Unit 1st

1.

a. **Nested class** -

Nested class in c++ is a type of class that is declared inside another class. The class inside which a nested class is declared is called an enclosing class.

The nested class is also a member variable of enclosing class and has the same access rights as the other members. However, the member functions of the enclosing class have no special access to the members of a nested class. The name of a nested class is local to its enclosing class. unless you use explicit pointers, references, or objects names, declaration in a nested class can only use visible constructors, including type names, static members, and enumerators from the enclosing class and global variables.

The nested class is considered a member of the enclosing class, adhering to encapsulation principles.

The members declared inside the nested class have usual-usage access to the members of enclosing class.

The nested class in c++ will be considered a member of enclosed class.

Example .

```
#include <iostream.h>
#include <conio.h>
class A
  {
public:
 class B
   {
  private:
    int num = 78;
  public:
   void getdata()
     {
   cout<< "The value of num is"<<num<<endl;
     }
   };
class C : public B
    {
  public:
  void func()
    {
  cout<<" class C is inherited from class B"<<endl
    }
 };
};
```

```
int main()
    {
    clrscr();
A::B obj;
obj.getdata();

A::C cc;
cc.fun();
cc.getdata();
 getch();
 return 0;

}
```

Output - The value of num is 78
class C is inherited from class B
The value of num is 78.

b. Arrays of objects

When a class is defined, only the specification for object is defined, no memory or storage is allocated. To use the data and access functions defined in the class, you need to create 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-1:

```
#include <iostream.h>
#include <conio.h>
class BaseClass
    {
public:
  virtual void Display()
        {
    cout<<"Display function of Base class"<<endl;
        }
    };

class DerivedClass-1 : public BaseClass
        {
public:
void Display()
        {
cout<<"Display function of Derived class1"<<
                                        endl;
        };
```

```
class DerivedClass-2: public BaseClass
    {
public:
void Display()
    {
cout<<"Display function of Derived class 2"<<endl;
    }
};

void main()
  {
clrscr();
Base *obj[2];
DerivedClass_1 obj1;
DerivedClass_2 obj2;
obj[0] = &obj1;
obj[1] = &obj2;
obj[0]->Display();
obj[1]->Display();
getch();
    }
```

Output - Display function of Derived class 1
Display function of Derived class 2

Example - 2 .

```
#include <iostream.h>
#include <conio.h>
class student
  {
  int id;
  int age;
  string name;
  string job;
public:
  void getdata();
  void putdata();
  int getid()
    {
    return id;
    }
  };
void student :: getdata()
  {
  cout<< "Enter Id : ";
  cin >> id;
  cout << "Enter age: ";
  cin >> age;
  cout<< "Enter name: ";
  cin >> name;
  cout << "Enter Job : ";
  cin >> job;
  }
```

```
void Student::putdata()
  {
  cout << id << " ";
  cout << age << " ";
  cout << name << " ";
  cout << job << " ";
  cout << endl;
  }

bool isDuplicate(Student st[], int n, int id)
  {
  for(int i=0; i<n; ++i)
    {
    if (st[i].getId() == id)
      return true; // Id already exists
    }
  return false; // Id is unique
  }

int main()
  {
  clrscr();
  Student st[30];
    int n, i;
  cout<<"Enter number of student - ";
  cin >> n;
```

```
for (i=0; i<n; i++)
    {
    st[i].getdata();
    while(isDuplicate(st, i, st[i].getId()))
      {
      cout<<"Error! ID already exists.
            Please enter a unique Id."<<endl;
      st[i].getdata();
      }
    }
cout<<"Student Data - "<<endl;
  for (i=0; i<n; i++)
      st[i].putdata();
  return 0;
    }
```

Output -

```
Enter number student - 2
Enter Id : 2
Enter age : 21
Enter Name : Deba
Enter Job : WD
Enter Id : 2
Enter age : 23
Enter Name : punk
Enter Job : DA
ERROR
```

Error: ID already exists. please enter a unique ID.

