# Basics of C++

Sajak Basnet

# Importance of C/C++ Programming Language in Computer Science

C/C++ programming languages are like the building blocks of computer science, playing a vital role in how software and technology work. These languages are like the superheroes behind the scenes, making sure everything runs smoothly. Think of C/C++ as the architects of the digital world. They allow programmers to create powerful and efficient programs that can do incredible things. From tiny devices to massive computer systems, C/C++ are the go-to languages when it comes to getting things done quickly and efficiently. One big reason for their importance is that they give programmers a lot of control. Imagine you're building a robot. C/C++ are like the tools that let you control every little detail, from how the robot moves to how it processes information. This level of control is crucial in creating programs that need to be super-fast and precise .C/C++ are also like the foundation of a skyscraper. Many other languages, like Java or Python, are built on top of them. Learning C/C++ gives you a deep understanding of how these other languages work, making it easier to learn and master them later. Moreover, C/C++ are everywhere. From your phone to your car's engine, these languages are running the show. They are the languages of choice for creating software that needs to be fast, reliable, and close to the hardware. In the world of video games, C/C++ are the magic behind the scenes. They make the characters move, the explosions happen, and the virtual worlds come alive.

In a nutshell, C/C++ are the neglected heroes of computer science. They are the languages that build, shape, and power the technology we use every day. Learning and mastering C/C++ opens doors to endless possibilities in software development, game design, system programming, and so much more.

# What’s Object-Oriented Programming?

Object-Oriented Programming (OOP) is the programming pattern that organizes code by modelling real-world entities as objects. There is other many programming pattern Unlike, others programming pattern OOP is unique and different from others. this programming promotes the use of classes and objects to structure and design software in a way that enhances modularity, reusability, and maintainability.

The major advantages of OOPs are:

1. Promote code Reusability.
2. Code maintenance and Organization.
3. Enhanced Flexibility and Extensibility.
4. Clear approach of solving problems.
5. Security and access control.
6. Easy modelling and structuring of complex applications.

# Pillars of OOPS

1. Objects and Classes
2. Inheritance
3. Polymorphism
4. Abstraction
5. Encapsulation

**Objects and Classes:** Objects and Classes stand as fundamental pillars of Object-Oriented Programming. Classes serve as templates or blueprints that mirror the attributes and behaviours of real-world entities, while objects express these templates in the actual world. To illustrate, consider the analogy of a "Human Being" class encapsulating traits like eye colour and hair colour (attributes) along with actions like walking, talking, and eating (methods). When we bring this class to life, individuals identified as men and women emerge as distinct objects, express the characteristics outlined in the class. This dynamic interplay between objects and classes lays the groundwork for constructing organized and efficient code structures while mirroring the relationships and complexities of the real world.

**Inheritance:** Inheritance is a fundamental concept in object-oriented programming (OOP) that involves creating new classes based on existing ones. It allows you to inherit properties and behaviours from a parent class into a derived class. This promotes the reuse of code and the organization of your program. For instance, imagine a "University" class serving as the parent, from which specialized classes like "Students" and "Teachers" are derived. Through inheritance, these specialized classes can access and build upon the attributes and methods of the "University" class, streamlining code development and enhancing code structure.

**Polymorphism**: Polymorphism is a big word that simply means "many shapes or forms." In programming, it's a clever way of making one thing act differently depending on the situation. Imagine

a tool like a smartphone. you can use it for calling, texting, or taking photos. Even though it's the same tool, it can do different things based on what you need. That's what polymorphism does in programming – it lets one thing, like a function or a class, behave in various ways based on how it's used. It makes code more flexible and versatile, just like how your smartphone adapts to your different tasks.

**Abstraction**: Abstraction in programming, it means hiding the complicated details and showing only the important stuff. Think of a car. you don't need to know how the engine works inside to drive it. You just use the steering wheel, pedals, and buttons. Abstraction helps programmers focus on the parts they need without getting lost in the nitty-gritty details. It's like using a TV remote without knowing all the technical stuff inside you just press buttons to change channels and volume. Abstraction makes programming simpler and more manageable.

