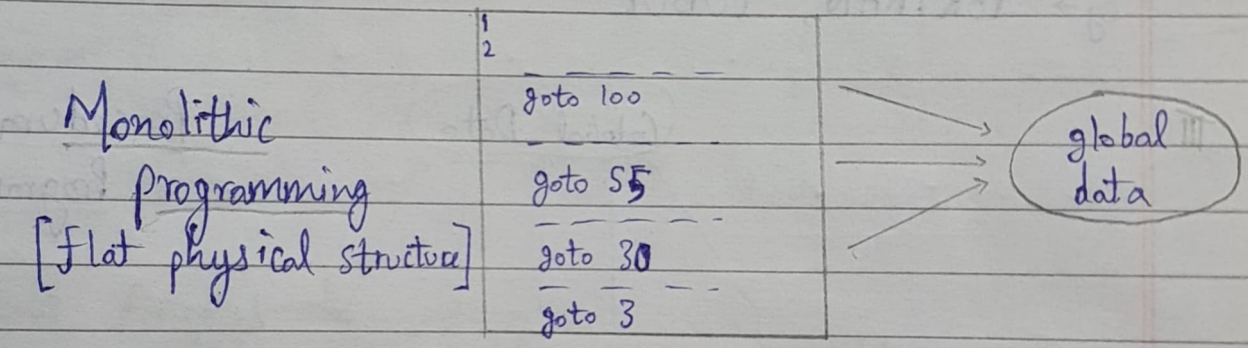
AL3OS OOPM

(I) Monolithik Programming

The program written in these languages exhibit relatively flat physical structure as shown in the figure.

They consist of only global data and sequencial code. Program flow control is achieved through the use of jumps.

eg → Assembly language and BASIC.



(II) Procedural Programming

Programs were considered as important intermediate points between problem & computer in.

Initially software abstraction achieved through procedural abstraction. grew directly out of this pragmatic view of software

The following are important features of procedural programming

i) Programs are organized in form of subroutines & all data items are global.

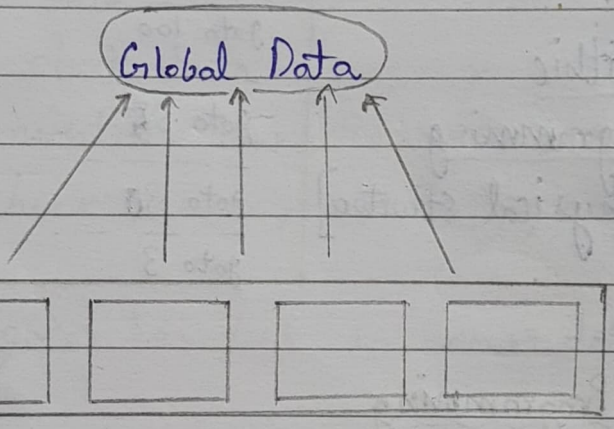
ii) Program control are through jumps (goto's) and calls to subroutines.

iii) Subroutines are abstracted to avoid repetitions.

iv) Suitable for medium size software applications.

v) Difficult to maintain & enhance the program code.

