Input /output streams: In C++, input and output are performed in the form of a sequence of bytes which are known as **streams**.

- **Input Stream:** This is a type of stream where the bytes flow from a device such as a keyboard to the main memory.

- **Output Stream:** This is a type of stream where the bytes flow in the opposite direction, that is, from main memory to the device such as display screen.

Iostream: Iostream stands for standard input-output stream. It consists of ostream and istream. It is a header file available in C++ to perform input and output operations. This header file contains definitions of objects like cin, cout etc. One must include this header file to perform any input/output operations.
Syntax: include <iostream.h>

The standard Input/Output objects: The two instances cout and cin of iostream class are the most basic methods of taking input and printing output in C++.

- **Standard output stream (cout)**: The cout statement is the instance of the ostream class. It is used to produce output on the standard output device . The data needed to be displayed on the screen is inserted in the standard output stream (cout) using the insertion operator(**<<**).

- **standard input stream (cin)**: The C++ cin statement is the instance of the class istream. It is used to read input from the standard input device. The extraction operator(**>>**) is used along with the object cin for reading inputs.

Example program- 1:
```
#include <iostream>
using namespace std;

int main()
{
    //prints the content enclosed in double quotes
    cout<<"This is C++ programming.";
    return 0;
}
```
Output: This is C++ programming.

Example program-2:

```
int main()
{
   int num=5;
   char c= 'A';

   cout<<num<<endl;   //prints integer
   cout<<"Number: "<<num<<endl;//prints integer after message
   cout<<c<<endl; //prints character
   cout<<"Character: "<<c<<endl;//prints character after message
   return 0;
}
```

Output: 5

```
         Number: 5
         A
         Character: A
```

Example program-3:

```
int main()
{
   int num;
   cout<<"Enter an integer: "<<endl;
   cin>>num;   //Taking input
   cout<<"Number: "<<num<<endl;

   return 0;
}
```

Output: Enter an integer:

```
         3 (entered by user)
         Number: 3
```

Manipulators: These are the functions or operators that can modify the input and output streams. These are used to manipulate or format the data in the desired way. They modify the I/O stream using insertion (<<) and extraction (>>) operators.

- One of the most commonly used manipulator is ‘endl’.
- It is defined by the ‘iostream’ library.
- It stands for endline and moves the cursor to the next line.
- If we do not use endl, the next output will be displayed in the same line.
- The endl has the same function as that of ‘\n.’

Example program-4:

```
int main()
{
   char a;
   int num;
   cout<<"Enter a character and an integer: "<<endl;
   cin>>a>>num;   //Taking two inputs
   cout<<"Character: "<<a<<endl;
   cout<<"Number: "<<num<<endl;

   return 0;
}
```

Output: Enter a character and an integer:

```
          a (entered by user)
          5 (entered by user)
          Character: a
          Number: 5
```

Example program-5:

```
int main()
{
   int num1=2;
   int num2=3;
   cout<<num1;     //next output will be displayed on same line

   cout<<num2<<endl;   //the next output will be displayed
                      //in another line

   cout<<"First number: "<<num1<<endl;
   cout<<"Second number: "<<num2<<endl;

   return 0;
}
```

Output: 23

```
          First number: 2
          Second number: 3
```

Example program-6:

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

int main()
{
   int id= 12;
   string name="Manoz";

   //printing in different style
   cout<<"Our lecturer name is "<<name<<" and "
   <<id<<" is his id. "<<endl;

   return 0;
}
```

Output: Our lecturer name is Manoz and 12 is his id.

LET's dive into the syllabus now. (You have to study all the topics along with their examples thoroughly).

Some common types of functions available in C++

1. Inline function
2. Static member function
3. Friend function

1. Inline function: It is a function that is expanded in line when it is called. That means, whenever an inline function is called, the whole code of the function gets inserted or substituted at the point of function call. The substitution is performed by the C++ compiler at compile time.

❖ *Why do we use inline function?*

- It is generally used when the function definitions are small, and the functions are called several times in a program.
- It saves time to transfer the control of the program from calling function to the definition of the called function and hence reduces the execution time of a program.

Syntax for defining an inline function:

