- As many as possible MCQs from GFG Topic-wise sections
- YouTube lecture series

- Empty classes have size = 1 (To identify the same uniquely)
- The default return type in CPP is *INT.*
- For all non-static integer variables, the default value is garbage. For all the static and global variables, the same is set to 0.
- Function declarations can provide default values as well. Default arguments must follow non-default values.

```
C/C++
void foo (int, int = 10)

int main() {
      foo(5); // Output: 15
}

void foo (int a, int b) { cout << a + b << "\n"; }
```

- While evaluating expressions with logical operators, remember that they are L2R associative and have SHORT-CIRCUITING built into them.
- *Switch* statements match the first condition in the clause and allow fall through to other clauses if the *BREAK* keyword is not used.
- Bitwise operators like (<<, >>) have precedence even lower than (+ -). Thus, we need to be careful when evaluating expressions involving them.
- All major operators are L2R associative, expect
  - Increment (++)
  - Decrement (--)
  - Logical and Bitwise NOT (! ~)
  - Ternary (? :)
  - Assignment and Compound Assignment (= += -= …)
- :: when used with a variable name, it refers to the global scope.
- *printf* returns the number of characters printed by the same by default.

- New keyword calls the constructor for the class, while malloc does not.

- To restrict the dynamic allocation of a class, we can declare private *new* and *new[]* operators so that the same can not be constructed. Similar techniques are also used with private copy constructors to disable shallow copying.

- A class can not have the destructors of any other class (even if it derives them).
- The destructors are always called in the reverse order of the calling of the constructors.
- When inheriting, the constructors of the parent classes are called first (in the correct order), and then the class's constructor is called.
- If you define any constructor for the class, then the default constructor IS NOT created.
- Whenever creating an object, the memory for the object is first allocated, and only after this are the constructors called. Allocating the memory of the class may involve calling the constructors of the data members.
  (Parent Classes -> Data Members -> Own Constructor)

```
C/C++

class A
{
public:
    A() { cout << "A "; }
};

class B
{
public:
    B() { cout << "B "; }
};

class C : public A
{
public:
    B b;
    C() { cout << "C "; }
};

int main()
{
    C c;
    /// Output: A B C
}
```

- Copy constructors are created by default (whether you manually define a constructor or not). They can also accept references to the objects (*const* or not *const*). They can be called in the following manner:

```
C/C++
class Foo
{
    Foo(const Foo &other) {}
};

int main()
{
    Foo f1;

    /// Both are equivalent
    Foo f2 = f1;
    Foo f3(f1);
}
```

- Destructors can be private and virtual. Constructors can be private but can't be virtual.
- Destructors can be called from the instance object, but the same is not recommended as it would just execute the destructor's code, but the memory related to the instance would not be freed. Calling *delete* does both of them.
- If the constructor errors, then the object is NOT created, and thus ,the destructor for the same is never called.

```
C/C++

class Test
{
    static int count;
public:
    Test()
    {
        count++;
        cout << count << " ";
        if (count == 2)
            throw 12;
    }
    ~Test() { cout << "D "; }
};

int Test::count = 0;
int main()
{
    try
    {
        Test t1[5];
    }
    catch (int x)
    {
        cout << "Caught";
    }
}
// Output is 1 2 D Caught
// The destructor for the 2nd and all the items after the same is not called
```

- When an object is passed into a method, then the object's copy constructor is called and NOT the default constructor. If the same is passed by reference, no constructor must be called.

- Base data types can store the objects of derived classes.
    - If a class inherits from multiple parent classes, it can be assigned to any of them.
- If the classes are used directly, then no overriding of functions occurs, and the base class is always used (an example of default copying). If pointers are used and the overridden function is marked as virtual, only then will the function from the derived class be used.
- Classes are inherited in *private* mode by default. Private members can’t be inherited, and the inherited members have visibility set as the minimum of the mode of inheritance and their original visibility.
- If there is multilevel inheritance, the function is linearly searched up in the inheritance hierarchy until a matching function is found. Ambiguity occurs when two functions with the same name are at the same level (refer to Diamond Problem).
    - If we override a function in the derived class, all of the overloaded functions with the same name (and different signatures) would be overridden.
    - Even after overriding, you can use the scope resolution operator to access the functions at any level up the inheritance chain.
- A base class pointer can point to a derived class object, but we can only access base class members or virtual functions using the base class pointer.
- When inheriting a class, the inherited class must have a default un-parameterized constructor or the initializer list syntax must be used to call the appropriate parameterized constructor.

```
C/C++
class Foo
{
public:
    Foo(int x) {}
};

class Bar : public Foo
{
public:
    Bar() : Foo(42) {}
};
```