**Encapsulation**: Encapsulation simple means encapsulating i.e., combining data (attributes) and function (methods) into single unit. This keeps the inner workings hidden from outside interference, simplifying how you interact with your code and helping to prevent unintended errors or changes. It's like having a secure shell around your code's components, making your program more organized and secure.

# Programs Demonstrating the concepts of OOPs.

## 1) Objects and Classes

```cpp
#include <iostream>
class Rectangle {
private:
    int length, width;

public:
    Rectangle(int l, int w) : length(l), width(w) {}
    int area() { return length * width; }
};

int main() {
    Rectangle r(5, 3);
    std::cout << "Area: " << r.area() << std::endl;
    return 0;
}
```

In above code example as you can see Rectangle class is serving as template or blueprint simply class and inside main, we created instance (object) of Rectangle class demonstrating concept of class and object in very simple way.

# 2) Constructors and Destructors

Constructors is a special member function that gets called when an object is created. It initializes the object's attributes and allocates resources if needed. Constructors have the same name as the class and do not have any return type, not even void. They can be overloaded means a class can have multiple constructors with different parameter lists.

A destructor is a special member function that gets called when an object goes out of scope or is explicitly deleted. The main use of destructor is that it is used to release any resources allocated by the object during its lifetime, like memory, file handles, etc. Like constructors' destructors also has same name as class preceded by the tilde (~) and do not take any arguments.

Simple program demonstrating constructors and destructors.

```
#include <iostream>
class Student {
public:
    Student() { std::cout << "Student object created!" <<
std::endl; }
    ~Student() { std::cout << "Student object destroyed!" <<
std::endl; }
};

int main() {
 Student s;  // Object created
    return 0;   // Object destroyed
}
```

# 3) Inheritance

```cpp
#include <iostream>
class Vehicle {
public:
    void honk() { std::cout << "Honk!" << std::endl; }
};

class Car : public Vehicle {
};

int main() {
    Car myCar;
    myCar.honk();
    return 0;
}
```

As you can see in above code example Vehicle class is the parent class and car is derived from Parent class Vehicle and as you can see car class (Derived class) is able to access honk() method of parent class Vehicle.

# 4) Polymorphism (virtual functions)

```
#include <iostream>
class Shape {
public:
virtual void draw() { std::cout << "Drawing a shape" << std::endl; }
};

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

int main() {
Shape* s = new Circle();
s->draw();
delete s;
return 0;
}
```

As you can see above code is trying to demonstrate the concept of polymorphism using the virtual function. Virtual functions are special kind of functions declare using **virtual** keyword and that can be overridden by base class. Don’t worry we will talk about virtual functions in detail.

# Types of polymorphism in OOPs

Mainly there are two types of polymorphisms in Object Oriented Programming:

1. Compile Time Polymorphism (Static)
2. Run Time Polymorphism (Dynamic)

**Compile Time Polymorphism (Static Polymorphism):** compile time polymorphism involves method or function **overloading** simply, multiple methods or functions with the same name but different parameter size. The appropriate method or function to be executed is determined at compile time based on the number, types, and order of arguments passed.

**Run time polymorphism (Dynamic Polymorphism):** It involves method **overriding**, where a subclass provides a specific implementation for a method that is already defined in its superclass. The method to be executed is determined at run time based on the actual type of the object, allowing a subclass to provide its own version of the method while still to stick to the method signature defined in the superclass.

Let’s see the code example.

# Compile time polymorphism

```
#include<iostream>

using namespace std;

class Calculator{
   private:
      double num1 , num2;

   public:
      // Calculator(double n1 , double n2) : num1(n1), num2(n2) {}

      double compute(double n1, double n2){return n1 + n2;}
      double compute(double n1, double n2,double n3){return n1 + n2 + n3;}
};

int main(){
   Calculator calculate;

   cout <<calculate.compute(55.78,78) <<endl;
   cout <<calculate.compute(55.78,78,100);
}
```

