OOPS Notes

1) **Inheritance:** Capability of a class to derive properties & characteristics from another class. Derived class inherits the properties of the base class, w/o changing properties of base class.
   Sub-Class → Derived class
   Super Class → Base Class

⇒ Avoids duplication of data.
⇒ Increase reusability.

2) **Polymorphism:** Ability of a message to be displayed in more than one form.

Polymorphism

Compile time

⇒ Polymorphism is achieved by function overloading or operator overloading.

⇒ **Function Overloading:** Multiple functions with the same name but different parameters.

⇒ **Operator Overloading:** Can ~~add~~ operate on different built in datatypes.

Runtime

⇒ It is achieved by Function overriding.

Function Overriding: It occurs when a derived class has a definition for one of the member of the base class.

3) Encapsulation: Defined as binding together the data and the functions that manipulates them. We cannot access any function from class directly. We need an object to access that function which is using the member variable of that class.
⇒ Leads to data Abstraction or hiding.

4) Abstraction: refers to providing only essential information about the data to the outside world, hiding the background details or implementation.
⇒ Can be implemented using classes & access specifiers
⇒ Avoids code duplication & increases reusability
⇒ Helps to increase security of an application.

5) Aggregation: (special form of Association)
⇒ represents Has-A relationship.
⇒ Unidirectional (eg. department has students)
⇒ Can exist w/o each other (college ◇— Student).
⇒ Increases code reusability

6) Composition:
⇒ Represents part-of relationship.
⇒ both entities are dependent on each other. (vehicle ◆— tyre)

7) Static local Variable:
⇒ Initialized to 0 by default.
⇒ Lifetime is throughout the program.

8) Static Member Variable:
⇒ Declared inside the class body, must be defined outside class
⇒ Do not belong to any object but to the whole class.

9) Static Member Function:
⇒ Can be invoked with or without object.
⇒ They can only access static members of the class.

10) Constructors:
⇒ Member function of a class.
⇒ Same name as of class.
⇒ No return type., & must be an instance member func.

1) Default : Made by compiler (no value & no code)
2) Parameterized: When a parameter is passed in a constructor
3) Constructor Overloading: when more than 1 constructor is made inside the class.

⇒ When no constructor is made in the class, compiler will create 2 constructors : 1) Copy, 2) Default.

∴ order of executing : from parent to child
order of calling : from child to parent.

- function pointer, Smart pointer
- Friend class, virtual destructor, static

Date:

11) Destructors:
⇒ Instance member function of a class.
⇒ Same name as class but preceded by (~) symbol.
⇒ Can never be static & has no return type.
⇒ Takes no argument (No overloading possible).
⇒ Invoked implicitely when object is going to destroy.
⇒ It should be defined to release resources allocated to an object.

12) Friend Function:
⇒ Not a member function of a class to which it is friend.
⇒ Must be defined outside the class to which it is friend.
⇒ Can access any member of the class to which " " "
⇒ Cannot access members of the class directly.
⇒ Can be declared as both public & private as not a member function
⇒ Member function of one class can become friend to another class.

13) Object Pointer: A pointer which contains address of an ~~pointer~~ object.

14) This Pointer: Local object Pointer in every instance member function containing address of caller object.

15) Method Overriding:
⇒ Creating a function in the child class with the same name as in the parent class.
⇒ Changes for parent class can be made in child class.

16) Method Hiding: Creating the same name function in child class with different arguments.

17) Virtual Function :
It is used to tell the compiler to late bind the function of the class. Late Binding means dynamic binding, binding at run time.

18) Base Class Pointer :
⇒ BCP can point to the object of any of it's descendant class.
⇒ But the converse is not true.

19) Pure Virtual Function : A do nothing function is a pure virtual function.

20) Abstract Class : A class containing Pure Virtual Func.

Inheritance Example:

```
class Animal {
  public:
    void eat() {
      cout << "I can Eat!" << endl;
    }
    void sleep() {
      cout << "I can sleep!" << endl;
    }
};
class Dog : public Animal {
  public:
    void bark() {
      cout << "I can bark!" << endl;
    }
};

main() {
  Dog dog1;
  dog1.eat();
  dog1.sleep();
  dog1.bark();
  return 0;
}
```

Polymorphism - Compile Time: (Function overloading)

```
class A {
  public:
    void func (int x) {
        cout << "value of x is: " << x << endl;
    }
    void func (double x) {
        cout << "Value of x is:" << x << endl;
    }
    void func (int x, int y) {
        cout << "Value of x & y is " << x << y << endl;
    }
};
main () {
    A obj1;
    obj1.func(7);
    obj1.func(9.132);
    obj1.func(85,64);
    return 0;
}
```

Runtime Polymorphism :

```
class Base {
  public:
    virtual void print() {
        cout << "print base class" << endl;
    }
    void show() {
        cout << "Show base class << endl;
    }
};

class Derived : public Base {
  public:
    void print() {
        cout << "Print Base Class" << endl;
                          Derived
    }
    void show() {
        cout << "Show Derived Class" << endl;
    }
};
main() {
  Base* p;
  Derived d;
  p = &d;
  p -> print();   // Virtual function binded at runtime
  p -> show();    // Non-Virtual function binded at Compile time
  return 0;
}
```

Output: Print Derived class
Show Base Class.

Date:

Encapsulation:

```
class Encapsulation {
  private:
    int x;
  public:
    void set (int a) {
        x = a;
    }
    int get ( ) {
        return x;
    }
};

main ( ) {
    Encapsulation obj;
    obj.set (5);
    cout << obj.get ();
    return 0;
}
```

Abstraction:

```
Class Abstraction {
    private:
        int a, b;
    public:
        void set (int x, int y){
            a = x;
            b = y;
        }
        Void display () {
            cout << " a = " << a << endl;
            cout << " b = " << b << endl;
        }
};

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

Output: a = 10
b = 20.