Enter Id : 4
Enter age : 23
Enter name: punk
Enter job: DA

Student Data -

2	21	Deva	WD
4	23	punk	DA

C. pointers.

pointer is a derived data type that refers to another data variable by storing the variable's memory rather than data. A pointer variable defines where to get the value of a specific data variable instead of defining actual data.

Declaring and Initializing pointers -

```
data-type * pointer-variable;
data-type* pointer-variable;
data-type *pointer-variable;
```

Here pointer-variable is the name of pointer and data-type can be int, char, float and so on.

example - int *ptr, a;
ptr = &a;

here ptr contains address variable a

Example-1

```
#include <iostream.h>
#include <conio.h>

int main()
  {
  clrscr();
int i = 10;
int *ptr1 = &i;
int **ptr2 = &ptr1;
   *ptr1 = *ptr1 + i;

Cout<< ptr1 <<endl; //memory address of i
Cout<<*ptr1<<endl; //value of i

Cout<< ptr2 <<endl; //memory address of ptr1
Cout<< *ptr2<<endl; // value pointed to by ptr2, which
    //is the memory address of i (since ptr1 holds
                                 that address)

Cout<< **ptr2 <<endl; //value pointed to by the value pointed
  getch();
}  //to by ptr2, which is the value of variable i.
```

output :-

```
0x7ffedb176cf4
20
0x7ffedb176ce8
0x7ffedb176cf4
20
```

Example-2 · Array of pointers ·

```
#include<iostream.h>
#include<conio.h>
int main()
    {
    clrscr();
    int *p = new int[5];
    for(int i=0; i<5; i++)
            {
        p[i] = 10 * (i+1);
            }
Cout<< *p <<endl;
Cout<< *p+1 <<endl;
cout<< *(p+1)<<endl;
Cout<< 2[p] <<endl; // *(p+2)
Cout<< p[2]<<endl;
   *p++;
  Cout<< *p;
   getch();
   return 0;
    }
```

Output -

```
10
11
20
30
30
20
```

2.
a. Constructor function.

A constructor is a special member function whose task is to initialize the objects of its class. It is special because its name is the same as the class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it constructs the values of data members of the class.

Features of constructors -

- They should be declared in public section.
- They are invoked automatically when objects are created.
- They don't have return types, not even void and therefore, and they can't return values.
- They can't be inherited, though a derived class can call the base class constructor.
- They can have default arguments.
- constructors can't be virtual.
- We can't refer to their addresses.
- An object with a constructor (or destructor) can't be used as a member of a union.
- They make 'implicit calls' to the operators new and delete when memory allocation is required.
- When a constructor is declared for a class, initialization of the class objects becomes mandatory.

Types of constructors.

- Default constructor
- parameterized constructor
- Copy constructor

Default constructor -

A default constructor is a constructor that doesn't take any argument. It has no parameters. It is also called a zero-argument constructor.

- Even if we don't define any constructor explicitly, the compiler will automatically provide a default constructor implicitly.

parameterized constructor -

parameterized constructors make it possible to pass arguments to constructors. These arguments help initialize an object when it is created.

To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.

Example -

```
#include<iostream.h>
#include<conio.h>
  int i;
  class abc
     {
  public:
    abc()
     {
     i=89;
  cout<<"the value of i is:"<<i<<endl;
     }
abc(int i, int j=90) //Constructor overloading
     {
   cout<<i<<endl;
  cout<<j<<endl;
     }
  };
int main()
   {
  clrscr();
   abc aa; //Default Constructor object
   abc bb(7); //parameterized constructor object
   abc cc(7,89); //7 will override the default value.
    getch();
     }
```

Output - The value of i is : 89
7
90
7
89

b. ~~Destructor function~~ -

Defining Constructor outside class.

```
#include <iostream.h>
#include <conio.h>
class student
    {
  int rno;
  char name[50];
  public:
    student();
    void display();
    };
student :: student()
    {
cout << "Enter the ROLLNO: ";
cin >> rno;
cout << "Enter the Name: ";
cin >> name;
}
```

```
void student :: display()
    {
cout<<endl<<rno<<"\t"<<name<<"\t";
    }
int main()
    {
  clrscr();
   student s;
   s.display();
    return 0;
      getch();
    }
```

output -

```
Enter the RollNo: 48
Enter the Name: Deba

48  Deba
```

Copy Constructor -

A copy constructor is a member function that initializes an object using another object of same class.

