1. **Class and Objects**

Classes and objects are fundamental concepts in object-oriented programming (OOP). In OOP, a class is a blueprint for creating objects that have similar properties and behaviors. An object, on the other hand, is an instance of a class.

A class defines the attributes and methods that an object of that class will have. Attributes are the characteristics of the object, while methods define the actions or behaviors of the object. For example, a class "Car" might have attributes such as "make," "model," and "year," and methods such as "start_engine" and "accelerate."

To create an object of a class, you first need to instantiate it. This involves using the class to create a new object in memory. Once you have an object of a class, you can access its attributes and methods using the dot notation.

2.Feature/characteristics of OOPs.

Object-oriented programming (OOP) is a programming paradigm that is based on the concept of objects, which are instances of classes. OOP has several features or characteristics that set it apart from other programming paradigms. Here are some of the main features of OOP:

1. **Encapsulation: OOP allows you to encapsulate data and methods within objects, which means that they can only be accessed by the object itself or by other objects that are authorized to access them.**
2. **Inheritance: OOP allows you to create new classes based on existing ones. Inheritance allows you to reuse code and extend the functionality of existing classes.**
3. **Polymorphism: OOP allows you to use a single interface to represent multiple types of objects. Polymorphism makes it easy to write code that can work with different types of objects.**
4. **Abstraction: OOP allows you to focus on the essential features of an object, while hiding the implementation details. This makes it easier to design, implement, and maintain complex systems.**
5. **Modularity: OOP allows you to break down a system into smaller, independent modules, each of which can be developed and tested separately. This makes it easier to maintain and update the system over time.**
6. **Message Passing: OOP allows objects to communicate with each other by sending messages. This allows objects to work together to accomplish complex tasks.**

These features of OOP make it a powerful paradigm for developing complex software systems. By encapsulating data and methods within objects, using inheritance to reuse code, and using polymorphism to work with multiple types of objects, OOP makes it easier to design, implement, and maintain complex software systems.

3. Compile time and Runtime polymorphism.

Polymorphism is one of the fundamental features of object-oriented programming (OOP), which allows a single interface to be used to represent multiple types of objects. In OOP, polymorphism can be achieved at both compile time and runtime.

Compile-time Polymorphism:

1. **Compile-time polymorphism is also known as static polymorphism or method overloading. In this type of polymorphism, the compiler determines which function to call based on the number and types of arguments passed to the function. Method overloading allows you to define multiple functions with the same name but different parameter lists.**

For example, consider the following C++ code:

```
int add(int x, int y);
float add(float x, float y);
```

In this example, we have defined two functions with the same name "add", but different parameter lists. The first function takes two integers as arguments, while the second function takes two floats. When we call the "add" function with two integers, the compiler will select the first function, while if we call it with two floats, the compiler will select the second function.

Runtime Polymorphism:

2. **Runtime polymorphism is also known as dynamic polymorphism or method overriding. In this type of polymorphism, the function to be called is determined at runtime based on the type of object that the function is called on. Method overriding allows you to define a function in a subclass with the same name and signature as a function in its superclass.**

For example, consider the following Java code:

```
class Animal {
  public void speak() {
    System.out.println("The animal speaks");
  }
}

class Dog extends Animal {
  public void speak() {
    System.out.println("The dog barks");
  }
}

class Cat extends Animal {
  public void speak() {
    System.out.println("The cat meows");
  }
}
```

In this example, we have defined three classes: Animal, Dog, and Cat. The Animal class has a "speak" method that prints "The animal speaks" to the console. The Dog and Cat classes override this method with their own implementation. When we call the "speak" method on a Dog object, the "speak" method of the Dog class will be called, and when we call it on a Cat object, the "speak" method of the Cat class will be called.

In summary, compile-time polymorphism is achieved through method overloading, while runtime polymorphism is achieved through method overriding. Both types of polymorphism allow you to write more flexible and reusable code in object-oriented programming.

