IEEE SCT SB

IEEE

Code in 10 days

Day 7

dotSlash
Workshop series

Topics for Today

- Programming Paradigms
- OOP Concepts

Programming Paradigm

- A programming paradigm defines a methodology of designing and implementing programs using the key features and building blocks of a language.

- Types:
  - Procedural Programming
  - Object Oriented Programming

Programming paradigms

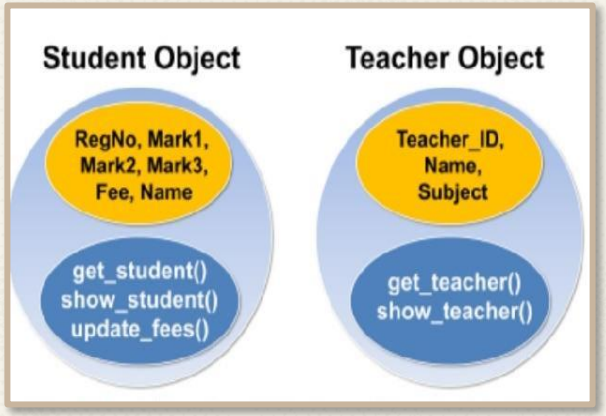
Procedural Oriented Programming	Object Oriented Programming
In procedural programming, program is divided into small parts called ***functions***.	In object oriented programming, program is divided into small parts called ***objects***.
Procedural programming follows ***top down approach***.	Object oriented programming follows ***bottom up approach***.
There is no access specifier in procedural programming.	Object oriented programming have access specifiers like private, public, protected etc.
Adding new data and function is not easy.	Adding new data and function is easy.
Procedural programming does not have any proper way for hiding data so it is ***less secure***.	Object oriented programming provides data hiding so it is ***more secure***.
In procedural programming, overloading is not possible.	Overloading is possible in object oriented programming.
In procedural programming, function is more important than data.	In object oriented programming, data is more important than function.
Procedural programming is based on ***unreal world***.	Object oriented programming is based on ***real world***.

OOP Paradigm

- 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 its related functions into a single entity.

OOP Paradigm

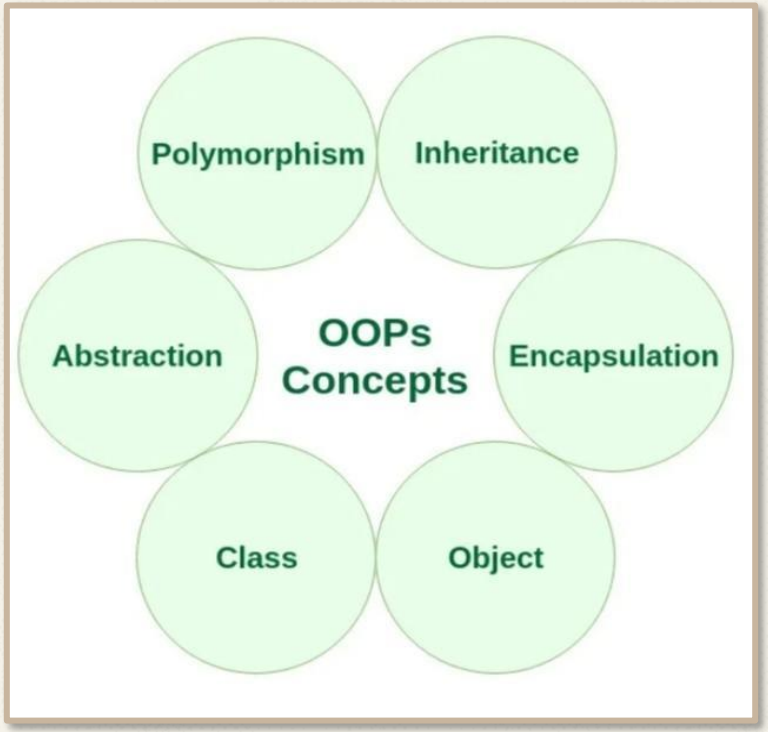
For example, by implementing OOP in a school software, we can create a Student object by clubbing student data and its functions, as well as a Teacher object by clubbing teachers' data and its functions. Now the functions of one object will not be able to access the data of other object without permission



Advantages of OOP

- OOP provides a clear modular structure for programs.
- It is good for defining abstract data types.
- Implementation details are hidden from other modules and have a clearly defined interface.
- It is easy to maintain and modify the existing code as new objects can be created without disturbing the existing ones.
- It can be used to implement real life scenarios.
- It can define new data types as well as new operations for operators.