eg → FORTRAN, COBOL



Procedural Programming

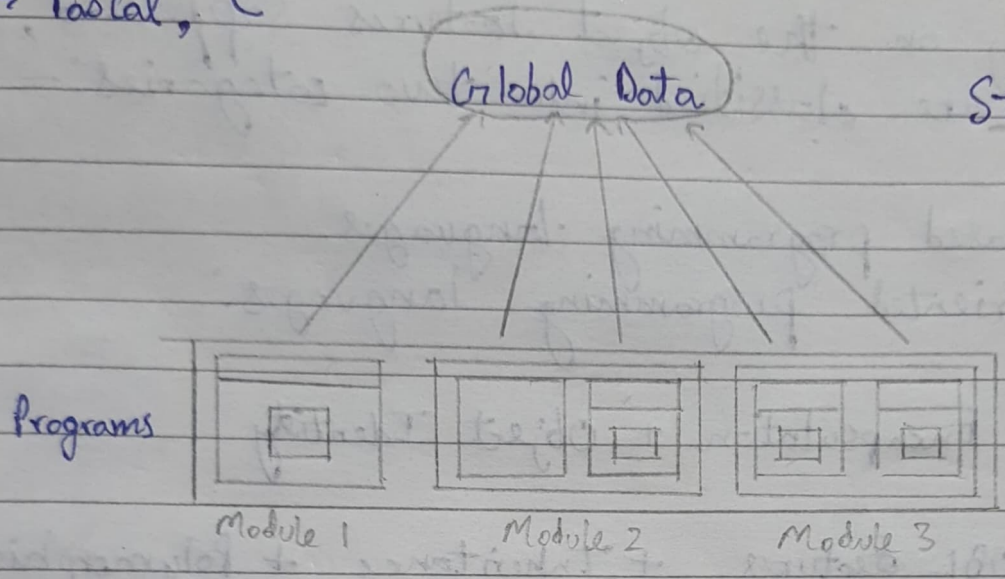
(III) Structured Programming

* Structured programming has evolved as a mechanism to address the growing issue of programming in the large.
* Larger programming projects consist of large development teams, developing different parts of the same project independently.
* The usage of separately compiled modules (algorithmic decomposition) was the answer for managing

large development teams.

* Programs consist of multiple modules & in turn each module has a set of functions of related types.

eg → Pascal, C



Structured Programming

(IV) Object Oriented Programming

* The easy way to master the management of complexity in the development of a software system is through the use of data abstraction.
* Procedure abstraction is suitable for the description of abstract operations, but it is not suitable for the description of abstract objects.
* This is a serious drawback in many applications
* The emergence of data driven methods provides a disciplined approach through the problem of data abstraction in algorithmic oriented languages.
* It has resulted in the development of object-based

language, supporting only data abstraction.

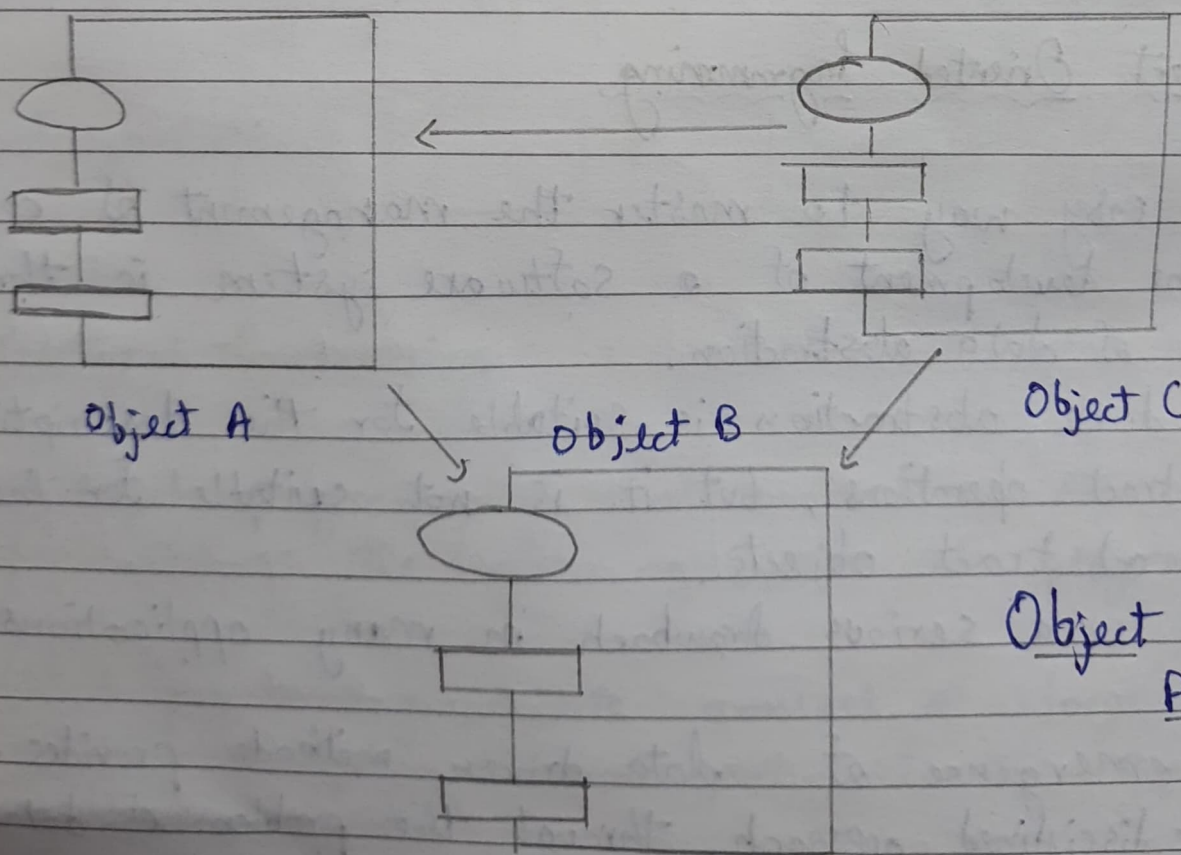
* Object based languages do not support features such as inheritance & polymorphism.

