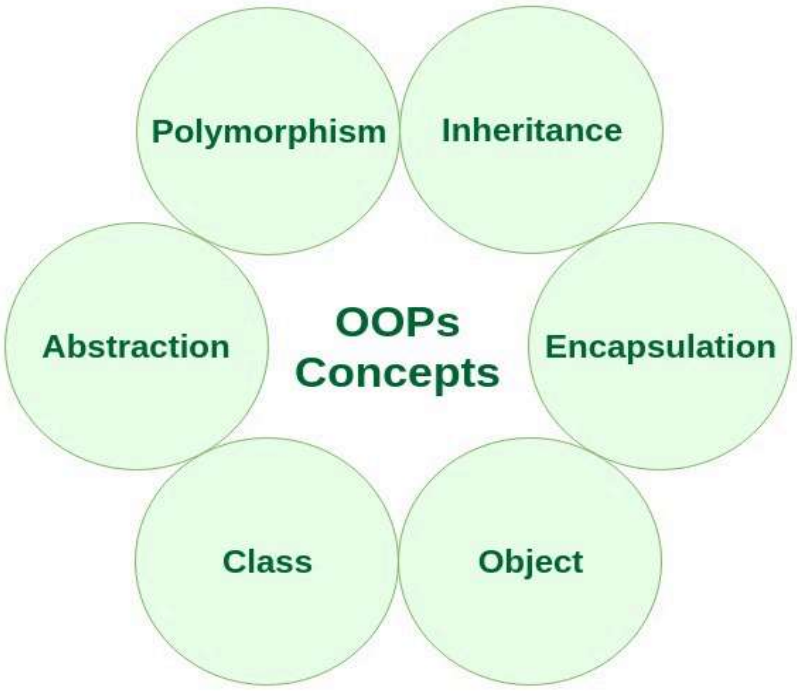
Object-oriented programming

- 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.

- Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc. in programming.

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

1. Encapsulation
2. Abstraction
3. Polymorphism
4. Inheritance



Object

- An Object is an identifiable entity with some characteristics and behavior.
- An Object is an instance of a Class.
- When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.
- Objects take up space in memory and have an associated address like a record in pascal or structure or union.
- When a program is executed the objects interact by sending messages to one another.
- Each object contains data and code to manipulate the data.
- Objects can interact without having to know details of each other's data or code, it is sufficient to know the type of message accepted and the type of response returned by the objects.

```
// C++ Program to show the syntax/working of Objects as a
// part of Object Oriented Programming
#include <iostream>
using namespace std;
class person {
    char name[20];
    int id;
public:
    void getdetails() {}
};
int main()
{
    person p1; // p1 is a object
    return 0;
}
```

Class

- The building block of C++ that leads to Object-Oriented programming is a Class.
- A Class is like a blueprint for an object.
- A Class is a user-defined data type that has data members and member functions which can be accessed and used by creating an instance of that class.
- Data members are the data variables and member functions are the functions used to manipulate these variables together these data members and member functions define the properties and behavior of the objects in a Class.

For Example: Consider the Class of Cars. There may be many cars with different names and brands but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range, etc. So here, the Car is the class, and wheels, speed limits, and mileage are their properties.

- In the above example of class Car, the data member will be speed limit, mileage, etc and member functions can apply brakes, increase speed, etc.

We can say that a **Class in C++** is a blueprint representing a group of objects which shares some common properties and behaviors.

Defining Class in C++

A class is defined in C++ using the keyword c**lass** followed by the name of the class.

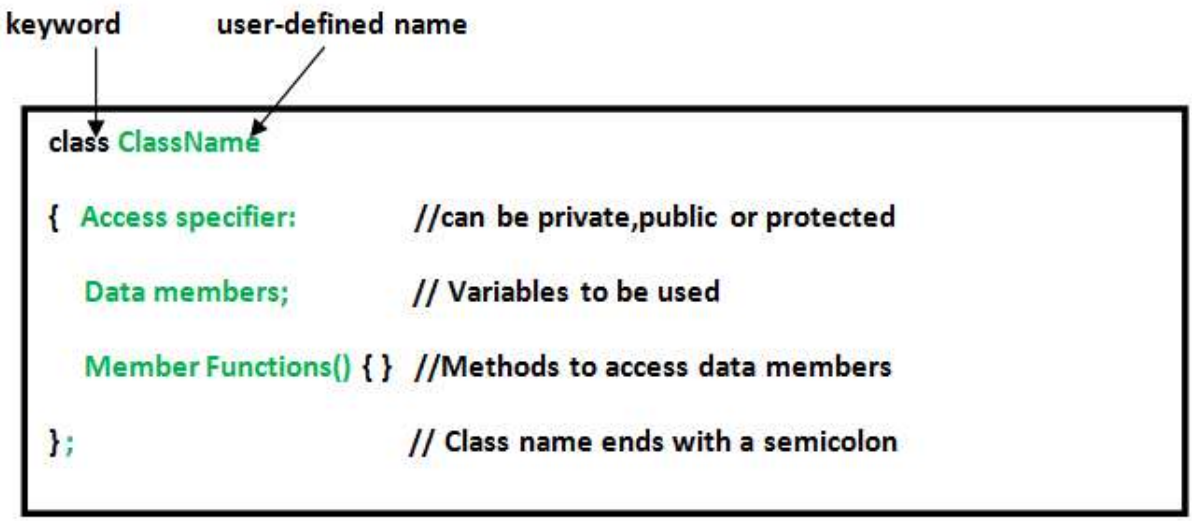
The following is the syntax:

```
class ClassName {
    access_specifier:
    // Body of the class
};
```

Note : The access specifier defines the level of access to the class's data members.

Example

```
class ThisClass {
    public:
    int var;     // data member
    void print() {           // member method
         cout << "Hello";
     }
};
```



Note : To use the data and access functions defined in the class, you need to create objects.

Syntax to Create an Object

- We can create an object of the given class in the same way we declare the variables of any other inbuilt data type.

```
ClassName ObjectName;
```

Example

```
MyClass obj;
```

Accessing Data Members and Member Functions

- The data members and member functions of the class can be accessed using the dot('.') operator with the object.