Basic concepts of OOP



Objects

- An Object is an identifiable entity with some characteristics and behaviour.
- 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.

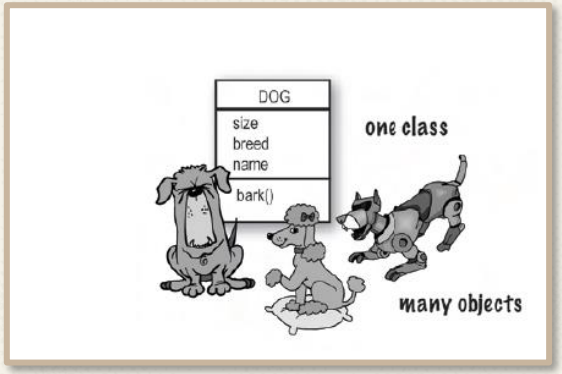
Classes

- The building block of C++ that leads to Object-Oriented programming is a Class.
- It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.
- It is a template/blueprint that represents a group of objects that share common properties and relationships.

Classes

- A class is a user-defined data-type which has data members and member functions.

- Data members are the data variables and member functions are the functions used to manipulate these variables and together, these data members and member functions define the properties and behaviour of the objects in a class.



Syntax

```
class class_name
    {
        private:
            [variable declarations;]
            [function declarations;]
        protected:
            [variable declarations;]
            [function declarations;]
        public:
            [variable declarations;]
            [function declarations;]
     }
```

//hidden data/methods

//unimportant implementation details

//exposed important details

Access Specifiers

- Private

  The members of the class cannot be accessed (or viewed) from outside the class.
- Public

  The members of the class are accessible from outside the class.
- Protected

  The members of the class cannot be accessed from outside the class, however, they can be accessed in inherited classes.

Access Specifiers

- Private – No Direct Access
- Protected – No Direct Access
- Public – Direct Access

Access-control specifiers	Accessible to	
	Own class members	Objects of a class
Private	Yes	No
Protected	Yes	No
Public	Yes	Yes

Example

- Consider a class of Cars. There may be many cars with different names and of different 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, Car is the class and wheels, speed limits, mileage are their properties.

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

- Functions such as gearChange(), slowDown(), brake() etc would be its member functions. Now create a object of this class named FordFigo which uses these data members and functions and give them its own values. We can create as many objects as we want using the class.

Example

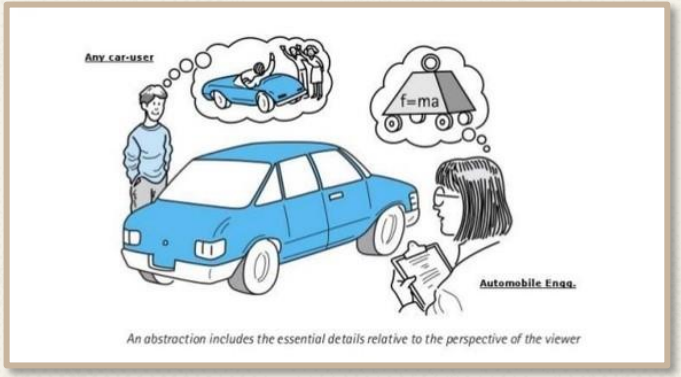
```
//Class name is Car
class Car
{
   //Data members
   char name[20];
   int speed;
   int weight;
   double mileage;

public:
   //Functions
   void brake()  {
       }
   void slowDown()  {
      }
};
```

```
int main()
{
  //ford is an object
   Car ford;
 }
```

Data Abstraction

- Abstraction is a process of hiding irrelevant details from user.
- For example, When you send an SMS you just type the message, select the contact and click send, the phone shows you that the message has been sent, what actually happens in background when you click send is hidden from you as it is not relevant to you.