Depending on the object features supported, the language are classified into two categories:-

i) Object based programming languages.
ii) Object Oriented programming languages.

OOBL = Encapsulation + Object Identity

OOPL = OBL features + Inheritance + Polymorphism



Object Oriented Programming

OOP	POP
i) A programming paradigm based on the concept of objects which contains data in the form of fields known as attributes, and code in the form of procedure known as methods	A programing paradigm that is based on the concept of procedure calls.
ii) It stands for object oriented Programming.	ii) It stands for Procedure oriented Programming.
iii) Emphasis on objects.	iii) Emphasis on functioning.
iv) Divides the program into multiple objects.	iv) Divides the program into multiple function.
v) Modification is easier as objects are independent.	v) Modification are difficult as they can affect the entire Program.
vi) objects communicate with each other by passing messages.	vi) Functions communicate with each other by passing parameters.
vii) Each object controls its own data.	vii) Functions share global variable
viii) It is possible to hide data.	viii) There is no data hiding mechanism.
ix) It has access specifiers.	ix) Do not have access specifiers.
x) Supported by C++, Java, Python.	x) Supported by C, Pascal, FORTRAN, COBOL

→ access specifier → by default private.
→ Main() mai class ke funcs ko call karne ke liye objects banate hai

⇒ example of class & obj.

```
class Abc {
    int a,b,c;
    public void add() { cout << "Enter value of a & b";
                        cin >> a >> b;
                        c = a+b;
                        cout << "Sum is = " << c; }
};   // class ki defination ke baad semicolon aata hai

void main() { Abc obj;        // Abc = class name, obj = object name
              obj.add(); }
```

```
class Student { int age, marks, roll_no;

public void print_student_details() {
        cout << "Age = " << age << endl;
        cout << "Roll No. = " << roll_no << endl;
        cout << "Marks = " << marks << endl; }

public void set_student_details() {
        cout << "Enter student's age, roll no. & marks";
        cin >> age >> roll_no >> marks; }
};

int main() { Student harish;
             harish.set_student_details();
             harish.print_student_details(); }
```

Array

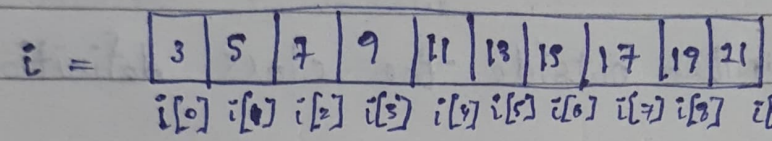
An array is used to store a collection of data.

Or

An array is a collection of variables of the same type.

Syntax → type arrayName [arraySize];

eg → int i[10];




```
// example program
int main () {  int number[6] = {8, 3, 11, 15, 31, 62};

    cout << "The numbers are ";
for (const int &n : number) { cout << n << "  ";}

  cout << "\n The numbers are ";

for (int i=0 ; i<6 ; i++ ) { cout << number [i] << "  ";}
  return 0;
}
```

Scope Resolution Operator (::)

It is used to define a function outside the class and declare inside the class.

eg→ class Hi { public : void func(); } ;

void Hi :: func() { cout << " func() called " ; }

int main () { Hi h ;
h.func();
return 0; }

Inline function with class.

→ Syntax :- // [written outside the class definition]

inline returnType className :: functionName (formalParameters)
{ function body }

→ When to use inline function:-
i> If a function is very small.
ii> If the time spent to function call is more than the function body execution time.
iii> If function call is frequent.
iv> If fully developed & tested program is running slowly.

Constructor

* Constructors are special member functions in a class.
* They have the same ~~name~~ name as the class.
* They do not have a return type.
* They are used to initialize an object in memory.

* They are of three types:-
(i) Default constructor (ii) Parametrized constructor (iii) Copy const.