4. Variable scopes.

5. static (variables, Functions, Objects).
In programming, the term "static" can refer to several different things, including static variables, static functions, and static objects. Here is an explanation of each:

Static variables:

1. **Static variables are variables that are declared with the static keyword and are associated with a class rather than with an instance of the class. They are stored in the data segment of memory and are initialized to zero by default. Static variables are created when the program starts and are destroyed when the program exits. They are accessible only within the scope of the class that declares them and can be used to share data between instances of the class.**

Static functions:

2. **Static functions are functions that are declared with the static keyword and are associated with a class rather than with an instance of the class. They are also known as class methods. Static functions can be called without creating an instance of the class, and they can access only static data members of the class. They are often used to implement utility functions that do not depend on the state of a particular object.**

Static objects:

3. **Static objects are objects that are declared with the static keyword and are associated with a class rather than with an instance of the class. They are**

created when the program starts and are destroyed when the program exits. Static objects can be used to store data that needs to be shared between instances of the class, or to maintain global state in a program. They are accessible only within the scope of the class that declares them.

In summary, static variables, functions, and objects are associated with a class rather than with an instance of the class. They are created when the program starts and are destroyed when the program exits. Static variables can be used to share data between instances of a class, static functions can be used to implement utility functions, and static objects can be used to store data that needs to be shared between instances of a class or to maintain global state in a program.

6.Inheritence (Type and Mode)

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows one class to inherit properties and behaviors from another class. There are two types of inheritance:

Single Inheritance:

1. In single inheritance, a class inherits properties and behaviors from a single parent class. That is, there is only one base class from which the derived class is derived. For example, in the following code, class B is derived from class A using single inheritance:

Multiple Inheritance:

2. In multiple inheritance, a class can inherit properties and behaviors from multiple parent classes. That is, a derived class can have multiple base classes. For example, in the following code, class C is derived from both class A and class B using multiple inheritance:

There are also two modes of inheritance:

Public Inheritance:

1. **In public inheritance, the public members of the base class are inherited as public members of the derived class, the protected members of the base class are inherited as protected members of the derived class, and the private members of the base class are not inherited. This means that public members of the base class are accessible to the outside world through the derived class.**

Protected Inheritance:

2. **In protected inheritance, the public and protected members of the base class are inherited as protected members of the derived class, and the private members of the base class are not inherited. This means that public and protected members of the base class are accessible to the derived class and its subclasses, but not to the outside world.**

In summary, inheritance is a fundamental concept in OOP that allows a class to inherit properties and behaviors from one or more parent classes. There are two types of inheritance, single and multiple, and two modes of inheritance, public and protected

7. Virtual (Functions and Class)

Virtual functions and virtual classes are important concepts in object-oriented programming (OOP) that allow for dynamic polymorphism.

Virtual Functions:

1. **A virtual function is a member function that is declared with the virtual keyword in the base class and is intended to be overridden in the derived class. When a virtual function is called through a pointer or reference to a base class object, the actual function called is determined at runtime based on the type of the object pointed to or referred to. This allows for dynamic binding of the function call to the appropriate derived class implementation. Here is an example:**

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

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

int main() {
    Base* ptr = new Derived;
    ptr->foo(); // calls Derived::foo()
    delete ptr;
    return 0;
}
```

In this example, the `foo()` function is declared as virtual in the `Base` class and is overridden in the `Derived` class. When the `foo()` function is called through a pointer to a `Base` object that actually points to a `Derived` object, the `Derived` version of the function is called at runtime.

Virtual Classes:

2. A virtual class is a class that has at least one virtual function declared in it. A virtual class cannot be instantiated directly; it must be subclassed. The main purpose of virtual classes is to provide a common interface for a group of related classes, which allows them to be used interchangeably in code that uses dynamic binding. Here is an example:

```
class Shape {
public:
    virtual void draw() = 0;
};
```

```
class Circle : public Shape {
public:
    void draw() override {
        // draw a circle
    }
};

class Square : public Shape {
public:
    void draw() override {
        // draw a square
    }
};

int main() {
    Shape* shapes[2];
    shapes[0] = new Circle;
    shapes[1] = new Square;
    for (int i = 0; i < 2; i++) {
        shapes[i]->draw(); // calls the appropriate derived class implementation
```