Example -

```
#include <iostream.h>
#include <conio.h>

class sample
    {
     int id;
    public:
     sample() //default constructor
        { }
    sample(int ok) //parameterized constructor
        {
        id= ok;
        }
    sample (sample& t) //copy constructor
        {
        id= t.id;
        }
    void display ()
        {
        cout << " ID = " << id;
        }
    };
```

```
int main()
  {
  clrscr();
Sample obj1(10);
obj1.display();
 cout << endl;

Sample obj2(obj1); //copy constructor called
                   //or obj2 = obj1

obj2.display();
  getch();
   }
```

Output -

```
ID =10
ID =10
```

b' Destructor function.

A destructor is used to destroy the objects that have been created by a constructor. Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde(~).

A destructor never takes any argument nor does it return any value. It will be invoked implicitly by the compiler upon exit from the program to clean up storage that is no longer accessible. It is a good practice to declare destructors in a program since it releases memory space for future use.

- Destructors can also be defined inside or outside of class.

Syntax: ~ class-name () { }

for outside class — classname :: ~ class-name () { }

Features of Destructors -

- Destructor is invoked automatically by the compiler when its corresponding constructor goes out of scope and releases the memory space that is no longer required by program.
- Destructor neither requires any argument nor returns any value therefore it can't be overloaded.
- Destructor can't be declared as static and const.
- Destructor should be declared in the public section of the program

Example -

```
#include <iostream.h>
#include <conio.h>

class sample
    {
public:
    sample()
        {
        cout<<"Objey created"<<endl;
        }
    ~sample()
        {
        cout<<"Objey destroyed"<<endl;
        }
};

int main()
    {
    sample s;
    return 0;
    }
```

output -

```
Objey created
Objey destroyed
```

3. Dynamic allocation operator : new() and delete()

Dynamically allocated memory is allocated on Heap, and non-static and local variable get memory allocated on Stack. We are free to allocate and deallocate memory whenever we need it and whenever we don't need it anymore.

How is it different from memory allocated to normal variables ? -

For normal variable like "int a" memory is automatically allocated and deallocated. For dynamically allocated memory like "int *p = new int[]", it is the programmer's responsibility to deallocate memory when no longer needed. If the programmer doesn't deallocate memory, it causes a memory leak (memory is not deallocated until the program terminates).

C++ supports these functions and also has two operators new and delete, that perform the task of allocating and freeing the memory in a better and easier way.

Since dynamic memory allocation happens at runtime, it makes it possible to efficiently allocate memory even when we don't know the variable size in advance. This also allows for allocation to varying sizes without any issues.

The new operator denotes a request for memory allocation on the free store. If sufficient memory is available, a new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

Syntax for new operator.

```
pointer-variable = new data-type;
int *p = new int;
```

delete operator.

```
delete pointer-variable;
delete p;
```

Example -1

```
#include <iostream.h>
#include <conio.h>
int main()
  {
  clrscr();
  int *i;
   i = new int;
   *i = 87;
  cout << *i ;
   delete i;
   getch();
  }
```

output - 87

Example-3 .

```
#include <iostream.h>
#include <conio.h>
  class tests
      {
    public:
    tests()
      {
    cout << "Good" << endl;
      }
    ~tests()
      {
    cout << "Morning" << endl;
      }
    };

int main()
    {
  tests *t = new tests[3];
    delete *[] t;
    return 0;
    }
```

output -

```
Good
Good
Good
Morning
Morning
Morning
```

Unit IInd

1. Destructors in derived class -

When you call destructor on derived class over a pointer or a reference, where base class has virtual destructor, the most derived destructor will be called first and then the rest of derived classes in reversed order of constructions. This is to make sure that all memory has been properly cleaned. It would not work if the most derived class wasn't called last because by that time the base class would not exists in memory and you would get segmentation fault. This is by design. The destructor on the base class must be called in order for it to release its resources. A derived class should only clean up its own resources and leave the base class to clean up itself.