Inheritance // reusability of code
Types:-

i) Single Inheritance
ii) Multilevel Inheritance
iii) Multiple Inheritance
iv) Hierarichal Inheritance
v) Hybrid Inheritance

~~Cons~~
Parametrized Constructor

```
class ParamA { private int b,c;
               public:
               ParamA(int b1, c1) { b=b1;
                                    c=c1; }

    int getX() { return b; }
    int getY() { return c; }
};

int main() { ParamA P1(10,15);
    cout << "P1.b=" << P1.getX() << "P1.C=" << P1.getY();
    return 0;
}
```

// Multilevel Inheritance.

```
class Base{
   public:
   void show1() { cout << "This is base class "; }
};

class Derived1 : public Base {
      public:
      void show2() { cout<< "This derived 1 class "; }
   };

class Derived2: public Base Derived1 {
   public:
   void show3() { cout<< "This derived 2 class " ; }
   };

int main() { Derived1 obj ;
             obj.show1 () ;
             obj.show2() ;
             obj.show3() ;
            return 0; }
```

Static data member

* Static data member a class member that are declared using 'static' keyword.
* There is only one copy of static data member in the class, even if there are many class objects. This is

because all the objects share static datamember.

* The static datamember is always initialized to 0, when the first class object is created.

static keyword

* static keyword has different meanings when used with different types. We can keyword with:

* static variable in functions: When a variable is declared as static. It gets memory space for the lifetime of the program.

* static member variable in class: static member variables (data members) are not initialized using constructor, because these are not dependent on object initialization.

* static function in class: It can access only static data members and calling member function directly with class name, because these are not dependent on object initialization.

* Program:-

```
// static function in class
class X{
    public:
    static void f() { // can access only static data members}
};
int main() { X::f();
    // calling member function directly with class name, no need to create objects.
}
```

Copy Constructor

* It is used to create a copy of an object.
* It is used when the compiler has to create a temporary object of a class object.
* Copy constructor requires at least one argument with reference to an object of a class & copy its argument into it.
* It is used to declare & initialize an object using reference of another variable.
* Program →

```
class CS {
    int i;
public:
    CS() // default
    { i = 10; }
    CS (CS &obj1) // copy
    { i = obj1.i; }

    void show() { cout << i; }
};

int main() { CS obj1;
             CS obj2(obj1);
             obj1.show();
             obj2.show();
           }
```

Abstract Class (3rd unit)

* Sometimes, implementation of all function can not be provided in a base class because we don't know the implementation. Such a class is called abstract class.

eg → *Let Shape be a base class.
*we cannot provide implementation of function draw() in Shape, but we know every derived class must have implementation of draw()
* Similarly, an Animal class doesn't have implementation of move() (assuming that all animals move) but all animals must know how to move.

* We cannot create objects of abstract classes.

syntax →
```
class Test {
    public:
        virtual void show() = 0; // pure virtual function
};
```

Pure virtual function

* A pure virtual function is implemented by classes which are derived from an abstract class.

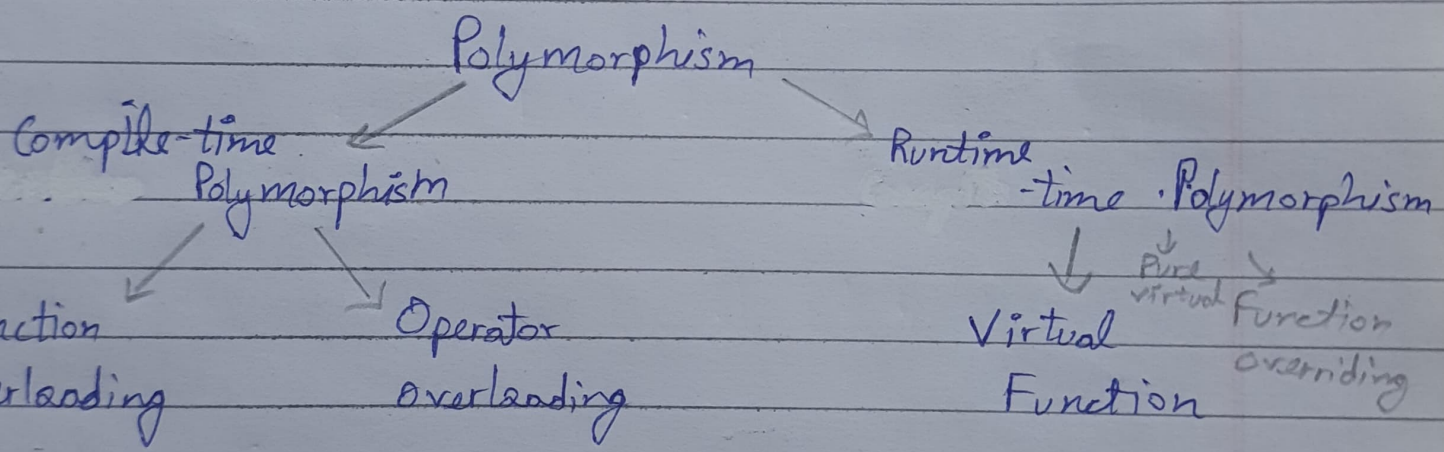
⇒
```
class Base { //Abstract class
    int x;
public:
    virtual void fun() = 0;
    int getX() { return x; }
};

class Derived : public Base {
    int y;
public : void fun() {
            cout << "fun() called";
        }
};

int main() {
    Derived d;
    d.fun();
    return 0;
}
```

