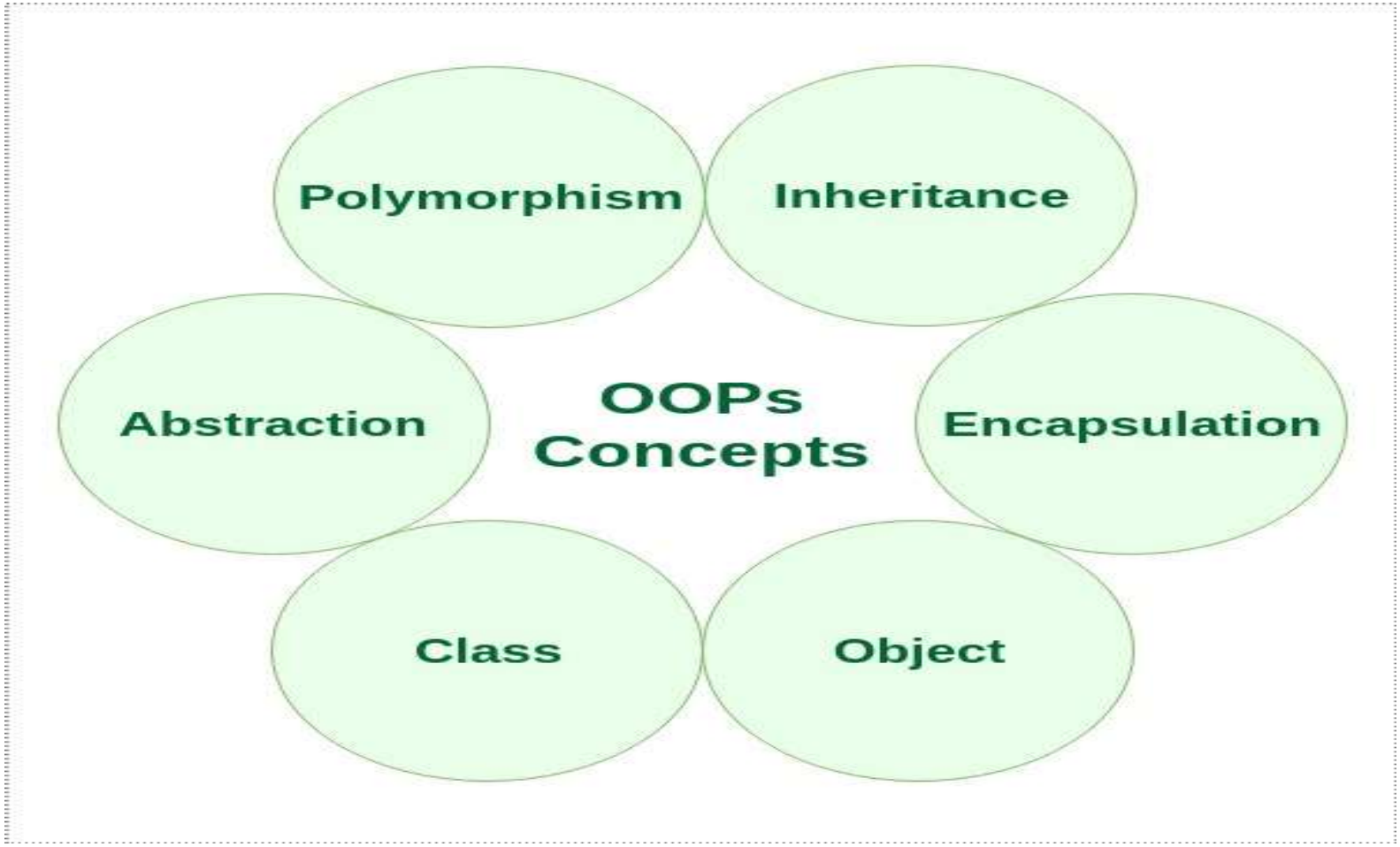
Object Oriented Programming

Unit -2

Object oriented programming

- Object-oriented programming (OOP) is a computer programming model that organizes software design around data, or objects, rather than functions and logic. An object can be defined as a data field that has unique attributes and behaviour.
- OOP focuses on the objects that developers want to manipulate rather than the logic required to manipulate them. This approach to programming is well-suited for programs that are large, complex and actively updated or maintained.