# Run time polymorphism.

```
#include <iostream>

// Base class
class Animal {
public:
   // Virtual function
   virtual void makeSound() {
      std::cout << "Animal makes a generic sound" << std::endl;
```

```
    }
};

// Derived class
class Dog : public Animal {
public:
    // Override the virtual function from the base class
    void makeSound() override {
        std::cout << "Dog barks" << std::endl;
    }
};

// Another derived class
class Cat : public Animal {
public:
    // Override the virtual function from the base class
    void makeSound() override {
        std::cout << "Cat meows" << std::endl;
    }
};

int main() {
    Animal* animal1 = new Dog();
    Animal* animal2 = new Cat();

    // Call the overridden functions
    animal1->makeSound(); // Output: Dog barks
    animal2->makeSound(); // Output: Cat meows

    delete animal1;
    delete animal2;

    return 0;
}
```

# Concept of Abstraction

In OOP Abstraction can be achieved through use of abstract base classes (Class which cannot be instantiated) and using pure virtual functions.

```cpp
#include <iostream>

// Abstract base class
class Shape {
public:
    // Pure virtual function to calculate area
    virtual double calculateArea() const = 0;

    // Abstract base classes can have regular member functions
    void display() const {
        std::cout << "This is a shape." << std::endl;
    }
};

// Derived class Circle
class Circle : public Shape {
private:
    double radius;

public:
    Circle(double r) : radius(r) {}

    // Implementation of the pure virtual function
    double calculateArea() const override {
        return 3.141592653589793 * radius * radius;
    }
};

// Derived class Rectangle
class Rectangle : public Shape {
private:
    double width;
    double height;
```

```
public:
    Rectangle(double w, double h) : width(w), height(h) {}

    // Implementation of the pure virtual function
    double calculateArea() const override {
        return width * height;
    }
};

int main() {
    Circle circle(5.0);
    Rectangle rectangle(4.0, 6.0);

    // Using polymorphism to calculate and display the areas
    Shape* shape1 = &circle;
    Shape* shape2 = &rectangle;

    std::cout << "Circle Area: " << shape1->calculateArea() << std::endl;
    std::cout << "Rectangle Area: " << shape2->calculateArea() << std::endl;

    return 0;
}
```

We already have used virtual functions somewhere above but what the heck are pure virtual functions. We I've seen that virtual functions are the functions declared with '**virtual**' keyword in base class and can be overridden by derived class. Pure virtual functions are also the same but just difference is that it doesn't contains any definitions (logic) in base class just declaration.

Pure virtual functions look something like this from above code POV:

```
virtual double calculateArea () = 0;
```

Its equal to 0 simply means it doesn't has any logic or defination

# Concept of Encapsulation

In C++ encapsulation simply means wrapping up methods and attributes into a single unit and can be achieved using access specifiers **private**, **public,** and **protected**. They control the accessibility and visibility of data members and members functions.

**Public**: Members declared as public access specifier can be accessible from anywhere. Both from inside and outside of the class.

**Private**: Members declared as private access specifier can only within class. Visibility is only within class. If no access specifier is given to methods and attributes, then by default it always private. Accessibility and visibility are only inside of class.

**Protected**: Its same as private access specifier but the main difference is that attributes and methods declared with protected access specifier can be accessible from derived class. It's mostly use in inheritance.

Check this code example.

```
#include <iostream>

class MyClass {
public:
    // Public member
    int publicVar;

    // Public member function
    void publicMethod() {
        std::cout << "This is a public method." << std::endl;
    }

protected:
```

```
    // Protected member
    int protectedVar;

private:
    // Private member
    int privateVar;

public:
    // Constructor
    MyClass() : publicVar(1), protectedVar(2), privateVar(3) {}

    // Display all member variables
    void display() {
        std::cout << "publicVar: " << publicVar << std::endl;
        std::cout << "protectedVar: " << protectedVar << std::endl;
        std::cout << "privateVar: " << privateVar << std::endl;
    }
};

int main() {
    MyClass obj;

    // Access public members
    obj.publicVar = 10;
    obj.publicMethod();

    // Access protected and private members is not allowed from outside the class
    // obj.protectedVar = 20;  // Error
    // obj.privateVar = 30;    // Error

    obj.display();

    return 0;
}
```

This code example demonstrates all the concept of access specifiers.

Now Let's look at very special type of function in C++ which is called **friend function**. So, what are friend functions. These are the special type of functions in C++ capable of accessing private data members of class. Above somewhere we studied about private access specifiers any data member declared with private access specifier is only accessible within that class.

What if we want to access private data members of some class to perform some logic at this situation friend function can be very useful.

