1. What is Object oriented Programming? Explain Encapsulation, Polymorphism and Inheritence?

* Object Oriented Programming:- ***O***bject-***o***riented ***p***rogramming (**OOP**) refers to a type of computer programming (software design) in which [programmers](https://www.webopedia.com/TERM/P/programmer.html) define not only the [data type](https://www.webopedia.com/TERM/D/data_type.html) of a [data structure](https://www.webopedia.com/TERM/D/data_structure.html), but also the types of operations ([functions](https://www.webopedia.com/TERM/F/function.html)) that can be applied to the data structure.
* The Basic feature of OOPS is [**Abstraction**](https://www.webopedia.com/TERM/A/abstraction.html)**:** The process of picking out (abstracting) common features of objects and procedures.
* [**Class**](https://www.webopedia.com/TERM/C/class.html)**:** A category of objects. The class defines all the common properties of the different objects that belong to it.
* [**Encapsulation**](https://www.webopedia.com/TERM/E/encapsulation.html)**:** The process of combining elements to create a new entity. A procedure is a type of encapsulation because it combines a series of computer instructions.
* [**Information hiding**](https://www.webopedia.com/TERM/I/information_hiding.html)**:** The process of hiding details of an object or function. Information hiding is a powerful programming technique because it reduces complexity.
* [**Inheritance**](https://www.webopedia.com/TERM/I/inheritance.html)**:** a feature that represents the "is a" relationship between different classes.
* [**Interface**](https://www.webopedia.com/TERM/I/interface.html)**:** the languages and codes that the applications use to communicate with each other and with the hardware.
* [**Messaging**](https://www.webopedia.com/TERM/M/message_passing.html)**:** Message passing is a form of communication used in parallel programming and object-oriented programming.
* [**Object**](https://www.webopedia.com/TERM/O/object.html)**:** a self-contained entity that consists of both data and procedures to manipulate the data.
* [**Polymorphism**](https://www.webopedia.com/TERM/P/polymorphism.html)**:** A programming language's ability to process objects differently depending on their data type or class.
* [**Procedure**](https://www.webopedia.com/TERM/P/procedure.html)**:** a section of a program that performs a specific task.
* Encapsulation:- **Data encapsulation** is a mechanism of bundling the data, and the functions that use them and **data abstraction** is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

E.g:- #include <iostream>

using namespace std;

class Adder {

public:

// constructor

Adder(int i = 0) {

total = i;

}

// interface to outside world

void addNum(int number) {

total += number;

}

// interface to outside world

int getTotal() {

return total;

};

private:

// hidden data from outside world

int total;

};

int main() {

Adder a;

a.addNum(10);

a.addNum(20);

a.addNum(30);

cout << "Total " << a.getTotal() <<endl;

return 0;

}

O/P:- Total 60

Polymorphism:-

The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a greek word. In object-oriented programming, we use 3 main concepts: inheritance, encapsulation and polymorphism.

**There are two types of polymorphism in C++:**

* **Compile time polymorphism:** It is achieved by function overloading and operator overloading which is also known as static binding or early binding.
* **Runtime polymorphism:** It is achieved by method overriding which is also known as dynamic binding or late binding.

**Inheritence:-**

In C++, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.

In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

1. E.g:- #include <iostream>
2. **using** **namespace** std;
3. **class** Account {
4. **public**:
5. **float** salary = 60000;
6. };
7. **class** Programmer: **public** Account {
8. **public**:
9. **float** bonus = 5000;
10. };
11. **int** main(**void**) {
12. Programmer p1;
13. cout<<"Salary: "<<p1.salary<<endl;
14. cout<<"Bonus: "<<p1.bonus<<endl;
15. **return** 0;
16. }

O/p:- Salary: 60000

Bonus: 5000

1. Define Class and Object.

Class:- A class definition starts with the keyword **class** followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations. For example, we defined the Box data type using the keyword **class** as follows −

class Box {

public:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

The keyword **public** determines the access attributes of the members of the class that follows it. A public member can be accessed from outside the class anywhere within the scope of the class object. You can also specify the members of a class as **private** or **protected** which we will discuss in a sub-section.

Object:- A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class Box −

Box Box1; // Declare Box1 of type Box

Box Box2; // Declare Box2 of type Box

Both of the objects Box1 and Box2 will have their own copy of data members.

1. Define Friend Function.

If a function is defined as a friend function in C++ then the protected and private data of a class can be accessed using the function.

By using the keyword **friend** compiler knows the given function is a friend function.

For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword **friend.**

Declaration of friend function in C++

1. **class** class\_name
2. {
3. **friend** data\_type function\_name(argument/s);
4. };

C++ friend function Example

Let's see the simple example of C++ friend function used to print the length of a box.