```
        delete shapes[i];
    }
    return 0;
}
```

In this example, the `Shape` class is declared as virtual and has a pure virtual function `draw()`. This allows it to be used as a common interface for the `Circle` and `Square` classes. The `draw()` function is overridden in both the `Circle` and `Square` classes to provide specific implementations for each shape. When the `draw()` function is called on a pointer to a `Shape` object that actually points to a `Circle` or `Square` object, the appropriate derived class implementation is called at runtime.

8. Abstract class and Interface.

In object-oriented programming (OOP), an abstract class and an interface are both used to define a set of methods that must be implemented by a derived class. However, they are different in their implementation and usage.

Abstract Class:

1. An abstract class is a class that cannot be instantiated directly and is meant to serve as a base class for other classes to inherit from. It contains one or more pure virtual functions, which means that these functions do not have a default implementation and must be overridden in the derived class. An abstract class can also contain concrete functions with implementations, as well as data members. Here is an example:

```
class Shape {
public:
    virtual double area() = 0;
```

```
    virtual double perimeter() = 0;
    virtual ~Shape() {}
};

class Rectangle : public Shape {
public:
    double area() override {
        // calculate the area of the rectangle
    }
    double perimeter() override {
        // calculate the perimeter of the rectangle
    }
};

class Circle : public Shape {
public:
    double area() override {
        // calculate the area of the circle
    }
    double perimeter() override {
        // calculate the perimeter of the circle
    }
};

int main() {
    Shape* shapes[2];
    shapes[0] = new Rectangle;
    shapes[1] = new Circle;
    for (int i = 0; i < 2; i++) {
        std::cout << "Area: " << shapes[i]->area() << std::endl;
        std::cout << "Perimeter: " << shapes[i]->perimeter() << std::endl;
        delete shapes[i];
    }
```

```
    return 0;
}
```

In this example, the `Shape` class is an abstract class that defines the pure virtual functions `area()` and `perimeter()`. The `Rectangle` and `Circle` classes inherit from `Shape` and implement these functions. The `main()` function creates an array of `Shape` pointers that point to `Rectangle` and `Circle` objects, and calls the `area()` and `perimeter()` functions through these pointers, which calls the appropriate implementation in the derived classes.

Interface:

2. **An interface is a collection of abstract methods with no implementation. It defines a set of methods that a class must implement, but does not provide any implementation itself. In C++, an interface can be defined using a pure abstract class, similar to an abstract class, but with all the member functions being pure virtual functions. Here is an example:**

```
class Printable {
public:
    virtual void print() = 0;
    virtual ~Printable() {}
};

class Document : public Printable {
public:
    void print() override {
```

```
        // print the document
    }
};


class Image : public Printable {
public:
    void print() override {
        // print the image
    }
};


int main() {
    Printable* items[2];
    items[0] = new Document;
    items[1] = new Image;
    for (int i = 0; i < 2; i++) {
        items[i]->print(); // calls the appropriate implementation in the derived classes
        delete items[i];
    }
    return 0;
}
```

```
}
```

In this example, the `Printable` class is an interface that defines the pure virtual function `print()`. The `Document` and `Image` classes inherit from `Printable` and implement this function. The `main()` function creates an array of `Printable` pointers that point to `Document` and `Image` objects, and calls the `print()` function through these pointers, which calls the appropriate implementation in the derived classes.

In summary, an abstract class is a class that cannot be instantiated

9. Friend function and Friend class.

In C++, a friend function or class is a function or class that is granted access to the private and protected members of another class. This allows the friend function or class to access and modify the private or protected data members of the class, which would otherwise not be accessible.

Friend Function:

1. **A friend function is a function that is declared with the keyword `friend` inside the class, but is not a member function of the class. It can be declared in any scope, such as global scope or another class, and can be called like a normal function, but has access to the private and protected members of the class. Here is an example:**

```
class MyClass {

private:

   int x;

public:

   MyClass(int a) : x(a) {}
```

```
    friend void printX(MyClass obj);
};

void printX(MyClass obj) {
    std::cout << obj.x << std::endl;
}

int main() {
    MyClass obj(5);
    printX(obj); // calls the friend function to print the private member x of MyClass
    return 0;
}
```

In this example, the `printX()` function is declared as a friend function inside the `MyClass` class, which allows it to access the private member `x`. The `main()` function creates a `MyClass` object and calls the `printX()` function to print the private member `x`.

Friend Class:

2. A friend class is a class that is declared with the keyword `friend` inside another class, and has access to the private and protected members of the class that declared it as a friend. A friend class can be used to provide access to the

private and protected members of a class to another class, without using public member functions or friend functions. Here is an example:

```
class MyClass {

private:

   int x;

   friend class MyFriendClass;

};



class MyFriendClass {

public:

   void printX(MyClass obj) {

      std::cout << obj.x << std::endl;

   }

};



int main() {

   MyClass obj(5);

   MyFriendClass friendObj;

   friendObj.printX(obj); // calls the member function of the friend class to print the private member x of MyClass

   return 0;

}
```

In this example, the `MyFriendClass` class is declared as a friend class inside the `MyClass` class, which allows it to access the private member `x`. The `MyFriendClass` class has a member function `printX()` that takes a `MyClass` object as a parameter and prints its private member `x`. The `main()` function creates a `MyClass` object and a `MyFriendClass` object, and calls the `printX()` function of the `MyFriendClass` object to print the private member `x` of the `MyClass` object.

In summary, friend functions and classes are useful when it is necessary to grant access to the private and protected members of a class to another function or class. However, they should be used with caution, as they can break encapsulation and make the code more difficult to maintain.

10. Call by value, refrence.

In programming, when a function is called, arguments can be passed to the function in two ways: by value and by reference.

Call by value:

1. In call by value, a copy of the value of the argument is passed to the function. The function works with this copy of the argument, and any changes made to the argument inside the function are not reflected in the original value of the argument. This means that the original value of the argument remains unchanged. Here is an example:

```
void increment(int x) {
    x++;
    std::cout << "Inside function: " << x << std::endl;
}

int main() {
    int num = 5;
```

```
    increment(num); // passing num by value
    std::cout << "Outside function: " << num << std::endl;
    return 0;
}
```

In this example, the `increment()` function takes an integer `x` as an argument, increments its value by 1, and prints its value inside the function. The `main()` function creates an integer `num` with a value of 5 and passes it to the `increment()` function by value. The output shows that the value of `num` outside the function is still 5, because the function works with a copy of the value of `num`.

Call by reference:

2. **In call by reference, a reference to the original value of the argument is passed to the function. This means that any changes made to the argument inside the function are reflected in the original value of the argument. Here is an example:**

```
void increment(int &x) {
    x++;
    std::cout << "Inside function: " << x << std::endl;
}

int main() {
    int num = 5;
    increment(num); // passing num by reference
    std::cout << "Outside function: " << num << std::endl;
    return 0;
```

```
}
```

In this example, the `increment()` function takes an integer reference `x` as an argument, increments its value by 1, and prints its value inside the function. The `main()` function creates an integer `num` with a value of 5 and passes it to the `increment()` function by reference. The output shows that the value of `num` outside the function is now 6, because the function works with a reference to the original value of `num`.

In summary, call by value and call by reference are two ways of passing arguments to functions. Call by value works with a copy of the value of the argument, while call by reference works with a reference to the original value of the argument, allowing changes made inside the function to be reflected in the original value of the argument.

11. This pointer

In C++, `this` is a pointer that holds the memory address of the current object. It is a keyword that is used to refer to the current instance of a class within its own member functions.

When a member function is called on an object, a pointer to that object is automatically passed as a hidden argument to the function. This pointer is named `this`. The `this` pointer allows the member function to access the object's data members and member functions.

Here is an example:

```
class Person {
public:
  void setName(std::string name) {
    this->name = name;
  }
```