NOTE - Constructor in inheritance -

- Constructor execute from parent to child but Constructor calls from child to parent
- Destructor execute from child to parent

making base class destructor virtual guarantees that the object of derived class is destructed properly, i.e both base class and derived class destructors are called.

Example -

```
#include<iostream.h>
#include<conio.h>
class A
  {
  public:
    A()
    {
    cout<<"Base class constructor"<<endl;
    }
virtual ~A()
    {
    cout<<"Base class destructor"<<endl;
    }
  };
class B: public A
    {
    public:
      B()
      {
      cout<<"Derived class constructor"<< endl;
      }
~B()
  {
  cout<<"Derived class destructor"<< endl;
  }
};
```

```
int main()
    {
    greeter();
B *b = new BC;
A *a = b;
 delete a;
// delete b; segmentation fault
 getch();
 return 0;
 }
```

output -

```
Base class constructor
Derived class constructor
Derived class destructor
Base class destructor
```

9. public, protected and private Inheritance -

private Inheritance -

In private inheritance, the public and protected members of base class become private members in the derived class. private inheritance is often used when the derived class wants to use the implementation details of base class but does not want to expose them to outside world. The derived class can access the base class's members, but they are treated as private and can't be accessed by objects of derived class or any other class.

Protected Inheritance -

In protected inheritance, the public and protected members of the base class become protected members in the derived class. protected inheritance is less common and has limited use cases. It provides a level of access between private and public inheritance. The derived class and its derived classes can access the protected members of base class, but objects of the derived class or unrelated classes can't access them.

public Inheritance -

In public inheritance, the public members of the base class become public members in the derived class, and the protected members become protected members. public inheritance is the most common form of inheritance and represents an "is-a" relationship between the base and derived classes.

The derived class and any other classes can access the public members of base class, while the protected members are accessible only by the derived class and its derived classes.

Example -

```
#include<iostream.h>
#include<conio.h>
 class Grandfather
    {
  private:
     void pension()
       {
        cout<<"pension is 45000"<<endl;
       }
  protected:
     void medicine()
       {
       cout<<"medicine price is 5000"<<endl;
       }
  public:
   void Roomrent()
     {
     cout<<"Roomrent is 5000"<<endl;
     }
   void P()
     {
      pension();
      }
     };
```

```
class father : public grandfather
        {
    public:
        void display()
            {
            Rooment();
            medicine();
            }
        };

class child: protected Grandfather
        {
    public:
        void display()
            {
            Rooment();
            medicine();
            }
        };

class Greatgrandchild: private grandfather
        {
    public:
        void display()
            {
            Rooment();
            medicine();
            }
        };
```

```
int main()
    {
    clrscr();
  Father f;
  f.display();
   f.p();

 Child c;
  c.display();
// c.Roomrent(); //can't access protected member
                 //directly outside of child

// c.p(); //can't access private member directly
          // outside of Grandfather

 GreatgrandChild g;
    g.display();
 // g.Roomrent(); //can't access private member
          //directly outside of Grandfather

// g.p(); //can't access private member directly
          //outside of Grandfather
  getch();
 return 0;
  }
```

Output -

Rooment is 5000
Medicine price is 5000
pension is 45000
Rooment is 5000
medicine price is 5000
Rooment is 5000
Medicine price is 5000

3. Function overriding.

Overriding Base class members in a Derived class.

- function overriding is also called method overriding.
It allows a derived class to override or replace a function inherited from its parent class, providing a tailored implementation to suit its unique requirements.
- function overriding enhance adaptability and customization of codebase.
- The child class inherits all the data members and the member functions present in parent class.
- function overriding means creating a newer version of the parent class function in the child class.

