Computer programming midterm answers

1.What is OOP in C++

Object-Oriented Programming (OOP) is a programming paradigm that is based on the idea of objects. An object is an instance of a class that contains both data and methods to operate on that data. C++ is a popular language for OOP due to its support for features such as encapsulation, inheritance, and polymorphism.

In C++, classes are used to define objects. A class is a user-defined data type that contains data members and member functions. Data members are variables that hold the data for an object, while member functions are methods that operate on that data.

Encapsulation is a fundamental concept in OOP that involves hiding the implementation details of a class's data members and methods from outside code. This is achieved by making the data members private and providing public member functions to access and modify them. This helps to ensure that the object's data is only modified in a controlled and predictable way.

Inheritance is another important concept in OOP that allows for the creation of derived classes that inherit the data members and methods of a base class. This allows for code reuse and helps to avoid duplication of code. Polymorphism is also supported in C++ through virtual functions, which allows for multiple objects of different classes to be treated as if they were of the same class.

2.Working with pointers in C++

Pointers in C++ are variables that store the memory addresses of other variables. They are used to manipulate memory directly and can be used to dynamically allocate and deallocate memory.

Working with pointers in C++ requires an understanding of memory management, as well as pointer arithmetic and dereferencing. Pointer arithmetic involves adding or subtracting integers from pointers to move them to different locations in memory. Dereferencing a pointer

involves using the * operator to access the value stored at the memory address pointed to by the pointer.

It is important to use pointers carefully in C++ to avoid common issues such as memory leaks and invalid pointer references. Memory leaks occur when dynamically allocated memory is not properly deallocated, while invalid pointer references occur when a pointer is used to access memory that has already been deallocated.

3.What is List and working principles

In C++, a list is a sequence container that allows for constant time insertions and removals from the beginning or the end of the list. The list is implemented as a doubly-linked list, which means that each element in the list has a pointer to both the previous and the next element in the list.

The list container provides a number of member functions for manipulating the list, including insert, erase, push_front, push_back, pop_front, pop_back, and more. These member functions can be used to add, remove, and access elements in the list.

Here is an example of using a list in C++:

```cpp
#include <iostream>
#include <list>

int main() {
    std::list<int> myList;

    myList.push_back(1);
    myList.push_back(2);
    myList.push_back(3);

    for (auto it = myList.begin(); it != myList.end(); ++it) {
        std::cout << *it << std::endl;
    }

    return 0;
}
```

In this example, a list container is created using the std::list class, and elements are added to the list using the push_back member function. The for loop iterates over the list using an iterator, and each element is printed to the console.

4.Functions of List's and Set's

List and set are two commonly used container classes in C++, and they each have their own set of functions.

Functions of List:

push_front: Adds an element to the beginning of the list.

push_back: Adds an element to the end of the list.

pop_front: Removes the first element from the list.

pop_back: Removes the last element from the list.

insert: Inserts an element at a specified position in the list.

erase: Removes an element at a specified position in the list.

size: Returns the number of elements in the list.

front: Returns the first element in the list.

back: Returns the last element in the list.

Functions of Set:

insert: Inserts an element into the set.

erase: Removes an element from the set.

size: Returns the number of elements in the set.

find: Searches the set for an element and returns an iterator to it if found.

count: Returns the number of occurrences of a specific element in the set.

clear: Removes all elements from the set.

Both list and set are useful for storing collections of data in C++. Lists are typically used when frequent insertions and removals from the beginning or end of the container are required, while sets are used when uniqueness and ordering of the elements are important.

5.What is constructors and write an example

What is constructors and write an example:

Constructors are special member functions in C++ that are called when an object is created. They are used to initialize the object's data members to their initial values. Constructors have the same name as the class and no return type. They can be overloaded to allow for different ways of initializing an object.

Here is an example of a class with a constructor:

```cpp
#include <iostream>

class Person {
public:
    std::string name;
    int age;

    Person(std::string n, int a) {
        name = n;
        age = a;
    }
};

int main() {
    Person p("John", 25);
    std::cout << p.name << " is " << p.age << " years old." <<
        std::endl;
    return 0;
}
```

In this example, the Person class has a constructor that takes two parameters, a name and an age, and initializes the object's name and age data members. In the main function, a new Person object is created using the constructor, and its data members are accessed and printed to the console.

6.Constant aggregates,Constant pointers

In C++, a constant aggregate is an aggregate type (such as an array, struct, or union) that is declared const. This means that the values of the elements in the aggregate cannot be modified after initialization. For example:

```
const int arr[] = {1, 2, 3};
```

In this example, the array arr is a constant aggregate, and the values of its elements cannot be modified.

A constant pointer is a pointer that is declared const, meaning that the value of the pointer (i.e., the memory address it points to) cannot be modified after initialization. However, the value of the object pointed to by the pointer can still be modified. For example:

```
int x = 5;
const int* ptr = &x;
```

In this example, ptr is a constant pointer that points to the variable x. The value of ptr (i.e., the memory address of x) cannot be modified, but the value of x can still be modified.

7.Access specifiers in C++

In C++, access specifiers are keywords used to define the access level of class members. There are three access specifiers in C++: public, private, and protected.

Public members can be accessed by any code that can access the object of the class. Private members can only be accessed by member functions of

the same class. Protected members can be accessed by member functions of the same class or member functions of derived classes.

Here is an example:

```
#include <iostream>

class Person {
public:
    std::string name;
    int age;

    void print() {
        std::cout << name << " is " << age << " years old."
            << std::endl;
    }

private:
    int salary;
};

int main() {
    Person p;
    p.name = "John";
    p.age = 25;
    p.print();
    // p.salary = 50000; // This line will cause a compile
        error
    return 0;
}
```

In this example, the Person class has a public member function print() that can be accessed by any code that can access the object of the class. The class also has a private data member salary that can only be accessed by member functions of the same class. In the main function, an object of the Person class is created, and its public data members are accessed and printed to the console. The attempt to access the private data member salary results in a compile error.

8.Function overloading

Function overloading is a feature in C++ that allows a programmer to define multiple functions with the same name, but with different parameter types or numbers of parameters. When a function is called with arguments, the compiler selects the appropriate function to call based on the types and number of arguments passed.

Here is an example:

```cpp
#include <iostream>

int add(int a, int b) {
    return a + b;
}

double add(double a, double b) {
    return a + b;
}

int main() {
    std::cout << add(1, 2) << std::endl;
    std::cout << add(2.5, 3.5) << std::endl;
    return 0;
}
```

In this example, there are two functions named "add", one that takes two integer parameters and another that takes two double parameters. When the add function is called with integers, the first add function is called, and when it is called with doubles, the second add function is called.

9.What is structs in C++ and write example

In C++, a struct is a user-defined data type that groups together related data members of different data types into a single unit. The struct is similar to a class, but with default public access to its members.

Here is an example:

```cpp
#include <iostream>

struct Person {
    std::string name;
    int age;
};

int main() {
    Person p;
    p.name = "John";
    p.age = 25;
    std::cout << p.name << " is " << p.age << " years old." << std
        ::endl;
    return 0;
}
```

In this example, a struct named "Person" is defined with two data members, a string "name" and an integer "age". In the main function, an object of the Person struct is created, and its data members are accessed and printed to the console.