```
    std::string getName() {
        return this->name;
    }

private:
    std::string name;
};

int main() {
    Person person1;
    person1.setName("Alice");
    std::cout << person1.getName() << std::endl;
    return 0;
}
```

In this example, the `Person` class has a `setName()` function that takes a `name` parameter and sets the object's `name` data member to that value using the `this` pointer. The `getName()` function returns the object's `name` data member using the `this` pointer.

In the `main()` function, a `Person` object `person1` is created, its name is set to "Alice" using the `setName()` function, and its name is printed using the `getName()` function.

In summary, the `this` pointer is a keyword in C++ that is used to refer to the current object within its own member functions. It allows the member functions to access the object's data members and member functions.

12. Abstraction

Abstraction is a fundamental concept in object-oriented programming that refers to the act of representing complex real-world entities in a simplified way. It involves focusing only on the essential features of an entity and ignoring its unnecessary details, making it easier to understand and use.

In programming, abstraction can be achieved through the use of abstract classes and interfaces. Abstract classes are classes that cannot be instantiated, but serve as

blueprints for other classes that inherit from them. They define a set of abstract methods that must be implemented by any class that inherits from them. Abstract classes can also have non-abstract methods and fields that are shared by all the classes that inherit from them.

Interfaces, on the other hand, are like abstract classes, but they define only abstract methods and constants. Classes that implement an interface must provide an implementation for all the methods defined in the interface. Interfaces are used to define a contract that a class must adhere to, without dictating how the class should be implemented.

Here is an example of using abstraction in C++:

```
// Abstract class
class Shape {
public:
  virtual void draw() = 0; // pure virtual function
};

// Concrete classes that inherit from Shape
class Rectangle : public Shape {
public:
  void draw() {
    std::cout << "Drawing a rectangle" << std::endl;
  }
};

class Circle : public Shape {
public:
  void draw() {
    std::cout << "Drawing a circle" << std::endl;
  }
};

int main() {
```

```
    Shape* shape1 = new Rectangle();
    Shape* shape2 = new Circle();
    shape1->draw();
    shape2->draw();
    return 0;
}
```

In this example, the `Shape` class is an abstract class that defines a pure virtual function `draw()`, which is a method that must be implemented by any class that inherits from `Shape`. The `Rectangle` and `Circle` classes are concrete classes that inherit from `Shape` and provide their own implementations of the `draw()` method. In the `main()` function, two `Shape` pointers are created and assigned to new `Rectangle` and `Circle` objects. The `draw()` method is then called on each of the objects using the `Shape` pointers, which allows the polymorphic behavior of the program.

In summary, abstraction is a fundamental concept in object-oriented programming that involves representing complex real-world entities in a simplified way. In C++, abstraction can be achieved through the use of abstract classes and interfaces, which define a set of methods that must be implemented by any class that inherits from them.

13. Exception Handling

Exception handling is a programming mechanism that allows a program to handle errors or exceptional conditions that occur during runtime. When an exceptional condition occurs, the program throws an exception object that contains information about the error, and the exception is then caught and handled by the program.

In C++, exceptions are handled using the try-catch block. The try block contains the code that may throw an exception, and the catch block handles the exception if it is thrown. The catch block specifies the type of exception it can handle and the code to handle the exception.

Here is an example:

```
#include <iostream>
```

```
int main() {
   try {
      int x = 10, y = 0;
      if (y == 0) {
         throw "Division by zero";
      }
      int z = x / y;
      std::cout << z << std::endl;
   }
   catch (const char* message) {
      std::cout << "Error: " << message << std::endl;
   }
   return 0;
}
```

In this example, the program attempts to divide an integer `x` by zero, which is an exceptional condition that will cause the program to throw an exception. The `throw` statement throws an exception object containing the message "Division by zero". The catch block catches the exception and prints the error message to the console.

There are several types of exceptions that can be thrown in C++, including built-in types like `int`, `char`, and `float`, as well as user-defined types. When defining a user-defined exception type, it is best to derive from the `std::exception` class.

In addition to the try-catch block, C++ also provides the `throw` keyword, which allows a program to explicitly throw an exception, and the `finally` keyword, which can be used to specify code that should be executed regardless of whether an exception is thrown or not.

In summary, exception handling is a programming mechanism that allows a program to handle errors or exceptional conditions that occur during runtime. In C++, exceptions are handled using the try-catch block, which catches the exception and handles it by executing the code in the catch block.

14. Constructor and Destructor.

Constructor and destructor are special member functions in C++ that are automatically called when an object is created and destroyed, respectively.

A constructor is a special member function that is called automatically when an object is created. It is used to initialize the data members of the object. Constructors can have parameters, and they can be overloaded to provide different ways of initializing the object. If a class does not define a constructor, the compiler will generate a default constructor that takes no arguments.

Here is an example of a constructor:

```
class MyClass {
public:
  MyClass(int x, int y) {
    this->x = x;
    this->y = y;
  }
private:
  int x;
  int y;
};
```

In this example, `MyClass` defines a constructor that takes two `int` arguments and initializes the data members `x` and `y`.

A destructor is a special member function that is called automatically when an object is destroyed. It is used to free any resources that were allocated by the object. Destructors have no parameters, and they cannot be overloaded. If a class does not define a destructor, the compiler will generate a default destructor that does nothing.

Here is an example of a destructor:

```
class MyClass {
public:
  ~MyClass() {
    // Free any resources here
  }
private:
  // Data members here
};
```

In this example, `MyClass` defines a destructor that frees any resources that were allocated by the object.

It is important to note that a class can have multiple constructors and destructors, and they can be overloaded to provide different ways of initializing or cleaning up the object. Constructors and destructors are automatically called when an object is created or destroyed, respectively, and they play an important role in managing the lifetime of objects in a C++ program.

15. Copy constructor

A copy constructor is a special type of constructor in object-oriented programming that creates a new object as a copy of an existing object. It is used to initialize an object with the values of another object of the same class.

In C++, a copy constructor is a constructor that takes a reference to an object of the same class as its parameter. It is used to create a new object that is a copy of the object passed as a parameter.

Here's an example of a copy constructor in C++:

```
class MyClass {
public:
    // default constructor
    MyClass() {
        // initialize members
    }

    // copy constructor
    MyClass(const MyClass& other) {
        // copy members
    }

    // other member functions and variables
};

member functions and variables };
```

The copy constructor takes a constant reference to an object of the same class (`const MyClass& other`) and initializes the new object with the values of the members of the object passed as a parameter. If the object has any dynamically allocated memory, the copy constructor should make a deep copy to avoid issues with memory management.

The copy constructor is called automatically in certain situations, such as when an object is passed by value to a function, or when an object is returned by value from a function.

16. copy assignemnt operator

The copy assignment operator in C++ is a special member function that allows an object to be assigned the value of another object of the same class. It is denoted by the "=" operator and has the following signature:

```
class_name& operator=(const class_name& other);
```

The copy assignment operator is used when an already initialized object is assigned a new value of the same type. It is called whenever an assignment statement is used with an existing object on the left-hand side and another object of the same type on the right-hand side.

Here is an example of a simple copy assignment operator for a class called `Person`:

```
class Person {

public:
```

```
    // Constructor
    Person(const std::string& name, int age) : name_(name), age_(age) {}

    // Copy assignment operator
    Person& operator=(const Person& other) {
        name_ = other.name_;
        age_ = other.age_;
        return *this;
    }

private:
    std::string name_;
    int age_;
};
```

In this example, the copy assignment operator is defined to copy the `name_` and `age_` members from the `other` object to the current object, and then return a reference to the current object (`*this`). This allows for chained assignment, such as `Person a, b, c; a = b = c;`.

17. References variable

In C++, a reference variable is an alias or an alternative name for an existing variable. It allows accessing and modifying the value of the original variable using a different name or identifier. A reference variable provides a way to pass arguments to functions by reference instead of by value, which can improve performance and reduce memory usage.

Here's an example of how to declare a reference variable in C++:

c

Copy code

```
int main     int     10  int

               cout         endl                cout         endl

return 0
```

In this example, `y` is a reference to `x`. When `y` is modified, it actually modifies the value of `x`. The output shows that both `x` and `y` have the same value of 11.

It's important to note that a reference variable must be initialized when it is declared and cannot be changed to refer to a different variable later. Additionally, a reference variable cannot refer to a null value or be null itself.

Reference variables are commonly used in C++ to pass arguments to functions by reference, which allows the function to modify the original value of the argument. They are also used to implement operator overloading and to create aliases for complex data types.

18.Overloading (Function, Constructor, Operator)

Overloading is a feature in C++ that allows multiple functions or operators to have the same name but different parameters. This makes it possible to create functions and operators that behave differently depending on the type and number of their arguments. Overloading can be applied to functions, constructors, and operators.

Function Overloading:

Function overloading allows multiple functions with the same name to be defined in a class or namespace. These functions must have different parameter lists. The compiler determines which function to call based on the arguments passed to the function.

Here is an example of function overloading:

```
class MyClass {
public:
  void print(int x) {
    std::cout << "Integer: " << x << std::endl;
  }
  void print(float x) {
    std::cout << "Float: " << x << std::endl;
  }
};

int main() {
  MyClass obj;
  obj.print(10);
  obj.print(3.14f);
  return 0;
```

```
}
```

In this example, `MyClass` defines two functions with the same name `print` but different parameter lists. The first function takes an integer parameter, and the second function takes a float parameter. The program creates an object of `MyClass` and calls both functions with different arguments.

Constructor Overloading:

Constructor overloading allows multiple constructors with the same name to be defined in a class. These constructors must have different parameter lists. The compiler determines which constructor to call based on the arguments passed to the constructor.

Here is an example of constructor overloading:

```
class MyClass {
public:
    MyClass() {
        std::cout << "Default constructor called" << std::endl;
    }
    MyClass(int x) {
        std::cout << "Parameterized constructor called with " << x << std::endl;
    }
};

int main() {
    MyClass obj1;
    MyClass obj2(10);
    return 0;
}
```

In this example, `MyClass` defines two constructors with the same name `MyClass` but different parameter lists. The first constructor takes no parameters, and the second constructor takes an integer parameter. The program creates two objects of `MyClass` and calls both constructors with different arguments.

Operator Overloading:

Operator overloading allows operators to be redefined to work with user-defined data types. This makes it possible to use operators with user-defined data types just like built-in data types. Operator overloading is achieved by defining a function with the same name as the operator, followed by the keyword `operator`.

Here is an example of operator overloading:

```
class Complex {
public:
    Complex operator+(const Complex& other) const {
        Complex result;
        result.real = real + other.real;
        result.imaginary = imaginary + other.imaginary;
        return result;
    }
private:
    double real;
    double imaginary;
};

int main() {
    Complex c1, c2, c3;
    // Code to initialize c1 and c2
    c3 = c1 + c2; // Operator+ is called with c1 and c2 as arguments
    return 0;
}
```

In this example, `Complex` defines an operator `+` that takes another `Complex` object as its parameter and returns a new `Complex` object that is the sum of the two. The program creates three objects of `Complex` and adds `c1` and `c2` using the `+` operator. The `+` operator is called with `c1` and `c2` as its arguments, and it returns a new `Complex` object that is assigned to `c3`.

18. Function overriding and Inline function.

Function Overriding:

Function overriding is a feature in C++ that allows a derived class to provide its own implementation for a function that is already defined in its base class. The function must have the same name, return type, and parameter list in both the base and derived classes. The function in the derived class overrides the function in the base class.

Here is an example of function overriding:

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

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

```
int main() {
    Base* obj1 = new Base();
    obj1->print(); // Output: Base class print function

    Base* obj2 = new Derived();
    obj2->print(); // Output: Derived class print function
    return 0;
}
```

In this example, `Base` defines a virtual function `print`, and `Derived` overrides the function. The program creates two objects, one of type `Base` and one of type `Derived`. When the `print` function is called on each object, the output depends on the object's type. When the `print` function is called on the object of type `Base`, the `Base` class `print` function is called. When the `print` function is called on the object of type `Derived`, the `Derived` class `print` function is called because it overrides the `print` function in the base class.

Inline Function:

Inline functions are functions that are expanded by the compiler at the point where they are called, rather than being executed as a separate function call. Inline functions are used to reduce the overhead of function calls, especially for small functions that are called frequently. To declare a function as inline, the `inline` keyword must be used before the function's return type.

Here is an example of an inline function:

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

int main() {
    int x = 5, y = 10, z;
    z = add(x, y);
    return 0;
}
```

In this example, `add` is declared as an inline function. The program calls `add` with two integers `x` and `y` and assigns the result to `z`. The `add` function is expanded inline by the compiler, so there is no function call overhead. The resulting code is equivalent to:

```
int x = 5, y = 10, z;
z = x + y;
```

Note that the compiler may choose not to inline a function, even if it is declared as inline, if it determines that doing so would not result in any performance benefits.