- Virtual objects are shared between the classes. Copy constructor first initializes all the parent classes by calling their (default) constructors and then invokes the body of the function.

```
C/C++
class A
{
public:
    A() { cout << 1; }
    A(const A &a) { cout << 2; }
};

class B : public virtual A
{
public:
    B() { cout << 3; }
    B(const B &b) { cout << 4; }
};

class C : public virtual A
{
public:
    C() { cout << 5; }
    C(const C &c) { cout << 6; }
};

class D : public B, public C
{
public:
    D() { cout << 7; }
    D(const D &d) { cout << 8; }
};

int main()
{
    D d1;
    D d2 = d1;
    /// Output: 13571358
    return 0;
}
```

- The overloading of the postfix operator involves a dummy parameter.
- Overloading the new keyword ONLY allows changing the memory allocation method but does nothing with the constructor. The same is determined by the class invocation following the *new* keyword.
- Dot, conditional, and scope resolution operators can't be overloaded. Conversion operators can be overloaded, but it can't be done globally.
- Conversion operators are automatically called when the typecasting of the classes occurs when passing functions or assignments to a declared variable of a specific type.
- *Rule of Three:* If you are writing one of the copy constructors or assignment operator destructors, you should probably write all three.
- Overloading of binary operators is done in ma anner analogous to the same:

```
C/C++
class Complex {
private:
    int real, imag;

public:
    Complex(int r = 0, int i = 0)
    {
        real = r;
        imag = i;
    }

    Complex operator+(Complex const& obj)
    {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }

    void print() { cout << real << " + i" << imag << '\n'; }
};
```

- Abstract classes *CAN* have constructors.

- The size of static variables is NOT included in the size of the instance object.
- The initialization of the static variables only happens once in the whole program.

```C/C++
void foo(int y)
{
    static int x = y;
    x++;
    cout << x << endl;
}

int main()
{
    foo(0);
    foo(1);
    foo(2);
    /// Ouput: 1 2 3
}
```

- Static attributes in a class are created when they are USED FOR THE FIRST time (or referred/accessed) in the program source code.
- *this* is passed to all the non-static function calls as a hidden argument.
- Static functions can not be virtual.

- In case you are overloading the same function with multiple definitions for different data types of arguments, the type of the argument MUST EXACTLY match that of at least one of the definitions.

- They can only call other constant functions.
- They can be invoked by both constant and non-constant objects.
- They can be overridden

```C/C++
struct Foo {
public:
    void foo() { cout << "1\n"; }
    void foo() const { cout << "2\n"; }
};

int main() {
    Foo a;
    a.foo();

    const Foo b;
    b.foo();
    // Output: 1 2
}
```

- Different meanings when used with pointers:

```C/C++
const int *ptr              /// Pointer to constant integer
int *const ptr              /// Constant pointer to integer
const int *const ptr        /// Constant pointer to constant integer
```

- A constant integer can only be assigned to a pointer to a constant integer (the pointer itself may or may not be constant).

- char -> 1B
- short int -> 2B
- int -> 4B
- double -> 8B

- The size of the structure is a multiple of the largest data types present in the same.
- Efficient packing is done of the variables, but the order of the variables must be respected.

- *Auto*: Default storage class, Stack-based, Garbage default value, Block scoped, and end of life at the end of the block
- *Static*: Data Segment, 0 default value, Scope is within the block, but life is the end of the program
- *Register:* Stored in the register, Garbage default value, Block scoped, and end of life at the end of the block.
- *Extern*: Extends the visibility of the variables and functions to the whole program. Stored in the data segment, have 0 default initialization, and have the entire program as scope. Life is till the end of the program.

- Templates are an example of runtime polymorphism.
- The compiler creates a new instance of a template function for every data type. Thus, the static members of a template are shared for the same data type objects.
- We can pass non-type arguments to templates, including integers and constants. Template arguments can also have default values or types associated with them.
- Nontype parameters must be constant.
- Templates can also be specialized for particular classes:

C/C++
```
#include <iostream>
using namespace std;

template <class T>
T max (T &a, T &b)
{
    return (a > b)? a : b;
}

/// Specialization of the max function for the type 'int'
template <>
int max <int> (int &a, int &b)
{
    cout << "Called ";
    return (a > b)? a : b;
}

int main ()
{
    int a = 10, b = 20;
    cout << max <int> (a, b);
    /// Ouput: Called 20
}
```

- Macros are expanded by the preprocessor before compilation properly; templates are expanded at compile time. Thus, macros are more difficult to debug and less efficient than templates.