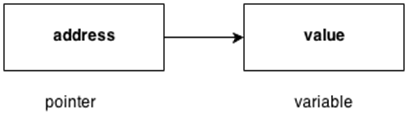
1. #include <iostream>
2. **using** **namespace** std;
3. **class** Box
4. {
5. **private**:
6. **int** length;
7. **public**:
8. Box(): length(0) { }
9. **friend** **int** printLength(Box); //friend function
10. };
11. **int** printLength(Box b)
12. {
13. b.length += 10;
14. **return** b.length;
15. }
16. **int** main()
17. {
18. Box b;
19. cout<<"Length of box: "<< printLength(b)<<endl;
20. **return** 0;
21. }

Output:

Length of box: 10

1. Write a program to read three numbers x, y and z and evaluate R given by R=z(x-y). Use exception handling to throw an exception in case of division by zero is attempted.
2. What is Pointer in C++? Explain ‘This pointer’.

Pointer: - The pointer in C++ language is a variable, it is also known as locator or indicator that points to an address of a value.



This Pointer: - In C++ programming, **this** is a keyword that refers to the current instance of the class. There can be 3 main usage of this keyword in C++.

* It can be used **to pass current object as a parameter to another method.**
* It can be used **to refer current class instance variable.**
* It can be used **to declare indexers.**

1. Explain reference parameters.

Reference parameters are useful in two cases:

**Change values**. Use a reference parameter when you need to change the value of an actual parameter variable in the call. When a function computes only one value it is considered a better style to return the value with the return statement. However, if a function produces more than one value, it is common to use reference parameters to return values, or a combination of the return value and reference parameters.

**Efficiency**. To pass large structures more efficiently. This is especially common for passing structs or class objects. If no changes are made to the parameter, it is should be declared const.

**Reference parameters pass an *address*, not a *value***

When you declare a reference parameter, the function call will pass the *memory address* of where the actual parameter, instead of copying the parameter value into the formal parameter.

**Declare reference parameters with a &**

To indicate a reference parameter, an ampersand (&) is written in the function prototype and header after the parameter type name. For example,

void assign(int**&** to, int from) {

to = from; // Will change the actual parameter in the call.

}

has two parameters, *to* is a reference parameter as indicated by the ampersand, and *from* is a value parameter. This ampersand must be in both the prototype and the function header.

1. What are the two dynamic allocation operator in C++? Explain.

* The two dynamic allocation operator in C++ is new() and delete().

## Dynamic Memory Allocation

* We can dynamically allocate storage space while the program is running, but we cannot create new variable names "on the fly"
* For this reason, dynamic allocation requires two steps:
  1. Creating the dynamic space.
  2. Storing its **address** in a **pointer** (so that the space can be accesed)
* To dynamically allocate memory in C++, we use the **new** operator.
* De-allocation:
  1. Deallocation is the "clean-up" of space being used for variables or other data storage
  2. Compile time variables are automatically deallocated based on their known extent (this is the same as scope for "automatic" variables)
  3. It is the programmer's job to deallocate dynamically created space
  4. To de-allocate dynamic memory, we use the **delete** operator

### Allocating space with new

* To allocate space dynamically, use the unary operator **new**, followed by the *type* being allocated.
* new int; // dynamically allocates an int
* new double; // dynamically allocates a double
* If creating an array dynamically, use the same form, but put brackets with a size after the type:
* new int[40]; // dynamically allocates an array of 40 ints
* new double[size]; // dynamically allocates an array of size doubles
* // note that the size can be a variable
* These statements above are not very useful by themselves, because the allocated spaces have no names! BUT, the new operator returns the starting address of the allocated space, and this address can be stored in a pointer:
* int \* p; // declare a pointer p
* p = new int; // dynamically allocate an int and load address into p
* double \* d; // declare a pointer d
* d = new double; // dynamically allocate a double and load address into d
* // we can also do these in single line statements
* int x = 40;
* int \* list = new int[x];
* float \* numbers = new float[x+10];

Notice that this is one more way of *initializing* a pointer to a valid target (and the most important one).