```
inline return-type  function-name (parameters) {
    // code statements
}
```

However, In-lining a function is just request to the compiler, and not the command. It depends on the compiler whether to accept or decline the request. Compiler may not perform in-lining of a function in certain circumstances such as:

- If a function contains a loop. (for, while, do-while)
- If a function contains static variables.
- If a function is recursive.
- If a function contains switch or go-to statements.

An example of a use of an inline function is given below. *[Example –program-7]*

Example program- 7:

```
inline int sum(int x, int y){  //inline function
 return x+y;
}

int main(){
  int a=5;
  int b=2;
  cout<<"The sum is: "<<sum(a,b)<<endl;
}
```

Output: The sum is: 7

Explanation: In the above program, the 'sum( )' function is declared as 'inline'. Therefore, the code which is inside the inline function is substituted at a place in the 'main( )' function where the inline function is called.

2. Static data member *(you have to know the basics of OOP, so explained later)*
3. Friend function *(you have to know the basics of OOP, so explained later)*

Default argument: It is an argument that is defined with a default value in a function declaration.

```
For e.g., void getData (int a=4) {   //this function takes integer 'a' as
                                    //argument which has default value 4.
          }
```

If the calling function does not pass any value to the default argument, then the complier automatically assigns the defined default value. If the calling function passes some value, then the default value is overridden. Once the default value is defined for an argument in the function, all the subsequent arguments after it must have a default value.

Advantage of using default argument:

- Default argument helps to reduce the code and the size of program.
- It increases the capability of the existing function.

An example of a use of default argument is given below. *[Example program-8]*

Example program- 8:

```
void sum(int x, int y=1){
 int sum=x+y;
 cout<<"The sum: "<<sum<<endl;
}

int main(){
 sum(5);
 sum(5,2);
}
```

Output: The sum: 6

```
The sum: 7
```

Explanation: In this program, the argument y in the function sum( ) is set as a default argument and is given a value of 1. Now, when a single parameter (5) is passed to it, x takes the value 5, and y takes the default value 1. When two parameters (5,2) are passed, x takes the value 5, and the default value in y is overridden as 2.

Q. Practice more examples that use more than one default argument. *[It was demonstrated in the class.]*

Q. Explain with example how the use of default arguments reduces the redundant code (reduces the code in the program/ reuse of code). *[It was explained/ demonstrated in the class]*

Reference variable: A reference variable is a reference to an existing variable. It provides an alternative name for an existing variable.

- It is actually an internal pointer.
- It should be initialized at the time of declaration and cannot be NULL.

- The operator '&' is used to declare reference variable. (but it is not read as 'address of')
- It cannot be changed to refer another variable.

Syntax: datatype &reference_variable_name = variable name;

For e.g. int a =10;
int &b = a; // here, b is a reference variable

An example of a use of reference variable is given below. *[Example program- 9]*

Example program- 9:

```
int main(){
 int a=10;

 //creating reference variable
 int &b= a;
 cout<<a<<endl;
 cout<<b<<endl;

 b=20;
 cout<<a<<endl;
 cout<<b<<endl;

 a=100;
 cout<<a<<endl;
 cout<<b<<endl;
}
```

```
Output: 10
        10
        20
        20
        100
        100
```

Explanation: In the above program, b is a reference variable that contains the reference of a. Both 'a' and 'b' point to the same memory location. So, both a and b give the same value. If b is modified, a will also be modified and vice-versa.

Example program-10:
Q. Write a program that swaps two numbers using reference variable.

```
void interchange(int &x, int &y){
  int temp=x;
  x=y;
  y=temp;
}

int main()
{
  int a=2, b=5;
  cout<<"Value of a: "<<a<<endl;
  cout<<"Value of b: "<<b<<endl;

  interchange(a,b);
  cout<<"Value of a: "<<a<<endl;
  cout<<"Value of b: "<<b<<endl;

