UNIT 2

- **Object Oriented Programming (OOP):**

Object-oriented programming – As the name suggests uses **objects** in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

There are some basic concepts that act as the building blocks of OOPs.

- Objects
- Classes
- Encapsulation
- Abstraction
- Inheritance
- Polymorphism
- Dynamic Binding
- Message Passing

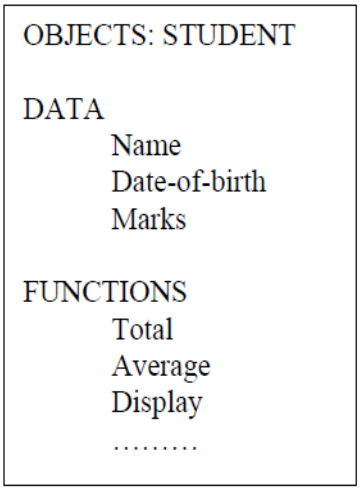
- **Objects :**

Objects are the basic run-time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program must handle.

The fundamental idea behind the object-oriented approach is to combine both data and function into a single unit and these units are called objects.

The term objects mean a combination of data and program that represent some real-world entity. For example: consider an example named Amit; Amit is 25 years old and his salary is 2500. The Amit may be represented in a computer program as an object. The data part of the object would be (name: Amit, age: 25, salary: 2500)

The program part of the object may be a collection of programs (retrieve of data, change age, change of salary). In general, even any user-defined type-such as an employee may be used. In the Amit object the name, age and salary are called attributes of the object.



- **Classes :**

A group of objects that share common properties for the data part and some program part are collectively called as a class.

We just mentioned that objects contain data, and code to manipulate that data. The entire set of data and code of an object can be made a user-defined data type with the help of class. In fact, objects are variables of the type class. Once a class has been defined, we can create any number of objects belonging to that class. Each object is associated with the data of type class with which they are created. A class is thus a collection of objects similar types. For examples, Mango, Apple and orange members of class fruit. Classes are user-defined that types and behave like the built-in types of a programming language. The syntax used to create an object is not different then the syntax used to create an integer object in C. If fruit has been defines as a class, then the statement

Fruit Mango;
Will create an object **mango** belonging to the class **fruit.**

- **Encapsulation :**

The wrapping up of data and function into a single unit (called class) is known as encapsulation. The data is not accessible to the outside world and only those functions which are wrapped in the class can access it. These functions provide the interface between the objects data and the program.

Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section etc. The finance section handles all the financial transactions and keeps records of all the data related to finance. Similarly, the sales section

handles all the sales-related activities and keeps records of all the sales. Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name “sales section”.

- **Abstraction :**

Data abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, wait, and cost, and function operate on these attributes. They encapsulate all the essential properties of the object that are to be created.
The attributes are some time called **data members** because they hold information. The functions that operate on these data are sometimes called methods or **member function**.

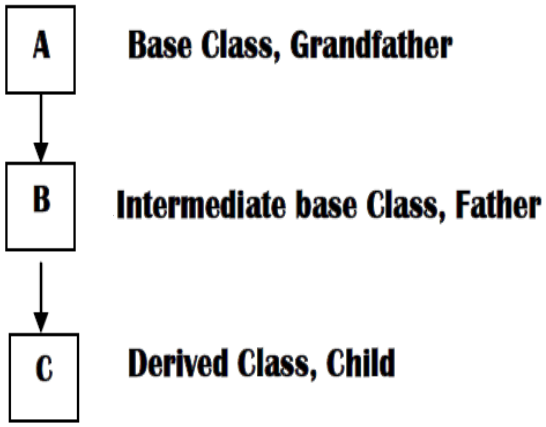
Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

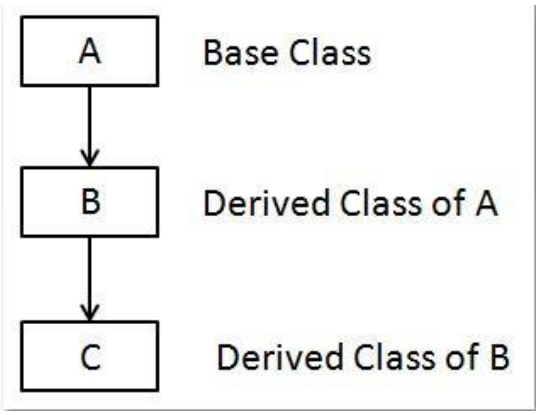
- **Inheritance :**

Inheritance is the process by which objects of one class acquired the properties of objects of another classes. It supports the concept of hierarchical classification.

Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

It’s pretty clear with the diagram that in Multilevel inheritance there is a concept of grandparent class. If we take the example of below diagram then class C inherits class B and class B inherits class A which means B is a parent class of C and A is a parent class of B. So in this case class C is implicitly inheriting the properties and method of class A along with B that’s what is called inheritance.





- **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.

A person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So, the same person posses different behaviour in different situations. This is called polymorphism.

A language feature that allows a function or operator to be given more than one definition. The types of the arguments with which the function or operator is called determines which definition will be used.

Overloading may be operator overloading or function overloading.

It is able to express the operation of addition by a single operator say '+'. When this is possible you use the expression x + y to denote the sum of x and y, for many different types of x and y; integers, float and complex no. You can even define the + operation for two strings to mean the concatenation of the strings.

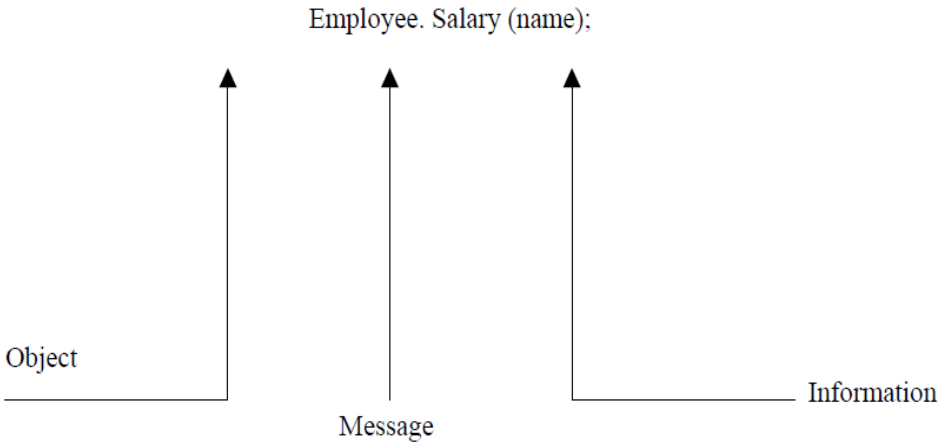
- **Dynamic Binding :**

Binding refers to the linking of a procedure call to the code to the executed in response to the call. Dynamic binding means the code associated with a given procedure call is not known until the time of the call at run-time. It is associated with a polymorphic reference depends upon the dynamic type of that reference.

- **Message Passing :**

Objects communicate with one another by sending and receiving information to each other. A message for an object is a request for execution of a procedure and therefore will invoke a

function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function and the information to be sent.



- **Object Oriented Language :**

Object-oriented programming is not the right of any particular languages. Like structured programming, OOP concepts can be implemented using languages such as C and Pascal. However, programming becomes clumsy and may generate confusion when the programs grow large. A language that is specially id designed to support the OOP concepts makes it easier to implement them.

The languages should support several of the OOP concepts to claim that they are object-oriented. Depending upon the features they support, they can be classified into the following two categories:

1) Object-based programming languages, and
2) Object-oriented programming languages.

Object-based programming is the style of programming that primarily supports encapsulation and object identity. Major feature that are required for object based programming are:

- Data encapsulation
- Data hiding and access mechanisms
- Automatic initialization and clear-up of objects
- Operator overloading

Languages that support programming with objects are said to the objects-based programming languages. They do not support inheritance and dynamic binding. Ada is a typical object-based programming language.

Object-oriented programming language incorporates all of object-based programming features along with two additional features, namely, inheritance and dynamic binding. Object-oriented programming can therefore be characterized by the following statements:

Object-based features + inheritance + dynamic binding

- ## Classes in C++ :

A class is a template or a blueprint that binds the properties and functions of an entity. You can put all the entities or objects having similar attributes under a single roof, known as a class. Classes further implement the core concepts like encapsulation, data hiding, and abstraction. In C++, a class acts as a data type that can have multiple objects or instances of the class type.