10.OOP concepts in C++

Object-oriented programming (OOP) is a programming paradigm that focuses on organizing code into reusable and modular objects that interact with each other to solve problems. In C++, the main OOP concepts are:

- Classes and objects: Classes are blueprints for creating objects, and objects are instances of classes that have their own unique data and behaviors.
- Encapsulation: Encapsulation is the practice of hiding the implementation details of a class from the user and providing a public interface for accessing its data and behaviors.
- Inheritance: Inheritance is the process of creating a new class from an existing class, where the new class inherits the data and behaviors of the existing class and can add or modify its own data and behaviors.
- Polymorphism: Polymorphism is the ability of objects of different classes to be used interchangeably, allowing for more flexible and modular code.

11.Encapsulation

Encapsulation is an OOP concept that refers to the practice of hiding the implementation details of a class from the user and providing a public interface for accessing its data and behaviors. Encapsulation allows for better code organization, modularity, and security by preventing users from directly accessing or modifying a class's data and behaviors without going through the public interface.

Here is an example:

```cpp
#include <iostream>

class BankAccount {
private:
    int accountNumber;
    double balance;

public:
    BankAccount(int accNum, double bal) {
        accountNumber = accNum;
        balance = bal;
    }

    void deposit(double amount) {
        balance += amount;
    }

    void withdraw(double amount) {
        if (amount <= balance) {
            balance -= amount;
        }
    }

    double getBalance() {
        return balance;
    }
};

int main() {
```

12.Inheritance

Inheritance is a fundamental feature of object-oriented programming (OOP) that allows classes to inherit properties and methods from a parent class. In C++, inheritance is achieved using the syntax `class DerivedClass : accessSpecifier BaseClass`, where `DerivedClass` is the class that is inheriting, `BaseClass` is the parent class being inherited from, and `accessSpecifier` defines the level of access that the derived class has to the members of the base class (public, protected, or private). Inheritance can be of several types such as single, multiple, hierarchical, and multilevel inheritance. Here is an example of inheritance:

```cpp
class Animal {
   public:
      void eat() {
         cout << "Eating..." << endl;
      }
};

class Dog : public Animal {
   public:
      void bark() {
         cout << "Barking..." << endl;
      }
};

int main() {
   Dog myDog;
   myDog.eat();
   myDog.bark();
   return 0;
}
```

In the above example, the `Dog` class is derived from the `Animal` class using public inheritance. This means that the public members of the `Animal` class are accessible in the `Dog` class. The `Dog` class also has its own method, `bark()`. When we create an object of the `Dog` class and call the `eat()` and `bark()` methods, both methods are executed, since the `Dog` class inherits the `eat()` method from the `Animal` class.

13.Polimorphism

Polymorphism is a concept in OOP that refers to the ability of objects to take on many forms. In C++, polymorphism is achieved using two techniques: function overloading and function overriding. Function overloading allows multiple functions with the same name but different parameters to exist in the same scope, while function overriding allows a derived class to provide its own implementation of a method that is already defined in its parent class. Polymorphism allows us to write code that can work with objects of different classes without knowing their exact type. Here is an example of polymorphism using function overriding:

```
class Animal {
   public:
      virtual void makeSound() {
         cout << "The animal makes a sound..." << endl;
      }
};

class Dog : public Animal {
   public:
      void makeSound() {
         cout << "The dog barks..." << endl;
      }
};

class Cat : public Animal {
   public:
      void makeSound() {
         cout << "The cat meows..." << endl;
      }
};
```

```
int main() {
   Animal* myAnimal = new Animal();
   Animal* myDog = new Dog();
   Animal* myCat = new Cat();

   myAnimal->makeSound();
   myDog->makeSound();
   myCat->makeSound();

   delete myAnimal;
   delete myDog;
   delete myCat;
   return 0;
}
```

In the above example, the **`Animal`** class has a virtual method **`makeSound()`**. The **`Dog`** and **`Cat`** classes inherit from **`Animal`** and provide their own implementation of the **`makeSound()`** method. When we create objects of the **`Animal`**, **`Dog`**, and **`Cat`** classes and call the **`makeSound()`** method on each of them, the appropriate implementation of **`makeSound()`** is executed for each object, based on its type

14.Pointers to functions

In C++, functions can also be treated as variables, and we can use pointers to functions to pass functions as arguments to other functions or to store them in arrays. Here is an example:

```
#include <iostream>

using namespace std;

int add(int a, int b) {
   return a + b;
}

int subtract(int a, int b) {
   return a - b;
}

int main() {
   int (*p)(int, int); // declare a pointer to a function

   p = add; // assign the address of the add function to p
   cout << p(2, 3) << endl; // call the add function using p

   p = subtract; // assign the address of the subtract function
       to p
```

```
   cout << p(5, 2) << endl; // call the subtract function using p

   return 0;
}
```

In the above example, we declare a pointer `p` to a function that takes two integer arguments and returns an integer value. We then assign the address of the `add()` function to `p` and call it using the pointer. We then assign the address of the `subtract()` function to `p` and call it using the pointer.

15.Scanf, printf, cin cout difference

`scanf()` and `printf()` are input/output functions in C, while `cin` and `cout` are input/output objects in C++. The main difference between them is that `scanf()` and `printf()` use format specifiers to specify the data type of the input or output, while `cin` and `cout` use the `<<` and `>>` operators to read or write data.

Here is an example using `scanf()` and `printf()`:

```c
#include <stdio.h>

int main() {
   int age;
   char name[20];

   printf("Enter your name: ");
   scanf("%s", name);
   printf("Enter your age: ");
   scanf("%d", &age);

   printf("Your name is %s and you are %d years old.\n", name, age
       );
   return 0;
}
```

And here is the same example using `cin` and `cout`:

```c
#include <iostream>

using namespace std;

int main() {
   int age;
   string name;

   cout << "Enter your name: ";
   cin >> name;
   cout << "Enter your age: ";
   cin >> age;

   cout << "Your name is " << name << " and you are " << age << "
       years old." << endl;
   return 0;
}
```

In the `scanf()` and `printf()` example, we use the `%s` and `%d` format specifiers to read in a string and an integer, respectively. We also use the `&` operator to pass the address of the `age` variable to `scanf()`. In the `cin` and `cout` example, we use the `>>` operator to read in the string and integer values. We also use the `endl` manipulator to print a newline character at the end of the output.

16.Constructor overloading

> In C++, a constructor can be overloaded just like any other function. This means that we can define multiple constructors for a class with different parameter lists. When an object of that class is created, the appropriate constructor will be called based on the arguments passed.

Here is an example of a class with two constructors:

```
class Person {
   private:
      string name;
      int age;

   public:
      Person(string n, int a) {
         name = n;
         age = a;
      }

      Person() {
         name = "Unknown";
         age = 0;
      }

      void display() {
         cout << "Name: " << name << endl;
         cout << "Age: " << age << endl;
      }
```

In the above example, we define two constructors for the `Person` class. The first constructor takes two parameters, `n` and `a`, and initializes the `name` and `age` member variables with those values. The second constructor takes no parameters and initializes the `name` and `age` member variables with default values. We can then create objects of the `Person` class using either constructor:

```
Person p1("John", 30);
p1.display(); // output: Name: John Age: 30

Person p2;
p2.display(); // output: Name: Unknown Age: 0
```

17.Selecting overloaded constructor

When there are multiple constructors for a class, the appropriate constructor is selected based on the arguments passed when an object is created. If there is an exact match between the argument types and the parameter types of one of the constructors, that constructor will be called. Otherwise, the compiler will try to convert the arguments to the appropriate types, and if a match is found, that constructor will be called.