Advantages of function Override -

Polymorphism: Overriding allows a program to execute the appropriate version of a function based on the object's type during runtime, enhancing flexibility and adaptability, thereby making it a backbone of polymorphism.

Code reusability: - Developers can reuse existing code by inheriting functions from a base class and overriding them in derived classes. This promotes a more efficient and organized code structure.

Customization: - Overriding enables the customization of inherited functionality in derived classes. This is particularly useful when a specific class requires a modified or enhanced version of a function inherited from its parent class.

Enhances code readability - The override function in C++ helps create class hierarchies, promoting a structured and organized codebase. This hierarchical arrangement simplifies code maintenance and readability.

Keeping code clean - With method overriding, you can use the same function name for different tasks, making code cleaner. This reduces confusion in naming and simplifies the code, making it easier to understand.

Saves memory - overriding enhances memory efficiency by avoiding unnecessary duplication, ensuring a consistent and streamlined code structure. This not only promotes clarity but also facilitates ease of maintenance.

Adaptability :- The overridde function is to create specialized versions of functions in derived classes and enhances the overall adoptability of the code. This is crucial for accommodating varying requirements in different parts of the program.

Example -

```
#include <iostream.h>
#include <conio.h>

class fun
  {
public:
 void play()
   {
  cout<< "playing in fun1"<<endl;
    }

void run()
  {
 cout << " Running in fun1"<<endl;
   }
};
```

Output :-

Playing in fun1
Playing in fun-2
playing in fun-3

Unit - IIIrd

1. Operator overloading (unary & binary) .

Operator overloading provides a flexible option for the creation of new definitions for most of the C++ operators.

Operator can overload all the C++ operators except the following -

- Class member access operators (.*)
- Scope resolution operator (::)
- Size operator (sizeof)
- Conditional operator (?)

The reason why we can't overload these operators may be attributed to the fact that these operators take class name as their operand instead of values, as in the case with other normal operators.

- operator functions must be either member functions (nonstatic) or friend functions. A basic difference between them is that a friend function will have only one argument for unary operators and two for binary operators, while a member function has no arguments for unary operators and only one for binary operators.

Example.

Overloading of unary Operator -

In the unary operator function, no arguments should be passed. It works only with one class object. It is the overloading of an operator operating on a single operand.

```
#include <iostream.h>
#include <conio.h>

class base
  {
public:
  int i= 9, j=7;
  void operator ++()
    {
    i++;
    j++;
   cout<<"i: "<< i<<endl;
   cout<<"j: "<< j<<endl;
    }
  };
int main()
  {
  base b;
    ++b;
  return 0;
  }
```

output -

```
i: 10
j: 8
```

The function operator() takes no argument. Then, what does this operator function do? changes the sign of data members of the object. Since this function is a member function of the same class, it can directly access the members of the object which activated it.

Overloading of Binary operator -

In the binary operator overloading function, there should be one argument to be passed. It is the overloading of an operator operating on two operands.

```
#include<iostream.h>
#include <conio.h>
class Base
    {
public:
  int i=9, j=7;
  void operator+()
    {
      ++i;
      ++j;
  // cout<<"i: "<<i<<endl;
  // cout<<"j: "<<j<<endl;
    }
};
```

```
int main()
  {
  cin>>b.i+b.j;
  return 0;
  getch();
  return 0;
  }
```

Output - 16

2. Virtual Destructor -

A constructor can not be virtual. First, to create an object the constructor of the object class must be of the same type as the class. But, this is not possible with a virtually implemented constructor. second, at the time of calling a constructor, the virtual table would not have been created to resolve any virtual function calls. Thus, a virtual constructor itself would not have anywhere to look up to. As a result, it is not possible to declare a constructor as virtual.

We can use virtual destructor to make sure that the different destructors in an inheritance hierarchy are called in order, particularly when the base class pointer is referencing to a derived type object.