Object oriented programming



C++ What are Classes and Objects?

Classes and objects are the two main aspects of object-oriented programming.

Look at the following illustration to see the difference between class and objects:

class	objects
Fruit	Apple Banana Mango

•So, a class is a template for objects, and an object is an instance of a class.

•When the individual objects are created, they inherit all the variables and functions from the class.

Classes/Objects

Example

Create a class called "MyClass":

```
class MyClass {        // The class
 public:               // Access specifier
 int myNum;            // Attribute (int variable)
string myString;     // Attribute (string
 variable)
};
```

Inheritance

- Inheritance is an important pillar of OOP(Object Oriented Programming). It is the mechanism in java by which one class is allow to inherit the features(fields and methods) of another class.

- **Reusability:** 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.

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.
- Polymorphism allows you to define one interface and have multiple implementations.

Dynamic Binding

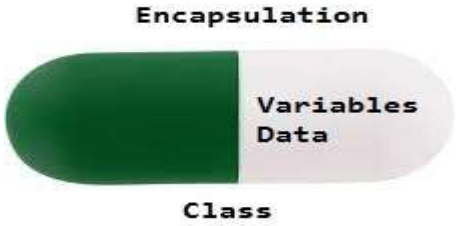
- C++ provides facility to specify that the compiler should match function calls with the correct definition at the run time; this is called dynamic binding or late binding or run-time binding.
- Dynamic binding is achieved using virtual functions. Base class pointer points to derived class object. And a function is declared virtual in base class, then the matching function is identified at run-time using virtual table entry.

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.

Encapsulation

- Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates.
- Other way to think about encapsulation is, it is a protective shield that prevents the data from being accessed by the code outside this shield.



Abstraction

- Data Abstraction is the property by virtue of which only the essential details are displayed to the user.The trivial or the non-essentials units are not displayed to the user

Advantage

Object-oriented programming has several advantages over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the C++ code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

Scope resolution operator in C++

- The :: (scope resolution) operator is used to get hidden names due to variable scopes so that you can still use them.
- The scope resolution operator can be used as both unary and binary. You can use the unary scope operator if a namespace scope or global scope name is hidden by a particular declaration of an equivalent name during a block or class.
- For example, if you have a global variable of name my_var and a local variable of name my_var, to access global my_var, you'll need to use the scope resolution operator.

Example

```
#include <iostream>
 using namespace std;
 int my_var = 0;
int main(void)
{
 int my_var = 0;
::my_var = 1;
// set global my_var to 1
my_var = 2;
   // set local my_var to 2
cout << ::my_var << ", " << my_var;
 return 0;
}
```

Output:-
1, 2

Example

```
#include <iostream>
using namespace std;
class X
 {
  public:
static int count;
 };
 int X::count = 10;  // define static data member
int main ()
 {
 int X = 0;   // hides class type X
 cout << X::count << endl;   // use static member of
      class X
}
```

Output:-
10

Scope of Variables in C++

- A scope is a region of the program and broadly speaking there are three places, where variables can be declared −
  - Inside a function or a block which is called local variables,
  - In the definition of function parameters which is called formal parameters.
  - Outside of all functions which are called global variables.
- Local variables can be used only by statements that are inside that function or block of code. Local variables are not known to functions on their own.

Example

```
#include <iostream>
using namespace std;
int main ()
 {    // Local variable declaration:
int a, b;
 int c;    // actual initialization
  a = 10;
 b = 20;
c = a + b;
 cout << c;
  return 0;
 }
```

Output:-
30

Scope of Variables in C++

- Global variables are defined outside of all the functions, usually on top of the program.
- The global variables will hold their value throughout the lifetime of your program.
- A global variable can be accessed by any function.

Example

```
#include <iostream>
using namespace std;
// Global variable declaration:
 int g;
int main ()
{    // Local variable declaration:
 int a, b;
 // actual initialization
   a = 10;
  b = 20;
  g = a + b;
 cout << g;
return 0;
 }
```

Output:-
30

Scope of Variables in C++

- A program can have the same name for local and global variables but the value of a local variable inside a function will take preference.
- For accessing the global variable with same rame, you'll have to use the scope resolution operator.