Here is an example:

```
class Person {
   private:
      string name;
      int age;

   public:
      Person(string n, int a) {
         name = n;
         age = a;
      }

      Person(int a) {
         name = "Unknown";
         age = a;
      }

      void display() {
         cout << "Name: " << name << endl;
         cout << "Age: " << age << endl;
      }
```

```
};

int main() {
    Person p1("John", 30); // calls first constructor
    p1.display(); // output: Name: John Age: 30

    Person p2(40); // calls second constructor
    p2.display(); // output: Name: Unknown Age: 40

    return 0;
}
```

In the above example, the first object `p1` is created using the first constructor because there is an exact match between the argument types (`string` and `int`) and the parameter types of the first constructor. The second object `p2` is created using the second constructor because there is no exact match for the argument type (`int`), but it can be converted to the appropriate type.

18.Arrays of pointers to objects

In C++, it is possible to create arrays of pointers to objects. This can be useful in situations where we want to dynamically create objects and store them in an array. We can use a loop to create each object and assign its address to an element of the array.

For example, consider a class named "Rectangle" that has a constructor that takes two arguments (length and width):

```
class Rectangle {
  private:
     int length, width;
  public:
     Rectangle(int l, int w) { length = l; width = w; }
};
```

To create an array of pointers to Rectangle objects, we can use the new operator to allocate memory for the array:

```
Rectangle** rectangles = new Rectangle*[10];
```

This creates an array of 10 pointers to Rectangle objects. To create each object and assign its address to an element of the array, we can use a loop:

```
for (int i = 0; i < 10; i++) {
   rectangles[i] = new Rectangle(i+1, i+2);
}
```

This creates 10 Rectangle objects, each with a different length and width, and stores their addresses in the "rectangles" array.

To access an object in the array, we can use the array index operator and the pointer dereference operator:

```
Rectangle* rect = rectangles[3];
int length = rect->getLength();
int width = rect->getWidth();
```

This retrieves the address of the fourth object in the array, dereferences it to access the object itself, and calls the "getLength()" and "getWidth()" functions to retrieve the length and width of the rectangle.

When we are finished using the objects and the array, we should free the memory allocated using the delete operator:

```
for (int i = 0; i < 10; i++) {
   delete rectangles[i];
}
delete[] rectangles;
```

This deletes each object and the array of pointers to objects.

19.Objects inside objects

In C++, it is possible to define objects inside other objects. This is known as composition and is a form of object-oriented programming that allows us to create complex objects by combining simpler objects.

For example, consider a class named "Point" that represents a point in two-dimensional space with x and y coordinates:

```
class Point {
   private:
      int x, y;
   public:
      Point(int xval, int yval) { x = xval; y = yval; }
};
```

Now, suppose we want to define a class named "Rectangle" that represents a rectangle in two-dimensional space using two "Point" objects to represent the opposite corners of the rectangle:

```
class Rectangle {
   private:
      Point topLeft, bottomRight;
   public:
      Rectangle(Point tl, Point br) { topLeft = tl; bottomRight =
          br; }
};
```

In this example, the "Rectangle" class contains two "Point" objects - one representing the top-left corner of the rectangle and the other representing the bottom-right corner. When we create a "Rectangle" object, we pass two "Point" objects to the constructor to initialize these member variables.

We can now create "Rectangle" objects using the "Point" objects we have defined:

```
Point topLeft(0, 0);
Point bottomRight(5, 5);
Rectangle rect(topLeft, bottomRight);
```

This creates a "Rectangle" object with the top-left corner at (0,0) and the bottom-right corner at (5,5).

20.Difference between List and Linkedlist

In C++, List and Linked List are two data structures used to store a collection of elements. Both of them have their advantages and disadvantages, and choosing the right one depends on the use case.

A List is a sequential container that stores elements in a linear order. It allows for fast insertion and deletion of elements, as well as random access to elements using an index. A List is implemented using an array or a dynamic array.

On the other hand, a Linked List is a non-sequential container that stores elements in a chain of nodes. Each node contains an element and a pointer to the next node in the chain. Linked Lists allow for efficient insertion and deletion of elements at any point in the list, but random access to elements is not possible as it requires traversing the list from the beginning.

Some of the key differences between List and Linked List are:

1. Implementation: List is implemented using an array or a dynamic array, whereas Linked List is implemented using nodes and pointers.
2. Access: List allows for random access to elements using an index, whereas Linked List requires traversal of the list to access an element.
3. Insertion and Deletion: List has a constant time complexity for insertion and deletion at the beginning or end of the list, but linear time complexity for insertion or deletion in the middle. Linked List has a constant time complexity for insertion and deletion at any point in the list.
4. Memory Usage: List uses contiguous memory allocation, while Linked List uses non-contiguous memory allocation.
5. Memory Overhead: List has a smaller memory overhead per element than Linked List, as it only stores the elements and a pointer to the first and last elements. Linked List requires an additional pointer for each node.

In summary, List is a good choice when fast random access is required and the number of elements is known beforehand. On the other hand, Linked List is a good choice when efficient insertion and deletion are required and the number of elements is not known beforehand.

21. Virtual methods and its usage

In C++, virtual methods are an essential feature of polymorphism, allowing derived classes to override the behavior of methods defined in their base classes. When a base class declares a method as virtual, it enables late binding or dynamic dispatch, which means that the appropriate method implementation is determined at runtime based on the actual object type rather than the declared type.

To define a virtual method in C++, you need to declare it as virtual in the base class and provide an implementation. Here's an example:

```cpp
class Base {
public:
    virtual void foo() {
        // Base class implementation
    }
};

class Derived : public Base {
public:
    void foo() override {
        // Derived class implementation
    }
};
```

In this example, the foo() method is declared as virtual in the Base class. The Derived class then overrides this method using the override keyword, indicating that it intends to provide a specialized implementation. The override keyword helps catch errors at compile-time if the base class method is not being overridden correctly.

When you have a pointer or reference to a base class object that actually points to a derived class object, you can invoke the virtual method, and the appropriate implementation will be called based on the actual object type:

```cpp
Base* basePtr = new Derived();
basePtr->foo();  // Calls the derived class implement
```

Virtual methods are typically used in base classes to provide a common interface or behavior that can be specialized by derived classes. This allows you to write code that operates on base class pointers or references without needing to know the exact derived class type. It enables code reuse, extensibility, and flexibility in object-oriented programming.

22.Calling virtual method from base function

In C++, if you want to call a virtual method from a base class function, you can simply invoke the method as if it were a non-virtual function. The actual implementation of the derived class will be called based on the runtime type of the object.

Here's an example:

```cpp
#include <iostream>

class Base {
public:
    virtual void foo() {
        std::cout << "Base::foo()" << std::endl;
    }

    void bar() {
        std::cout << "Base::bar()" << std::endl;
        foo();  // Call the virtual method
    }
};

class Derived : public Base {
public:
    void foo() override {
        std::cout << "Derived::foo()" << std::endl;
    }
};

int main() {
    Base* basePtr = new Derived();
    basePtr->bar();  // Calls Base::bar() and Derived::foo()

    delete basePtr;
    return 0;
}
```

In this example, we have a base class Base with a virtual method foo() and a non-virtual method bar(). The bar() function calls the foo() method, which is virtual.