Example

```
#include <iostream>
using namespace std;
class AbstractionExample{
private:
  /* By making these data members private, they
are hidden from the outside world.
   * These data members are not accessible outside
   * the class. The only way to set and get their
   * values is through the public functions.
   */
  int num;
  char ch;

public:
  void setMyValues(int n, char c) {
    num = n; ch = c;
  }
  void getMyValues() {
    cout<<"Numbers is:
"<<num<< endl;
    cout<<"Char is: "<<ch<<endl;
  }
};
int main(){
  AbstractionExample obj;
  obj.setMyValues(100, 'X');
  obj.getMyValues();
  return 0;
}
```

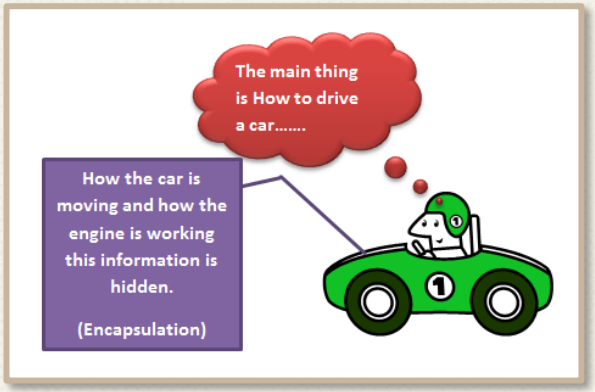
Output:

```
Numbers is: 100
Char is: X
```

Encapsulation

- Encapsulation is a process of combining data and function into a single unit, like a capsule. This is to avoid the access of private data members from outside the class.

- To achieve encapsulation, we make all data members of class private and create public functions, using them we can get the values from these data members or set the value to these data members.



Data Encapsulation Example

- In the previous example, we have two data members num and ch, and we have declared them as private so that they are not accessible outside the class, this way we are hiding the data. The only way to get and set the values of these data members is through the public getter and setter functions.

Example

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

class ExampleEncap
{
    private:
   /* Since we have marked these data members private,
    * any entity outside this class cannot access these data members directly, they have to use getter and setter functions.
    */
   int num;
   char ch;
public:
   /* Getter functions to get the value of data members.
    * Since these functions are public, they can be accessed outside the class, thus provide the access to data members  through them */
int getNum() const {
      return num;
   }

char getCh() const {
      return ch;
   }
 /* Setter functions are called for assigning the values to the private data members.
    */
   void setNum(int num) {
      this->num = num;
   }
   void setCh(char ch) {
      this->ch = ch;
   }
};
```

Example