Example

```
#include <iostream>
using namespace std;
 // Global variable declaration:
 int g = 20;
int main ()
 {    // Local variable declaration:
 int g = 10;
  cout << g;
// Local
cout << ::g;
// Global
return 0;
 }
```

Output:-
10
20

Access Specifier

- In C++, there are three access specifiers:
  - public - members are accessible from outside the class
  - private - members cannot be accessed (or viewed) from outside the class
  - protected - members cannot be accessed from outside the class, however, they can be accessed in inherited classes. You will learn more about Inheritance later.

we demonstrate the differences between public and private members:

Example:-

```
class MyClass {
  public:    // Public access specifier
    int x;   // Public attribute
  private:   // Private access specifier
    int y;   // Private attribute
};

int main() {
  MyClass myObj;
  myObj.x = 25;  // Allowed (public)
  myObj.y = 50;  // Not allowed (private)
  return 0;
}
```

Access Specifier

- By default, all members of a class are private if you don't specify an access specifier:

```
class MyClass {
  int x;   // Private attribute
  int y;   // Private attribute
};
```

Data Hiding

- Data hiding is an object-oriented programming technique of hiding internal object details i.e. data members. Data hiding guarantees restricted data access to class members & maintain object integrity.
- How data hiding works in C++. Following topics are covered in this:
  - » Encapsulation
  - » Abstraction
  - » Data Hiding

Encapsulation

- Encapsulation binds the data & functions together which keeps both safe from outside interference. Data encapsulation led to data hiding.

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

class Encapsulation
{
   private:
      int num;

   public:
         void set(int a)
      {
         num =a;
      }

            int get()
      {
         return num;
      }
};

int main()
{
   Encapsulation obj;

   obj.set(5);

   cout<<obj.get();
   return 0;
}
```

Output:-

Data Abstraction

- Data Abstraction is a mechanism of hiding the implementation from the user & exposing the interface.

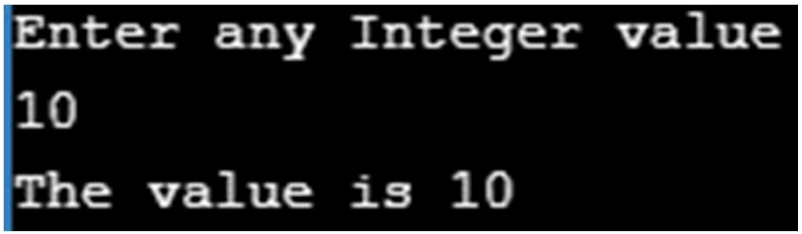
```
#include <iostream>
using namespace std;
   class Abstraction
{
   private:
      int num1, num2;

   public:

      void set(int a, int b)
      {
         num1 = a;
         num2 = b;
      }

      void display()
      {
         cout<<"num1 = " <<num1 << endl;
         cout<<"num2 = " << num2 << endl;
      }
};

int main()
{
   Abstraction obj;
   obj.set(50, 100);
   obj.display();
   return 0;
}
```

```
num1 = 50
num2 = 100
```

Data hiding

- Data hiding is a process of combining data and functions into a single unit. The ideology behind data hiding is to conceal data within a class, to prevent its direct access from outside the class.
- It helps programmers to create classes with unique data sets and functions, avoiding unnecessary penetration from other program classes.

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

 int num;  //by default private
   public:

   void read();
   void print();

};

void Base :: read(){
   cout<<"Enter any Integer value"<<endl; cin>>num;

}

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

int main(){
   Base obj;

   obj.read();
   obj.print();

   return 0;
}
```



Data members and Member functions in C++

- **"Data Member"** and **"Member Functions"** are the new names/terms for the members of a class, which are introduced in C++ programming language.
- 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.**

Data members and Member functions in C++

- There are two **types of data members/member functions in C++**:
  - Private members
  - 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.

