OOPS

1. **Dynamic Binding:** In dynamic binding, the code to be executed in response to function call is decided at runtime. C++ has virtual functions to support this.

2. **Access Modifiers:**
   - Access modifiers are used to implement an important aspect of Object-Oriented Programming known as Data Hiding.
   - There are 3 types of access modifiers available in C++:
     1. **Public:** All the class members declared under the public specifier will be available to everyone. The data members and member functions declared as public can be accessed by other classes and functions too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.
     2. **Private:** The class members declared as private can be accessed only by the member functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of a class.
     3. **Protected:** Protected access modifier is similar to private access modifier in the sense that it can't be accessed outside of it's class unless with the help of friend class, the difference is that the class members declared as Protected can be accessed by any subclass (derived class) of that class as well.
   - **Note:** If we do not specify any access modifiers for the members inside the class then by default the access modifier for the members will be Private.
   - **Note**: This access through inheritance can alter the access modifier of the elements of base class in derived class depending on the modes of Inheritance.

3. **Structure vs Class:**
   - In C++, a *structure* is the same as a *class* except for a few differences.
   - The most important of them is security. A Structure is not secure and cannot hide its implementation details from the end-user while a class is secure and can hide its programming and designing details.
   - Members of a class are private by default and members of a structure are public by default.
   - When deriving a struct from a class/struct, the default access-specifier for a base class/struct is public. And when deriving a class, the default access specifier is private.
   - Class can have null values, but the structure can not have null values.
   - Memory of structure is allocated in the stack while the memory of class is allocated in heap.
   - Class requires constructor and destructor, but the structure does not require it.

- Classes support polymorphism and also be inherited but the structure cannot be inherited.

4. Difference between C and C++ structures:

- **Member functions inside structure**: Structures in C cannot have member functions inside structure, but Structures in C++ can have member functions along with data members.
- **Direct Initialization:** We cannot directly initialize structure data members in C, but we can do it in C++.
- **Using struct keyword:** In C, we need to use struct to declare a struct variable. In C++, struct is not necessary. For example, let there be a structure for Record. In C, we must use "struct Record" for Record variables. In C++, we need not use struct and using 'Record' only would work.
- **Static Members:** C structures cannot have static members but is allowed in C++.
- **Constructor creation in structure**: Structures in C cannot have constructor inside structure, but Structures in C++ can have Constructor creation.
- **sizeof operator:** This operator will generate 0 for an empty structure in C whereas 1 for an empty structure in C++.
- **Data Hiding:** C structures do not allow concept of Data hiding but is permitted in C++.
- **Access Modifiers:** C structures do not have access modifiers as these modifiers are not supported by the language. C++ structures can have this concept as it is inbuilt in the language.

5. Can a class have an object of self-type?

- A class declaration can contain static object of self-type, it can also have pointer to self-type, but it cannot have a non-static object of self-type.

```
#include<iostream>

// A class can have a static member of self-type
class Test
{
    static Test self;   // works fine
};

// A class can have a pointer to self-type
class Test
{
    Test* self;       // works fine
};
```

```
// A class cannot have non-static object(s) of self-type.
class Test
{
    Test self;          // error
};

int main()
{
    Test t;
    return 0;
}
```

- If a non-static object is member then declaration of class is incomplete, and compiler has no way to find out size of the objects of the class.
- Static variables do not contribute to the size of objects. So, no problem in calculating size with static variables of self-type.
- For a compiler, all pointers have a fixed size irrespective of the data type they are pointing to, so no problem with this also.

6. Why is the size of an empty class not zero?

- When structure was introduced in C, there was no concept of Objects that time. So according to C standard, it was decided to keep size of empty structure to zero.
- In C++, Size of empty structure/class is of one byte as to call function at least empty structure/class should have some size (minimum 1 byte is required) i.e. one byte to make them distinguishable.
- Empty class means, a class that does not contain any data members (e.g. int a , float b, char c and string d etc.) However, an empty class may contain member functions.
- Simply a class without an object requires no space allocated to it. The space is allocated when the class is instantiated. So, to an object of an empty class, 1 byte is allocated by compiler, for it's unique address identification.
- If a class have multiple objects they can have different unique memory location. Suppose, if a class does not have any size, what would be stored on the memory location? That's the reason when we create an object of an empty class in C++ program, it needs some memory to get stored, and the minimum amount of memory that can be reserved is 1 byte. Hence, if we create multiple objects of an empty class, every object will have unique address.

7. Static data members:

- Static data member are class members that are declared using static keyword.
- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.

- They are initialized to zero when the first object of its class is created. No other initialization is permitted.
- They are visible only within the class, but its lifetime is the entire program.
- Static members are only declared in class declaration, not defined. They must be explicitly defined outside the class using scope resolution operator. If we try to access static member without explicit definition of it, we will get compilation error.
- Also, static members can be accessed without any object, by using scope resolution operator.

8. Static member function:

- Static member functions do not have *this pointer*.
- A static member function cannot be *virtual*.
- Member function declarations with the same name and the name parameter-type-list cannot be overloaded if any of them is a static member function declaration.
- A static member function can not be declared const, volatile, or const volatile.

