios>
#include <iostream>
using namespace std;
int main()
{
int num = 50;
cout << "Before setting the width: \n" << num << endl;
        // Using setw()
cout << "Setting the width"<< " using setw to 5: \n"<< setw(5);
        cout << num;
        return 0;
}
```

OUTPUT:

```
Before setting the width:
50
Setting the width using setw to 5:
  50
```

Case 1: Using setw() with cin to limit the number of characters to take from the input stream.

```
#include <iostream>
using namespace std;
int main()
{       string str;
        // setting string limit to 5 characters
        cin >> setw(5) >> str;
        cout << str;
        return 0;
}
```

Input:

GeeksforGeeks

Output:

Geeks

Case 2: Using setw() to set the character limit for string output.

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
        string str("GeeksforGeeks");
        // adding padding
        cout << "Increasing Width:\n"<< setw(20) << str << endl;
        // reducing width
        cout << "Decreasing Width:\n" << setw(5) << str;
        return 0;
}
```

Output:

Increasing Width:
        GeeksforGeeks
Decreasing Width:
GeeksforGeeks

Programming Questions:

- **Problem:** Given an array arr[] of n elements, write a function to search a given element x in arr[].

Input: arr[] = {10, 20, 80, 30, 60, 50,
110, 100, 130, 170}
x = 110;

Output: 6

Element x is present at index 6

Input: arr[] = {10, 20, 80, 30, 60, 50,
110, 100, 130, 170}
x = 175;

Output: -1

Element x is not present in arr[]

```
#include <iostream>
using namespace std;
int search(int arr[],
        int n, int x)
{
        int i;
        for (i = 0; i < n; i++)
                if (arr[i] == x)
                return i;
        return -1;
}
int main(void)
{       int arr[] = {2, 3, 4, 10, 40};
        int x = 10;
        int n = sizeof(arr) / sizeof(arr[0]);
                                                // Function call
                                                int result = search(arr, n, x);
                                                (result == -1) ?
                                                cout << "Element is not present in array" :
                                                cout << "Element is present at index " << result;
                                                return 0;
                                        }
```

Write a C++ program to implement recursive Binary Search

- [binay search code.docx]

Print calendar for a given year in C++

- Print pattern questions

- **String class functions**

Getline , push_back , pop_back

Output:

The initial string is :
The string after push_back operation is : s
The string after pop_back operation is :

```
#include <iostream>
#include <string> // for string class
using namespace std;
int main()
{
        string str;
        getline(cin, str);
        // Displaying string
        cout << "The initial string is : ";
        cout << str << endl;
        // Inserting a character
        str.push_back('s');
        // Displaying string
        cout << "The string after push_back operation
is : ";
        cout << str << endl;
        // Deleting a character
        str.pop_back();
        // Displaying string
        cout << "The string after pop_back operation
is : ";
        cout << str << endl;

        return 0;
}
```

getline()	This function is used to store a stream of characters as entered by the user in the object memory.
push_back()	This function is used to input a character at the end of the string.
pop_back()	Introduced from C++11(for strings), this function is used to delete the last character from the string

Function	**Definition**
capacity()	This function returns the capacity allocated to the string, which can be equal to or more than the size of the string. Additional space is allocated so that when the new characters are added to the string, the operations can be done efficiently.
resize()	This function changes the size of the string, the size can be increased or decreased.
length()	This function finds the length of the string.
shrink_to_fit()	This function decreases the capacity of the string and makes it equal to the minimum capacity of the string. This operation is useful to save additional memory if we are sure that no further addition of characters has to be made.

```
#include <iostream>
#include <string> // for string class
using namespace std;
int main()
{
        // Initializing string
        string str = "geeksforgeeks is for geeks";

// Displaying string
        cout << "The initial string is : "<< str << endl;

// Resizing string using resize()
        str.resize(13);

// Displaying string
cout << "The string after resize operation is : "<< str << endl;
```

```
// Displaying capacity of string
        cout << "The capacity of string is : " << str.capacity() << endl;

        // Displaying length of the string
        cout << "The length of the string is :" << str.length() << endl;

// Decreasing the capacity of string  // using shrink_to_fit()
        Str.shrink_to_fit();

        // Displaying string
        cout << "The new capacity after shrinking is : "<< str.capacity() <<
endl;
        return 0;
}
```

Output

```
The initial string is : geeksforgeeks is for geeks
The string after resize operation is : geeksforgeeks
The capacity of string is : 26
The length of the string is :13
The new capacity after shrinking is : 13
```

DYNAMIC CONSTRUCTOR

```
#include <iostream>
using namespace std;
class geeks {
        const char* p;
public:
        // default constructor
        geeks()
        {
        // allocating memory at run time
                p = new char[6];
                p = "geeks";
        }
void display() { cout << p << endl; }
};

int main()
{
        geeks obj;
        obj.display();
}
```

- In this we point data member of type char which is allocated memory dynamically by new operator and when we create dynamic memory within the constructor of class this is known as dynamic constructor.