  return 0;
}
```

```
Output: Value of a: 2
        Value of b: 5
        Value of a: 5
        Value of b: 2
```

Dynamic memory allocation: Reserving or providing space to a variable or an array is called memory allocation. The allocation in which memory is allocated dynamically is called dynamic memory allocation. Dynamically means that the memory of a variable or an array can be allocated in run time. It is opposite to that of static memory allocation where the memory is allocated during compile time.

In this type of allocation, the exact size of the variable is not known in advance. This process allocates memory on the segment known as heap. Pointers play a major role in dynamic memory allocation. In this allocation, the memory management is done manually by a programmer buy using **new** and **delete** operators.

Advantages/Uses:

- It allocates memory at run-time.

- It allows allocating the memory of variable size which is generally not possible at compile time.
- It provides flexibility to the programmers as they are free to allocate and de-allocate memory whenever they need and when they don't.

Process of dynamic memory allocation: In C++, the memory can be allocated and de-allocated dynamically using the new and delete operator respectively.

new operator: It is a special operator that performs the task of allocating memory dynamically.

It suggests a request for memory allocation on the heap segment of the memory which is also known as free store. If sufficient memory is available, then the new operator initializes the memory and returns the address of the newly allocated initialized memory to the pointer variable.

Syntax: pointer-variable = new data-type;
Example: int *p = new int;
Here, we have allocated memory dynamically for an integer variable using the *new* operator. The pointer *p* is used which points to the address of the memory location returned by the *new* operator.

The new operator can also be used to allocate a block (an array) of memory of a particular data-type dynamically.
Syntax: pointer-variable = new data-type[size];
Example: int *p = new int[10];
Above example allocates memory dynamically for 10 integers and the pointer *p* points to the address of the first element of the array.

delete operator: It is a special operator that performs the task of deallocating memory dynamically.
Once the memory is no longer needed, it should be free so that the memory becomes available again for other requests of dynamic memory. This is performed manually by the programmer using the delete operator.

Syntax: delete pointer-variable;
Example: delete p;
The above example releases the memory pointed by the pointer-variable p.

If a block of memory is allocated dynamically, then this block of memory should be de-allocated using delete operator if no longer needed.
Syntax: delete[ ] pointer-variable;
Example: delete[ ] p;
The above example releases the block of memory pointed by the pointer-variable p.

An example of a dynamic memory allocation and de-allocation is given below. *[Example program- 11]*

Example program-11:

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

int main()
{
    //declare an int pointer
    int *p;
    // dynamic memory allocation using the new keyword
    p= new int;
    // assign value to allocated memory
    *p = 10;
    // print the value stored in memory
    cout << *p; // Output: 10
    // deallocate the memory
    delete p;

    return 0;
}
```

Output: 10

Advantages of new operator over malloc:

- The *new* does not need the sizeof( ) operator whereas *malloc( )* needs to know the size before memory allocation.
- The *new* operator can be overloaded, but *malloc( )* cannot be overloaded.
- The *new* operator automatically returns the correct pointer-type, whereas *malloc* returns the void pointer which needs to be typecast to the desired type.
- The syntax of *new* is easier compared to that of *malloc.*
- The *new* throws an exception on failure whereas the *malloc* returns a NULL.

Example program-12:

Q. Write a program to store the numbers input by the user in an array using dynamic memory allocation, and display them.

```
int main()
{
  int num;
  cout<<"Enter the total numbers to store in an array: "<<endl;
  cin>>num;

  int *ptr = new int[num]; //dynamic memory allocation

  for(int i=0; i<num; i++){
    cout<<"Enter a number: "<<endl;
    cin>>ptr[i];
  }

  cout<<"The numbers stored in an array are: "<<endl;
  for(int i=0; i<num; i++){
    cout<<ptr[i]<<endl;
  }

  delete[] ptr;  //releases the memory pointed by ptr.

  return 0;
}
```

Output: Enter the total numbers to store in an array:

```
3 (entered by user)
Enter a number:
4 (entered by user)
Enter a number:
1 (entered by user)
Enter a number:
2 (entered by user)
The numbers stored in an array are:
4
1
2
```

Classes: Class is a user-defined data type that holds various variables and functions under one construct. It is regarded as a template or blueprint of the objects. It is a collection of the objects. Example of a class could be a template that represents a collection of books in a library, or a number of students at a college, or number of employees in an organization.