Consider an example of a railway station having several trains. A train has some characteristics like train_no, destination, train_type, arrival_time, and departure_time. And its associated operations are arrival and departure. You can define a class can for a train as follows:

```
class train
{
   // characteristics
   int train_no;
   char destination;
   char train_type;
   int arrival_time;
   int departure_time;
   // functions
   int arrival(delayed_time)
   {
      arrival_time += delayed_time;
      return arr_time;
   }
   int departure(delayed_time)
   {
      departure_time += delayed_time;
      return departure_time;
   }
}
```

The above class declaration contains properties of the class, train_no, destination, train_type, arrival_time, and departure_time as the data members. You can define the operations, arrival, and departure as the member functions of the class.

- **Syntax to Declare a Class in C++:**

```
class class_name
{
   // class definition
   access_specifier:        // public, protected, or private
   data_member1;   // data members
   data_member2;
   func1(){}       // member functions
   func2(){}
};
```

- **Description of the Syntax:**

  - **class:** This is the keyword used to declare a class that is followed by the name of the class.

  - **class_name:** This is the name of the class which is specified along with the keyword class.

  - **access_specifier:** It provides the access specifier before declaring the members of the class. These specifiers control the access of the class members within the class. The specifiers can be public, protected, and private.

  - **data_member:** These are the variables of the class to store the data values.

  - **member_function:** These are the functions declared inside the class.

• Objects in C++?

Objects in C++ are analogous to real-world entities. There are objects everywhere around you, like trees, birds, chairs, tables, dogs, cars, and the list can go on. There are some properties and functions associated with these objects. Similarly, C++ also includes the concept of objects. When you define a class, it contains all the information about the objects of the class type. Once it defines the class, it can create similar objects sharing that information, with the class name being the type specifier.

Consider the example of a railway station discussed in the previous section. After defining the class train, you can create similar objects for this class. For example, train_A and train_B. You can create the objects for the class defined above in the following way:

```
train train_A, train_B;
```

The syntax to create objects in C++:

class_name object_name;

The object object_name once created, can be used to access the data members and member functions of the class class_name using the dot operator in the following way:

```
obj.data_member = 10;   // accessing data member

obj.func();             // accessing member function
```

- **Significance of Class and Object in C++ :**

The concept of class and object in C++ makes it possible to incorporate real-life analogy to programming. It gives the data the highest priority using classes. The following features prove the significance of class and object in C++:

- **Data hiding:** A class prevents the access of the data from the outside world using access specifiers. It can set permissions to restrict the access of the data.

- **Code Reusability:** You can reduce code redundancy by using reusable code with the help of inheritance. Other classes can inherit similar functionalities and properties, which makes the code clean.

- **Data binding:** The data elements and their associated functionalities are bound under one hood, providing more security to the data.

- **Flexibility:** You can use a class in many forms using the concept of polymorphism. This makes a program flexible and increases its extensibility.

- **Member Functions in Classes :**

The member functions are like the conventional functions. It defines these methods inside a class and has direct access to all the data members of its class. When you define a member function, it

only creates and shares one instance of that function by all the instances of that class. The following syntax can be used to declare a member function inside a class:

```
class class_name
{
access_specifier:
   return_type member_function_name(data_type arg);
};
```

It specifies the access specifier before declaring a member function. It also specifies the return type and data type in the same way in which it declares a usual function.

- **Method Definition Outside and Inside of a Class :**

The following two ways can define a method or member functions of a class:

1. Inside class definition
2. Outside class definition

The function body remains the same in both approaches to define a member function. The difference lies only in the function's header. Now, have a deeper understanding of these approaches.

- **Inside Class Definition :**

This approach of defining a member function is generally preferred for small functions. It defines a member function inside a class in the same familiar way as it defines a conventional function. It specifies the return type of the function, followed by the function name, and it provides arguments in the function header. Then it provides the function body to define the complete function. The member functions that are defined inside a class are automatically inline. The following example illustrates defining a member function inside a class.

```
#include <iostream>
using namespace std;
// define a class
class my_class
{
public:
   // inside class definition of the function
   void sum(int num1, int num2)              // function header
```

```
    {
        cout << "The sum of the numbers is: ";  // function body
        cout << (num1 + num2) << "\n\n";
    }
};
int main()
{
    // create an object of the class
    my_class obj;
    // call the member function
    obj.sum(5, 10);
    return 0;
}
```

In the above example, you define the function sum() inside the class my_class. The function is automatically an inline function. Whenever you call this function by an object of its class, it inserts the code of the function's body there, which reduces the execution time of the program. Here, the statement obj.sum(5, 10) is replaced by the body of the function sum().

