Module 3

Structures in C++

In C++, a structure can have both variables and function as members. It can also declare some of the member as private so that they cannot be accessed directly by the external functions.

The keyword 'struct' can be omitted in declaration of structure variables.

By default, structure members are public.

```
struct student
{
  //body
};

student nishant, suhani, jiya;
```

↳ declaration in C++.

vs

```
struct student nishant, suhani, jiya;
```

↳ declaration in C

Module 3:

Classes and Objects

Classes are a user-defined data-type. The entire set of data and code of an object can be made with the help of class. Class is a way to bind data and its associated functions Together. It can also allows us to hide data and function.

Class can be created using keyword 'class'.

Syntax:

```
class class-name
{
    private:
        variable declaration;
        function declaration;

    public:
        variable declaration;
        function declaration;

};
```

These functions and variables are collectively called class member i.e. declared inside the class. Variable of class are called as data members and functions of class are called as member function or methods.

They are grouped under two sections - private and public also known as visibility labels or access specifiers. The data hiding is the key feature of OOPs. The use of private is optional because, by default, members of class are private. Public members can be accessed from outside the class. If both the labels are missing, then all members are private. Such class is completely hidden and does not serve any purpose.

Only the member function can have access to the private data and functions.
The binding of data and functions together into a single class-type variable referred to as encapsulation.

Objects

Object are the instance of a class. Once a class is defined, we can create any number of objects belonging to that class. Each object is associated with the data of type ~~unit~~ class with which they are created.

```
class class-name
{
  ---
} obj-name;
```

or.

```
classname obj-name;
```

Memory is allocated only when an object is created. Class provides a template only, doesnot create any memory space for object.

Accessing a class member.

The format for calling a member a member function.

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

Object-name. Public data member;

Example of inside class definition

```
class snj
{
    int a, b;
    public:
        int c = 0;
        void sum(int x, int y)
        {
            cout << "Sum is " << x+y << endl;
        }
};

int main()
{
    snj obj;
    obj.c = 40;
    obj.sum(40,50);
    return 0;
}.
```

Defining Member functions

Member functions can be defined in two places:

- Outside the class definition
- Inside the class definition. (above example)
  → replacing function declaration with definition inside the class.

1) Outside the class definition.

```
return-type class-name :: func-name(argument) declaration.
{
    function body
}.
```

The membership label 'class-name;' :: tells the compiler that the function function-name belongs to the class class-name.
Meaning that scope of the function is restricted to class-name specified in header line.

```
void item :: get(int a, int b)
{
    number = a;
    cost = b;
}
```

Since, there is no return type, it is void here.

- Several different class can use the same function name. The 'membership label' will resolve their scope.
- only Member function can access the private data of class.
- A member function can call other member function directly, without using dot operator.
- ~~Inside the~~

Making an outside function inline.

It is acheived by just using the qualifier inline in the header line of function definition.

```
inline void item :: get(int a, int b)
{
    number = a;
    cost = b;
}.
```

Nesting of Member functions

A member function can be called by using its name inside another member function of same class. This is known as nesting.

Array can be used as member variables in a class.

```
const int size = 50;
class array
{
    int a[size];
    public:
      void setval(void);
      void display(void);
};
```

Memory Allocation for objects

The member functions are created and placed in the memory space only once when they are defined as a part of Class. No separate space is allocated for member functions when the objects are created.

For member variables, space is allocated separately for each object.

Static Member functions

Like static member variable we can also have static member functions.

A member function that is declared static has the following properties:

- A static function can have access to only other static members declared in same class.
- A static member function can be called using class name instead of its objects.

classname :: function-name;

Array of objects

We can also have arrays of variables that are of type class just like struct. Such variables are called array of objects.

```
class employee
{
   - - -
   - - -
};

employee manager[3]; //array of manager.
```

Imp

Friendly / Friend Function

We know private members cannot be accessed from outside the class.

C++ allows the common function to be made friendly with both the classes, thereby allowing the function to have access to the private data of these classes. Such a function need not be a member of any of these classes.

To make an outside function "friendly" to a class, we have to simply declare this function as a friend of the class as shown

```
Class ABC
{
    ...
    ...
    public:
    ...
    ...
        friend void xyz(void); //declaration.
};
```

The function declaration should be preceded by the keyword friend. The function is defined elsewhere in the program like a normal C++ function.

The functions definition doesnot use either the ~~keybo~~ keyword friend or the scope operator ::.

The Functions that are declared with the keyword friend are known as friend functions.

A function can be declared as a friend in any number of classes. A friend function, although not a member function, has full access rights to the private member of class.