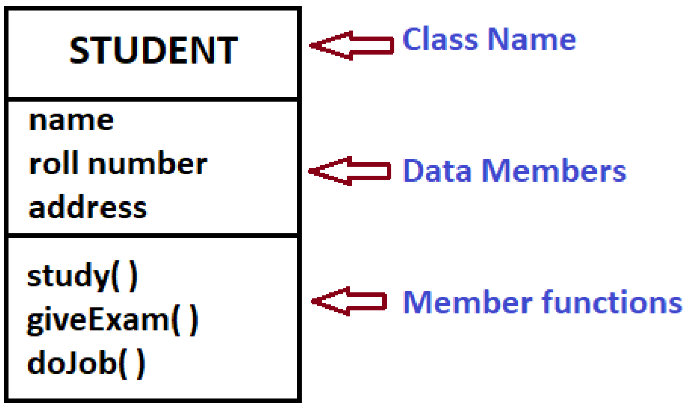
The variables are called data members and functions are called member functions. The member functions are used to manipulate the data members. The data members and member functions are accessed by creating an instance (object) of the class. These members (data, functions) define the properties and behavior of the instances in a class.

Objects: The objects are the real-world entities. The objects are the instances of a class. Example of object could be any entity such as particular book, a student, an employee, etc.

The data members and member functions defined inside class are accessed by creating objects. When a class is defined, only the specifications of the objects are defined and the memory is not allocated. The memory is allocated when the objects are created.

Case-example: For e.g. consider a number of students studying in a particular college. The students would have the same properties, such as names, roll number, address etc. But their identity is different. That means, the names, roll number and address of each student would be different. In addition to that, they may be performing similar activities such as study, give exam, do job etc.

Modeling with Object Oriented Programming: The above example can be modeled by object-oriented programming by creating objects and classes. At first, a class can be defined using the name 'Student'. The data members could be the name, roll number etc. and the member functions could be study, give exam etc. And each individual student could be represented by creating an instance (object) of the 'Student' class. The data members (eg. name) and member functions (eg. study) are accessed by using these objects.



State: The set of data fields with their current value is known as state. It is the defined attributes of the object.

For e.g. if we consider a particular student as an object, its attributes could be name, address, roll number etc. The state of the object is determined by the values of these attributes. If these attributes are defined as name= 'Aakash', rollNos='110' and address 'Bharatpur'. Then, they are the states of that object. The states are represented by data members inside the class.

Behaviors: These are the tasks that an object performs. If we consider a person as an object, which can speak, walk, eat etc., then these are the behaviors of the object person. Behaviors determine how an object acts and reacts in terms of state changes and message passing. The behaviors are defined by the member functions inside the class.

Methods: These are the functions defined within the class. The behavior of an object is defined by these methods. The methods are used to access the data members.

For e.g. getName( ), displayInfo( ) etc. are methods which are defined inside the class to access or modify the data members. These methods determine what type of functionality a class has, how it modifies its data, and its overall behavior.

Class declaration/definition: In C++, a class is defined by using keyword 'class' followed by the name of the class. The body of the class is defined inside the curly brackets and is terminated by a semi-colon (;) at the end.

Syntax:

```
class Class-name {
        // data members
        // member functions
 } ;
```

Example:

```
class Student {
      string name;  //data member
     int roll;       // data member
      void study( );  //member function
} ;
```

Declaring/Creating objects:

- The objects are created from classes.
- These are declared in a similar way as variables are declared.
- It is declared by using the class name followed by the name of the object.

Syntax: Class-name object-name;

Example: Student s1;

Accessing members of class: The members of the class can be accessed using the dot (.) operator with the object.

Syntax: object-name.dataMember-name;
Or, object-name.memberFunction-name( );

Example: s1.roll;
s1.study( );

Here, 's1' is the name of the object, which is followed by the dot(.) operator. It is accessing the data member with the name 'roll', and member function with the name 'study'.

[Example program- 13]

Member functions defined outside the class: Member functions of a class can be defined outside the class. The function body remains the same like defining the function inside the class; however, the function header is different. If the member functions are defined outside the class, its function declaration (function prototype) should be provided inside the class definition. The member function is declared inside the class like a normal function. This declaration informs the compiler that the function is a member of the class, but has been defined outside the class.