- **Outside Class Definition :**

Here, you use the scope resolution operator (::) to define a member function outside its class. Even though you define the member function outside the class, it still needs to be declared first inside the class. This approach of defining a member function of a class is the most preferred. The following syntax is used to define a member function outside its class:

```
void class_name :: function_name(arguments)
{
// function body
}
```

The class_name is the name of the class in which the function has been declared. The function_name is the name of the member function. It uses the scope resolution operator here to restrict the scope of the function to its class.

The following example illustrates how to define a member function outside a class.

```
#include <iostream>
using namespace std;
// define a class
```

```
class my_class
{
public:
   // declare the member function
   void sum(int num1, int num2);
};
// outside class definition of the function
void my_class::sum(int num1, int num2) // function header
{
   cout << "The sum of the numbers is: "; // function body
   cout << (num1 + num2) << "\n\n";
}
int main()
{
   // create an object of the class
   my_class obj;
   // call the member function
   obj.sum(5, 10);
   return 0;
}
```

In the above example, it first declares the function sum() inside the class my_class. It then defines the member function outside the class specifying the class name, followed by the scope resolution operator. The function body remains the same as in the inside class definition of the function.

- **Array of Objects :**

In C++, you can declare an array of any data type. This also includes the class type. The Array of objects is an array containing the elements of the class type. The definition of an array of objects is similar to the usual array definition with the data type replaced with the class name. The following syntax is used to define an array of objects:

```
class_name obj[20];
```

The array obj contains 20 elements of the type class_name. These elements are, obj[1], obj[2], obj[3], …… obj[20]. An array of objects is preferred when there is a requirement for numerous objects of the same class. Instead of creating obj1, obj2, obj3,....obj20, you can simply declare obj[20].

The following example illustrates the use of an array of objects.

```
#include <iostream>
using namespace std;
// define a class
class my_class
{
public:
   // declare the member function
   void printValue(int value)
   {
      cout << "The value is: " << value << "\n";
   }
};
int main()
{
   // create an object of the class
   my_class obj[5];
   // call printValue() function for all objects
   for (int i = 0; i < 5; i++)
   {
      obj[i].printValue(2 * i); // printValue() is called
                                // for particular object
   }
   cout << "\n\n";
   return 0;
}
```

In the above example, obj[5] is an array of objects, containing 5 elements. This is used to involve the printValue() function five times for a particular object to print a particular value.

- **Objects as Function Arguments :**

In C++, you can pass the objects of a class as arguments in the same way you pass a variable to a function as arguments. An object can be passed as an argument to a member function, a friend function, and a non-member function. The private and public members of the object are accessible to the member function and the friend function. However, the non-member function is only allowed to access the public members of the object.

You can pass the objects as arguments in the following ways:

1. Passing objects by value
2. Passing objects by reference

- **Passing Objects by Value :**

You can pass objects using the “call by value mechanism” to a function as arguments. Only the value of the object is passed to the function. It makes a separate copy of the object for the function. Any modifications done by the function to the object members do not modify the original object. The following example illustrates how to pass objects by value as arguments.

```
#include <iostream>
using namespace std;
// class to convert temperature
// from fahrenheit to celsius
class convTemperature
{
public:
   float f, c;
   void getTemp(float value)
   {
      f = value;
   }
   // function that accepts object as argument
   void findTemp(convTemperature obj)
   {
      obj.c = ((obj.f - 32) * 5) / 9;
      cout << "\nThe temperature in celsius is: " << obj.c << "\n";
   }
};
int main()
{
   // create objects of the class
   convTemperature obj1;
   obj1.getTemp(100);
   obj1.findTemp(obj1); // pass obj1 by value
   // garbage value of c will be printed
   // as the object was passed by value
   cout << "The value of c (object's copy) is: " << obj1.c << "\n";
```

```
    cout << "\n";
    return 0;
}
```

In the above example, the object obj1 of the class convTemperature is passed by value to the function findTemp(). The calculates the temperature in celsius and updates the value of the member variable "c". Since it passes the object by value, it only updates the function's version of "c" and the variable "c" of the object's copy remains unaffected. Therefore, the statement obj1.c prints a garbage value.