So, technically friend function is a special type of function that is not a member of a class but is allowed to access the private and also protected members (variables and functions) of that class. It's like giving special permission to a function to access the inner workings of a class, even though it's not part of the class.

Let's understand with code example:

```
#include <iostream>

// Declare the class
class MyClass {
private:
    int privateData;

public:
    MyClass() : privateData(0) {}

    // Declare the friend function
    friend void displayPrivateData(const MyClass&);
};

// Define the friend function
void displayPrivateData(const MyClass& obj) {
    std::cout << "Private Data: " << obj.privateData << std::endl;
}
```

```
int main() {
    MyClass myObject;

    // Call the friend function to access and display privateData
    displayPrivateData(myObject);

    return 0;
}
```

In this example:

1. We define a class `MyClass` with a private data member `privateData`. By default, this data member cannot be accessed from outside the class.

2. We declare a friend function `displayPrivateData` within the class. This function is not a member of the class but is declared as a friend. As a result, it has access to the private members of `MyClass`.

3. In the `main` function, we create an instance of `MyClass` named `myObject`.

4. We then call the `displayPrivateData` function, passing `myObject` as an argument. This function can access and display the private data member, `privateData`, even though it is not a member of the class.

Now let’s see where friend function can be useful in complex scenarios:

1. **Operator Overloading**: When overloading operators (e.g., `+`, `-`, `<<`, `>>`) for user-defined classes, you may need to access private members of the class. Friend functions can be used to implement these operators.

2. **Serialization and Deserialization**: When saving object data to a file (serialization) or loading data from a file (deserialization), you may need to access private members to read or write the object's state. Friend functions can be helpful in this context.

3. **Mathematical Functions**: For mathematical classes, like a `Vector` or `Matrix` class, you might need friend functions to implement mathematical operations that require access to private data members.

4. **Data Validation**: Friend functions can be used to validate and set private data members while ensuring that the validation logic remains encapsulated within the class.

5. **Custom Iterators**: When creating custom iterators for a class, you may need to access private data members to iterate through the class's internal data structures.

6. **Factory Methods**: In factory design patterns, friend functions can be used to create objects and initialize their private members in a controlled manner.

7. **Comparisons and Equality Checks**: For classes that need custom comparison operations (e.g., comparing objects for equality or ordering), friend functions can access private members to implement these comparisons.

8. **Legacy Code Integration**: When integrating with legacy code or external libraries, friend functions can be used to bridge the gap between the class's private members and external functions or data structures.

9**. Optimization**: In performance-critical applications, friend functions can provide a more efficient means of accessing and modifying data members compared to getter and setter methods.

10. **Testing and Debugging**: Friend functions can be useful for testing or debugging purposes, allowing access to internal class details for verification or analysis.

Like friend functions we have the concept of friend class in C++. Friend class is type of class which granted access to private and protected data members of another class. This is typically done using friend keyword in during class declaration. Let’s see the code example.

```
#include <iostream>

class MyClass {
private:
    int privateVar = 42;

public:
    // Declare FriendClass as a friend class
    friend class FriendClass;

    void PrintPrivateVar() {
        std::cout << "MyClass privateVar: " << privateVar <<
std::endl;
    }
};
```

```
class FriendClass {
public:
    void AccessPrivateVar(MyClass& myObject) {
        // FriendClass can access the privateVar of MyClass
        std::cout << "FriendClass accessing privateVar: " <<
myObject.privateVar << std::endl;
    }
};

int main() {
    MyClass myObject;
    FriendClass friendObject;

    myObject.PrintPrivateVar();          // Accessing from MyClass
    friendObject.AccessPrivateVar(myObject); // Accessing from
FriendClass

    return 0;
}
```

1. We have two classes, `MyClass` and `FriendClass`.
2. `MyClass` has a private member `secretValue` which is inaccessible from outside the class by default.
3. We declare `FriendClass` as a friend of `MyClass` using the `friend` keyword. This means that `FriendClass` can access the private members of `MyClass`.
4. In the `main` function, we create objects of both `MyClass` and `FriendClass`.
5. The `FriendClass` object (`friendObj`) calls a function `accessPrivateMember` on the `MyClass` object (`myObj`) and

accesses its private member `secretValue`. This access is possible because `FriendClass` is a friend of `MyClass`.