For example

```
class Cube
{
 public:
int side;
/* Declaring function getVolume with no
   argument and return type int.
*/
 int getVolume(); };
```

If we define the function inside class then we don't not need to declare it first, we can directly define the function.

```
class Cube
 {
public:
int side;
 int getVolume()
 {
return side*side*side;
//returns volume of cube
 }
 };
```

The main function for both the function definition will be same. Inside main() we will create object of class, and will call the member function using dot . operator.

```
class Cube
 { public:
 int side;
 int getVolume();
}
// member function defined outside class definition
int Cube :: getVolume()
{
return side*side*side;
}
```

Consider the 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.

```
#include <iostream>
 #include <string.h>
using namespace std;
#define MAX_CHAR 30
class person
 {
private:
char name [MAX_CHAR];
int age;
public:
void get(char n[], int a)
 {
 strcpy(name , n);
age = a;
 }
void put() { cout<< "Name: " << name <<endl;
cout<< "Age: " <<age <<endl;
}
};
 int main() {
 person PER;
  PER.get("Manju Tomar", 23);
PER.put();
return 0;
}
```

Acessing Data Member

- Accessing a data member depends solely on the access control of that data member. If its public, then the data member can be easily accessed using the direct member access (.) operator with the object of that class.
- If, the data member is defined as private or protected, then we cannot access the data variables directly. Then we will have to create special public member functions to access, use or initialize the private and protected data members. These member functions are also called **Accessors** and **Mutator** methods or **getter** and **setter** functions.

Accessing Public Data Members

- Following is an example to show you how to initialize and use the public data members using the dot (.) operator and the respective object of class.

```
class Student
{
public:
 int rollno;
string name;
 };
int main()
 {
Student A;
Student B;
 A.rollno=1;
A.name="Adam";
 B.rollno=2;
B.name="Bella";
cout <<"Name and Roll no of A is: "<< A.name << "-
     " << A.rollno;
cout <<"Name and Roll no of B is: "<< B.name << "-"
     << B.rollno;
 }
```

Output:-

Name and Roll no of A is: Adam-1

Name and Roll no of B is: Bella-2

Calling Class Member Function in C++

- Similar to accessing a data member in the class, we can also access the public member functions through the class object using the dot operator (.).
- Below we have a simple code example, where we are creating an object of the class Cube and calling the member function getVolume():

Calling Class Member Function in C++

```
int main()
{
 Cube C1;
 C1.side = 4;
// setting side value
 cout<< "Volume of cube C1 =
   "<< C1.getVolume();
 }
```

Output:-
Volume of cube C1 = 16

Accessing Private Data Members

- To access, use and initialize the private data member you need to create getter and setter functions, to get and set the value of the data member.
- The setter function will set the value passed as argument to the private data member, and the getter function will return the value of the private data member to be used. Both getter and setter function must be defined public.

```
class Student
{
private:
int rollno;
public:
int getRollno()
{
 return rollno;
 }
void setRollno(int i)
{
 rollno=i;
 }
};
int main()
 {
Student A;
 A.rollono=1;
cout<< A.rollno;
A.setRollno(1);
cout<< A.getRollno();
}
```

Types of Class Member Functions in C++

- We already know what member functions are, what they do, how to define member function and how to call them using class objects. Now lets learn about some special member functions which can be defined in C++ classes. Following are the different types of Member functions:
  - » Simple functions
  - » Static functions
  - » Const functions
  - » Inline functions
  - » Friend functions

Simple Member functions in C++

- These are the basic member function, which don't have any special keyword like static etc as prefix. All the general member functions, which are of below given form, are termed as simple and basic member functions.

```
return_type
 functionName(parameter_list)
 {
  function body;
  }
```

Static Member functions in C++

- Static is something that holds its position.
- Static is a keyword which can be used with data members as well as the member functions.
- A function is made static by using static keyword with function name. These functions work for the class as whole rather than for a particular object of a class.

Example

```
class X
{
public:
static void f()
{
// statement
 }
};
int main()
 {
 X::f();          // calling member function directly with class name
 }
```

Const Member functions in C++

- **Const** keyword in detail later(Const Keyword), but as an introduction, Const keyword makes variables constant, that means once defined, there values can't be changed.
- When used with member function, such member functions can never modify the object or its related data members.

```
// basic syntax of const Member Function
   void fun()
    const
     {
   // statement
     }
```

Inline functions in C++

- Inline functions are actual functions, which are copied everywhere during compilation, like preprocessor macro, so the overhead of function calling is reduced.
- All the functions defined inside class definition are by default inline, but you can also make any non-class function inline by using keyword **inline** with them.

```
inline void fun(int a)
  {
   return a++;
   }
```

Some Important points about Inline Functions

- We must keep inline functions small, small inline functions have better efficiency.
- Inline functions do increase efficiency, but we should not make all the functions inline. Because if we make large functions inline, it may lead to **code bloat**, and might affect the speed too.
- Hence, it is adviced to define large functions outside the class definition using scope resolution :: operator, because if we define such functions inside class definition, then they become inline automatically.
- Inline functions are kept in the Symbol Table by the compiler, and all the call for such functions is taken care at compile time.