Example -

```
#include <iostream.h>
#include <conio.h>
  class A
    {
  public:
    A()
     {
    cout<< "Base class constructor"<<endl;
     }
   ~A()
    {
   cout<<"Base class destructor"<<endl;
    }
  };
```

```
class B: public A
    {
    public:
    B()
    {
    cout<<"Derived class constructor"<<endl;
    }
   ~B()
    {
   cout<<"Derived class destructor"<<endl;
    }
   };
int main()
    {
   clrscr();
    B *b = new B();
    A *a = b;
    delete a;
   //delete b; segmentation fault
    getch();
    return 0;
    }
```

output -

```
Base class constructor
Derived class constructor
Base class destructor
```

In the above class declarations, both class A and the derived class B have their own destructors. Now, an A class pointer has been allocated to a B class object. When this object pointer is deleted using the delete operator, it will trigger the base class destructor and the derived class destructor won't be called at all. This may lead to a memory leak situation. To make sure that the derived class destructor is mandatorily called, we must declare the base class destructor as virtual, as shown below.

Example.

```
#include <iostream.h>
#include <conio.h>
    class A
      {
    public:
     A()
      {
     cout<<"Base class constructor"<<endl;
      }
    virtual ~A()
        {
     cout<<"Base class destructor"<<endl;
        }
    };
```

```
class B : public A
    {
public:
  B()
    {
    cout<<" Derived class constructor"<<endl;
    }
 ~B()
    {
    cout<<"Derived class destructor"<<endl;
    }
};

int main()
    {
    clrscr();
    B *b = new B();
    A *a = b;
    delete a;
  // delete b; segmentation fault
    getch();
    return 0;
    }
```

Output:

```
Base class constructor
Derived class constructor
Derived class destructor
Base class destructor
```

Virtual Derivation.

Below the 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 any 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 erroar.

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.

virtual can be written before or after the public. Now only one copy of data/function member will be copied to class C and class B and class A becomes the virtual base class. virtual base classes offer a way to save space and avoid ambiguities in class hierarchies that use multiple inheritances. When a base class is specified as a virtual base, it can act as an indirect base more than once without duplication of its data members. A single copy of its data members is shared by all the base classes that use virtual base.

Example.

```
#include<iostream.h>
#include<conio.h>

class A
  {
public:
void foo()
 {
cout<< "A foo()" << endl;
  }
 };

class B: public virtual A
  {
public:
void display()
  {
 cout<<" B display()" << endl;
  }
 };

class C: public virtual A
  {
public:
 void bage()
  {
  cout<< "C bage()" << endl;
    }
  };
```

```
class D: public B, public C
        {
        };

int main()
    {
    clrscr();
     D d;
    d.foo(); // call foo() from classA
    d.B::foo(); // call foo() from class B
    d.C::foo(); // call foo() from class C

    d.display();
    d.base();
    getch();
    return 0;
     }
```

output :

```
A foo()
A foo()
A foo()
B display()
C base()
```

3. Friend 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. For example, a LinkedList class may be allowed to access private members of Node.

We declare a friend class in c++ by using the friend keyword.

Example -

```
#include <iostream.h>
#include <conio.h>
  class A
    {
   private:
     int i = 9;
   protected:
     int j = 6;
   public:
     int k = 8;
   friend class F;
    };
```

```
class f
  {
  public:
  void sum(A& t)
       {
  cout<<"The value of i+j= "<< t.i+t.j<<endl;
  cout<<"The value of k is: "<<t.k;
       }
  };
int main()
  {
  clrscr();
  A a;
  f f;
  f.sum(a);
  getch();
  return 0;
  }
```

Output :- The value of i+j= 15
The value of k is: 8

Inline functions:

One of the objectives of using this functions in a program is to save memory space, when a function is likely to be called many times. However, everytime a function is called, it takes a lot of extra time in executing a series of instructions for tasks such as jumping to the function, saving registers, pushing arguments into the stack, and returning to the calling function. When a function is small, a substantial percentage of execution time may be spent in such overheads.

One solution to this problem is to use macro definitions, popularly known as macros. Preprocessor macros are popular in C. Macros are almost never necessary in C++. and they are error-prone. There are some problems with the use of macros in C++. Macro can't access private members of the class, Macros look like function calls but they are actually not.