The definition of the member function outside the class contains the return type, followed by the class name and the scope resolution operator (: :), which is followed by the function name. The scope resolution operator informs the compiler what class the member belongs to.

The *syntax* for defining a member function outside the class is as follows:

```
class Class-name {
        //data members
        return-type function-name (parameters);  //member function declaration
 };

return-type Class-name :: function-name(parameters) {
      // body of the member function
 }
```

Example:

```
class Member{
 public:
 void display(); //member function declaration
};
void Member::display(){
 cout<<"Member function defined outside the class. "<<endl;
}
```

Example program -13:

```
class Student{  //class name
      string name;    //data member
      int rollNos;    //data member
      public:
      void getData(){     //member function
         cout<<"Enter name:"<<endl;
         cin>>name;
         cout<<"Enter roll nos:"<<endl;
         cin>>rollNos;
      }
      void displayInfo(){    //member function
       cout<<"Name: "<<name<<endl;
       cout<<"Roll nos: "<<rollNos<<endl;
      }
};
int main()
{
   Student s1;  //Creating object s1;
   s1.getData();   //calling member function
   s1.displayInfo(); //calling member function
   return 0;
}
```

Output: Enter name:

```
          Aakash (entered by user)
          Enter roll nos:
```

```
34 (entered by user)
Name: Aakash
Roll nos: 34
```

Access specifiers/modifiers/modes: These are the specifiers that are used to define the accessibility of the members of the class. These are used to implement one important aspect of OOP known as data hiding. These specifiers set some restrictions on the members of the class so that these members are not directly accessed by the outside functions and classes. There are three types of access specifiers. These are:

1. Private
2. Protected
3. Public

1. Private: This is the specifier/modifier that makes the members of the class private and doesn't allow them to be accessible from outside the class. These members can only be accessed from within the class. The 'private' keyword is used to create private members (data and functions). If any mode is not specified for the members of the class they become by default 'private'.

The members declared as private are not allowed to be accessed directly by any object or function outside the class. However, there is a special function and a class known as friend function and a friend class that can access private members of a class.

Example of private member declaration:

```
class Student{
        private:    //below members become private
          name;
          rollNos;
    };
```

2. Public: This is the specifier/modifier that makes the members of the class public and allows them to be accessible from outside the class. The 'public' keyword is used to create public members (data and functions).

The members declared as public are available to everyone and so they can be accessed by other classes and functions. That means, these members are accessible from any part of the program.

Example of public member declaration:

```
class Student{
        public:    // below members become public
            name;
            rollNos;
};
```

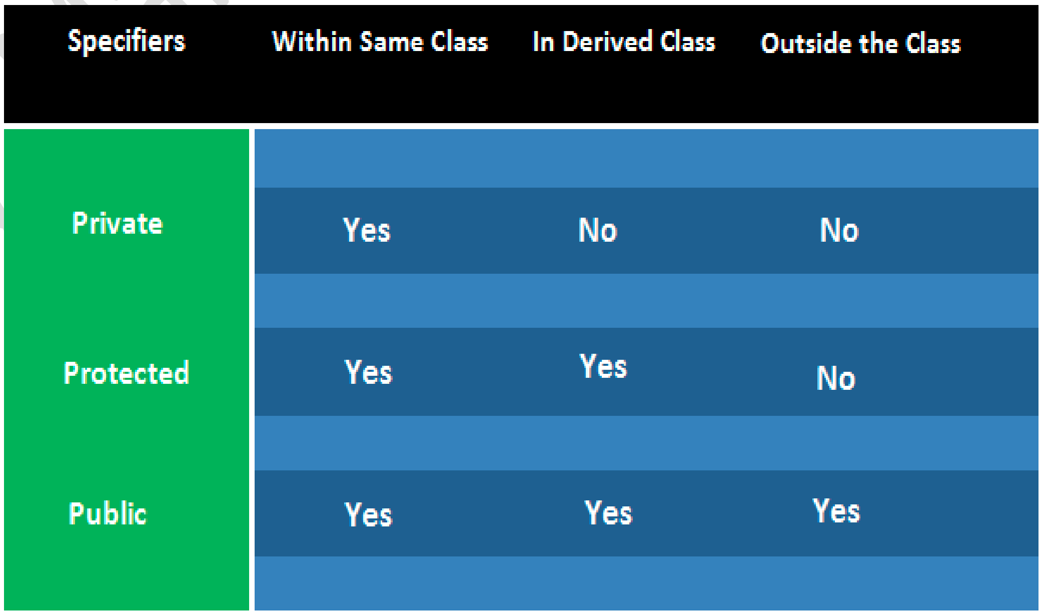
3. Protected: This is the specifier/modifier that makes the members of the class protected and doesn't allow them to be accessible from outside the class, except the derived classes. That means, these members can only be accessed from within the class and from derived classes. The 'protected' keyword is used to create protected members (data and functions).