output → fun() called

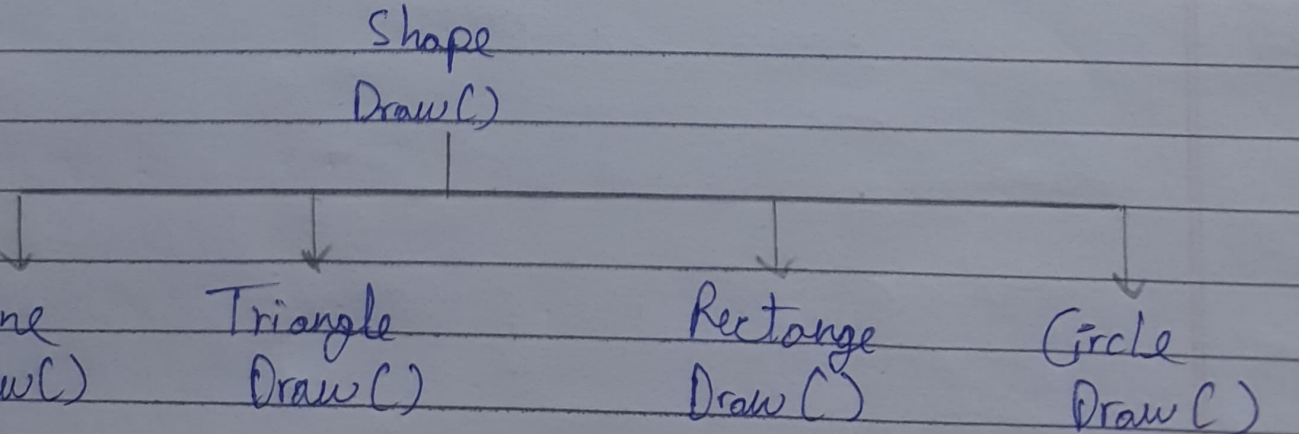
Polymorphism

* The word "Polymorphism" means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

* A real-life example of polymorphism is a person who at the same time can have different characteristics like, a man at the same time, may be a father, a husband and an employee. So, the same person exhibits different behaviour in different situations. This is called Polymorphism.
* Polymorphism is considered as one of the important feature of object-oriented programming.
* Types of polymorphism :-
  i) Compile-time polymorphism
  ii) Run-time polymorphism



Compile time polymorphism :-



Operator Overloading program

```
class CS {
    int a,b,c,d;
    public:
    CS (int p, int q, int r, int s)
    { a=p; b=q; c=r; d=s; }

    void show() { cout << a << b << c << d << endl; }
      // prefix increment overloading
    void operator ++ () { a++; b++; c++; d++; }

      // postfix increment overloading
    void operator ++ (int) { a++; b++; c++; d++; }

};

int main() {
    CS obj(1,2,3,4);
    cout << "Before Increment ";
    obj.show();
    obj++;
    ++obj;
    cout << "After increment ";
    obj.show();
    return 0; }
```

Output → Before Increment 1234
After increment 3456

Virtual Function

* A virtual function is a member function which is declared within the base class and is redefined (overridden) by a derived class. *
* When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function of that object & execute the derived class's version of the function.

i) Virtual functions ensure that the correct function is called for an object regardless of the type of reference (or pointer) used for function call.

ii) They are mainly used to achieve runtime polymorphism.

iii) Function are declared with a virtual keyword in base class.

iv) The resolving of function call is done at runtime.