For example, if the name of the object is *obj* and you want to access the member function with the name *printName()* then you will have to write:

```
obj.printName()
```

Example of Class and Object in C++

The below program shows how to define a simple class and how to create an object of it.

```
// C++ program to illustrate how create a simple class and

// object

#include <iostream>

#include <string>

using namespace std;
```

```cpp
// Define a class named 'Person'
class Person {
public:
   // Data members
   string name;
   int age;

   // Member function to introduce the person
   void introduce()
   {
       cout << "Hi, my name is " << name << " and I am "
            << age << " years old." << endl;
   }
};
```

```
int main()

{

   // Create an object of the Person class

   Person person1;



   // accessing data members

   person1.name = "Alice";

   person1.age = 30;



   // Call the introduce member method

   person1.introduce();



   return 0;

}
```

Output

```
Hi, my name is Alice and I am 30 years old.
```

Access Modifiers or Access Specifiers

- Access Specifier in a class are used to assign the accessibility to the class members.
- They set some restrictions on the class members so that they can't be directly accessed by the outside functions.
- Also known as access modifier.
- They are the keywords that are specified in the class.
- All the members of the class under that access specifier will have a specific access level.
- Access modifiers are used to implement an important aspect of Object-Oriented Programming known as **Data Hiding**.

There are 3 access specifiers that are as follows:

1. **Public:** Members declared as public can be accessed from outside the class.
2. **Private:** Members declared as private can only be accessed within the class itself.
3. **Protected:** Members declared as protected can be accessed within the class and by derived classes.

Note : If we do not specify the access specifier, the private specifier is applied to every member by default.

Example of Access Specifiers

```
// C++ program to demonstrate accessing of data members

#include <iostream>
```

```
using namespace std;

Class Student {

private:

   string studentname;

   // Access specifier

public:

   // Member Functions()

   void setName(string name) {

         studentname = name;

     }


   void printname() {

       cout << "studentname is:" << studentname;

     }

};

int main()

{

   // Declare an object of class Student

   Student st1;

   // accessing data member
```

```
    // cannot do it like: st1.studentname = "Abhi";

    st1.setName("Abhi");

    // accessing member function

    st1.printname();

    return 0;
}
```

```
Output
studentname is:Abhi
```

1. Public:

- All the class members declared under the public specifier will be available to everyone.
- The data members and member functions declared as public can be accessed by other classes and functions too.
- The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

Example:

```cpp
// C++ program to demonstrate public
// access modifier
#include <iostream>
using namespace std;

// class definition
class Circle

{
   public:

       double radius;

       double  compute_area()

       {
           return 3.14*radius*radius;
       }

};

// main function

int main()
{
   Circle obj;
   // accessing public data member outside class
   obj.radius = 5.5;

   cout << "Radius is: " << obj.radius << "\n";

   cout << "Area is: " << obj.compute_area();

   return 0;

}
```

Output:

```
Radius is: 5.5
Area is: 94.985
```

2. Private:

- The class members declared as *private* can be accessed only by the member functions inside the class.
- They are not allowed to be accessed directly by any object or function outside the class.
- Only the member functions or the friend functions / friend class are allowed to access the private data members of the class.

Example:

```
// C++ program to demonstrate private
// access modifier
#include<iostream>
using namespace std;
class Circle
{
   // private data member
   private:
double radius;

   // public member function
   public:
       double  compute_area()

       {   // member function can access private
           // data member radius
           return 3.14*radius*radius;
       }
  };
```

```
// main function

int main()

{

   // creating object of the class

   Circle obj;

   // trying to access private data member

   // directly outside the class

   obj.radius = 1.5;

   cout << "Area is:" << obj.compute_area();

   return 0;
}
```

Output:

```
In function 'int main()':
11:16: error: 'double Circle::radius' is private
        double radius;
               ^
31:9: error: within this context
    obj.radius = 1.5;
        ^
```

Note : We can access the private data members of a class indirectly using the public member functions of the class.

Example:

```
// C++ program to demonstrate private
// access modifier
#include<iostream>
using namespace std;
class Circle
{
   // private data member
   private:
       double radius;

   // public member function
   public:
       void compute_area(double r)
       {
// member function can access private
           // data member radius
           radius = r;
           double area = 3.14*radius*radius;
           cout << "Radius is: " << radius << endl;
           cout << "Area is: " << area;
       }
};

// main function
int main()
{
// creating object of the class
   Circle obj;
   // trying to access private data member
   // directly outside the class
   obj.compute_area(1.5);
         return 0;
}
```

Output:

```
Radius is: 1.5
Area is: 7.065
```

3. Protected:

- The protected access modifier is similar to the private access modifier in the sense that it can't be accessed outside of its class unless with the help of a friend class.
- The difference is that the class members declared as Protected can be accessed by any subclass (derived class) of that class as well.

Note: This access through inheritance can alter the access modifier of the elements of base class in derived class depending on the mode of inheritance.

Example:

```
// C++ program to demonstrate
// protected access modifier
#include <iostream>
using namespace std;

// base class
class Parent

{
   // protected data members

   protected:

   int id_protected;



};
```

```
// sub class or derived class from public base class

class Child : public Parent

{

   public:

   void setId(int id)

   {
       // Child class is able to access the inherited

       // protected data members of base class
       id_protected = id;
   }

   void displayId()

   {

       cout << "id_protected is: " << id_protected << endl;

   }
};
// main function

int main() {
   Child obj1;
   // member function of the derived class can
   // access the protected data members of the base class
   obj1.setId(81);

   obj1.displayId();
   return 0;

}
```

Output:

```
id_protected is: 81
```

Member Function

There are 2 ways to define a member function:

- Inside class definition
- Outside class definition

Till now, we have defined the member function inside the class, but we can also define the member function outside the class.

To define a member function outside the class definition,

- We have to first declare the function prototype in the class definition.
- Then we have to use the **scope resolution :: operator** along with the class name and function name.

Example

```
// C++ program to demonstrate member function
// definition outside class
#include <iostream>
using namespace std;

class Student {

public:
   string Studentname;
```

```
   int id;

   // printname is not defined inside class definition
   void printname();

   // printid is defined inside class definition

   void printid() {

cout << "Student id is: " << id;

}

};

// Definition of printname using scope resolution operator

// ::

void Student::printname()

{

   cout << "Student name is: " << Studentname;

}

int main()

{

   Student obj1;

   obj1.Studentname = "xyz";

   obj1.id = 15;

   // call printname()
   obj1.printname();
```

```
    cout << endl;

    // call printid()
    obj1.printid();
    return 0;
}
```

Output

```
Student name is: xyz
Student id is: 15
```

Constructors

- **Constructor** is a special method that is invoked automatically at the time an object of a class is created.
- It is used to initialize the data members of new objects generally.
- The name of the constructor is the same as its class name.
- It constructs the values i.e. provides data for the object.
- Constructors do not return values; hence they do not have a return type.
- Constructors are mostly declared in the public section of the class though they can be declared in the private section of the class.