- **Passing Objects by Reference :**

The other way of passing an object as an argument is by using the "call by reference mechanism". In this case, the memory address of the object is passed as an argument. So the function directly accesses the object and its members. All the modifications by the function are now done to the original members of the object. The following example illustrates how to pass objects by reference as arguments.

```
#include <iostream>
using namespace std;
// class to update the value of the data member
class my_class
{
public:
    int x
    // function that accepts object as argument
    void updateValue(my_class &obj)
    {
        obj.x = 100;
        cout << "\nThe modified value of x is: " << obj.x << "\n";
    }
};
int main()
{
    // create an object of the class
    my_class obj;
    obj.updateValue(obj); // pass obj by reference
    // same value of x will be printed
    // as the object was passed by reference
    cout << "The value of x (object's copy) is: " << obj.x << "\n";
```

```
    cout << "\n";
    return 0;
}
```

In the above example, you pass the object obj of the class my_class by reference to the function updateValue(). The function gets direct access to the memory address of the object obj. When the function makes a modification to the member variable "x" using the statement obj.x = 100, the original object gets modified. Therefore, the following statement now prints the same modified value of x.

```
cout << obj.x
```

- **Difference Between Class and Structure :**

Although a class and a structure may seem identical, they differ from each other to a great extent. Now, you will understand how a class differs from a structure in C++.

- **Class:**

A class can be defined as a user-defined data type that contains some data members and member functions whose access is regulated by the specified access specifiers.

The following program illustrates the working of a class in C++:

```
#include <iostream>
using namespace std;
class my_class
{
    //members are private by default
    int member1;
    int member2;
public:
    //default constructor
    my_class()
    {
        member1 = 10;
        member2 = 20;
    }
    //function printing the values of private data members
    void print()
```

```
    {
        cout << "Value of member1 is: " << member1 << endl;
        cout << "Value of member2 is: " << member2 << endl;
    }
};
int main()
{
    my_class obj;
    obj.print();
}
```

- **Structure:**

A structure can be defined as a user-defined data type that is used to group elements of different data types together.

The following program illustrates the working of a structure in C++:

```
#include <iostream>
using namespace std;
struct my_structure
{
    //members are public by default
    int member1;
    int member2;
    //assigning values to the data members
    my_structure()
    {
        member1 = 10;
        member2 = 20;
    }
    //function printing the values of public data members
    void print()
    {
        cout << "Value of member1 is: " << member1 << endl;
        cout << "Value of member2 is: " << member2 << endl;
    }
};
int main()
{
```

```
    my_structure obj;
    obj.print();
}
```

The following table highlights the key differences between a class and a structure in C++.

Class	Structure
A class is defined using the class keyword.	A structure is defined using the structure keyword.
All the data members, as well as the member functions of a class, are private by default.	All the data members, as well as the member functions of a structure, are public by default.
Concepts of OOPs like data abstraction and data encapsulation are supported by classes.	Structures do not support any concept of OOPs.
A class can have a NULL value.	A structure cannot acquire a NULL value.
You cannot implement classes in the C language.	You can implement structures in C as well as C++ language.
Members of a class can be specified as public and protected too.	Members of a structure can only be public and cannot be specified as private or protected.