Rules for virtual function:

1) Virtual function cannot be static.
2) A virtual function can be a friend function of another class.
3) Virtual functions should be accessed using pointer or reference of base class to achieve ru

4) The prototype of virtual func should by same in the base as well as derived class.

5) They are always defined in base class & overridden in a derived class.

It is not mandatory for derived class to overide (or re-define) the virtual function, in that case the base class version of func is used.

6) A class may have virtual destructor but it cannot have virtual constructor.

```
class Base {
  public:
     virtual void print() { cout << " print base class \n"; }

    void show() { cout << " show base class \n"; }
};

class Derived : public Base {
  public:
     void print() { cout << " print derived class \n"; }
    void show () { cout << " show derived class \n"; }
};

int main () { Base *bptr;
              Derived d;
              bptr = &d;
              bptr -> print ();
```

```
    bptr -> show();
    return 0;
}
```

Output → Print derived class
Show base class

Pure Virtual Function

* A pure virtual function is a virtual function in C++ for which we need not to write any function definition & only we have to declare it.
* It is declared by assigning 0 in the declaration.
* If there is no same name function in base class so we create pure virtual function for that base class.

Syntax: `virtual returnType functionName() = 0;`

Or

`virtual returnType functionName() { }`

```
class Base {
    public:
    virtual void showData() { } // Pure Virtual Function
};

class Derived : public Base {
        public:
        void showData() { cout << "Derived Class" << endl; }
};
```

```
int main(){
    Base obj1;
    Derived obj2;
    Base *bptr;
    bptr = &obj1;
    bptr->showData(); // base class function is called

    bptr = &obj2;
    bptr = showData(); // outputs Derived Class!

    obj1.showData(); // base class func called.

}
```

Function Overridding

* If derived class defines same function as defined in its base class, it is known as function overriding.
* In C++, It is used to achieve runtime polymorphism

```
class Animal {
    public:
    void speak() { cout << "This animal produces Sound!" << endl; }
};

class Cat : public Animal {
    public:
    void speak() { cout << "Meow!" << endl; }
};

class Dog : public Animal {
    public:
    void speak() { cout << "Dog!" << endl; }
};

int main() {
    Cat emi;
    Dog jack;
    emi.speak();
    jack.speak();
}
```

Output:

```
Meow!
Dog!
```

'this' Pointer in C++

To understand this pointer, it is important to know how objects look at functions and data members of a class.

i) Each object gets its own copy of the data member
ii) All access the same function definition as present in the code segment.

Meaning, each object gets its own copy of data members & all object share a single copy of member functions.

Now, question is that if only one copy of each member function exists and is used by multiple objects, How are the proper data members accessed & updated?

The compiler supplies an implicit pointer along with the names of the functions as 'this'.

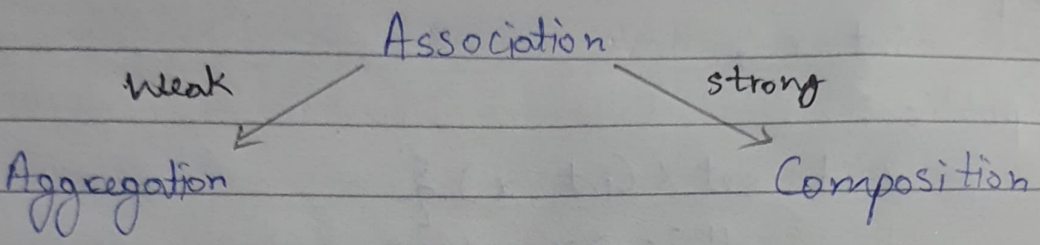
The 'this' pointer is passed as a hidden argument or non-static member function calls and is available as a local variable within the body of all non-static function.

'this' pointer is not-available in static member function as static member functions can be called without any object (with class name)

```
class Test {
    private:
        int x;
    public:
        void setX (int x) {
            this->x = x;
        }
        void print () {
            cout << "x is = " << x << endl;
        }
};

int main () {
    Test obj;
    int x = 20;
    obj.setX (x);
    obj.print ();
    return 0;
}
```