### Accessing dynamically created space

* So once the space has been dynamically allocated, how do we use it?
* For single items, we go through the pointer. Dereference the pointer to reach the dynamically created target:
* int \* p = new int; // dynamic integer, pointed to by p
* \*p = 10; // assigns 10 to the dynamic integer
* cout << \*p; // prints 10
* For dynamically created arrays, you can use either pointer-offset notation, or treat the pointer as the array name and use the standard bracket notation:
* double \* numList = new double[size]; // dynamic array
* for (int i = 0; i < size; i++)
* numList[i] = 0; // initialize array elements to 0
* numList[5] = 20; // bracket notation
* \*(numList + 7) = 15; // pointer-offset notation
* // means same as numList[7]

### Deallocation of dynamic memory

* To deallocate memory that was created with new, we use the unary operator **delete**. The one operand should be a pointer that stores the address of the space to be deallocated:
* int \* ptr = new int; // dynamically created int
* // ...
* delete ptr; // deletes the space that ptr points to

Note that the pointer ptr *still exists* in this example. That's a named variable subject to scope and extent determined at compile time. It can be reused:

ptr = new int[10]; // point p to a brand new array

* To deallocate a dynamic array, use this form:
* delete [] *name\_of\_pointer*;

Example:

int \* list = new int[40]; // dynamic array

delete [] list; // deallocates the array

list = 0; // reset list to null pointer

After deallocating space, it's always a good idea to reset the pointer to null unless you are pointing it at another valid target right away.

* **To consider:** So what happens if you fail to deallocate dynamic memory when you are finished with it? (i.e. why is deallocation important?)

## Application Example: Dynamically resizing an array

If you have an existing array, and you want to make it bigger (add array cells to it), you cannot simply append new cells to the old ones.  Remember that arrays are stored in consecutive memory, and you never know whether or not the memory immediately after the array is already allocated for something else.   For that reason, the process takes a few more steps.  Here is an example using an integer array.  Let's say this is the original array:

int \* list = new int[size];

I want to resize this so that the array called **list** has space for 5 more numbers (presumably because the old one is full).   
There are four main steps.

1. Create an entirely new array of the appropriate type and of the new size. (You'll need another pointer for this).
2. int \* temp = new int[size + 5];
3. Copy the data from the old array into the new array (keeping them in the same positions). This is easy with a for-loop.
4. for (int i = 0; i < size; i++)
5. temp[i] = list[i];
6. Delete the old array -- you don't need it anymore! (Do as your Mom says, and take out the garbage!)

delete [] list; // this deletes the array pointed to by "list"

1. Change the pointer. You still want the array to be called "list" (its original name), so change the list pointer to the new address.
2. list = temp;

That's it! The list array is now 5 larger than the previous one, and it has the same data in it that the original one had. But, now it has room for 5 more items.

1. Write a function template for sorting list of arrays?
2. Give an example to Inherit multiple base classes.

C++ Multi Level Inheritance Example

When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C++. Inheritance is transitive so the last derived class acquires all the members of all its base classes.

Let's see the example of multi level inheritance in C++.

1. #include <iostream>
2. **using** **namespace** std;
3. **class** Animal {
4. **public**:
5. **void** eat() {
6. cout<<"Eating..."<<endl;
7. }
8. };
9. **class** Dog: **public** Animal
10. {
11. **public**:
12. **void** bark(){
13. cout<<"Barking..."<<endl;
14. }
15. };
16. **class** BabyDog: **public** Dog
17. {
18. **public**:
19. **void** weep() {
20. cout<<"Weeping...";
21. }
22. };
23. **int** main(**void**) {
24. BabyDog d1;
25. d1.eat();
26. d1.bark();
27. d1.weep();
28. **return** 0;
29. }

Output:

Eating...

Barking?

Weeping?

Virtual Inheritence:- **Virtual inheritance** is a [C++](https://en.wikipedia.org/wiki/C%2B%2B) technique that ensures only one copy of a [base class](https://en.wikipedia.org/wiki/Base_class)'s member variables are [inherited](https://en.wikipedia.org/wiki/Inheritance_(computer_science)) by grandchild derived classes.

X

^ ^

/ \

virtual / \ virtual

A B

^ ^

\ /

\ /

C

Without virtual inheritance, if classes *A* and *B* both inherit from class *X*, and class *C* inherits from classes *A* and *B*, then class *C* will contain two copies of *X'*s member variables: one via *A*, and one via *B*. These will be accessible independently, using [scope resolution](https://en.wikipedia.org/wiki/Scope_resolution_operator).

Instead, if classes *A* and *B* inherit virtually from class *X*, then objects of class *C* will contain only one set of the member variables from class *X*.

We can re-declare our classes as follows:

**struct** Animal {

**virtual** ~Animal() { }

**virtual** void eat(){};

};

*// Two classes virtually inheriting Animal:*

**struct** Mammal : **virtual** Animal {

**virtual** void breathe(){};

};

**struct** WingedAnimal : **virtual** Animal {

**virtual** void flap(){};

};

*// A bat is still a winged mammal*

**struct** Bat : Mammal, WingedAnimal {

};