Static data members

- Static data members are class members that are declared using static keywords. A static member has certain special characteristics which are as follows:
- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is initialized before any object of this class is created, even before the main starts.
- It is visible only within the class, but its lifetime is the entire program.
- syntax- : static data_type data_member_name;

```
#include <iostream>
using namespace std;
class Employee
{
   int id;
   static int count;
public:
   void setData(void)
   {
      cout << "Enter the id" << endl;
      cin >> id;
      count++;
   }
   void getData(void)
   {
      cout << "The id of this employee is " << id << "
and this is employee number " << count << endl;
   }
};
```

```
// Count is the static data member of class Employee
int Employee::count; // Default value is 0
int main()
{
        Employee e1,e2, e3;
        // e1.id = 1;
   // e2.count=1; // cannot do this as id and count
are private
        e1.setData();
        e1.getData
        e2.setData();
         e2.getData();
        e3.setData();
         e3.getData();
        return 0;
        }
```

```
// C++ Program to demonstrate
// Static member in a class
#include <iostream>
using namespace std;

class Student {
public:
        // static member
        static int total;

        // Constructor called
        Student() { total += 1; }
};

int Student::total = 0;

int main()
{
        // Student 1 declared
        Student s1;
        cout << "Number of students:" << s1.total
<< endl;

        // Student 2 declared
        Student s2;
        cout << "Number of students:" << s2.total
<< endl;

        // Student 3 declared
        Student s3;
        cout << "Number of students:" << s3.total
<< endl;
        return 0;
}
```

Static member function

```
#include <iostream>
using namespace std;
class Box
{           private:
            static int length;
            static int breadth;
            static int height;
            public:
            static void print()
            {
cout << "The value of the length is: " << length << endl;
cout << "The value of the breadth is: " << breadth << endl;
cout << "The value of the height is: " << height << endl;
            }
};
int Box :: length = 10;
            int Box :: breadth = 20;
            int Box :: height = 30;

// Driver Code

int main()
{

            Box b;

            cout << "Static member function is called through Object name: \n" << endl;
            b.print();

            cout << "\nStatic member function is called through Class name: \n" << endl;
            Box::print();

            return 0;
}
```

Output:

Static member function is called through Object name:

- The value of the length is: 10
- The value of the breadth is: 20
- The value of the height is: 30

Static member function is called through Class name:

- The value of the length is: 10
- The value of the breadth is: 20
- The value of the height is: 30

WAP to find prime numbers between 20-50

```
#include <iostream>
using namespace std;
bool isPrime(int num) {
    if (num <= 1) {
        return false;
    }
    for (int i = 2; i <= num / 2; ++i) {
        if (num % i == 0) {
            return false;
        }
    }
    return true;
}
int main() {
    cout << "Prime numbers between
50 and 100 are:" << endl;
    for (int i = 20; i <= 50; ++i) {
        if (isPrime(i)) {
            cout << i << " ";
        }
    }
    cout << endl;
    return 0;
}
```

Explicit type casting

```
#include <iostream>
using namespace std;
int main()
{
        double x = 1.2;
        // Explicit conversion from double to int
        int sum = (int)x + 1;
        cout << "Sum = " << sum;
        return 0;
}
```

Implicit type casting done by compiler itself

```
#include <iostream>
using namespace std;
int main()
{
        int x = 10; // integer x
        char y = 'a'; // character c
// y implicitly converted to int. ASCII
        // value of 'a' is 97
        x = x + y;
// x is implicitly converted to float
        float z = x + 1.0;
cout << "x = " << x << endl
                << "y = " << y << endl
                << "z = " << z << endl;

        return 0;
}
```

polymorphism

- The word “polymorphism” means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