Association

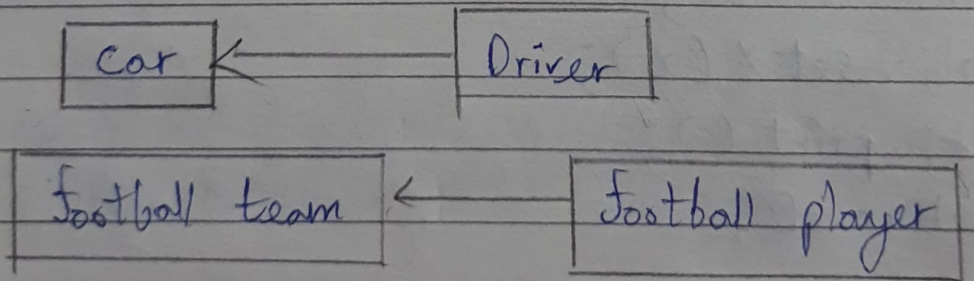
Association depicts the relationship between two classes.
It is of two types:



Aggregation:

Aggregation / weak association / loose coupling

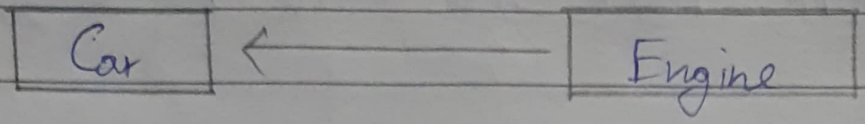
Two objects will have weak association / loose coupling when one object can live or exist without another object



Composition / strong association / Tight coupling

Two objects will have strong association / tight coupling when one object can not live or exist

without another object.



Friend function / Friendly function

The private members cannot be accessed from outside the class. i.e a non-member function cannot have an access to the private data of the class. However, there could be a situation where we would like two class to share a particular function.

eg → Consider a case where two classes 'Manager' & 'Scientist' have been defined. We would like to use a function incomeTax() to operate on the objects of both these classes.

In such situations, C++ allows the common function to be made friendly with both the classes thereby allowing the function to have access to the private data of these classes.

```
class sample {
    int a, b;
    public:
        void setValue() { a = 25; b = 40; }
        friend float mean (sample s);
};
```

```
float mean(sample s){
    return float(s.a + s.b)/2.0;
}

int main(){ sample X;
            X.setValue();
    cout << "Mean Value = " << mean(X) << "\n";
    return 0;
}
```

Friend class

* A friend class can access private & protected members of other class in which it is declared as friend.

* It is sometimes useful to allow a particular class to access private members of other class

```
class A {
    int x;
public:
    A(){ x = 10; }
    friend class B;
};

class B { public:
    void display(A &t){
        cout << endl << "The value of x = " << t.x;
    }
};
```

```
int main(){ A a;
            B b;
            b.display(a);
            return 0;
          }
```

Output :- The value of x = 10

```
class Largest{
    int a, b, m;
    public:
      void set_data();
      friend void find_max(Largest);
};

void Largest :: set_data(){
        cout << "Enter the First no: ";
        cin >> a;
        cout << "Enter the Second No: ";
        cin >> b;
}

void find_max(Largest t){
        if(t.a > t.b) t.m = t.a;
        else t.m = t.b
        cout << "Maximum Number is \t" << t.m;
    }
```

```
int main (){
    Largest l;
    l.set_data();
    Find_max (l);
    return 0;
}
```

Output :-

```
Enter the First no: 78
Enter the Second No: 532
Maximum Number is: 532
```

Exception Handling

Exception- When executing C++ code, different errors can occur: coding errors made by the programmer, errors due to wrong input or other things.

When an error occur C++ will normally stop & generate an error message.
The technical term for this is C++ will throw an error.

One of the advantages of C++ over C language is exception handling.
Exceptions are runtime anaMolies or abnormal conditions that a program encounters during its execution.

C++ provides the following a special keyword:
'try': represents a block of code that can throw an exception.

'Catch': represents a block of code that is executed when a particular exception is thrown.

'throw': used to throw an exception

There are two types of exception:
1) Synchronous exception
2) Asynchronous exception → beyond the control of program
eg → disk failure

```
double division (int a, int b) {
    if (b==0) {
        throw "Division by zero condition"
    }
    return (a/b);
}

int main() { int x=50;
             int y=0;
             double z=0;
    try {
        z= division(x,y);
        cout<<z << endl;
    }
    catch (const char *msg) {
        cout << msg << endl;
    }
    return 0;
}
```

Synchronous exception — The exceptions under the program control are known as synchronous exceptions.
eg → ArrayOut Of Bound
→ Arithematic Exception

Asynchronous Exception - Asynchronous exceptions are beyond the program control such as → Disk failure → Keyboard Interrupt

String in C++

* A string in C++ is a type of object representing a collection (or sequence) of different characters.
* Strings in C++ are a part of the standard string class (std :: string).
* The String class stores the characters of a string as a collection of bytes in ~~contigous~~ contiguous memory ~~long~~ location.