# Static Data Member and Static Member Functions

Static Data Members are special type of data members that are not associated with any specific class instance. Means static Data members can be accessed directly via class without even creating instance that class. In C++ we use scope resolution operator (::) to access static member of specific class.

Static member Functions are also like Static Data Members. Static member functions can also be called directly via class without creating instance of that class. Static member functions are often used for operation that are not dependent on the state of individual objects but are related to class some way.

Check out this code example:

```
#include <iostream>

using namespace std;

class MathUtils
{
private:
    static int number;

public:
    static int square(int x)
    {
        return x * x;
    }

    static int factorial(int n)
    {
        if (n <= 0)
        {
            return 1;
        }
        return n * factorial(n - 1);
    }

    static void incrementNumber()
    {
        number += 1;
    }
};

int MathUtils::number = 0;

int main()
{
    int square = MathUtils::square(6);
    int fact = MathUtils::factorial(5);
```

```
    MathUtils::incrementNumber();

    cout << "Static data member is " << MathUtils::number << endl;
    cout << square << endl << fact;
    return 0;
}
```

In this code:

1. A new static function `incrementNumber` inside the `MathUtils` class to increment the `number` data member.
2. We call `MathUtils::incrementNumber()` in the `main` function to increment the static data member.
3. Now, the `number` data member is manipulated correctly within the `incrementNumber` function.

With these changes, the code should work as expected, and you will see the updated value of the `number` static data member.

Now let’s talk about passing objects to function as arguments and returning objects from function. Passing objects to function and returning objects is one of the most useful technique its mainly used in various logics like solving mathematical problems, in complex Data Structures , Factory Methods , Data Transformation and so on. Let’s try to understand this concept using simple code example:

```
#include <iostream>

class Point {
private:
    int x;
    int y;

public:
    Point(int x, int y) : x(x), y(y) {}
```

```
    // A function to add two Point objects and return a new Point
object
    static Point addPoints(const Point& p1, const Point& p2) {
        int newX = p1.x + p2.x;
        int newY = p1.y + p2.y;
        return Point(newX, newY);
    }

    // A function to display the coordinates of a Point object
    void display() {
        std::cout << "x: " << x << ", y: " << y << std::endl;
    }
};

int main() {
    Point point1(3, 4);
    Point point2(1, 2);

    // Call the addPoints function to add two Point objects by
passing objects as arguments
    Point result = Point::addPoints(point1, point2);

    std::cout << "Resultant Point: ";
    result.display();

    return 0;
}
```

In this code:

1. We have a `Point` class with x and y coordinates and a constructor to initialize these coordinates.

2. Inside the `Point` class, we have a static member function `addPoints`. This function takes two `Point` objects as arguments (`p1` and `p2`), adds their coordinates, and returns a new `Point` object representing the sum.

3. In the `main` function, we create two `Point` objects (`point1` and `point2`) with specific coordinates.

4. We then call the `addPoints` function to add these two `Point` objects, and the result is stored in the `result` variable.

5. Finally, we display the coordinates of the resultant `Point` object using the `display` function.

# Operator Overloading

Operated overloading is a powerful and important feature in C++ that allows you to redefine the behaviour of C++ operators for user defined data types. this means you can make operators work with objects of your custom classes allowing you to write more expressive and natural code. The major use cases of operator overloading are:

1. **Mathematical Operations Overloading**: You can overload arithmetic operators like `+`, `-`, `*`, `/`, and `%` to work with custom numeric or mathematical classes. For example, you can create classes to represent complex numbers, matrices, or vectors and define operator overloads for these classes.

2. **Comparisons**: Overloading comparison operators (`==`, `!=`, `<`, `>`, `<=`, `>=`) is useful when you want to define custom comparison logic for objects. This is common when working with user-defined data types like strings or custom containers.

3. **Concatenation**: Operator overloading can be used to define custom concatenation behaviour for objects. For example, you can overload the `+` operator for string concatenation or for combining elements in a custom container.

4. **Input and Output**: Overloading the `<<` and `>>` operators allows you to define custom input and output operations for your classes. This is often used for formatting and displaying objects.