Another thing is that the macros are managed by the preprocessor and inline functions are managed by C++ compiler. It is true that all the functions defined inside the class are implicitly inline and the C++ compiler will perform inline calls of these functions, but C++ compiler can't perform inline if the function is virtual. The reason is called to a virtual function is resolved at runtime instead of compile-time. virtual means waiting until runtime and inline means during compilation, if the compiler doesn't know which function will be called, how it can perform inlining?

One other thing to remember is that it is only useful to make the function inline if the time spent during a function call is more compared to the function body execution time.

C++ has a different solution to this problem. To eliminate the cost of calls to small functions, C++ proposes a new feature called inline function. An inline function is a function that is expanded in line when it is invoked. That is, the compiler replaces the function call with the corresponding code.

The speed benefits of inline functions diminish as the function grows in size. At some point the overhead of the function call becomes small compared to the execution of the function, and the benefits of inline functions may be lost. In such cases, the use of normal functions will be more meaningful. Usually, the functions are made inline when they are small enough to be defined in one or two lines.

An inline function can increase the function size so much that it may not fit in the cache.

It makes the program to take up more ~~more~~ memory because the statements that define the inline function are reproduced at each point where the function is called.

Example -

```
#include<iostream.h>
#include<conio.h>
class Addition
    {
    int i,j;
    public:
    void add();
    void sub();
     };
inline void Addition::add()
     {
    cout<<"Enter first value:";
    cin>>i;
    cout<<"Enter second value:";
    cin>>j;
  cout<<"Addition of i and j is:"<<i+j<<endl;
    }
inline void  Addition::sub()
     {
    cout<<"Subtraction of i and j is:"<<i-j<<endl;
     }
int main()
 {
clrscr();
  Addition a;
  a.add();
  a.sub();
   getch();
    }
```

Output - Enter first value : 3
Enter second value : 2
Addition of i and j is : 5
Subtraction of i and j is : 1

Situations where inline expansion may not work.

- For functions returning values, if a loop, a switch, or a goto exists.
- If a function return type is other than void, and the return statement doesn't exist in a function body.
- If functions contain static variables.
- If inline functions are recursive.

Unit - IV

1. File Handling.

The I/O system of C++ handles file operations which are very much similar to the console input and output operations. It uses file streams as an interface between the programs and the files. The stream that supplies data to the program is known as input stream and the one that receives data from the program is known as output stream.

For achieving file handling we need to follow the following steps -

1 - Naming a file
2 - Opening a file
3 - Writing data into file
4 - Reading data from the file
5 - Closing a file

Classes for File stream operations -

For file handling classes include ifstream, ofstream and fstream. These classes are derived from fstreambase and from the corresponding iostream class.

: **ios** – ios stands for input output stream.
- This class used by all other derived classes for input and output operations.

istream .
- istream stands for input stream.
- This class is derived from class 'ios'.
- It handles input stream.
- The extraction operator (>>) is overloaded in this class to handle input streams from files to the program execution.
- This class declares input functions get(), getline() and read.

Ostream .
- Ostream stands for output stream.
- This class is derived from the class 'ios'.
- It handle output stream.
- The insertion operator (<<) is overloaded in this class to handle output streams to files from the program execution.
- This class declares output functions such as put() and write().

Streambuf .
This class contains a pointer which points to the buffers which is used to manage the input and output streams.

filebuf - Its purpose is to set the file buffers to read and write.

fstreambase - provides operations common to the file streams. serves as a base for fstream, ifstream and ofstream class.

ifstream - provides input operations. Contain open() with default input mode. Inherits the functions get(), getline(), read(), seekg() and tellg() functions from istream.

ofstream - provides output operations. Contains open() with default output mode. Inherits put(), tellp() and write() functions from ostream.

fstream - provides support for simultaneous input and output operations. Inherits all the functions from istream and ostream classes through iostream.

3. Opening and Closing Files
read() and write()

C++ provides us with four different operations for file handling. They are:

Open() - This is used to create a file
read() - This is used to read the data from the file.
write() - This is used to write new data to file.
close() - This is used to close the file.