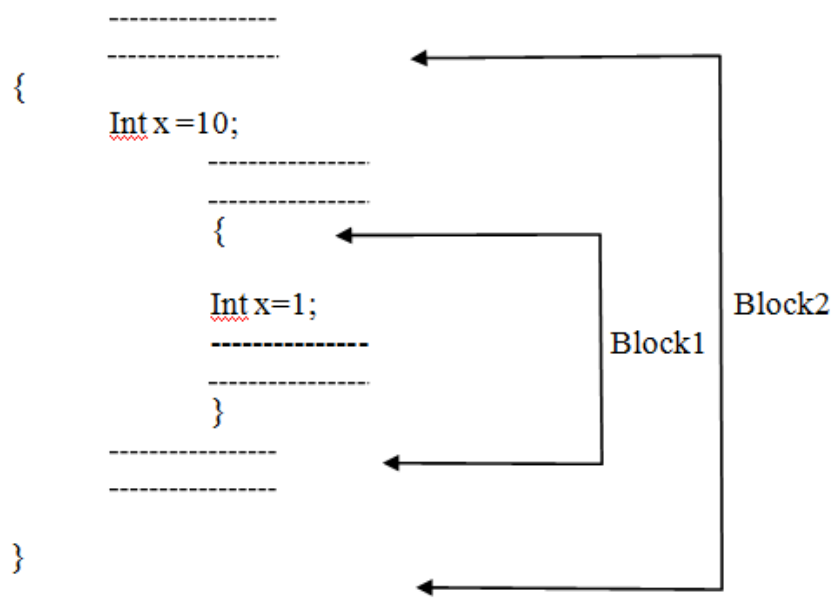
A Heap is used for allocating the memory of a class.	A Stack is used for allocating the memory of a structure.

- **Scope Resolution Operator :**

Like C, C++ is also a block-structured language. Block -structured language, Blocks and scopes can be used in constructing programs. We know same variables can be declared in different blocks because the variables declared in blocks are local to that function.

Blocks in C++ are often nested.

Example:



Block2 contained in block l .Note that declaration in an inner block hides a declaration of the same variable in an outer block and therefore each declaration of x causes it to refer to a different data object . With in the inner block the variable x will refer to the data object declared there in.

In C, the global version of a variable can't be accessed from within the inner block. C++ resolves this problem by introducing a new operator :: called the scope resolution operator .This can be used to uncover a hidden variable. Defining a member function within the class definition declares the function inline, even if you do not use the inline specifier. Defining a member function using scope resolution operator uses following declaration

return-type class-name::func-name(parameter- list) {
// body of function
}

Here the class-name is the name of the class to which the function belongs. The scope resolution operator (::) tells the compiler that the function func-name belongs to the class class-name. That is, the scope of the function is restricted to the class-name specified.

```
Class myclass {
int a;
public:
void set_a(intnum); //member function declaration
int get_a( ); //member function declaration
};
//member function definition outside class using scope resolution operator
void myclass :: set_a(intnum)
{
a=num;
}
int myclass::get_a( ) {
return a;
}
```

Another use of scope resolution operator is to allow access to the global version of a variable. In many situation, it happens that the name of global variable and the name of the local variable are same .In this while accessing the variable, the priority is given to the local variable by the compiler. If we want to access or use the global variable, then the scope resolution operator (::) is used. The syntax for accessing a global variable using scope resolution operator is as follows:-

:: Global-variable-name

- **Data members and Member functions in C++**

The variables which are declared in any class by using any fundamental data types (like int, char, float etc) or derived data type (like class, structure, pointer etc.) are known as Data Members. And the functions which are declared either in private section of public section are known as Member functions.

There are two types of data members/member functions in C++:

1. Private members
2. Public members

1) **Private members**

The members which are declared in private section of the class (using private access modifier) are known as private members. Private members can also be accessible within the same class in which they are declared.

2) **Public members**

The members which are declared in public section of the class (using public access modifier) are known as public members. Public members can access within the class and outside of the class by using the object name of the class in which they are declared.

Example:

```
class Test
{
        private:
                int a;
                float b;
                char *name;

                void getA() { a=10; }
                ...;

        public:
                int count;
                void getB() { b=20; }

                ...;
};
```

Here, a, b, and name are the private data members and count is a public data member. While, getA() is a private member function and getB() is public member functions.

- **C++ Access Specifiers – Private, Public and Protected :**

C++ access specifiers are used for determining or setting the boundary for the availability of class members (data members and member functions) beyond that class.

For example, the class members are grouped into sections, **private, protected** and **public.** These keywords are called **access specifiers** which define the accessibility or visibility level of class members.

Access specifier can be either private or protected or public. In general access specifiers are the access restriction imposed during the derivation of different subclasses from the base class.

- **private access specifier** - members cannot be accessed (or viewed) from outside the class.
- **protected access specifier**- members cannot be accessed from outside the class, however, they can be accessed in inherited classes.
- **public access specifier** - members are accessible from outside the class.

