**Notes**

**Benefits of Object Oriented**

• Ease of programming , maintainence , enhancements

• Map to real world entities

**Object Oriented Concepts**

**Major Elements of Object Oriented**

• **Abstraction**

Focus only on relevant details(data members and methods) and ignore/delete the non essential details from the class design

• **Encapsulation**

Binding of data members and methods into a single unit called class

• **Data Hiding**

Use of appropriate access specifiers(private or protected) to avoid the visibility and accessibility of class members outside the class design

• **Inheritance**

Designed for extendability and in the process of extending, we also wish to reuse the members of the base class with appropriate access specifiers

Inheritance establishes an Is-A relationship between the derived class and the base class

• **Polymorphism**

Many forms of behaviour of a given object is called polymorphism.It is classified into 2 types

• Static Polymorphism – demonstrated using method overloading concept

• Dynamic Polymorphism – demonstrated using method overriding concept

**Minor Elements of Object Oriented**

• **Class**

It is conceptual representation of an object and no memory is allocated for the data members during the class design

• **Object**

It is a real world entity having a well defined identity , state and behaviour.Size of an object is the sum of the memory of the non static data members in the class design. Static static members are allocated memory during the class loading phase.

Non static members are accessible only via an object and hence they have an object scope. Static members are accessible using the class name itself and hence they have a class scope

Idenity -> name of the class for which the object is created

-> Unique attribute in the class design as a data member(example StuID in a Student class) using which an object can be uniquely identified

->The bigger entity(example Organization) for which the object is a part(example Employee)

State -> values what we give for the data members when an object is created

Behaviour -> the methods in the class are the behaviours of an object

**Static Keyword**

* Data members marked as static are a single copy in memory and can be shared across te n number of instances of the class
* A static data member should be exposed outside the class for the preprocessor/class loading phase to recognize it and allocate the memory based on the its adta type . By default the static data member will be initialized with a default value and if need be , we can also initialize it.
* A static method in a class has 2 contraints

1) It cannot access the non static members of the class directly

2) We cannot use a this keyword to access the non static members within a static method

* Static members are class scope members and hence they can be accessed using the syntax classname::static member
* 2 methods which differ only in terms of static are not considered as overloaded methods since their scopes are different as shown below

static void disp(){ } //class scope

void disp(){ } //instance scope

* static data member can be prefixed with a const keyword in case we do not want the state of the data member to undergo a change
* Inner classes in C++ are implicitly prefixed with a static keyword and this cannot be removed programmatically
* If the inner class is marked as static , it does not treate the members of that class implicitly as static

**Const Keyword**

* A const keyword can be applied for both static as well as non static data members
* A state of a data member prefixed with a const keyword cannot undergo a change
* A method marked as const is said to be const method and within its scope the values of the data members cannot be chnaged unless we prefix a keyword mutable to the data member
* A combination of mutable and const can apply only for data members which are pointers
* A const object can invoke only const methods
* A non const object will first prefer to invoke a non-const method. But in case the non const method is not available it is capable of binding itself to the const method
* Two methods which differ only in terms of const keyword are considered as overloaded methods and the decision of the method binding is based on how the object is created.A constant object binds itself to a constant method while a non-const object binds itself with a non-const function
* If an instance data member is marked as a constant, it is mandatory to initialize it at the constructor initializer list
* If an static data member is marked as a constant, it is mandatory to initialize it outside the class and be exposed to the class loading stage
* A non constant object prefers to bind itself with a non-const function.But if a non-const method is not available,it is capable to bind itself to a constant method. But a constant object can bind only with a constant function

Example:

void display();

void display()const;

**Mutable Keyword**

* Mutable keyword is applicable only on instance data members
* Mutable keyword on a non-const instance data member enables the state of the member to undergo a change within a constant function
* The combination of mutable and const keyword on an instance data member can be applied only on pointer notations
* The application of a mutable keyword is relevant only on constant functions
* The mutable keyword cannot be used on methods/member functions

**Method Overloading Thumbrules**

* Methods are said to be overloaded if the below rules are followed

1) The names of the overloaded methods should be the same

2) The signature should be different

3) The parameter signature can defer either by the number of parameters , types of parameters or order of the parameters

4) The scope of the methods should be the same(i.e. they should be either static or non-static)

5) The binding of the method call with the method definition should happen at compile time

6) During the method call , importance is always given to exact match

7) If exact match is not there , then an implicitly convertable function prototype can be binded to the method call

8) During the method call , if the input data is casted to a different type , then importance will be given to casted information and accordingly the method call be binded to the casted method definition

9) A choice between and eact match versus and ellipse notation , importance is always given to an exact match

10) If the prototype of the exact match and ellipse notation turns out to be the same , it leads to error of ambiguity at compile time

**Namespaces**

* Namespaces are designed to avoid the clashes in the global scope
* Namespace cannot be designed for a local scope(example designing a namespace within a method is not allowed)
* A using directive can be applied to get access to the members of the namespace directly
* A using directive has no role to play in the presence of a local variable or a global variable
* Namespaces can be nested but make sure the names of the members in the namespaces are not the same as it leads to ambiguities if the using directive is applied on both the namespaces

**Function with default values**

* The default values should always be given in the order of right to left
* The default values can be specified either at the declaration or definition level and not at both the places
* Reinitialization of default values at different prototype levels is not allowed
* If all the parameters of a function is with a default value , we should not overload this method as it leads to redefinition and ambiguity error

**Reference Concepts**

* References are internally treated as pointers and does occupy memory
* References must be initialized
* The prototype of the reference should match the prototype of the original named location (If the original location is treated as a constance , it is mandatory that the reference to it should also be treated as a constant but the vice versa does not apply)
* After initialization is done, if the reference variable is assigned to another variable , it does not get updated to refer to the location but just picks up the value from the location and updates the original location value to which it was initialized.
* References can be looked upon as constant pointers internally
* References can be mapped to a pointer also and can be dereferenced

**Inheritance**

* The order of constructors is always the base constructor followed by the derived class constructor. The order of destructors is always the reverse of that of constructors
* The task of initializing the base class instance data members should be the responsibility of the base class itself. This task cannot be delicated to the derived class constructor though the base class instance data members are visible and accessible to the derived class constructor
* If an explicit call to base class constructor is not specified in the initialization list of the derived class constructor, implicitly there will be a call to the default constructor of the base class
* To demonstrate dynamic binding it is mandatory, to have virtual methods in the base class
* The overriding methods prototype should exactly match the prototype of the overridden method

**Exception handling**

* The keywords in C++ to handle an exception are try, catch and throw
* A try block should mandatorily be followed with a catch
* A try can have multiple catches but of the multiple catches, only one of the catch block whichever is appropriate will be considered and the other catches will be completely ignored
* If a try has multiple catches, it is mandatory that the generic catch is always the last catch
* If an exception has occurred which is not handled by your application, then the built in exception handler comes into existence but implicitly it invokes the terminate () method
* Try blocks can be nested and for exceptions that has occurred in the inner try, inner catch blocks will first be considered and if they are not appropriate, then the outer catches will be considered. If the outer catches are also not appropriate, then the exception propagates to the calling level