We also have a derived class Derived that overrides the foo() method with its own implementation.

In the main() function, we create a pointer basePtr of type Base* that points to an object of type Derived. When we call basePtr->bar(), it first calls Base::bar(), which then internally calls

foo(). However, since basePtr is actually pointing to a Derived object, the overridden Derived::foo() method is invoked.

The output of the program will be:

```
Base::bar()
Derived::foo()
```

As you can see, the virtual method foo() called from within the bar() function in the base class invokes the overridden implementation in the derived class.

23.Multiple inheritance

Multiple inheritance is a feature in C++ that allows a class to inherit from multiple base classes. This means that a derived class can inherit members and functionality from more than one parent class.

Here's an example to demonstrate multiple inheritance:

```cpp
#include <iostream>

// Base class 1
class Base1 {
public:
    void display1() {
        std::cout << "Base1::display1()" << std::endl;
    }
};

// Base class 2
class Base2 {
public:
    void display2() {
        std::cout << "Base2::display2()" << std::endl;
    }
};

// Derived class inheriting from Base1 and Base2
class Derived : public Base1, public Base2 {
```

```
public:
    void display3() {
        std::cout << "Derived::display3()" << std::endl;
    }
};

int main() {
    Derived obj;
    obj.display1();  // Call Base1::display1()
    obj.display2();  // Call Base2::display2()
    obj.display3();  // Call Derived::display3()

    return 0;
}
```

In this example, we have two base classes, Base1 and Base2. The derived class Derived inherits from both Base1 and Base2 using multiple inheritance. The derived class can access members and functions from both base classes.

The display1() function belongs to Base1, display2() belongs to Base2, and display3() is specific to Derived. The main() function demonstrates calling these functions on an object of the Derived class.

It's important to note that multiple inheritance can lead to certain complexities, such as the diamond problem and ambiguity when multiple base classes have members with the same name. In such cases, you may need to use scoping or explicitly resolve the ambiguity using the scope operator (::).

24.Exceptions in C++

Exceptions in C++ provide a mechanism to handle exceptional situations or errors that occur during program execution. An exception is an object that represents a specific type of error or exceptional condition. When an error occurs, you can throw an exception, and if it's not handled, the program will terminate.

Here's a basic overview of working with exceptions in C++:

Throwing an Exception:

To indicate an error or exceptional condition, you can throw an exception using the throw keyword. You can throw any object as an exception, but it's common to use predefined exception types or create custom exception classes.

Example:

```
throw std::runtime_error("An error occurred!");
```

Catching Exceptions:

To handle exceptions, you use a try-catch block. The try block contains the code that might throw an exception, and the catch block handles the exception if it occurs. You can have multiple catch blocks to handle different types of exceptions.

Example:

```
try {
    // Code that might throw an exception
} catch (const std::exception& ex) {
    // Exception handling code
}
```

Catching Specific Exceptions:

You can catch specific exceptions by specifying the exception type in the catch block. This allows you to handle different types of exceptions differently.

Example:

```
try {
    // Code that might throw an exception
} catch (const std::runtime_error& ex) {
    // Exception handling code for runtime_error
} catch (const std::exception& ex) {
    // Exception handling code for other exception
}
```

Handling Uncaught Exceptions:

If an exception is thrown but not caught, it will propagate up the call stack until it either reaches a matching catch block or terminates the program. To handle uncaught exceptions, you can define a global catch block using std::terminate() or std::set_terminate().

Example:

```
std::set_terminate([]() {
    std::cout << "Uncaught exception occurred!" << std::endl;
    std::terminate();
});
```

Creating Custom Exception Classes:

You can create your own exception classes by deriving from std::exception or any of its derived classes. Custom exception classes allow you to provide specific information about the error or exceptional condition.

Example:

```
class MyException : public std::exception {
public:
    const char* what() const noexcept override {
        return "Custom exception occurred!";
    }
};

throw MyException();
```

Exceptions provide a powerful mechanism for handling errors and exceptional conditions in a structured manner. By throwing and catching exceptions, you can gracefully handle errors and take appropriate actions in your C++ programs.

25.Handling method of exceptions

When working with exceptions in C++, you can handle them using the try-catch mechanism. The try block contains the code that might throw an exception, and the catch block is used to handle the thrown exception. Here's an example of how to handle exceptions in C++:

```cpp
#include <iostream>

int main() {
    try {
        // Code that might throw an exception
        int numerator = 10;
        int denominator = 0;
        if (denominator == 0) {
            throw std::runtime_error("Divide by zero exception"
                );
        }
        int result = numerator / denominator;
        std::cout << "Result: " << result << std::endl;
    } catch (const std::exception& ex) {
        // Exception handling code
        std::cout << "Exception caught: " << ex.what() << std
            ::endl;
    }

    return 0;
```

In this example, the try block contains the code that performs a division operation. If the denominator is zero, a std::runtime_error exception is thrown using the throw statement. The catch block is responsible for handling the thrown exception. It catches the exception by specifying the exception type (std::exception) and creates a constant reference to the caught exception object named ex. Inside the catch block, you can write code to handle the exception. In this case, it simply prints the exception message using the what() function.

If an exception is thrown within the try block, the program flow immediately transfers to the corresponding catch block. The catch block with the matching exception type is executed. If no matching catch block is found, the exception propagates up the call stack until it reaches a suitable catch block or terminates the program.

It's important to note that you can have multiple catch blocks to handle different types of exceptions. The catch blocks are checked in order, and the first matching catch block is executed.

By using the try-catch mechanism, you can handle exceptions and gracefully handle errors or exceptional conditions in your C++ code, allowing you to provide appropriate error handling and recovery logic.

26. Catching exceptions with try-catch blocks:

In C++, exceptions can be caught and handled using the try-catch blocks. The try block contains the code that might throw an exception, and the catch block is used to handle the thrown exception. Here's the general syntax:

```
try {
    // Code that might throw an exception
} catch (ExceptionType1& ex) {
    // Exception handling code for ExceptionType1
} catch (ExceptionType2& ex) {
    // Exception handling code for ExceptionType2
} catch (...) {
    // Exception handling code for any other exceptio
}
```

In this syntax, you can have multiple catch blocks, each handling a specific type of exception. The catch block with the matching exception type is executed when that type of exception is thrown. The ellipsis ... in the last catch block is used to catch any other unhandled exceptions.

27.What is STL library elements:

STL (Standard Template Library) is a powerful library in C++ that provides a collection of generic algorithms and data structures. It is part of the C++ Standard Library and offers a wide range of container classes, algorithms, and iterators. The key elements of the STL library include:

Containers: STL provides various container classes, such as vectors, lists, stacks, queues, sets, maps, and more. These containers allow you to store and manage collections of objects.

Algorithms: STL provides a set of generic algorithms that can operate on containers or any other suitable data structures. These algorithms include sorting, searching, manipulating, and performing various operations on the data.

Iterators: Iterators in STL provide a way to traverse through the elements of a container. They act as a generalized pointer and allow you to access and manipulate the elements of a container sequentially or non-sequentially.

Function Objects: Function objects (also known as functors) in STL are objects that behave like functions. They can be used with algorithms to define custom behavior or operations on the elements.