Characteristics of friend function

- It is not in the scope of the class to which it has declared as friend, so it cannot be called using the object of that class.
- It can be invoked like normal function without any help of any object.
- Unlike member functions, it cannot access the member names directly and has to use an object name and dot membership operator with each member name.
- It can be declared either in public or private part of class without affecting its meaning.
- Usually it has the objects as argument.

Returning Objects :-

A function cannot only receive objects as argument but also can return them.

Const Member function

If a member function does not alter any data in the class, then we may declare it as a const member function as follows:

```
Void mul(int, int) Const;
```

The qualifier const is appended to the function prototype (in both declaration and definition). The compiler will generate an error message if such function try to alter the data values.

Local classes

Classes can be defined and used inside a function or a block. Such classes are called local classes.

```
void test(int a)  // function
{
    class student  // local class
    {
        ---
        ---
    };
    ---
    ---
    student s1(a); // create student object
}
```

Local class can use global variables and static variables declared inside the function but cannot use automatic local variable. The global variables should be used with scope operator (::)

Enclosing ~~the~~ function cannot ~~access~~ access the private members of local class. We can achieve this by declaring the enclosing function as a friend.

Constructors

A Constructor is a 'special' member function whose task is to initialize the object of its class. It is special because it has same name 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 value of data members of class

```
Class integer
{
    int m,n;
    public:
        integer (void);   // constructor declared
        ....
        ....
};

integer :: integer (void)  // defined
{
    m=40;
    n=50;
}
```

The declaration –
integer int1; // object.
↓
not only creates the object int1 of type integer but also initializes its data member.

A constructor that accepts no parameter is called the default constructor

The default constructor for class A is A::A(). If no such constructor is defined, then the compiler supplies a default constructor.

A a;

invokes the default constructor of compiler to create the object a.

The constructor functions have some special characteristics

- declared in public section
- automatically invoked when the objects are create
- do not have any return type, not even void.
- cannot be inherited, through a derived class
- can't be virtual
- can't refer to their addresses.

Parameterized Constructors

The constructors that can take arguments are called parameterized constructors.

Example of inline constructor function

```
class integer
{
    int m=0, n=0;
  public:
    integer(int x, int y)
    {
        m=x;
        n=y;
    }
};
```

integer int1; X → may not work.

We need to pass initial values as arguments to the constructor function when an object is declared. This can be done in two ways.

- by calling the constructor explicitly
  integer int1 = integer(0,100);

- by calling it implicitly.

integer int(40,50);

A constructor can accept a reference to its own class as a parameter. The statement

```
class A
{
    ....
    public:
        A(A&);
};
```

is valid. In such cases, the constructor is called the copy constructor.

Copy Constructor

The parameters of a constructor can be of any type except that of the class to which it belongs. for Eg;

```
class A
{
    ...
    public:
        A(A);
};
```

is illegal.

However, a constructor can accept a reference to its own class as a parameter.

```
Class A
{
    ...
    public:
        A(A&);
};
```

is valid. In such cases, the constructor is called the copy constructor.

A copy constructor is used to declare and initialize an object from another Object.

```
Integer I2(I1);
```

↓

This would define the object I2 and at same time initialize it to value of I1.

```
integer I2 = I1;
```

The process of initializating ~~a~~ through a copy constructor is known as copy initialization.

We use copy constructor ~~as one~~ in overloaded constructor.

Dynamic Constructors

The constructor can also be used to allocate memory while creating objects. This will enable the System to allocate the right amount of memory for each object when the objects are not of the same size, thus resulting in the saving of memory.

Allocation of memory to objects at the time of their construction is known as dynamic construction of objects.

The memory is allocated with the help of 'new' operator.

Const objects

```
const matrix x(m,n);
      ↓        ↓
   Class      Object
   name
```

Any attempt to modify the values of m, n will generate compile time error.

Destructors

It is used to destroy the objects that have been created by a Constructor. It is also a member function whose name is same as class name but is preceded by a tilde.

```
~integer() { ... }.
```

A destructor never takes any arguments nor does it return any value.

It is a good practice to declare destructors in a program since it releases memory space for future use.

Whenever 'new' is used to allocate memory in constructor, we use 'delete' to free that memory.

Accessor and Mutator Function

In C++, accessor and mutator functions are like special doors guarding the data within your class. They provide controlled access to your class's private member variables, ensuring data integrity and proper object behaviour.

Accessor Function.

- Also known as a getter function.
- Allows us to read the value of a private member variable from outside the class.
- typically has no arguments and returns the value of the member variable.
- provides a safe and controlled way to access data.

Mutator Function

- Also known as a setter function
- Allows you to modify the value of a private member variable from outside the class.
- Types takes an argument corresponding to new value
- Enables controlled changes to data.

Benefits

- Data encapsulation
- Data validation
- Centralized control
- Improved code maintainability.