Syntax of Constructors

The prototype of the constructor looks like this:

```
<class-name> (){
...
}
```

Types of Constructor Definitions

There are 2 methods by which a constructor can be declared:

1. Defining the Constructor Within the Class

```
<class-name> (list-of-parameters) {
   // constructor definition
}
```

Example:

```
// Example to show defining
// the constructor within the class
#include <iostream>
using namespace std;
// Class definition
class student {
    int rno;
    char name[50];
    double fee;
public:
    student()
    {
        // Constructor within the class

        cout << "Enter the RollNo:";
        cin >> rno;
        cout << "Enter the Name:";
        cin >> name;
        cout << "Enter the Fee:";
        cin >> fee;
    }
```

```
    // Function to display the data
    // defined via constructor
    void display()
    {
        cout << endl << rno << "\t" << name << "\t" << fee;
    }
};

int main()
{

    student s;


    /*
    constructor gets called automatically
    as soon as the object of the class is declared
    */

    s.display();
    return 0;
}
```

2. Defining the Constructor Outside the Class

```
<class-name> {

   // Declaring the constructor
   // Definiton will be provided outside
   <class-name>();

   // Defining remaining class
}

<class-name>::<class-name>(list-of-parameters) {
     // constructor definition
}
```

Example:

```
#include <iostream>
using namespace std;
class student {
    int rno;
    char name[50];
    double fee;

public:
    /*
    To define a constructor outside the class,
    we need to declare it within the class first.
    Then we can define the implementation anywhere.
    */
    student();

    void display();
};
```

```
/*
Here we will define a constructor
outside the class for which, we are creating it.
*/
student::student()
{
    // outside definition of constructor

    cout << "Enter the RollNo:";
    cin >> rno;
    cout << "Enter the Name:";
    cin >> name;
    cout << "Enter the Fee:";
    cin >> fee;
}

void student::display()
{
    cout << endl << rno << "\t" << name << "\t" << fee;
}

// driver code
int main()
{
    student s;
    /*
    constructor gets called automatically
    as soon as the object of the class is declared
    */

    s.display();
    return 0;
}
```

Types of Constructors

- Constructors can be classified based on in which situations they are being used.
- There are 4 types of constructors.
  1. **Default Constructor**: No parameters. They are used to create an object with default values.
  2. **Parameterized Constructor**: Takes parameters. Used to create an object with specific initial values.
  3. **Copy Constructor**: Takes a reference to another object of the same class. Used to create a copy of an object.
  4. **Move Constructor**: Takes a value reference to another object. Transfers resources from a temporary object.

1. Default Constructors

- A **default constructor** is a constructor that either takes no arguments or has default values for all its parameters.
- It is also referred to as a **zero-argument constructor** when it explicitly accepts no arguments.
- If a default constructor is not explicitly defined by the programmer in the source code, the compiler automatically generates one during the compilation process.

- However, if a programmer explicitly defines a default constructor, the compiler does not generate it. Instead, the explicitly defined default constructor is called implicitly wherever needed.
- In cases where a class is derived from a base class with a default constructor, or a class contains an object of another class with a default constructor, the compiler must insert code to ensure these default constructors are invoked appropriately for the base class or the embedded objects.

Default Constructors and Inheritance:

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

class Base {
public:
    // Compiler "declares" constructor
};

class A {
public:
    // User defined constructor
    A() { cout << "A Constructor" << endl; }

    // Uninitialized
    int size;
};

class B : public A {
    // Compiler defines default constructor of B,
    // and inserts stub to call A constructor
    // Compiler won't initialize any data of A
};
```

```
class C : public A {
public:
    C()
    {
        // User defined default constructor of C
        // Compiler inserts stub to call A's constructor
        cout << "C Constructor" << endl;

        // Compiler won't initialize any data of A
    }
};

class D : public A {
    A a;
public:
    D()
    {
        // User defined default constructor of D
        // a - constructor to be called, compiler inserts
        // stub to call A constructor
        cout << "D Constructor" << endl;

        // Compiler won't initialize any data of 'a'
    }
};

// Driver Code
int main()
{
    Base base; // Only Base constructor (default provided by
the compiler) is called
    B b; // Calls A's constructor due to inheritance
(compiler-generated constructor for B)
    C c; // Calls A's constructor first, then C's constructor
    D d; // Calls A's constructor for member 'a', then D's
```

```
constructor

    return 0;
}
```

2.Parameterized Constructor

- Parameterized constructor is a type of constructor that can accept arguments.
- Parameterized Constructors make it possible to pass arguments to initialize an object when it is created.
- To create a parameterized constructor, simply add parameters to it the way you would to any other function.
- When you define the constructor's body, use the parameters to initialize the object.

```
#include <iostream>
using namespace std;
class A {
public:
    int x;
     // Parameterized constructor
     A(int val) {
          x = val;
    }
};
```

```
int main() {

     // Creating object and calling parameterized constructor
    A a(10);

    cout << a.x;
    return 0;
}
```

```
// Output
10
```

1. Syntax of Parameterized Constructor

The parameterized constructor can be defined using the below syntax:

```
class ClassName {
     public:
     // parameterized constructor starts here
     className (parameters...) {
           // body
     }
};
```

2. Example of Parameterized Constructor

```
#include <iostream>
#include <string>

using namespace std;
class Student {
   int rno;
   string name;
   double fee;

public:
  // Declaration of parameterized constructor
   Student(int, string, double);
   void display();
};
// Parameterized constructor outside class
Student::Student(int no, string n, double f) {
   rno = no;
   name = n;
   fee = f;

}
void Student::display() {

   cout << rno << "\t" << name << "\t" << fee << endl;

}
int main() {

  // Creating object with members initialized to
  // given values
   Student s(1001, "Ram", 10000);
   s.display();
   return 0;
}

// Output
1001  Ram   10000
```

The parameterized constructors can be called explicitly or implicitly:

```
Student s = Student(1001, "Ram", 10000);    // Explicit call
Student s(1001, "Ram", 10000);          // Implicit call
```

When the parameterized constructor is defined and no default constructor is defined explicitly, the compiler will not implicitly create the default constructor and hence create a simple object as:

```
Student s;
```

```
// will flash an error.
```

Note: 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. However, it is not necessary, but it's considered to be the best practice to always define a default constructor.

3. Default Arguments with Parameterized Constructor

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