Getter and Setter Functions in C++

- Accessing **private** data variables inside a class. we use access functions, which are inline to do so.

```
class Auto
{
// by default private
int price;
public:
 // getter function for variable price
int getPrice()
{ return price; }
  // setter function for variable price
void setPrice(int x)
 { i=x; }
 };
```

Here getPrice() and setPrice() are inline functions, and are made to access the private data members of the class Auto. The function getPrice(), in this case is called **Getter or Accessor** function and the function setPrice() is a **Setter or Mutator** function.

Limitations of Inline Functions

- Large Inline functions cause Cache misses and affect performance negatively.
- Compilation overhead of copying the function body everywhere in the code on compilation, which is negligible for small programs, but it makes a difference in large code bases.
- Also, if we require address of the function in program, compiler cannot perform inlining on such functions. Because for providing address to a function, compiler will have to allocate storage to it. But inline functions doesn't get storage, they are kept in Symbol table.

Forward References in C++

```
class ForwardReference
 {
int i;
public:
// call to undeclared function
int f() {
 return g()+10;
}
int g()
{
return i;
}
 };
 int main()
{
ForwardReference fr;
 fr.f();
 }
```

Friend functions in C++

- Friend functions are actually not class member function.
- Friend functions are made to give **private** access to non-class functions. You can declare a global function as friend, or a member function of other class as friend.

Example

```
class WithFriend
{
 int i;
public:
friend void fun();
 // global function as friend
 };
void fun()
 {
 WithFriend wf;
wf.i=10;
// access to private data member
cout << wf.i;
 } int main()
 { fun();
//Can be called directly
 }
```

Reason??

why C++ is not called as a **pure Object Oriented language**

- When we make a class as friend, all its member functions automatically become friend functions.
- Friend Functions is a reason, why C++ is not called as a **pure Object Oriented language**. Because it violates the concept of **Encapsulation**.

Example

```
class Other
{
 void fun();
 };
class WithFriend
 {
private:
int i;
public:
void getdata();
friend void Other::fun();
 friend class Other;
};
```

Passing an Object as argument

- To pass an object as an argument we write the object name as the argument while calling the function the same way we do it for other variables.

Syntax:-

```
function_name(object_name);
```

Program to show passing of objects to a function

```
#include <bits/stdc++.h>
using namespace std;
  class Example {
public:
    int a;
    void add(Example E)
    {
        a = a + E.a;
    }
};
int main()
{
     Example E1, E2;
    E1.a = 50;
    E2.a = 100;
     cout << "Initial Values \n";
    cout << "Value of object 1: " << E1.a
         << "\n& object 2: " << E2.a
         << "\n\n";
        E2.add(E1);
         cout << "New values \n";
    cout << "Value of object 1: " << E1.a
         << "\n& object 2: " << E2.a
         << "\n\n";
     return 0;
}
```

```
Initial Values Value of object 1: 50
 & object 2: 100
 New values Value of object 1: 50
& object 2: 150
```

Returning Object as argument

Syntax:

object = return object_name;

Program to show passing of objects to a function

```
#include <bits/stdc++.h>
using namespace std;
 class Example {
public:
   int a;
    Example add(Example Ea, Example Eb)
   {
      Example Ec;
      Ec.a = Ec.a + Ea.a + Eb.a;
       return Ec;
   }
};
int main()
{
   Example E1, E2, E3;
   E1.a = 50;
   E2.a = 100;
   E3.a = 0;
    cout << "Initial Values \n";
   cout << "Value of object 1: " << E1.a
       << ", \nobject 2: " << E2.a
       << ", \nobject 3: " << E3.a
       << "\n";
    E3 = E3.add(E1, E2);
      cout << "New values \n";
   cout << "Value of object 1: " << E1.a
       << ", \nobject 2: " << E2.a
       << ", \nobject 3: " << E3.a
       << "\n";      return 0; }
```

Output:-

Initial Values
Value of object 1: 50,
object 2: 100,
object 3: 0
New values Value of object 1: 50,
object 2: 100,
object 3: 200