Exceptions

- The statements that may cause problems are put in the try block. Also, the statements that should not be executed after a problem occurs are put in the try block.
- Exceptions bubble up the call stack until they are caught.
- *catch(...)* is the catch-all syntax that catches exceptions of any type. It must be the last catch block; otherwise, a compile-time error will be raised.
- All the objects in the *try* block are first dropped, and only then the control is transferred to the catch block.
- Destructors are only called for objects whose constructors are complete without encountering errors.
- C++ compiler doesn't check and enforce a function to list the exceptions it can throw. In Java, it is enforced.
- If both base and derived classes are caught as exceptions, then the catch block of the derived class must appear before the base class. If we put the base class first, then the derived class catch block will never be reached. In Java, the compiler itself does not allow catching a base class exception before it is derived. In C++, the compiler might warn about it but compiles the code. This behavior exists as the compiler automatically typecasts the error to fix the arguments.
- C++ has no *finally* clause, so it is recommended to follow the RAII pattern (implementing objects with destructors to clean up the resources).

- If you are getting confused about pointers and values, assuming the variables' addresses and simulating the whole program is helpful.
- The *double-number-sign* or *token-pasting operator (##)*, sometimes called the merging or combining operator, is used in object-like and function-like macros. It permits separate tokens to be joined into a single token and, therefore, can't be the first or last token in the macro definition.

```
C/C++

#define f(g, h) g##h
int main()
{
    int ab = 10;
    cout << f(a, b); // Equivalent to cout << ab | Prints 10
}
```

SOLID Principles

1. **Single Responsibility Principle (SRP)**

A class should have only one reason to change, meaning it should only have one job or responsibility.

Example: Imagine a User class that handles user data. If it's also responsible for saving data to a database and logging activities, it's doing too much. Instead, the User class should handle user-related logic. A UserRepository class should handle data persistence, and a Logger class should handle logging.

SRP promotes separation of concerns, which makes debugging easier because each class has a clear role. It's especially practical in large projects, reducing dependencies and making unit testing easier. However, too strictly adhering to SRP can lead to an overly fragmented codebase, so balance is key.

2. **Open-Closed Principle (OCP)**

Software entities should be open for extension but closed for modification. You should be able to add new functionality without changing existing code.

Example: Suppose you have a PaymentProcessor class that handles different payment methods. Instead of modifying the class to add each new payment type, create a PaymentMethod interface that implements each payment type (like CreditCardPayment or PayPalPayment). The PaymentProcessor can then accept any PaymentMethod implementation without being modified.

OCP is practical when you expect frequent changes or additions to the system, such as adding new features. It keeps the codebase stable, and adding new functionalities becomes easier. However, implementing OCP can sometimes lead to an excess of small, similar classes, so it should be used in scenarios where flexibility is essential.

3. **Liskov Substitution Principle (LSP)**

Derived classes should be substitutable for their base classes without altering the correctness of the program.

Example: Consider a Rectangle class with width and height properties and a Square class that inherits from Rectangle. If Square redefines width and height always to be equal, it can break LSP when used in place of a Rectangle in functions that expect a rectangle. Instead, Square should not inherit from Rectangle if the relationship breaks logical expectations.

LSP promotes the use of inheritance with caution. Violating LSP can lead to unexpected behavior and subtle bugs, especially when classes are part of an inheritance chain. It's practical to ensure that any subclass doesn’t break the intended behavior of the base class in complex inheritance hierarchies.

4. **Interface Segregation Principle (ISP)**

A client should not be forced to implement interfaces it doesn’t use. In other words, interfaces should be small and specific to particular client needs.

Example: Instead of a large Worker interface with methods like code(), test(), and design(), break it down into smaller interfaces like Coder, Tester, and Designer. A class implementing Tester won’t have unnecessary methods like design().

ISP helps to keep interfaces focused and makes code cleaner and more maintainable. When interfaces are too large or general, classes implementing them often end up with unused methods, leading to bloated or confusing code. ISP is very useful in code that relies heavily on abstractions, as it keeps implementations streamlined.

5. **Dependency Inversion Principle (DIP)**

High-level modules should not depend on low-level modules. Both should depend on abstractions. Additionally, abstractions should not depend on details; details should depend on abstractions.

Example: Instead of a UserService class directly depending on a concrete UserRepository class, it should depend on an interface, IUserRepository. The concrete class SQLUserRepository would then implement IUserRepository. This way, UserService can work with any IUserRepository implementation, making it easier to swap databases or mock the repository for testing.

DIP reduces tight coupling, which makes components more flexible and easier to test. This is especially useful in large systems with multiple dependencies and potential implementation changes. It promotes dependency injection, making it practical in layered architectures and following inversion-of-control (IoC) patterns.