- **Class member visibility :**
  - A class member cannot have an arbitrary access.
  - There is a controlled access to each and every member in a class to provide security for the data members in OOP.
  - Information hiding prevents the function of a C++ program from accessing directly the internal representation of a class type. This means that data is hidden within a class, so that it cannot be accessed by functions outside.

There are three member access specifiers in C++ :

1. Public
2. Private
3. Protected

Class member visibility :

1. **Public access specifier –** A data member or member function declared as public can be accessed from anywhere within a program.
2. **Private access specifier –**
   - ✓ A data member or member function declared as private can be accessed from within the class. A private member of a class can be accessed only by the member functions of that class.
   - ✓ They can also be accessed by the friends of the class.
3) **Protected access specifier –**
   - ✓ It is a member access specifier and comes into picture when we deal with inheritance.
   - ✓ When an original class or a base class is being inherited to another class the derived class, there is a need of specifying the access control mechanism for the members of the derived class.
   - ✓ There may be any number of private, public and protected sections.
   - ✓ Usually we write private members first and then the public members.
   - ✓ Protected access specifier is used to declare derived class members.

- **Accessing Class Members :**

The main() cannot contain statements that access class members directly. Class members can beaccessed only by an object of that class. To access class members, use the dot (.) operator. The dot operator links the name of an object with the name of a member. The general form of the dot operator is shown here:

object.member

Example

ob1.set_a(10);

The private members of a class cannot be accessed directly using the dot operator, but through the member functions of that class providing data hiding. A member function can call another member function directly, without using the dot operator.

C++ program to find sum of two numbers using classes

```
#include<iostream.h>
#include<conio.h>
class A{
int a,b,c;
public:
void sum(){
cout<<"enter two numbers";
cin>>a>>b;
c=a+b;
cout<<"sum="<<c;
}
};
int main(){
A u;
u.sum();
getch();
return(0);
}
```

- **Data hiding :**

Data hiding is a technique of hiding internal object details, i.e., data members. It is an object-oriented programming technique. Data hiding ensures, or we can say guarantees to restrict the data access to class members. It maintains data integrity.

Data hiding means hiding the internal data within the class to prevent its direct access from outside the class.

If we talk about data encapsulation so, **Data encapsulation** hides the private methods and class data parts, whereas **Data hiding** only hides class data components. Both **data hiding** and **data encapsulation** are essential concepts of object-oriented programming. **Encapsulation** wraps up the complex data to present a simpler view to the user, whereas **Data hiding** restricts the data use to assure data security.

Data hiding also helps to reduce the system complexity to increase the robustness by limiting the interdependencies between software components. Data hiding is achieved by using the private access specifier.

Example: We can understand data hiding with an example. Suppose we declared an Account class with a data member balance inside it. Here, the account balance is sensitive information. We may allow someone to check the account balance but won't allow altering the balance attribute. So, by declaring the balance attribute private, we can restrict the access to balance from an outside application.
The access specifiers to understand the data hiding. Access specifiers define how the member's functions and variables can be accessed from outside the class.

Example
In this example, there is a class with a variable and two functions. Here, the variable "num" is private so, it can be accessed only by the members of the same class, and it can't be accessed anywhere else. Hence, it is unable to access this variable outside the class, which is called data hiding.

```
#include<iostream>
using namespace std;
class Base{

   int num;  //by default private
   public:

   void getData();
   void showData();

};
void Base :: getData()
{
```

```
    cout<< "Enter any Integer value" <<endl;
    cin>>num;

}
void Base :: showData()
{
    cout<< "The value is " << num <<endl;
}

int main(){
    Base obj;
    obj.getData();
    obj.showData();
    return 0;
}
```

Output :

```
Enter any Integer value
2
The value is 2
```

Inline function :

If make a function as inline, then the compiler replaces the function calling location with the definition of the inline function at compile time.

Any changes made to an inline function will require the inline function to be recompiled again because the compiler would need to replace all the code with a new code; otherwise, it will execute the old functionality.

Syntax for an inline function:

```
inline return_type function_name(parameters)
{
  // function code?
}
```

Inside the main() method, when the function fun1() is called, the control is transferred to the definition of the called function. The addresses from where the function is called and the

definition of the function are different. This control transfer takes a lot of time and increases the overhead.