* Syntax: We use the string keyword to create a string in C++
However, we also need to include the standard string class in our program before using this keyword

```
string str_name = "This is a C++ String";
```

In C language,

```
char str_array[7] = {'s', 'c', 'a', 'l', 'e', 'r', '\0'};
```

Index of characters:

S	c	a	l	e	r	\0
0	1	2	3	4	5	6

Because we also have to ~~have~~ add a null ~~ed~~ character in the array, the array's length is 1 greater than the ~~leg~~ length of the string.
Alternatively, we can define the above string like

this as well, char str_array[] = "Scaler";

Then array str_array will still hold 7 characters because C++ automatically adds a null character at the end of the string.
If we use the sizeof() function to check the size of the array above, it will return 7 as well.

Different ways of defining a String:

using string keyword is the best and the most convenient way of defining a string. We should prefer using the string keyword because it is easy to write & understand.

```
string str_name = "hello";
```
or
```
string str_name("hello");
```

C style string

```
char str_name[6] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

Alternatively,
```
char str_name[] = "hello";
```
or
```
char str_name[6] = "hello";
```

```
char str_name[50] = "hello";
```

Example program :-

```
int main (){
    char str[3];
    cout << "Enter the name of your city:" << endl;
    cin >> str;
    cout << "your city is ; " << tr;
    return 0;
}
```

Concatenation of strings

~~Two or~~ Combining two or more strings to form a resultant string is called the concatenation of strings

If we wish to concatenate two or more strings, C++ provides us the functionality to do the same using strcat().

To use the strcat() function, we need to include the cstring header file in our program.

The strcat() function takes two character arrays as the input. It concatenates the second array to the end of the first array.

The syntax for strcat() is

strcat (char_array1, char_array2);

```
int main () {
    string str1 = "  ", str2 = "  ";
    cout << "Enter string 1: ";
    cin >> str1;
    cout << "Enter string 2: ";
    cin >> str2;
    string res = str1 + str2;
    cout << "The resultant string is =" << res;
}
```

```
int main () {
    char str1 = "Artificial";
    char str2 = "Intelligence";
    cout << "Concatenated string is: ";
    strcat (str1, str2);
    cout << str1;
    return 0;
}
```

Copying a string

Using the inbuilt function strcpy() from string.h header file to copy one string to the other we can copy the string.

strcpy() accepts a pointer to the destination array & source array as a parameter & after copying, it returns a pointer to the destination string.

Project Topics:

1) ATM system
2) Library Management system
3) Employee Management System
4) Hospital Management System
5) Hotel Management System
6) Attendance Management System
7) Store Management System

class diagrams
whole documentation written
2 minimum - 5 maximum

Parameter Passing.

There are different ways in which parameter data can be passed into and out of methods & functions

Terminology:
1) Formal parameters → void swap (int x, int y)

=> Actual parameters →

```
int main(){
    swap(4,5);
}
```

Formal parameter → A variable and its type as they appear in the prototype of the function or method.

Actual parameter → The variable or expression corresponding to a formal parameter that appears in the function or method call in the calling environment.

Important ways of parameter passing :-

1> Pass by Value:

This way is used in mode semantics. Changes made to formal parameter do not get transmitted back to the caller.

Any modifications to the formal parameters variable inside the called function /method affects only the separate storage location and will not be reflected in the actual parameter in the calling environment.

This way is also known as Call by value.

Eg→

```
void func(int a, int b){
    a += b;
    cout << "In function value of a & b: " << a << b;
}
```

```
int main(){
    int x=10, y=20;
    Func(x, y);
    cout << "The value of x & y:" << x << y;
    return 0;
}
```

2) Call/ Pass by reference:

In this technique, changes made to formal parameters do get transmitted back to the caller through parameter passing.

Any changes to formal parameter are reflected in the actual parameter in the calling environment as formal parameter ~~fct~~ receives a reference (or pointer) to the actual data.

This method is also called as Call by reference.
This method is efficient in both time & space.

eg→

```
Void swap(int *x, int *y){
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```

```
int main() {
    int x=300, y=200;
    swap (&x, &y);
cout << "Value of x=" << x << endl;
 cout << "Value of y=" << y;
    return 0;
}
```