The members declared as protected are not allowed to be accessed directly by any object or function outside the class, except the derived class. However, friend function and friend classes can also access protected members. The protected members are similar to private members. The only difference is that the protected members of a class can be accessed from its derived classes whereas the private members cannot be accessed.

Example of protected member declaration:

```
class Student{
        protected:  //below members become protected
           name;
           rollNos;
};
```

Summary Table:



Specifiers	Within Same Class	In Derived Class	Outside the Class
Private	Yes	No	No
Protected	Yes	Yes	No
Public	Yes	Yes	Yes

[Example program- 14]

Example program- 14:

```
class Student{  //class name

     //If the modifier is not specified,
     //the members are by default private
     private:     // can be accessed by only
              //the member functions of Student class
     string name;
     int rollNos;

     protected:    //can be accessed by member functions
                  //of student class and derived classes
     void getName(){     //member function
        cout<<"Enter name:"<<endl;
        cin>>name;
     }

     public://can be accessed from anywhere
     void getRollNos(){    //member function
        cout<<"Enter RollNos:"<<endl;
        cin>>rollNos;
     }
     void displayInfo(){   //member function
      cout<<"Name: "<<name<<endl;
      cout<<"Roll nos: "<<rollNos<<endl;
     }
};
void main()
{
   Student s1;  //Creating object s1;
   s1.name;   //INVALID because the name is private
   s1.rollNos;  //INVALID because the rollNos is private
   s1.getRollNos(); //VALID call because getRollNos is public
   s1.getName();    //INVALID because getName is protected.
                    //However, it can be accessed from derived classes.
   s1.displayInfo(); //VALID call because displayInfo is public
}
```

Example program -15:

Q. Write a program that creates a class Teacher with data members, tid and subject, and member functions for reading and displaying data members. Both the member functions should be defined outside the class. Create two objects of the class Teacher and use them to display the information.

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

class Teacher{
  int tid;           //data member
  string subject;     //data member

  public:
  void getInfo();      //member function declared inside the class
  void displayInfo(); //member function declared inside the class
};

void Teacher::getInfo(){ //member function defined outside the class
  cout<<"Enter Teacher id: "<<endl;
  cin>>tid;
  cout<<"Enter name of subject: "<<endl;
  cin>>subject;
}
void Teacher::displayInfo(){ //member function defined outside the class
  cout<<"Teacher id: "<<tid<<endl;
  cout<<"Name of subject: "<<subject<<endl;
}

int main()
{
 Teacher t1, t2;   //creating two objects of class Teacher

 t1.getInfo();  //calls 'getInfo()' to initialize data members of t1 object
 t2.getInfo();  //calls 'getInfo()' to initialize data members of t2 object

 t1.displayInfo();  //calls displayInfo() to show the information of t1 object
 t2.displayInfo();  //calls displayInfo() to show the information of t2 object

 return 0;
}
```

Output: Enter Teacher id:

```
10 (entered by user)
Enter name of subject:
C++
Enter Teacher id:
12
Enter name of subject:
Maths
Teacher id: 10
Name of subject: C++
Teacher id: 12
Name of subject: Maths
```

Constructor: A constructor is a special type of member function of a class that is automatically called when an object is created.

Constructor has the same name as that of the class. It does not have any return type. It is placed in public section of class. A default constructor for an object is generated automatically if we do not specify a constructor.

Use/Need of constructor: A constructor is used to initialize the objects of a class. That mean, it is used to initialize the data members of an object as soon as it is created. It is used to provide initial (default) values to the data members of the object. This saves us from the garbage values assigned to data members, during initializing.

Example:
```
class Student{
        public:
            Student( )      //creating a constructor
            {
                //code
            }
         };
```

Types of constructors: An object of a class can be initialized by using following types of constructors:

1. Default constructor
2. Copy constructor
3. Parameterized constructor

1. Default constructor: It is a type of constructor that doesn't require any arguments when called. It has no parameters.

If we do not define any constructor in our class (explicitly), then the compiler will automatically create a default constructor (implicitly) with an empty code and no parameters. In that case, the default values of the data members would be 0.

An example of a default constructor is shown below. *[Example program – 16]*

Example program -16:

```
using namespace std;

class A{
   int x;
   public:
   A(){      //default constructor
       x=0;
   }
};

int main()
{
   A obj;    //object created // default constructor is called
              // when the object is created
   return 0;
}
```

Explanation: When an object 'obj' is created, then the default constructor is called automatically. The constructor then initializes the data member 'x' of object 'obj' to be 0.

2. Parameterized constructor: A constructor with parameters is known as parameterized constructor. In other words, it is a type of constructor that requires arguments when called.

There is no default parameterized constructor and we have to explicitly define it. Generally, it is the preferred method to initialize the objects.

Uses/benefits of parameterized constructor:

- If we want to initialize the data members of various objects with different values, then parameterized constructor is used.
- It can be used to overload constructors.

An example of a parameterized constructor is shown below. *[Example program – 17]*

3. Copy constructor: It is a type of constructor that initializes an object using another object of the same class. It is used to copy the data of one object to the other. That means, the value of the data members of one object is copied to the data members of another object.

If we don't define a copy constructor inside a class (explicitly), then the compiler will automatically call a default copy constructor (implicitly). Copy constructor uses a reference variable as an argument.

A general function prototype of the copy constructor:

```
class Student
{
   public:
      Student ( Student &obj ) {   //copy constructor
        ......... //code statements
      }
};
```

A copy constructor is called when an object is passed as an argument to another object. It can be called by following two ways:

a) Class_name   object2 = object1;
b) Class_name   object2(object1);

Where,
object2= target object where the data of object1 is copied
object1= source object whose data are copied to another object 'object2'

An example of a copy constructor is shown below: *[Example program -18]*

Example program – 17:

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

class A{
   int x;
   public:
   A(int a){     //Parameterized constructor
     x=a;
   }
};

int main()
{
   //object created
   A obj(5);     // parameterized constructor is called

   return 0;
}
```

Explanation: When an object 'obj' is created, and a value '5' is passed to it, then the parameterized constructor is automatically called. The constructor then initializes the data member 'x' of object 'obj' to be 5.

Example program -18:

```
class A{
   int x;
   public:
   A(){     //default constructor
     x=0;
   }

   A(A &obj){    //copy constructor
    x=obj.x;
   }
};

int main()
{
   //Creating one object
   A obj1;     //default constructor is called

   //Creating second object
   A obj2(obj1); //Copy constructor is called

   //Creating third object
   A obj3=obj1;  //Copy constructor is called

   return 0;
}
```

Explanation: When an object 'obj1' is created, a default constructor is called and the value of x of object obj1 is initialized to 0. When an object 'obj2' is created and a previous object 'obj1' is passed to it as an argument, then the copy constructor is called. Similarly when an object 'obj3' is created and previous 'obj1' is assigned to it, then the copy constructor is called. Both the last two methods will do the same thing, so we can choose one of these two methods while calling the copy constructor. The copy constructor then copies the value of the data member 'x' of object 'obj1' to the data member 'x' of the next objects 'obj2' and 'obj3'.

Things to understand: Whenever we define one or more non-default constructors (with parameters) for a class, a default constructor (without parameters) should also be explicitly defined as the compiler will not provide a default constructor in this case.