Allocators: Allocators in STL are used to allocate and deallocate memory for containers. They provide a way to control the memory management of containers and can be customized to meet specific requirements.

The STL library is designed to provide generic and reusable components, promoting code reusability and efficiency. It offers a standardized and efficient way to work with data structures and algorithms in C++.

28. What is vectors

In C++, a vector is a dynamic array container provided by the STL (Standard Template Library). It allows you to store and manipulate a sequence of elements of the same type. Vectors provide a flexible and efficient way to manage resizable arrays.

Some key characteristics of vectors include:

Dynamic Size: Vectors can grow or shrink in size dynamically based on the number of elements you add or remove. You don't need to specify the size during declaration.

Random Access: Vectors support efficient random access to elements. You can access elements using an index, allowing for constant-time access.

Continuous Memory: The elements in a vector are stored in a contiguous block of memory, ensuring efficient memory access and cache utilization.

Dynamic Insertion and Deletion: Vectors provide methods to insert or delete elements at any position, allowing you to dynamically modify the vector's contents.

Here's a simple example of using a vector in C++:

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> numbers;

    // Adding elements to the vector
    numbers.push_back(10);
    numbers.push_back(20);
    numbers.push_back(30);

    // Accessing elements using index
    std::cout << numbers[0] << std::endl;  // Output: 10

    // Iterating over the vector
    for (const auto& num : numbers) {
        std::cout << num << " ";
    }
    // Output: 10 20 30

    return 0;
}
```

In this example, we create a vector called numbers to store integers. Elements are added to the vector using the push_back() method. The elements can be accessed using the subscript operator ([]) or iterated over using a range-based for loop.

29.Difference between vectors and lists:

Vectors and lists are both container classes provided by the STL in C++, but they have some differences in their characteristics and usage:

Storage: Vectors store elements in a contiguous block of memory, while lists use a doubly-linked list structure. This means that vector elements are stored in a continuous chunk of memory, allowing for efficient random access, whereas list elements are scattered in memory and require traversal to access each element.

Insertion and Deletion: Vectors offer efficient insertion and deletion at the end (push_back() and pop_back() operations) but are relatively slower for insertions or deletions in the middle or beginning. Lists, on the other hand, provide efficient constant-time insertion and deletion at any position.

Random Access: Vectors allow constant-time random access to elements using the subscript operator ([]) or iterator arithmetic. Lists do not support random access and require sequential traversal from the beginning or end to reach a specific element.

Iterator Validity: When elements are added or removed from a vector, iterators and references to elements may become invalidated. In a list, iterators and references remain valid even after insertion or deletion.

Memory Overhead: Vectors have a smaller memory overhead compared to lists since they do not require additional pointers for linking elements like lists do.

Choosing between vectors and lists depends on your specific requirements. If you need frequent random access and efficient storage of elements, vectors are a better choice. If you need efficient insertion and deletion at arbitrary positions or if iterator validity is a concern, lists may be more suitable.

30. What is a stack:

In computer science, a stack is an abstract data type that follows the LIFO (Last-In, First-Out) principle. It behaves like a physical stack of objects, where you can only add or remove items from the top.

In the context of programming, a stack is a data structure that supports two main operations:

Push: Adds an element to the top of the stack.

Pop: Removes the topmost element from the stack.

Additionally, a stack may provide other operations such as peek (accessing the top element without removing it) and checking if the stack is empty.

Stacks are often used to solve problems involving depth-first search, backtracking, parsing expressions, function call stack management, and more. They can be implemented using various data structures, such as arrays or linked lists.

Here's a simple example of using a stack in C++ using the STL stack container:

```cpp
#include <iostream>
#include <stack>

int main() {
    std::stack<int> stack;

    // Pushing elements onto the stack
    stack.push(10);
    stack.push(20);
    stack.push(30);

    // Accessing and removing the top element
    int topElement = stack.top();  // Accessing the top element
    stack.pop();                   // Removing the top element

    std::cout << "Top element: " << topElement << std::endl;
    std::cout << "Stack size: " << stack.size() << std::endl;