```
int main(){
   ExampleEncap obj;
   obj.setNum(100);
   obj.setCh('A');
   cout<<obj.getNum()<<endl;
   cout<<obj.getCh()<<endl;
   return 0;
}
```

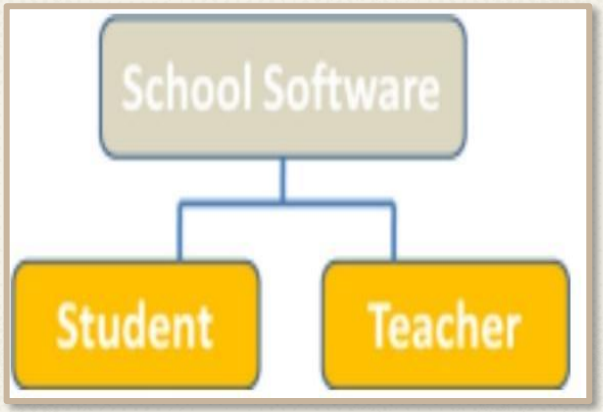
Output:

Modularity

- Modularity is a concept of partitioning a program into modules that can be considered and written on their own, with no consideration of any other module.
- These modules are later linked together to build the complete software.
- These modules communicate with each other by passing messages.

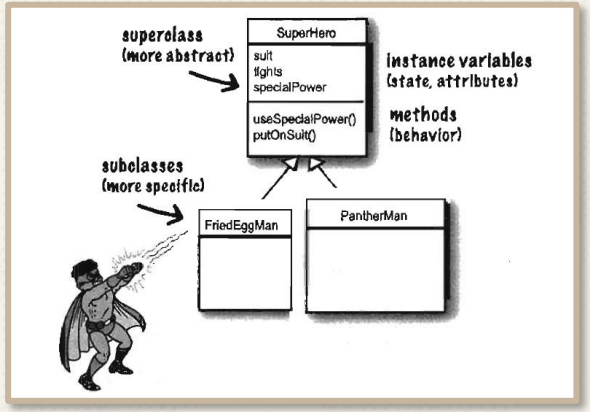
Modularity

- In object oriented-programming, modularity is implemented with the help of class.
- For example, in our school software we can separate everything related to students and teachers into two separate modules using the concept of class.

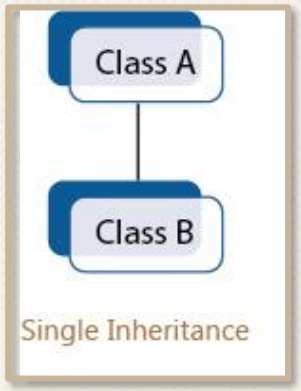


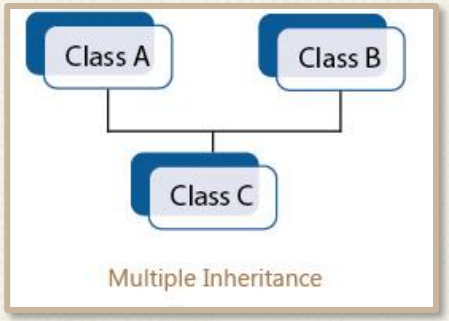
Inheritance

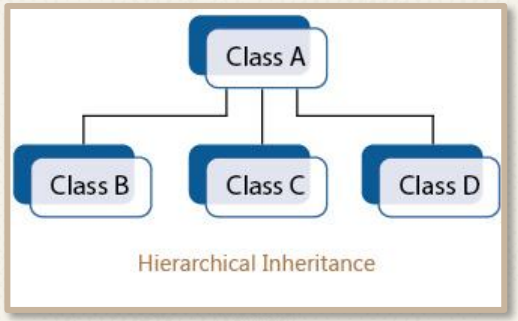
- Inheritance is the capability of one class of things to inherit capabilities or properties of another class.
- Types:
  - Single Inheritance
  - Multiple Inheritance
  - Hierarchical Inheritance
  - Multi-level Inheritance
- The child class is called the derived class, and the parent class is called the base class.

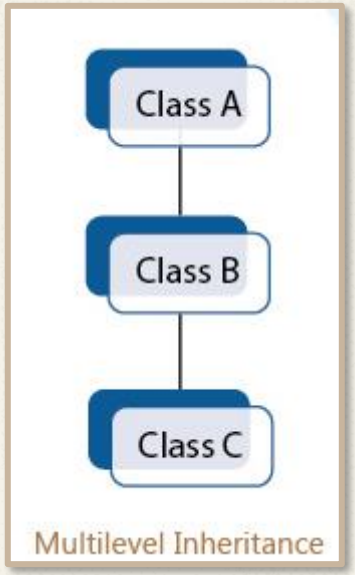


Types of Inheritance









Example of Single Inheritance

```
// C++ program to explain
// single inheritance

#include <iostream>
using namespace std;
class A
{
 //data member
  public:
                int var1 =100;
};
class B: public A
{
  public:
                int var2 = 500;
};
int main(void) {
  B obj;
  cout<<obj.var1<<endl;
  cout<<obj.var2;
}
```

Output:

Example of Multiple Inheritance

```
// C++ program to explain
// multiple inheritance
#include <iostream>
using namespace std;

// first base class
class Vehicle {
public:
        Vehicle()
        {
        cout << "This is a Vehicle" << endl;
        }
};

// second base class
class FourWheeler {
public:
        FourWheeler()
        {
        cout << "This is a 4 wheeler Vehicle" <<
endl;
        }
};

// sub class derived from two base
classes
class Car: public Vehicle, public
FourWheeler {

};

// main function
int main()
{
        // creating object of sub class
will invoke the constructor of base
classes
        Car obj;
        return 0;
}
```

Example of Hierarchical Inheritance

```
// C++ program to implement
// Hierarchical Inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle
{
public:
                Vehicle()
                {
                cout << "This is a Vehicle" << endl;
                }
};