Opening files in C++.

To read or enter data to a file, we need to open it first. This can be performed with the help of 'ifstream' for reading and 'ifstream' or 'ofstream' for writing or appending to the file. All these three objects have open() function pre-built in them.

Writing to File.

We will use fstream or ofstream object to write data into the file and to do so. We will use stream insertion operator (<<) along with the text enclosed within the double-quotes.

With the help of open() function, we will create a new file and then we will set the mode to 'ios::out' as we have to write the data to file.

Closing a file -

Closing a file is a good practice, and it is must to close the file. whenever the c++ program comes to an end, it clears the allocated memory, and it closes the file.

Reading from file .

Getting the data from the file is an essential thing to perform because without getting the data, we can't perform any task. We can perform the reading of data from a file with the cin to get data from the user.

Example -

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
    {
    fstream file;
    file.open("void.txt", ios::out);
if(!file)
    {
cout<<"Error while creating/opening
                the file";
    }
else
    {
cout<<"File created successfully";
file<<" Good Morning "<<endl;
file<<" 56";
    }
```

```
file.close();
File.open("void.txt", ios::in);
if(!File){
  cout<<"Error while opening the file";
  }
  else
   {
cout<<endl<<"Contents of the file:"<<endl;
   int a;
while (File>>a)
      {
    cout<<a;
    }
   }
File.close();
return 0;
  }
```

3. File pointers and their Manipulation.

The read operation from a file involves get pointer. It points to a specific location in the file and reading starts from that location. Then, the get pointer keeps moving forward which lets us read entire file. Similarly, we can start writing to a location where put pointer is currently pointing. The get and put are known as file position pointers and these pointers can be manipulated.

The functions which manipulate file pointer are as follows:

Function	Description
seekg()	Moves the get pointer to a specific location in the file.
seekp()	moves the put pointer to a specific location in the file.
tellg()	Returns the position of get pointer
tellp()	Returns the position of put pointer.

Manipulators.

Manipulators are helping functions that can modify the input/output stream. It ~~does not~~ only modifies the I/O stream using insertion(<<) and extraction(>>) operators.

Types of Manipulators.

Manipulators without arguments:

The most important manipulators defined by the IOstream library are provided below.

endl: It is defined in ostream. It is used to enter a new line and after entering a new line it flushes the output stream.

ws: It is defined in istream and is used to ignore the whitespace in the string sequence.

ends: It is also defined in ostream and it inserts a null character into the output stream.

flush: It is also defined in ostream and it flushes the output stream.

Example.

```
#include<iostream>
#include <istream>
#include <sstream>
#include <string>
using namespace std;
int main()
    {
    istringstream str(" Debdatta");
    string a;
```

```
//Extracting the trimmed a
getline(str >> std::ws, a);
//printing the trimmed a
cout << a << endl;
cout << "Good Morning" << flush;

cout << "\nc";
cout << "a" << ends;
cout << "+" << endl;
cout << "a" << ends
return 0;
}
```

output -

```
Debadatta
Good Morning
ca'+
a'
```

4. seekg() and tellg()

seekg() - moves the get pointer to a specific location in the file.

tellg() - Returns the position of get pointer.

Example :

```
#include<iostream>
#include<fstream>
using namespace std;
class student
    {
  int rol;
  char Name[20];
public:
  void display(int k);
    };
void student::display(int k)
    {
ifstream fs("student.dat",ios::in |ios::binary);
if(!fs.is_open())
    {
  cout<<"Error opening file"<<endl;
    return;
    }
```

```
fs.seekg(k * sizeof(student));
fs.read((char *) this, sizeof(student));
fs.
cout << "current position: "
     << "student no: "
     << fs.tellg() / sizeof(student) + 1;
fs.seekg(0, ios::end);
cout << " of "
     << fs.tellg() / sizeof(student)
     << endl;
fs.close();
}
int main()
{
int k = 7;
student s;
s.display(k);
return 0;
}
```

output - Current position: student no: 1 of 0