    return 0;
}
```

In this example, we create a stack called stack to store integers. Elements are added to the stack using the push() method, and the top element can be accessed using the top() method. The pop() method removes the top element from the stack. The size() method returns the number of elements in the stack.

Stacks provide an efficient way to manage elements following the LIFO principle, making them useful in a wide range of applications.

31.What is queue?

In C++, a queue is a data structure that follows the First-In-First-Out (FIFO) principle. It represents a collection of elements in which new elements are inserted at the rear (also known as the "back") and existing elements are removed from the front. This means that the element that has been in the queue the longest is the first one to be removed.

The queue data structure provides two main operations:

1.Enqueue: It adds an element to the rear of the queue. This operation is also referred to as "push" or "insert".

2.Dequeue: It removes the element from the front of the queue. This operation is also known as "pop" or "remove".

C++ provides a standard template library (STL) container called std::queue that implements a queue data structure. To use it, you need to include the <queue> header file.

32. Class's and function's variables life time in C++

In C++, the lifetime of variables depends on where they are declared and how they are scoped. Variables declared within a class or function have different lifetimes:

1. Class Member Variables: These variables are declared within a class and are associated with objects of that class. They exist as long as the object exists. When an object is created, memory is allocated for its member variables, and when the object is destroyed, the memory is released. The lifetime of class member variables is tied to the lifetime of the object.

2. Local Variables: These variables are declared within a function or a block of code and are only accessible within that function or block. Local variables come into existence when the function or block is entered and are destroyed when the function or block is exited. The lifetime of local variables is limited to the scope in which they are declared. They are typically stored on the stack.

3. Static Variables: Static variables are declared with the `static` keyword. Within a function, a static variable retains its value between function calls. Static variables have a lifetime that extends throughout the program's execution, from the moment they are initialized until the program terminates. They are typically stored in a separate static data area.

4. Global Variables: Global variables are declared outside of any function or class, typically at the beginning of a file. They have a lifetime that extends throughout the program's execution, similar to static variables. Global variables are accessible from any part of the program, and their lifetime ends when the program terminates. They are typically stored in a separate global data area.

It's important to note that the initialization and destruction of variables may also involve constructors and destructors, respectively, depending on their types and whether they are objects or primitive types.

Understanding the lifetime of variables is crucial for managing memory and avoiding potential issues like accessing variables after they have gone out of scope or using uninitialized variables.

33. Static components of the class

In C++, static components of a class are shared among all objects of that class. They are associated with the class itself rather than individual objects. Static components include static member variables and static member functions.

1.Static Member Variables: These are variables declared with the static keyword inside a class. They are shared by all objects of the class and have the same value across all instances. Static member variables are typically used to store class-wide information or shared data. They are initialized once, before any

object of the class is created, and their memory is allocated in a separate static data area. Static member variables can be accessed using the class name followed by the scope resolution operator (::) or within the class using the class name itself.

2.Static Member Functions: These are functions declared with the static keyword inside a class. Like static member variables, static member functions belong to the class rather than individual objects. They can be called using the class name followed by the scope resolution operator (::). Static member functions do not have access to non-static member variables or functions of the class, as they do not operate on a specific object. They are typically used for utility functions or operations that are not dependent on object-specific data.

Static member variables and functions can be accessed and used without creating an instance of the class. They are commonly used for tasks such as counting instances of a class, providing common functionality across objects, or storing shared resources.

34. Static class variables

In C++, static class variables are static member variables that belong to the class itself rather than individual objects of the class. They are shared among all objects of the class and have the same value for all instances. Static class variables are declared with the static keyword inside the class definition. Static class variables are commonly used when you need to maintain information or state that is shared among all instances of the class. They can be accessed and modified independently of any specific object of the class, and changes to the static class variable are reflected across all instances.

35. Static versus non-static components

In C++, static and non-static components have different characteristics and usage scenarios within a class. Here's a comparison between static and non-static components:

1. Static Components:

- Static Member Variables: Static member variables belong to the class itself rather than individual objects. They are shared among all instances of the class and have the same value across all objects. They are declared with the `static` keyword. Static member variables are typically used for class-wide data or shared resources.

- Static Member Functions: Static member functions are associated with the class rather than specific objects. They can be called without creating an instance of the class. Static member functions do not have access to non-static member variables or functions, as they do not operate on a specific object. They are declared with the `static` keyword and are often used for utility functions or operations that are not dependent on object-specific data.

2. Non-Static Components:

- Non-Static Member Variables: Non-static member variables are associated with individual objects of the class. Each object has its own copy of non-static member variables, and their values can vary across different instances. Non-static member variables are declared without the `static` keyword. They are used to represent object-specific data or state.

- Non-Static Member Functions: Non-static member functions operate on specific objects of the class. They have access to non-static member variables and can modify the state of individual objects. Non-static member functions are declared without the `static` keyword and are used to encapsulate behavior and operations specific to objects.

Key differences between static and non-static components:

- Static components are shared among all instances of the class, while non-static components are specific to individual objects.

- Static member variables have a single instance for the entire class, whereas non-static member variables have a separate instance for each object.

- Static member functions do not have access to non-static member variables or functions, while non-static member functions can access both static and non-static members.

- Static components can be accessed without creating an object of the class, whereas non-static components are accessed through object instances.

When deciding whether to use static or non-static components, consider the shared nature of the data or behavior you want to represent. If the data or behavior should be consistent across all objects, static components may be appropriate. On the other hand, if the data or behavior is specific to individual objects, non-static components should be used.

36. Passing an object by value

When passing an object by value in C++, a copy of the object is made and passed to the function or method. Any modifications made to the object within the function will not affect the original object outside of the function. Passing an object by value involves the following steps:

1.A new copy of the object is created, including all member variables and their values.

2.The copy of the object is passed to the function as a parameter.

3.Inside the function, the copy of the object is modified or used as needed.

4.Any changes made to the object within the function are local to that function and do not affect the original object.

37. Passing an object by reference

When passing an object by reference in C++, a reference to the original object is passed to the function or method. Any modifications made to the object within the function will affect the original object outside of the function. Passing an object by reference involves the following steps:

1.A reference variable is created in the function parameter list, which refers to the original object.

2.The reference is bound to the original object when the function is called.

3.Inside the function, the reference can be used to access and modify the original object.

4.Any changes made to the object within the function will directly affect the original object outside the function.

38. Friend classes

In C++, a friend class is a class that has access to the private and protected members of another class. It is a way to grant special access privileges to specific classes, allowing them to manipulate the private members of the class as if they were their own. Friend classes are declared using the friend keyword within the class definition. Friend classes can be useful in certain scenarios where you need to provide specific access to certain classes, while still maintaining encapsulation and data hiding. However, it's important to use friend classes judiciously, as granting excessive access can undermine the encapsulation and information hiding principles of object-oriented programming.

39. Git version control system and its usage

Git is a distributed version control system widely used for managing source code and tracking changes in software development projects, including those written in C++. Git provides a set of commands and features that enable collaboration, versioning, branching, merging, and tracking of changes in code repositories.

Here's an overview of how Git is used in C++ development:

1. Initializing a Git Repository: To start using Git in a C++ project, you typically initialize a Git repository in the root directory of your project. This is done by running the command `git init` in the terminal or using a Git client.

2. Tracking Files: Git allows you to track the changes in your project files. You can add specific files or entire directories to the Git repository using the `git add` command. For example, to add a C++ file named "main.cpp", you would use the command `git add main.cpp`.

3. Committing Changes: Once files are added, you can create a commit to save a snapshot of the changes. Commits act as milestones in the project history. You use the `git commit` command along with a commit message to create a commit. For example, `git commit -m "Added main.cpp file"`.

4. Branching and Merging: Git allows you to create branches to work on new features, bug fixes, or experiments without affecting the main codebase. You can create a branch using `git branch` and switch to it using `git checkout`. Once you are done with the changes in a branch, you can merge it back into the main branch using `git merge`.

5. Collaborating and Remote Repositories: Git supports collaboration by allowing you to connect to remote repositories hosted on services like GitHub, GitLab, or Bitbucket. You can push your local commits to a remote repository using `git push`, fetch the latest changes from the remote repository using `git fetch`, and merge those changes into your local branch using `git merge` or `git pull`.

6. Managing Versions: Git provides powerful tools to manage versions and navigate through the project history. You can view the commit log, compare different versions, revert changes, and create tags to mark specific versions or releases.

These are just a few of the fundamental concepts and commands in Git. Git offers a wide range of features and workflows that can be customized to fit the needs of a C++ development project. Learning Git and its various commands can greatly enhance collaboration, enable effective version control, and provide a history of changes for C++ projects.

40.Git version control systems commands

The Git version control system commands are not specific to any programming language, including C++. However, I can provide you with a list of commonly used Git commands that are used in the context of managing C++ projects. These commands can be executed in the command line or through Git clients:

1. `git init`: Initializes a new Git repository in the current directory.

2. `git clone <repository_url>`: Creates a copy of a remote Git repository on your local machine.

3. `git add <file>`: Adds a file or changes to the staging area, preparing them for the next commit. For example, `git add main.cpp`.

4. `git commit -m "<commit_message>"`: Creates a new commit with the changes in the staging area along with a commit message. For example, `git commit -m "Added main.cpp file"`.

5. `git status`: Shows the current status of the repository, including modified files and untracked files.

6. `git branch`: Lists all the branches in the repository. The current branch is indicated with an asterisk.

7. `git checkout <branch_name>`: Switches to the specified branch. For example, `git checkout feature-branch`.

8. `git merge <branch_name>`: Merges the specified branch into the current branch. For example, `git merge feature-branch`.

9. `git push`: Pushes the local commits to a remote repository. For example, `git push origin master` pushes the local commits to the remote repository named "origin" on the "master" branch.

10. `git pull`: Fetches the latest changes from a remote repository and merges them into the current branch.

11. `git fetch`: Retrieves the latest changes from a remote repository without merging them into the current branch.

12. `git log`: Shows the commit history, including commit messages, authors, dates, and commit hashes.

13. `git diff`: Displays the differences between the current code and the last committed version.

14. `git tag`: Lists all the tags in the repository, marking specific versions or releases.

These are some of the common Git commands used in C++ projects. It's recommended to refer to Git documentation or online resources for more detailed information about each command and their various options and use cases.

41.Write code of Adding new element to array's given index(Array size will be icrease)

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

int main() {
  int arr[100] = {1, 2, 3, 4, 5};
  int n = 5;
  int index = 2;
  int value = 10; //ortaya salınacaq rəqəm

  //elementlərin sağa çəkilməsi
  for(int i = n-1; i >= index; i--) {
    arr[i+1] = arr[i];
  }

  // yeni rəqəmin qoyulması
  arr[index] = value;
  n++; // arrayi böyütməsi
```

```
  for(int i = 0; i < n; i++) {
    cout << arr[i] << " ";
  }
  cout << endl;
  return 0;
}
```