```
public:
   int data;
  // Parameterized constructor with default values

   A(int x = 5) { data = x; }

};

int main() {

   A a1; // Will not throw error
   A a2(25);
   cout << a1.data << endl;
   cout << a2.data;
   return 0;
}
```

Output

```
5
25
```

4. Member Initializer Lists in Constructor

- Member initializer list provides a clean and compact way to initialize data members of the class using parameterized constructor.

Syntax of Member Initializer List

```
// parameterized constructor inside a class
ClassName (p1, p2, ...) : mem1(p1),  mem2(p2) ... {};
```

- Different members can be initialized by separating then by a comma(,). Although, the number of elements in the initializer list must match the number of parameters in the constructor.

Example:

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

// class definition
class A {
public:
   // Member initialization list

   A(int v1, double v2) : x(v1) , y(v2){}

   int x;
   double y;
};
int main() {

   // Creating and initializing the object
   A a(10, 11.5);

   // Printing object data members' values
   cout << a.x << " " << a.y;

   return 0;

}
```

Output

```
10 11.5
```

5. Applications of Parameterized Constructor

- It is used to initialize the various data elements of different objects with different values when they are created.
- It is used to overload constructors.

3. Copy Constructor

- A **copy constructor** is a type of constructor that initializes an object using another object of the same class.
- In simple terms, a constructor which creates an object by initializing it with an object of the same class, which has been created previously , is known as a **copy constructor**.
- The process of initializing members of an object through a copy constructor is known as **copy initialization**.
- It is also called **member-wise initialization** because the copy constructor initializes one object with the existing object, both belonging to the same class on a member-by-member copy basis.

Syntax of Copy Constructor

Copy constructor takes a reference to an object of the same class as an argument:

```
className (const ClassName &obj)
{
   member1 = obj.member1;
   // for dynamic resources
   ......
}
```

Examples of Copy Constructor in C++

Example 1: User Defined Copy Constructor

```
// C++ program to illustrate the use of copy constructor
#include <iostream>
#include <string.h>

using namespace std;
// Class definition for 'student'
class student {
   int rno;
   string name;
   double fee;

public:
   // Parameterized constructor
   student(int, string, double);

   // Copy constructor
   student(student& t)

   {
       rno = t.rno;
       name = t.name;
       fee = t.fee;
       cout << "Copy Constructor Called" << endl;
   }
   // Function to display student details
   void display();

};
```

```
// Implementation of the parameterized constructor
student::student(int no, string n, double f)
{
   rno = no;
   name = n;
   fee = f;
}

// Implementation of the display function
void student::display()
{
   cout << rno << "\t" << name << "\t" << fee << endl;
}

int main()

{
   // Create student object with parameterized constructor
   student s(1001, "Manjeet", 10000);
   s.display();

   // Create another student object using the copy
   // constructor
   student manjeet(s);
   manjeet.display();
   return 0;
}

// Output
1001    Manjeet    10000
Copy Constructor Called
1001    Manjeet    10000
```

Example 2: Default Copy Constructor

- An implicitly defined copy constructor will copy the bases and members of an object in the same order that a constructor would initialize the bases and members of the object.

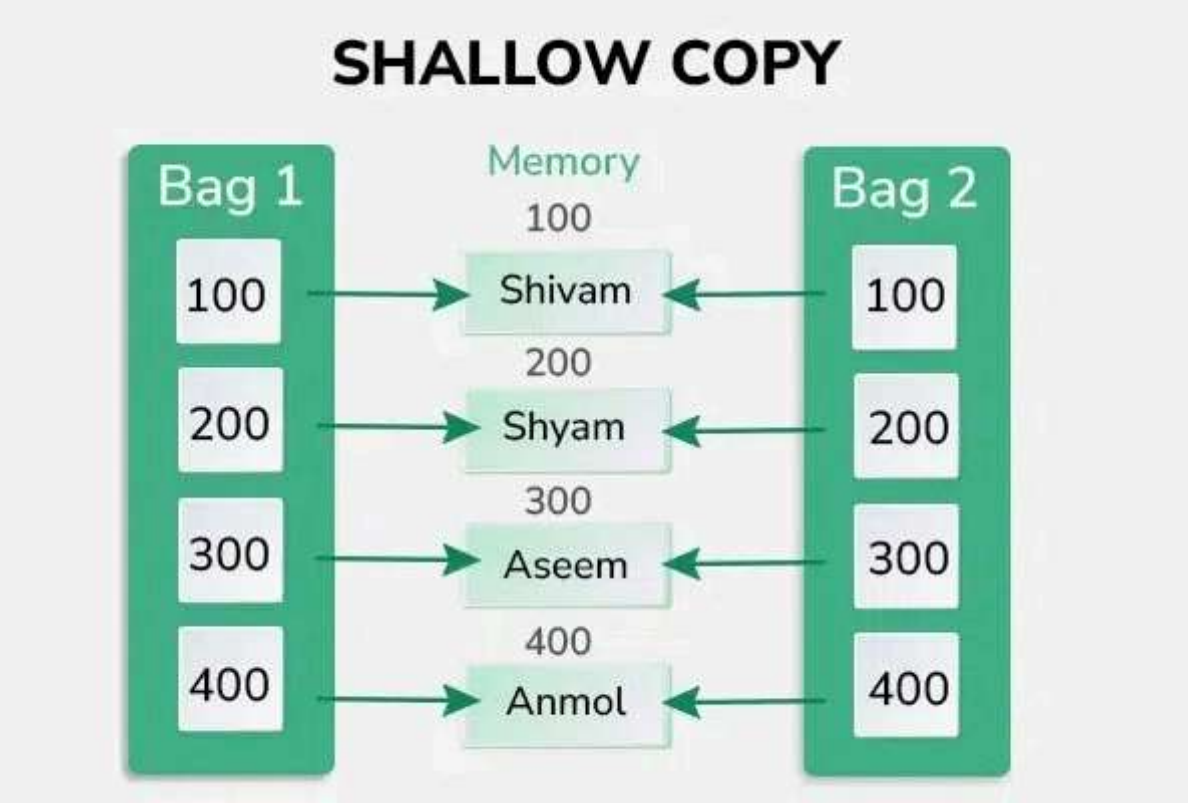
```
// Implicit copy constructor Calling
#include <iostream>
using namespace std;
class Sample {
   int id;
public:
   void init(int x) { id = x; }
   void display() { cout << endl << "ID=" << id; }
};

int main()

{

   Sample obj1;

   obj1.init(10);

   obj1.display();
   // Implicit Copy Constructor Calling

   Sample obj2(obj1); // or obj2=obj1;

   obj2.display();

   return 0;

}

Output
ID=10
ID=10
```

Need of User Defined Copy Constructor

- If we don't define our own copy constructor, the C++ compiler creates a default copy constructor for each class which works fine in general. However, we need to define our own copy constructor only if an object has pointers or any runtime allocation of the resource like *a file handle*, a network connection, etc because the default *constructor does only shallow copy*.