// first sub class
class Car: public Vehicle
{

};

// second sub class
class Bus: public Vehicle
{

};

// main function
int main()
{
                // creating object of sub class will
invoke the constructor of base class
                Car obj1;
                Bus obj2;
                return 0;
}
```

Example of Multilevel Inheritance

```
// C++ program to implement
// Multilevel Inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle
{
public:
        Vehicle()
        {
        cout << "This is a Vehicle" << endl;
        }
};

// first sub_class derived from class vehicle
class fourWheeler: public Vehicle
{ public:
        fourWheeler()
        {
        cout<<"Objects with 4 wheels are vehicles"<<endl;
        }
};

// sub class derived from the derived base class fourWheeler
class Car: public fourWheeler{
public:
        Car()
        {
        cout<<"Car has 4 Wheels"<<endl;
        }
};

// main function
int main()
{
        //creating object of sub class will invoke the constructor of base classes
        Car obj;
        return 0;
}
```

Inheritance and Access Modifiers

Modes of Inheritance

1. Public mode: If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.

2. Protected mode: If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.

3. Private mode: If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.

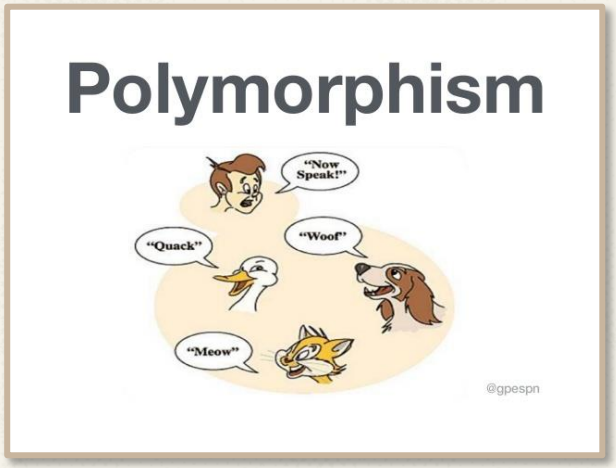
Inheritance and Access Modifiers

Public, Protected and Private Base Classes

Base Class Specifer	public inheritance	protected inheritance	private inheritance
public	public in derived class	protected in derived class	private in derived class
protected	protected in derived class	protected in derived class	private in derived class
private	hidden	hidden	hidden

Polymorphism

- Polymorphism is a feature using which an object behaves differently in different situation.
- Function overloading and Operator overloading are examples of polymorphism.
- In function overloading we can have more than one function with same name but different numbers, type or sequence of arguments.
- There are two types of polymorphism:
  - Compile Time polymorphism
  - Run Time polymorphism.



Example

```
#include <iostream>
using namespace std;
class Sum {
  public:
    int add(int num1,int num2){
      return num1 + num2;
    }
    int add(int num1, int num2, int num3){
      return num1 + num2 + num3;
    }
};
int main(void) {
   //Object of class Sum
   Sum obj;

   //This will call the second add function
   cout<<obj.add(10, 20, 30)<<endl;

   //This will call the first add function
   cout<<obj.add(11, 22);
   return 0;
}
```

Output:

```
60
33
```

Program

//Write a C++ program to create a student class. Program will read and print details of N students

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

class student
{
  private:
       int   rollno;
       char  name[20];
       float percentage;
  public:
    //member function to read details
    void readDetails(){
      cout << "Enter Roll number:";
      cin >> rollno;
      cout << "Enter Name:";
      cin >> name;
      cout << "Enter Percentage:";
      cin >> percentage;
    }
    //member function to display details
    void printDetails(){
      cout << "Student details:\n";
      cout << "RollNo=" << rollno;
      cout << ", Name=" << name;
      cout << ", Percentage=" << percentage;
    }
};
```

```
int main()
{
    student std[5];    //array of students object
    int n, i;

    cout << "Enter total number of students: ";
    cin >> n;
for(i=0; i<n; i++){
        cout << "Enter details of student " << i+1 << ":\n";
        std[i].readDetails();
    }

    for(i=0; i<n; i++){
        cout << "\nDetails of student " << i+1 << ":\n";
        std[i].printDetails();
    }
    return 0;
}
```

Thank You