9. ‘this’ pointer:

- In ‘*this’* pointer each object gets its own copy of the data member and all-access the same function definition as present in the code segment.
- Meaning each object gets its own copy of data members and all objects share a single copy of member functions.
- The compiler supplies an implicit pointer along with the names of the functions as ‘this’.
- The ‘this’ pointer is passed as a hidden argument to all non-static member function calls and is available as a local variable within the body of all non-static functions.
- ‘this’ pointer is not available in static member functions as static member functions can be called without any object (with class name).
- For a class X, the type of this pointer is ‘X*’. Also, if a member function of X is declared as const, then the type of this pointer is ‘const X *’
- When local variable’s name is same as member’s name then we can use ‘this’ to assign value of local variable to member variable.
- C++ lets object destroy themselves by calling the following code:
```
delete this;
```
- Ideally *delete* operator should not be used for *this* pointer. However, if used, then following points must be considered.
    - *delete* operator works only for objects allocated using operator *new*. If the object is created using new, then we can do *delete this*, otherwise behavior is undefined.
    - Once *delete this* is done, any member of the deleted object should not be accessed after deletion.

10. Local class:

- A class declared inside a function becomes local to that function and is called Local Class in C++.
- A local class type name can only be used in the enclosing function.
- All the methods of Local classes must be defined inside the class only.
- A Local class cannot contain static data members. It may contain static functions though.
- Member methods of local class can only access static and enum variables of the enclosing function. Non-static variables of the enclosing function are not accessible inside local classes.
- Local classes can access global types, variables and functions. Also, local classes can access other local classes of same function.

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

void fun()
{
      class Test  // local to fun
      {
        /* members of Test class */
      };
}

int main()
{
    return 0;
}
```

11. Friend class:

- A friend class can access private and protected members of other class in which it is declared as friend.
- A friend function can be:
  - A member of another class .
  - A global function.
- Friendship is not mutual. If class A is a friend of B, then B doesn't become a friend of A automatically.
- Friendship is not inherited.
- It can be declared either in the private or the public part.
- The function definition does not use either the keyword friend or scope resolution operator.

12. Opaque Pointer:

- Opaque as the name suggests is something we can't see through.
- Opaque pointer is a pointer which points to a data structure whose contents are not exposed at the time of its definition.
- Following pointer is opaque. One can't know the data contained in STest structure by looking at the definition.

```
struct STest* pSTest;
```

- It is safe to assign NULL to an opaque pointer.

```
pSTest = NULL;
```

13. References:

- When a variable is declared as a reference, it becomes an alternative name for an existing variable.
- A variable can be declared as a reference by putting '&' in the declaration.
- A reference can be thought of as a constant pointer (not to be confused with a pointer to a constant value!) with automatic indirection, i.e the compiler will apply the ***** operator for us.
- **Example:**

Input:

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

int main()
{
    int x = 10;

    // ref is a reference to x
    int& ref = x;

    // Value of x is now changed to 20
    ref = 20;
    cout << "x = " << x << endl ;

    // Value of x is now changed to 30
    x = 30;
    cout << "ref = " << ref << endl ;

    return 0;
}
```

Output:

```
x = 20
ref = 30
```

- **Applications:**
  - **To modify local variables of the caller function:** A reference (or pointer) allows called function to modify a local variable of the caller function.
  - **For passing large sized arguments:** If an argument is large, passing by reference (or pointer) is more efficient because only an address is really passed, not the entire object.
  - **To avoid Object Slicing:** If we pass an object of subclass to a function that expects an object of superclass then the passed object is sliced if it is pass by value.
  - **To achieve Run Time Polymorphism in a function:** We can make a function polymorphic by passing objects as reference (or pointer) to it.

- **References vs Pointers:**
  - A pointer can be declared as void but a reference can never be void.

    ```
    int a = 10;
    void* aa = &a;    // it is valid
    void &ar = a;      // it is not valid
    ```
  - Reference variable is an *internal pointer* .
  - **Reassignment:** A pointer can be re-assigned. This property is useful for implementation of data structures like linked list, tree, etc. On the other hand, a reference cannot be re-assigned, and must be assigned at initialization.
  - **Memory address:** A pointer has its own memory address and size on the stack whereas a reference shares the same memory address (with the original variable) but also takes up some space on the stack.
  - **NULL value:** Pointer can be assigned NULL directly, whereas reference cannot. The constraints associated with references (no NULL, no reassignment) ensure that the underlying operations do not run into exception situation.
  - **Indirection:** You can have pointers to pointers offering extra levels of indirection. Whereas references only offer one level of indirection.
  - **Arithmetic operations:** Various arithmetic operations can be performed on pointers whereas there is no such thing called Reference Arithmetic. (but we can take the address of an object pointed by a reference and do pointer arithmetics on it as in &obj + 5).
- **Can references refer to invalid location in C++?**
  - In C++, Reference variables are safer than pointers because reference variables must be initialized and they cannot be changed to refer to something else once they are initialized. But there are exceptions where we can have invalid references.

    a) Reference to value at uninitialized pointer.

    ```
    int *ptr;
    int &ref = *ptr;  // Reference to value at some random memory location
    ```

    b) Reference to a local variable is returned.

    ```
    int& fun()
    {
      int a = 10;
    ```

```
  return a;
}
```

Once fun() returns, the space allocated to it on stack frame will be taken back. So, the reference to a local variable will not be valid.

14. Default Arguments:

- A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the caller of the function doesn't provide a value for the argument with a default value.
- When Function overloading is done along with default values. Then we need to make sure it will not be ambiguous. The compiler will throw error if ambiguous.
- **Key Points:**
    - Default arguments are different from constant arguments as constant arguments can't be changed whereas default arguments can be overwritten if required.
    - Default arguments are overwritten when calling function provides values for them.
    - During calling of function, arguments from calling function to called function are copied from left to right.
    - Once default value is used for an argument in function definition, all subsequent arguments to it must have default value. It can also be stated as default arguments are assigned from right to left.