42.Write code of Creating Linked List with structs

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

struct Node {
  int data;
  Node* next;
};

int main() {
  Node* head = NULL;
  Node* current = NULL;

  // ilk node yaradın və onu siyahının başı olaraq təyin edin
  head = new Node;
  head->data = 1;
  head->next = NULL;
  current = head;

  // ikinci node yaradın və onu siyahının sonuna əlavə edin
```

```
  current->next = new Node;
  current = current->next;
  current->data = 2;
  current->next = NULL;

  // üçüncü node yaradın və onu siyahının sonuna əlavə edin
  current->next = new Node;
  current = current->next;
  current->data = 3;
  current->next = NULL;

  // print the list
  current = head;
  while(current != NULL) {
    cout << current->data << " ";
    current = current->next;
  }
  cout << endl;

  current = head;
  while(current != NULL) {
    Node* temp = current;
    current = current->next;
    delete temp;
  }
  return 0;
}
```

43.Write code of Creating encapsulated class in C++

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

class Person {
  private:
    string name;
    int age;
  public:
    void setName(string n) {
      name = n;
    }
    void setAge(int a) {
      age = a;
    }
    string getName() {
      return name;
    }
    int getAge() {
      return age;
    }
};

int main() {
  Person p1; // Person classinin nümunəsini yaradın
  p1.setName("Vaqkus"); // adı "istədiyiniz hər hansı bir ad" olaraq təyin edin
  p1.setAge(33); // set age to 30
  cout << "Name: " << p1.getName() << endl; // print name
  cout << "Age: " << p1.getAge() << endl; // print age
  return 0;
}
```

44.Write code of Creating subclass and access super class members in within them C++

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

// superclass müəyyənləşdirin
class SuperClass {
public:
   int x; // Member variable
   SuperClass(int x) : x(x) {} // Constructor
   void printX() { // Member function
      cout << "x = " << x << endl;
   }
};

// Superclassdan miras qalan subclassi müəyyənləşdirin
class SubClass : public SuperClass {
public:
   SubClass(int x) : SuperClass(x) {} // Constructor
   void printXPlusOne() { // Member function
      cout << "x + 1 = " << x + 1 << endl; // inherited member variable daxil olun
   }
};

int main() {
   SubClass sc(5); //subclassın bir nümunəsini yaradın
   sc.printX(); // inherited member function cağirin
   sc.printXPlusOne(); // subclass member function cağirin
   return 0;
}
```

45.Write code of Creating polimorphism from class objects and use casting operator in C++

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

// base class müəyyənləşdirin
class Base {
public:
   virtual void print() { // Virtual function
      cout << "Base" << endl;
   }
};

// derived class müəyyənləşdirin
class Derived : public Base {
public:
   void print() override { //  virtual function override edilmesi
      cout << "Derived" << endl;
   }
};

int main() {
   Base* b = new Derived(); //  derived object ucun pointer yaradin
   b->print(); // Call virtual function (prints "Derived")
   static_cast<Derived*>(b)->print(); //casting operator islederek derived class function caqirilmasi (prints "Derived")
   delete b; // yaddasin temizlenmesi
   return 0;
```

```
}
```

46.Write code of Creating overriding of function of a class

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

// base class müəyyənləşdirin
class Base {
public:
   virtual void print() { // Virtual function
      cout << "Base" << endl;
   }
};

// derived class müəyyənləşdirin
class Derived : public Base {
public:
   void print() override { // Override virtual function
      cout << "Derived" << endl;
   }
};

int main() {
   Base* b = new Derived(); // derived object ucun pointer yaradin
   b->print(); // Call virtual function (prints "Derived")
   delete b; // yaddasin temizlenmesi
   return 0;
}
```

47.Write code of example to pass by value and pass by referance for functions

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

// Qiymet(value) üzrə arqument alan funksiya
void byValue(int x) {
   x = 10; // Modify the local copy of x
}

// Istinad(reference) üzrə arqument alan funksiya
void byReference(int& x) {
   x = 10; // Modify the original x
}

int main() {
   int a = 5;
   int b = 5;

   byValue(a); // Funksiyaya a-nı qiymet (value) kimi ötürün
   cout << "a = " << a << endl; // Prints "a = 5"

   byReference(b); // Funksiyaya b-ni istinad (reference) kimi ötürün
   cout << "b = " << b << endl; // Prints "b = 10"

   return 0;
}
```

48.Write code of Creating friend function in C++

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

class MyClass {
private:
   int x;
public:
   MyClass(int x) : x(x) {}
   friend void printX(MyClass& mc); // friend function müəyyənləşdirin
};

// friend function təyin edin
void printX(MyClass& mc) {
   cout << "x = " << mc.x << endl; // Access private member variable
}

int main() {
   MyClass mc(5);
   printX(mc); // Call friend function (prints "x = 5")
   return 0;
}
```

49.Write code of copying of constructors with different ways

Copying Constructors in C++:

There are two common ways to copy constructors in C++: the default copy constructor and the user-defined copy constructor.

a) Default Copy Constructor:

```
class MyClass {
public:
```

```
    int data;

    MyClass(const MyClass& other) {
        data = other.data;
    }
};
```

b) User-defined Copy Constructor:

```
class MyClass {
public:
    int data;

    MyClass(int value) : data(value) {}

    MyClass(const MyClass& other) {
        data = other.data;
    }
};
```

<u>50.Write code of usage of destructions</u>

```
#include <iostream>

class MyClass {
public:
    MyClass() {
        std::cout << "Constructor called." << std::endl;
    }
```

```
    ~MyClass() {
        std::cout << "Destructor called." << std::endl;
    }
};

int main() {
    MyClass obj; // Create an object of MyClass

    // Rest of the program

    return 0;
}
```

In the code above, we define a class MyClass with a constructor and a destructor. In the main function, we create an object obj of MyClass. When the object obj goes out of scope (i.e., at the end of the main function), the destructor of MyClass is automatically called.

51.Write C++ code for adding and removing element to Stack

```
#include <iostream>
#include <stack>

int main() {
    std::stack<int> myStack;

    // Adding elements to the stack
    myStack.push(10);
    myStack.push(20);
    myStack.push(30);

    // Removing elements from the stack
    while (!myStack.empty()) {
```

```
    int element = myStack.top();
    myStack.pop();
    std::cout << "Popped element: " << element << std::endl;
  }

  return 0;
}
```

52.Write C++ code for adding and removing element to Queue

```
#include <iostream>
#include <queue>

int main() {
  std::queue<int> myQueue;

  // Adding elements to the queue
  myQueue.push(10);
  myQueue.push(20);
  myQueue.push(30);