Types of Polymorphism

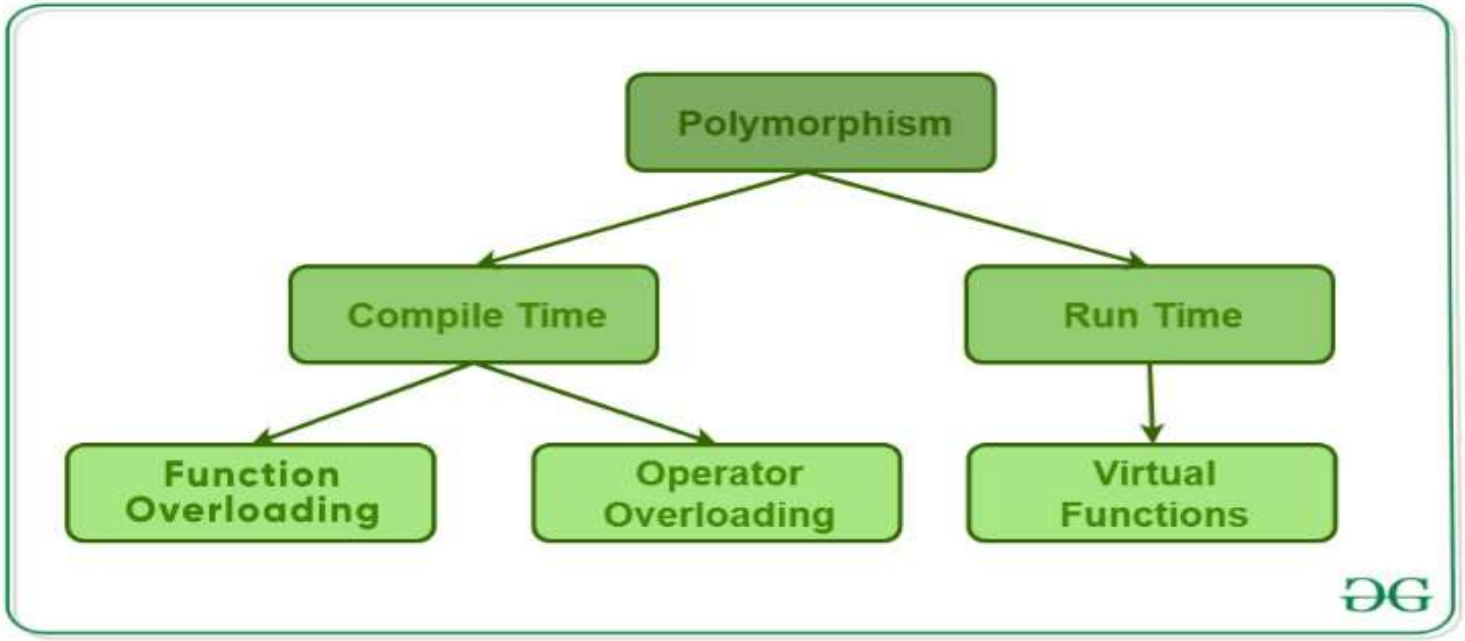
- Compile-time Polymorphism
- Runtime Polymorphism

- **Compile-time polymorphism,** also known as static polymorphism, refers to the mechanism in programming languages where the selection of which function to execute is done at compile time. This is based on the types of arguments passed to the function and is resolved during the compilation phase of the program.
- **Runtime polymorphism**, also known as dynamic polymorphism, is a feature of object-oriented programming languages like C++. It allows a function or method to behave differently based on the object it is called with. In C++, runtime polymorphism is achieved through virtual functions and function overriding.

1. **Function Overloading**

When there are multiple functions with the same name but different parameters, then the functions are said to be overloaded, hence this is known as Function Overloading. Functions can be overloaded by changing the number of arguments or/and changing the type of arguments.

Including <bits/stdc++.h> is a common practice in C++ programming to include all standard library headers in one go. This header file is not part of the C++ standard library, but it's supported by many compilers, including GCC and Clang.



Operator overloading:

- Operator overloading is a compile-time polymorphism. It is an idea of giving special meaning to an existing operator in C++ without changing its original meaning.

Syntax- return type className :: operator op (arg_list)

```
{
   //Function body;
}
```

The lists of such operators are:

1. Class member access operator (. (dot), .* (dot-asterisk))
2. Scope resolution operator (::)
3. Conditional Operator (?:)
4. Size Operator (sizeof)

Operator overloading of “+” operator

```
#include <iostream>
class Complex {
private:
    float real;
    float imag;
public:
   // Constructor to initialize real and imaginary parts
    Complex(float r = 0.0, float i = 0.0) : real(r), imag(i) {}
// Overload the '+' operator
    Complex operator+(const Complex& other) const {
        return Complex(real + other.real, imag + other.imag);
    }
 // Function to display the complex number
    void display() const {
        std::cout << real << " + " << imag << "i" << std::endl;
    }
};
```

```
int main() {
    Complex c1(3.5, 2.5), c2(1.2, 3.3);
    Complex c3 = c1 + c2;  // Using the overloaded '+' operator

    std::cout << "Complex number c1: ";
    c1.display();
    std::cout << "Complex number c2: ";
    c2.display();
    std::cout << "Sum of c1 and c2: ";
    c3.display();

    return 0;
}
```