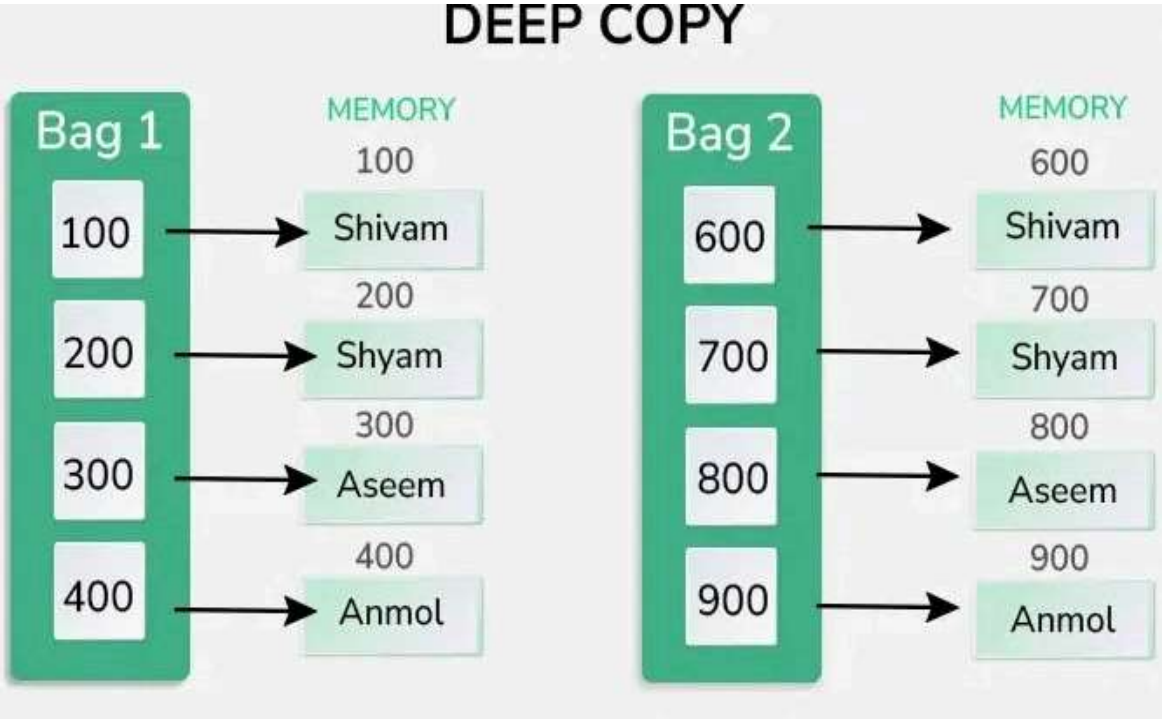
Shallow Copy

- means that only the pointers will be copied, not the actual resources that the pointers are pointing to. This can lead to dangling pointers if the original object is deleted.



Deep copy

- *is possible only with a user-defined copy constructor.* In a user-defined copy constructor, we make sure that pointers (or references) of copied objects point to new copy of the dynamic resource allocated manually in the copy constructor using new operators.



Note : Destructors,Pointer : -

Destructors

- Destructor is an instance member function that is invoked automatically whenever an object is going to be destroyed.
- Meaning, a destructor is the last function that is going to be called before an object is destroyed.

Syntax to Define Destructor

The syntax for defining the destructor within the class:

```
~ <class-name>() {
   // some instructions
}
```

Just like any other member function of the class, we can define the destructor outside the class too:

```
<class-name> {
public:
    ~<class-name>();
}

<class-name> :: ~<class-name>() {
  // some instructions
}
```

Characteristics of a Destructor

- A Destructor is also a special member function like a constructor.
- Destructor destroys the class objects created by the constructor.
- Destructor has the same name as their class name preceded by a tilde (~) symbol.
- It is not possible to define more than one destructor.
- The destructor is only one way to destroy the object created by the constructor. Hence, the destructor cannot be overloaded.
- It cannot be declared static or const.
- Destructor neither requires any argument nor returns any value.

- It is automatically called when an object goes out of scope.
- Destructor release memory space occupied by the objects created by the constructor.
- In a destructor, objects are destroyed in the reverse of an object creation.

Example 1

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

class Test {

public:
   // User-Defined Constructor
   Test() { cout << "\n Constructor executed"; }

   // User-Defined Destructor
   ~Test() { cout << "\nDestructor executed"; }

};

main(){
   Test t;
   return 0;
}
```

Output

```
Constructor executed
Destructor executed
```

Example 2

```
#include <iostream>
using namespace std;
// It is static so that every class object has the same
// value
static int Count = 0;

class Test {
public:
   // User-Defined Constructor
   Test()
   {
       // Number of times constructor is called
       Count++;
       cout << "No. of Object created: " << Count << endl;
   }
   // User-Defined Destructor
   ~Test()
   {
       // It will print count in descending order
       cout << "No. of Object destroyed: " << Count
            << endl;
       Count--;
       // Number of times destructor is called
   }
};
// driver code
int main()
{
   Test t, t1, t2, t3;
   return 0;
}
```

Output

```
No. of Object created: 1
No. of Object created: 2
```

```
No. of Object created: 3
No. of Object created: 4
No. of Object destroyed: 4
No. of Object destroyed: 3
No. of Object destroyed: 2
No. of Object destroyed: 1
```

Note: *Objects are destroyed in the reverse order of their creation. In this case, t3 is the first to be destroyed, while t is the last.*

When is the destructor called?

A destructor function is called automatically when the object goes out of scope or is deleted. Following are the cases where destructor is called:

1. Destructor is called when the function ends.
2. Destructor is called when the program ends.
3. Destructor is called when a block containing local variables ends.
4. Destructor is called when a delete operator is called.

How to call destructors explicitly?

Destructor can also be called explicitly for an object. We can call the destructors explicitly using the following statement:

```
object_name.~class_name()
```

When do we need to write a user-defined destructor?

If we do not write our own destructor in class, the compiler creates a default destructor for us. **The default destructor works fine unless we have dynamically allocated memory or pointer in class.** When a class contains a pointer to memory allocated in the class, we should write a destructor to release memory before the class instance is destroyed. This must be done to avoid memory leaks.

Pointers and References

- Pointers and References both are mechanisms used to deal with memory, memory address, and data in a program.
- Pointers are used to store the memory address of another variable whereas references are used to create an alias for an already existing variable.

Pointers

- **Pointers** are a symbolic representation of addresses.
- They enable programs to simulate call-by-reference and create and manipulate dynamic data structures.
- Pointers store the address of variables or a memory location.

Syntax

```
datatype *var_name;
```

for example, **int *_ptr_**; //ptr points to an address that holds int data.

Example of Pointers in

```
include <iostream>
using namespace std;

int main()
{
    int x = 10; // variable declared
    int* myptr; // pointer variable

    // storing address of x in pointer myptr
    myptr = &x;

    cout << "Value of x is: ";
    cout << x << endl;

    // print the address stored in myptr pointer variable
    cout << "Address stored in myptr is: ";
    cout << myptr << endl;

    // printing value of x using pointer myptr
    cout << "Value of x using *myptr is: ";
    cout << *myptr << endl;

    return 0;
}
```

Output

```
Value of x is: 10
Address stored in myptr is: 0x7ffd2b32c7f4
Value of x using *myptr is: 10
```

Application of Pointers

Following are the **Applications of Pointers:**

- To pass arguments by reference: Passing by reference serves two purposes
- For accessing array elements: The Compiler internally uses pointers to access array elements.
- To return multiple values: For example in returning square and the square root of numbers.
- Dynamic memory allocation: We can use pointers to dynamically allocate memory. The advantage of dynamically allocated memory is that it is not deleted until we explicitly delete it.
- To implement data structures.
- To do system-level programming where memory addresses are useful.

Features and Use of Pointers

- The Pointers have a few important features and uses like it saves memory space, they are used to allocate memory dynamically, it is used for file handling, etc. Pointers store the address of variables or a memory location.
  Example: pointer “ptr” holds the address of an integer variable or holds the address of memory whose value(s) can be accessed as integer values through “ptr”.

```
int *ptr;
```

References

- When a variable is declared as a reference, it becomes an alternative name for an existing variable.
- A variable can be declared as a reference by putting '&' in the declaration. There are 3 ways to pass C++ arguments to a function:

1. call-by-value
2. call-by-reference with a pointer argument
3. call-by-reference with a reference argument

Example of References

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

int main()
{
    int y = 10;

    // ref is a reference to x.
    int& myref = y;

    // changing value of y to 20
    y = 30;
    cout << "value of y is " << y << endl;
    cout << "value of myref after change in value of y is: "
         << myref << '\n';

    return 0;
}
```

Output

```
value of y is 30
value of myref after change in value of y is: 30
```

Pointers vs References

Pointers vs References :

- Both references and pointers can be used to change the local variables of one function inside another function.
- Both of them can also be used to save copying of big objects when passed as arguments to functions or returned from functions, to get efficiency gain.
- Despite the above similarities, there are the following differences between references and pointers.

Note *A pointer can be declared as void but a reference can never be void.For example:*

```
int a = 10;
void*aa = &a;. //it is valid
void &ar = a; // it is not valid
```

- References are less powerful than pointers
- Once a reference is created, it cannot be later made to reference another object; it cannot be reseated. This is often done with pointers.
- References cannot be NULL. Pointers are often made NULL to indicate that they are not pointing to any valid thing.
- A reference must be initialized when declared. There is no such restriction with pointers
- **Passing by pointer Vs Passing by Reference :** In C++, we can pass parameters to a function either by pointers or by reference. In both cases, we get the same result. So what should be preferred and why?
- **Passing Reference to a Pointer :** In this article let's compare the usage of a "pointer to pointer" VS "Reference to pointer" in some cases.

we must write a copy constructor.

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

class String {
private:
    char* s;
    int size;

public:
    String(const char* str = NULL);
    ~String() { delete[] s; }

       // Copy constructor
    String(const String&);
    void print() {
        cout << s << endl;
    }
    void change(const char*);
};

// Definitions of constructor and memeber functions
String::String(const char* str) {
    size = strlen(str);
    s = new char[size + 1];
    strcpy(s, str);
}

String::String(const String& old_str) {
    size = old_str.size;
    s = new char[size + 1];
    strcpy(s, old_str.s);
}
```

```
void String::change(const char* str) {
    delete[] s;
    size = strlen(str);
    s = new char[size + 1];
    strcpy(s, str);
}

int main() {
    String str1("KhanQuiz");

      // Create str2 from str1
    String str2 = str1;

    str1.print(); // what is printed ?
    str2.print();

    str2.change("KhanByKhan");

    str1.print(); // what is printed now ?
    str2.print();
    return 0;
}
```

Output

```
KhanQuiz
KhanQuiz
KhanQuiz
KhanByKhan
```

What would be the problem if we remove the copy constructor from the above code?

- If we remove the copy constructor from the above program, we don't get the expected output. The c**hanges made to str2 reflect in str1** as well which is never expected. Also, **if the str1 is destroyed, the str2's data member s will be pointing to the deallocated memory.**
-

When is the Copy Constructor Called?

- **When an object of the class is returned by value.**
- **When an object of the class is passed (to a function) by value as an argument.**
- **When an object is constructed based on another object of the same class.**
- **When the compiler generates a temporary object.**

Shallow Copy and Deep Copy in C++

- creating a copy of an object means to create an exact replica of the object having the same literal value, data type, and resources. There are two ways that are used by C++ compilers to create a copy of objects.
- 1. Copy Constructor
- 2. Assignment Operator

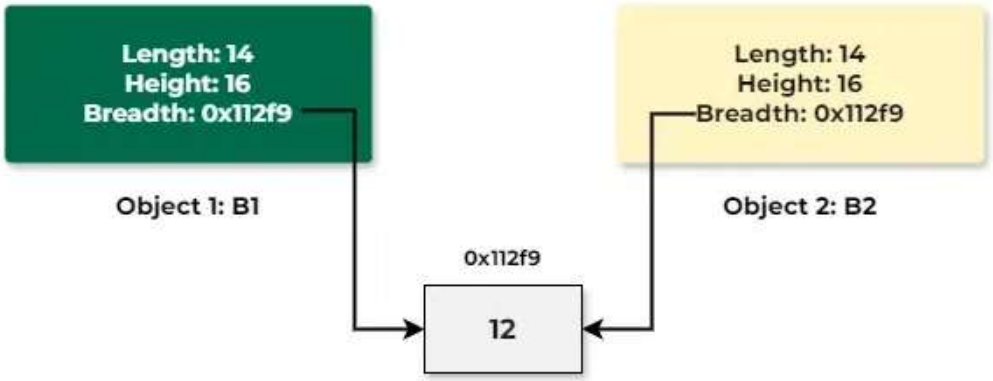
```
// Copy Constructor
Geeks Obj1(Obj);
or
Geeks Obj1 = Obj;