When the inline function is encountered, then the definition of the function is copied to it. In this case, there is no control transfer which saves a lot of time and also decreases the overhead.

Example.

```
#include <iostream>
using namespace std;
inline int add(int a, int b)
{
   return(a+b);
}
int main()
{
   cout<<"Addition of 'a' and 'b' is:"<<add(2,3);A
   return 0;

}
```

Advantages of inline function

- In the inline function, we do not need to call a function, so it does not cause any overhead.
- It also saves the overhead of the return statement from a function.
- It does not require any stack on which we can push or pop the variables as it does not perform any function calling.
- An inline function is mainly beneficial for the embedded systems as it yields less code than a normal function.

Disadvantages of inline function

The following are the disadvantages of an inline function:

- The variables that are created inside the inline function will consume additional registers. If the variables increase, then the use of registers also increases, which may increase the overhead on register variable resource utilization. It means that when the function call is replaced with an inline function body, then the number of variables also increases, leading to an increase in the number of registers. This causes an overhead on resource utilization.
- If we use many inline functions, then the binary executable file also becomes large.

- The use of so many inline functions can reduce the instruction cache hit rate, reducing the speed of instruction fetch from the cache memory to that of the primary memory.
- It also increases the compile-time overhead because whenever the changes are made inside the inline function, then the code needs to be recompiled again to reflect the changes; otherwise, it will execute the old functionality.
- Sometimes inline functions are not useful for many embedded systems because, in some cases, the size of the embedded is considered more important than the speed.
- It can also cause thrashing due to the increase in the size of the binary executable file. If the thrashing occurs in the memory, then it leads to the degradation in the performance of the computer.

- **Friend function :**

If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.

By using the keyword friend compiler knows the given function is a friend function.

For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword friend.

- **Declaration of friend function in C++**

```
class class_name
{
   friend data_type function_name(argument/s);          // syntax of friend function.
};
```

- **Characteristics of a Friend function:**

  - ✓ The function is not in the scope of the class to which it has been declared as a friend.
  - ✓ It cannot be called using the object as it is not in the scope of that class.
  - ✓ It can be invoked like a normal function without using the object.
  - ✓ It cannot access the member names directly and has to use an object name and dot membership operator with the member name.
  - ✓ It can be declared either in the private or the public part.

Example

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

```
class Box
{
   private:
      int length;
   public:
      Box(): length(0) { }
      friend int printLength(Box); //friend function
};
int printLength(Box b)
{
  b.length += 10;
   return b.length;
}
int main()
{
   Box b;
   cout<<"Length of box: "<< printLength(b)<<endl;
   return 0;
}
```

Output:

Length of box: 10

- **Passing and returning Object from a function :**

➢ **Pass object to a function-**

An object can be passed to a function just like we pass structure to a function. Here in class A we have a function disp() in which we are passing the object of class A. Similarly we can pass the object of another class to a function of different class.

```
#include <iostream>
using namespace std;
class A {
public:
  int n=100;
  char ch='A';
  void disp(A a){
    cout<<a.n<<endl;
    cout<<a.ch<<endl;
  }
```

```
};
int main() {
   A obj;
   obj.disp(obj);
   return 0;
}
```

Output:

```
100
A
```

- **Return object from a function :**

In this example we have two functions, the function input() returns the Student object and disp() takes Student object as an argument.

```
#include <iostream>
using namespace std;
class Student {
public:
   int stuId;
   int stuAge;
   string stuName;
   /* In this function we are returning the
    * Student object.
    */
   Student input(int n, int a, string s){
      Student obj;
      obj.stuId = n;
      obj.stuAge = a;
      obj.stuName = s;
      return obj;
   }
   /* In this function we are passing object
    * as an argument.
    */
   void disp(Student obj){
      cout<<"Name: "<<obj.stuName<<endl;
      cout<<"Id: "<<obj.stuId<<endl;
      cout<<"Age: "<<obj.stuAge<<endl;
```

```
    }
};
int main() {
    Student s;
    s = s.input(1001, 29, "Negan");
    s.disp(s);
    return 0;
}
```

Output:

```
Name: Negan
Id: 1001
Age: 29
```