  // Removing elements from the queue
  while (!myQueue.empty()) {
    int element = myQueue.front();
    myQueue.pop();
    std::cout << "Removed element: " << element << std::endl;
  }

  return 0;
```

```
}
```

53.Write C++ code for handling dividing by zero exception

```
#include <iostream>
#include <stdexcept>

int divide(int a, int b) {
   if (b == 0) {
      throw std::runtime_error("Division by zero!");
   }

   return a / b;
}

int main() {
   try {
      int result = divide(10, 0);
      std::cout << "Result: " << result << std::endl;
   } catch (const std::runtime_error& error) {
      std::cout << "Exception caught: " << error.what() << std::endl;
   }

   return 0;
}
```

54.Write C++ code for accesing static member from non-static one

```
#include <iostream>

class MyClass {
public:
   static int staticMember;
   int nonStaticMember;

   void printMembers() {
      std::cout << "Static Member: " << staticMember << std::endl;
      std::cout << "Non-Static Member: " << nonStaticMember << std::endl;
   }
};

int MyClass::staticMember = 10;

int main() {
   MyClass obj;
   obj.nonStaticMember = 20;
   obj.printMembers();

   return 0;
}
```

In the code above, we define a class MyClass with a static member staticMember and a non-static member nonStaticMember. The static member is accessed using the class name, while the non-static member is accessed using the object instance.

55.Write C++ code for finding count of all objects of classes

```
#include <iostream>
```

```
class MyClass1 {
   static int count;

public:
   MyClass1() {
      count++;
   }

   ~MyClass1() {
      count--;
   }

   static int getCount() {
      return count;
   }
};

int MyClass1::count = 0;

class MyClass2 {
   static int count;

public:
   MyClass2() {
      count++;
   }
```

```
    ~MyClass2() {
        count--;
    }

    static int getCount() {
        return count;
    }
};

int MyClass2::count = 0;

int main() {
    MyClass1 obj1_1;
    MyClass1 obj1_2;
    MyClass2 obj2_1;
    MyClass2 obj2_2;

    std::cout << "Count of MyClass1 objects: " << MyClass1::getCount() <<
std::endl;
    std::cout << "Count of MyClass2 objects: " << MyClass2::getCount() <<
std::endl;

    return 0;
}
```

56.Write git commands for adding files to local repository

To add C++ files to a local Git repository, you can use the following Git commands:

1.Initialize a new Git repository (if not already initialized):

git init(charsp)

2. Check the status of the repository to see which files are not yet tracked:

git status(lua)

3. Add a single C++ file to the staging area:

git add <filename.cpp> (csharp)

Replace <filename.cpp> with the name of the C++ file you want to add. For example, git add main.cpp.

4. Add multiple C++ files to the staging area at once:

git add <file1.cpp> <file2.cpp> <file3.cpp> ...(php)

Replace **<file1.cpp>, <file2.cpp>, <file3.cpp>,** and so on with the names of the C++ files you want to add. For example, **git add file1.cpp file2.cpp file3.cpp.**

5. Add all modified and new C++ files to the staging area:

git add .(csharp)

The . symbol represents the current directory.

6. Verify the changes you've made are staged correctly by checking the status again:

git status(lua)

Once you have added the C++ files to the staging area, you can proceed with committing the changes using the git commit command.

57.Write git commands for adding files to remote repository

To add C++ files to a remote Git repository, you can use the following Git commands:

1. Clone the remote repository to your local machine (if not already cloned):

```bash
git clone <remote_repository_url>
```

Replace `<remote_repository_url>` with the URL of the remote Git repository. For example, `git clone https://github.com/username/repo.git`.

2. Navigate to the cloned repository directory:

```bash
cd <repository_directory>
```

Replace `<repository_directory>` with the name of the directory where the repository was cloned.

3. Check the status of the repository to see which files are not yet tracked:

3. Check the status of the repository to see which files are not yet tracked:

```lua
git status
```

4. Add a single C++ file to the staging area:

```csharp
git add <filename.cpp>
```

Replace `<filename.cpp>` with the name of the C++ file you want to add. For example, `git add main.cpp`.

5. Add multiple C++ files to the staging area at once:

```php
git add <file1.cpp> <file2.cpp> <file3.cpp> ...
```

Replace `<file1.cpp>`, `<file2.cpp>`, `<file3.cpp>`, and so on with the names of the C++ files you want to add. For example, `git add file1.cpp file2.cpp file3.cpp`.

6. Add all modified and new C++ files to the staging area:

csharp Copy code

```
git add .
```

The `.` symbol represents the current directory.

7. Verify the changes you've made are staged correctly by checking the status again:

lua Copy code

```
git status
```

8. Commit the changes:

sql Copy code

```
git commit -m "Add C++ files"
```

8. Commit the changes:

sql Copy code

```
git commit -m "Add C++ files"
```

Replace the commit message `"Add C++ files"` with an appropriate message describing the changes.

9. Push the changes to the remote repository:

perl Copy code

```
git push origin <branch_name>
```

Replace `<branch_name>` with the name of the branch where you want to push the changes. For example, `git push origin main`.

Note: Before pushing the changes, ensure that you have the necessary permissions to write to the remote repository. If you do not have write access, you may need to fork the repository or request access from the repository owner.

58.Write git commands for download file from remote repository

To download a file from a remote repository using Git, you need to clone the repository. Here are the steps to clone a remote repository and download a file for a C++ project:

1. Open your terminal or Git Bash.
2. Navigate to the directory where you want to download the file.
3. Use the `git clone` command followed by the URL of the remote repository:

```shell
git clone <repository-url>
```

   Replace `<repository-url>` with the URL of the remote repository you want to clone.
4. Change your current directory to the cloned repository:

```shell
cd <repository-name>
```

   Replace `<repository-name>` with the name of the repository you just cloned.
5. List the files in the repository to find the desired file:

```shell
ls
```

Regenerate response

   This command will display all the files and directories in the repository.
6. Once you locate the file you want to download, you can either copy it manually or use the `git checkout` command to download the file to your current directory:

```shell
git checkout <branch-name> -- <path-to-file>
```

   Replace `<branch-name>` with the name of the branch where the file is located and `<path-to-file>` with the relative path to the file within the repository.
   For example, if the file you want to download is located in the `src` directory on the `main` branch, the command would be:

```shell
git checkout main -- src/file.cpp
```

   This will download the `file.cpp` to your current directory.

Now you have successfully downloaded the file from the remote repository to your local machine. You can use it for your C++ project as needed.

59.Write C++ code for throw errors in some cases

```
#include <iostream>
#include <stdexcept>

void divideNumbers(int dividend, int divisor) {
   if (divisor == 0) {
      throw std::runtime_error("Division by zero error!");
   }

   int result = dividend / divisor;
   std::cout << "Result: " << result << std::endl;
}

int main() {
   int dividend, divisor;

   std::cout << "Enter dividend: ";
   std::cin >> dividend;

   std::cout << "Enter divisor: ";
   std::cin >> divisor;

   try {
      divideNumbers(dividend, divisor);
   } catch (const std::runtime_error& error) {
      std::cerr << "Error: " << error.what() << std::endl;
```

```
    }

    return 0;
}
```

---

60. Write C++ code for constant pointers

```
#include <iostream>

int main() {
    int value = 5;
    int anotherValue = 10;

    // Constant pointer to an integer
    int* const ptr = &value;

    std::cout << "Value: " << *ptr << std::endl; // Output: Value: 5

    *ptr = 8;
    std::cout << "Modified value: " << *ptr << std::endl; // Output: Modified value: 8

    // Compilation error: ptr cannot be reassigned
    // ptr = &anotherValue;

    return 0;
}
```