// Default assignment operator
Geeks Obj2;
Obj2 = Obj1;
```

- Depending upon the resources like dynamic memory held by the object, either we need to perform **Shallow Copy** or **Deep Copy** in order to create a replica of the object. In general, if the variables of an object have been dynamically allocated, then it is required to do a Deep Copy in order to create a copy of the object but one may wonder what is the shallow copy and deep copy?

What is Shallow Copy?

- An object is created by simply copying the data of all variables of the original object.
- This works well if none of the variables of the object are defined in the heap section of memory but if some variables are dynamically allocated memory from the heap section, then the copied object variable will also reference the same memory location.
- This will create ambiguity and run-time errors, dangling pointers. Since both objects will reference the same memory location, then changes made by one will reflect those changes in another object as well. Since we wanted to create a replica of the object, this purpose will not be filled by Shallow copy.

Shallow Copy



Shallow Copy

```cpp
// C++ program for the above approach
#include <iostream>
using namespace std;

// Box Class
class box {
private:
    int length;
    int breadth;
    int height;
      int *p;

public:
    // Function that sets the dimensions
    void set_dimensions(int length1, int breadth1,
                        int height1,int x)
    {
        length = length1;
        breadth = breadth1;
        height = height1;
          p=new int;
      *p=x;
    }

    // Function to display the dimensions
    // of the Box object
    void show_data()
    {
        cout << " Length = " << length
             << "\n Breadth = " << breadth
             << "\n Height = " << height
             << "\n P int pointing to = "<<p
             << endl;
    }
};
```

```
// Driver Code
int main()
{
    // Object of class Box
    box B1, B3;

    // Set dimensions of Box B1
    B1.set_dimensions(14, 12, 16,100);
    B1.show_data();

    // When copying the data of object
    // at the time of initialization
    // then copy is made through
    // COPY CONSTRUCTOR
    box B2 = B1;
    B2.show_data();

    // When copying the data of object
    // after initialization then the
    // copy is done through DEFAULT
    // ASSIGNMENT OPERATOR
    B3 = B1;
    B3.show_data();

    return 0;
}
```

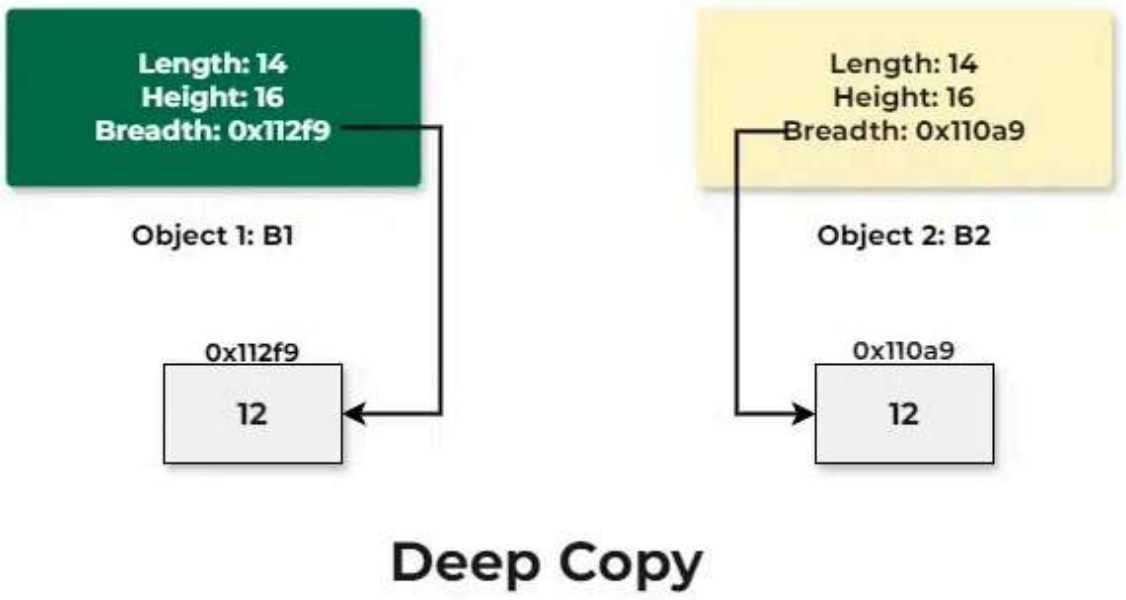
Output

```
Length = 14
 Breadth = 12
 Height = 16
 Length = 14
 Breadth = 12
```

```
Height = 16
Length = 14
Breadth = 12
Height = 16
```

What is Deep Copy?

- An object is created by copying data of all variables, and it also allocates similar memory resources with the same value to the object.
- In order to perform Deep copy, we need to explicitly define the copy constructor and assign dynamic memory as well, if required.
- Also, it is required to dynamically allocate memory to the variables in the other constructors, as well.



Deep Copy

```cpp
// C++ program to implement the
// deep copy
#include <iostream>
using namespace std;

// Box Class
class box {
private:
    int length;
    int* breadth;
    int height;

public:
    // Constructor
    box()
    {
        breadth = new int;
    }

    // Function to set the dimensions
    // of the Box
    void set_dimension(int len, int brea,
                       int heig)
    {
        length = len;
        *breadth = brea;
        height = heig;
    }

    // Function to show the dimensions
    // of the Box
    void show_data()
    {
        cout << " Length = " << length
             << "\n Breadth = " << *breadth
```

```
             << "\n Height = " << height
             << endl;
    }

    // Parameterized Constructors for
    // for implementing deep copy
    box(box& sample)
    {
        length = sample.length;
        breadth = new int;
        *breadth = *(sample.breadth);
        height = sample.height;
    }

    // Destructors
    ~box()
    {
        delete breadth;
    }
};

// Driver Code
int main()
{
    // Object of class first
    box first;

    // Set the dimensions
    first.set_dimension(12, 14, 16);