Constructor overloading: It is a method in which more than one constructor are defined in a class having the same name (name of the class) but with different number or types of arguments. It is a process in which the constructors are overloaded in a similar way as functions. The corresponding constructor is called based on the number and type of the arguments passed.

An example of a constructor overloading is shown below. *[Example program – 19]*

Example program- 19:

```
class Utech{
 int x;   //data member
 int y;  //data member

 public:
 Utech(){   //constructor with no arguments
   x=0;
   y=0;
 }

 Utech(int a){   //constructor with single argument
   x=a;
   y=0;
 }

 Utech(int a, int b){ //constructor with two arguments
   x=a;
   y=b;
 }
 void displayInfo(){
  cout<<"Value of x: "<<x<<endl;
  cout<<"Value of y: "<<y<<endl;
 }
};

int main()
{
 /**Constructor Overloading **/
 Utech u1;
 Utech u2(5);
 Utech u3(2,5);

 u1.displayInfo();
 u2.displayInfo();
 u3.displayInfo();

 return 0;
}
```

```
Output: Value of x: 0
        Value of y: 0
        Value of x: 5
        Value of y: 0
        Value of x: 2
        Value of y: 5
```

Example program- 20:

Q. Write a program to calculate the area and volume of a room using OOP concept.

- Use Data members: length, breadth and height, which should be private.
- Initialize the data members using constructor.
- Define member functions for calculating area and volume of a room and for displaying the result.
- Create an object of the class and use it to display the information.

(This was done in our lab-sessions; submit it as a part of lab-assignment).

Destructor: Destructor is a special type of member function of a class that is invoked automatically whenever an object is going to be destroyed. It is the last function that is called before an object goes out of scope such as when the function ends, the program ends, or a pointer variable is deleted etc.

- Destructor has no return type and also does not have any arguments.
- There is only one destructor in a class.
- It is declared in the public section of the class as similar to constructor.
- It has the same name as the class preceded by a symbol (~).

Use/Need of destructor:

- Destructors are used to release dynamically allocated memory in a class
- They are used to free memory, release resources and to perform other clean up.

Example:
```
class Student{
    public:
        ~Student( )      // destructor
        {
            //code
        }
    };
```
An example of a destructor is shown below. *[Example program -21]*

Example program- 21:

```
class Utech{
 int x;   //data member

 public:
   Utech(){    //constructor
     x=0;
     cout<<"Constructor called"<<endl;
   }

   ~Utech(){    //destructor
     cout<<"Destructor called"<<endl;
   }
};

int main()
{
   Utech u1;
   return 0;
}
```

Output:
```
Constructor called
Destructor called
```

Explanation: Whenever an object is created, a constructor is called automatically. And the destructor is called just before the object is going out of scope. In the above program, the object 'u1' is going out of scope when the main( ) function ends. So, just before the function ends, the destructor is called.

Static Data member: A data member that has a single copy and is shared by all the instances (objects) of the class is known as static data member. That means, only one copy of it exists for the whole class. So, it belongs to the whole class and not to the individual objects. Any objects can use the same copy of it. It is also known as class data member or class member variable.

Static data member is declared inside the class body. It must be explicitly defined outside the class using the scope resolution operator.

Example:
```
class Student{
           int x;
             static int y;    //static member variable declaration
        };
        int Student::y;   //static member variable definition outside the class
```

An example of a use of a static data member is given below. *[Example program-22]*

Example program- 22:

```
class Model{
  int x;
  static int y; //static member variable declaration
  public:
  void getSomething(int a){
   y=a;
  }
  void displaySomething(){
   cout<<"The value of y: "<<y<<endl;
  }
};
int Model::y=0; //static member variable definition
int main(){
  Model paul, samikshya;
  paul.getSomething(5);
  samikshya.displaySomething();
  return 0;
}
```

Output: The value of y: 5

Explanation: In the above program, data member 'y' is declared as a static member. It is defined outside the class and is initialized as 0. Two objects 'paul' and 'samikshya' of the class Model are created from inside main( ) function. A member function getSomething(5) is called using paul object that modifies the value of y to be 5. Now, when the displaySomething( ) is called by using samikshya object, the value of y is displayed to be 5. This is because, y is static and both the objects share the same copy of it.