    // Display the dimensions
    first.show_data();

    // When the data will be copied then
    // all the resources will also get
```

```
    // allocated to the new object
    box second = first;

    // Display the dimensions
    second.show_data();

    return 0;
}
```

Output

```
Length = 12
 Breadth = 14
 Height = 16
 Length = 12
 Breadth = 14
 Height = 16
```

Encapsulation :

- Encapsulation is defined as the wrapping up of data and information in a single unit.
- In Object Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulate them.

Two Important property of Encapsulation

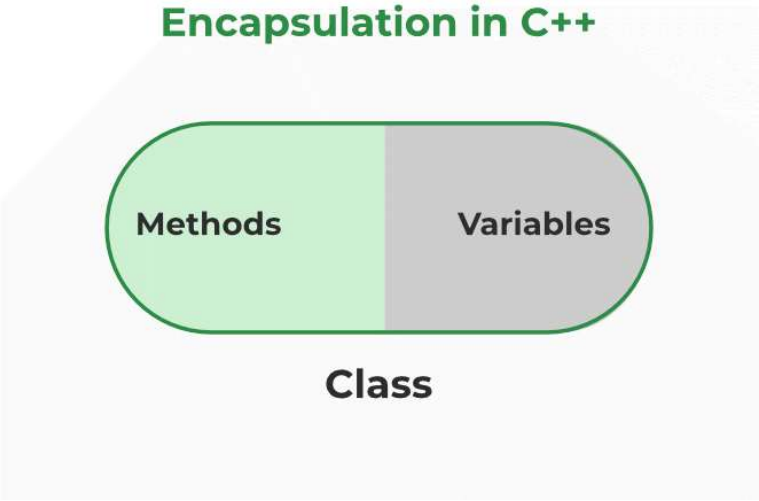
1. **Data Protection :**
    a. Encapsulation protects the internal state of an object by keeping its data members private.
    b. Access to and modification of these data members is restricted to the class's public methods,ensuring controlled and secure data manipulation.
2. **Information Hiding :**
    a. Encapsulation hides the internal implementation details of a class from external code.
    b. Only the public interface of the class is accessible,providing abstraction and simplifying the usage of the class while allowing the internal implementation to be modified without impacting external code.

```
#include <iostream>
using namespace std;
class temp{
     int a;
  int b;
  public:
  int solve(int input){
    a=input;
    b=a/2;
    return b;
  }
};
//driver code
int main() {
  int n;
  cin>>n;
  temp half;
  int ans=half.solve(n);
  cout<<ans<<endl;

}
```

Features of Encapsulation

1. We can not access any function from the class directly. We need an object to access that function that is using the member variables of that class.
2. The function which we are making inside the class must use only member variables, only then it is called *encapsulation*.

3. If we don’t make a function inside the class which is using the member variable of the class then we don’t call it encapsulation.
4. Encapsulation improves readability, maintainability, and security by grouping data and methods together.
5. It helps to control the modification of our data members.



Encapsulation also leads to data abstraction. Using encapsulation also hides the data, as in the above example, the data of the sections like sales, finance, or accounts are hidden from any other section.

```cpp
#include <iostream>
#include <string>

using namespace std;

class Person {
  private:
    string name;
    int age;
  public:
    Person(string name, int age) {
      this->name = name;
      this->age = age;
    }
    void setName(string name) {
      this->name = name;
    }
    string getName() {
      return name;
    }
    void setAge(int age) {
      this->age = age;
    }
    int getAge() {
      return age;
    }
};

int main() {
  Person person("John Doe", 30);

  cout << "Name: " << person.getName() << endl;
  cout << "Age: " << person.getAge() << endl;

  person.setName("Jane Doe");
```

```
  person.setAge(32);

  cout << "Name: " << person.getName() << endl;
  cout << "Age: " << person.getAge() << endl;

  return 0;
}
```

```
Output
Name: John Doe
Age: 30
Name: Jane Doe
Age: 32
```

Abstraction in C++

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

Types of Abstraction:

1. **Data abstraction** – This type only shows the required information about the data and ignores unnecessary details.
2. **Control Abstraction** – This type only shows the required information about the implementation and ignores unnecessary details.

Abstraction using Classes

- We can implement Abstraction in C++ using classes.
- The class helps us to group data members and member functions using available access specifiers.
- A Class can decide which data member will be visible to the outside world and which is not.

Abstraction in Header files

- One more type of abstraction in C++ can be header files.
- For example, consider the pow() method present in the math.h header file.
- Whenever we need to calculate the power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating the power of numbers.

Abstraction using Access Specifiers

Access specifiers are the main pillar of implementing abstraction in C++.

We can use access specifiers to enforce restrictions on class members.

For example:

- Members declared as **public** in a class can be accessed from anywhere in the program.
- Members declared as **private** in a class, can be accessed only from within the class.
- They are not allowed to be accessed from any part of the code outside the class.

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

class implementAbstraction {
private:
    int a, b;

public:
    // method to set values of
    // private members
    void set(int x, int y)
    {
        a = x;
        b = y;
    }

    void display()
    {
        cout << "a = " << a << endl;
        cout << "b = " << b << endl;
    }
};

int main()
{
    implementAbstraction obj;
    obj.set(10, 20);
    obj.display();
    return 0;
}
```

Output

```
a = 10
b = 20
```

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

class Vehicle
{
  private:
          void piston()
        {
            cout<<"4 piston\n";
        }

        void manWhoMade()
        {
            cout<<"Markus Librette\n";
        }
    public:
        void company()
        {
            cout<<"GFG\n";
        }
        void model()
        {
            cout<<"SIMPLE\n";
        }
        void color()
        {
            cout<<"Red/GREEN/Silver\n";
        }
        void cost()
        {
            cout<<"Rs. 60000 to 900000\n";
        }
        void oil()
        {
            cout<<"PETROL\n";
```

```
        }
};
int main()
{

    Vehicle obj;
    obj.company();
    obj.model();
    obj.color();
    obj.cost();
    obj.oil();
}
```

Output

```
GFG
SIMPLE
Red/GREEN/Silver
Rs. 60000 to 900000
PETRO
```

Advantages of Data Abstraction

- Helps the user to avoid writing the low-level code
- Avoids code duplication and increases reusability.
- Can change the internal implementation of the class independently without affecting the user.
- Helps to increase the security of an application or program as only important details are provided to the user.

- It reduces the complexity as well as the redundancy of the code, therefore increasing the readability.
- New features or changes can be added to the system with minimal impact on existing code.