5. **Assignment**: Overloading the assignment operator (`=`) allows you to customize how objects of your class are assigned values. This can be helpful in resource management or ensuring deep or shallow copying, as needed.

6. **Increment and Decrement**: You can overload the `++` and `--` operators for both pre-increment/post-increment and pre-decrement/post-decrement operations. This is useful for objects that have a natural notion of incrementing or decrementing.

7. **Function Objects (Functors):** Function objects are objects that can be called like functions. You can overload the function call operator ` () ` to create functors, which can be used in algorithms that expect callable objects.

8. **Indexing**: You can overload the `[]` operator to provide array-like access to elements of your custom classes or containers.

9. **Type Conversion**: Overloading type-casting operators allows you to define how an object of one type can be converted to another type. For example, you can define how a custom numerical type can be converted to a standard type.

10. **Custom Iterators**: When creating custom iterators for your data structures, overloading the `*`, `->`, `++`, and `--` operators can make your iterators work seamlessly with the standard library algorithms.

11. **Smart Pointers**: Overloading the `->` and `*` operators is often done when creating custom smart pointers to encapsulate resource management, like memory or file handles.

12. **Matrix and Vector Operations**: Overloading operators is common when working with mathematical objects like matrices and vectors, allowing you to write code that closely resembles mathematical notation.

```
#include <iostream>

class Complex {
private:
    double real;
    double imaginary;

public:
    Complex(double r, double i) : real(r), imaginary(i) {}

    // Overload the + operator to add two Complex numbers
    Complex operator+(const Complex& other) {
        double newReal = real + other.real;
        double newImaginary = imaginary + other.imaginary;
        return Complex(newReal, newImaginary);
    }

    // Display the Complex number
    void display() {
        std::cout << real << " + " << imaginary << "i" << std::endl;
    }
};

int main() {
    Complex c1(3.0, 2.0);
    Complex c2(1.0, 4.0);

    Complex sum = c1 + c2;

    std::cout << "c1: ";
```

```
    c1.display();
    std::cout << "c2: ";
    c2.display();
    std::cout << "Sum of c1 and c2: ";
    sum.display();

    return 0;
}
```

In this code:

- We define a `Complex` class with two private members, `real` for the real part and `imaginary` for the imaginary part.
- We overload the `+` operator by defining an `operator+` function that takes another `Complex` object as a parameter. It returns a new `Complex` object with the sum of the real and imaginary parts.
- In the `main` function, we create two `Complex` objects, `c1` and `c2`, and add them together using the `+` operator.
- We then display the values of `c1`, `c2`, and the result of the addition.

# C++ general rules of operator overloading

1. Only built-in operators can be overloaded. New operators cannot be created.
2. Arity (number of operands) cannot be changed.
3. Precedence and associativity of the operators cannot be changed.
4. Overloaded operators cannot have default arguments except the function call operator () which have default arguments.
5. Operators cannot be overloaded for built in types only. At least one operand must be used defined type.
6. Assignment (=), subscript([]) , function call (‘()’), and members selection(->) operators must be defined as member functions
7. Except the operators specified in point 6, all other operators can be either member functions or a non-member function.
8. Some operators cannot be overloaded includes:
    - `::` (scope resolution operator)
    - `.*` (pointer to member operator)
    - `.` (member access operator)
    - `?:` (ternary conditional operator)

# Inheritance

Somewhere above we've already discussed about inheritance. so, what's inheritance, simply it's the process of passing traits and characteristics from parent to child. Technically inheritance in OOP is defined as the mechanism that allows a new class to acquire and extend the properties and behaviours of an existing class, facilitating code reuse and hierarchy construction.

Benefits of inheritance:

1. Code Reusability:
   - Inheritance enables the reuse of code from existing classes, reducing redundancy, and promoting a modular approach to software development.

2. Polymorphism:
   - Inheritance facilitates polymorphism, allowing objects of different classes to be treated as objects of a common base class, promoting flexibility and adaptability in the code.

3. Extensibility:
   - Derived classes can extend the functionality of base classes by adding new attributes and behaviours, enhancing the overall capabilities of the software.

4. Modularity:
   - Inheritance promotes a modular design, allowing developers to focus on individual classes without being concerned about the entire system, which can lead to better organization and maintenance.

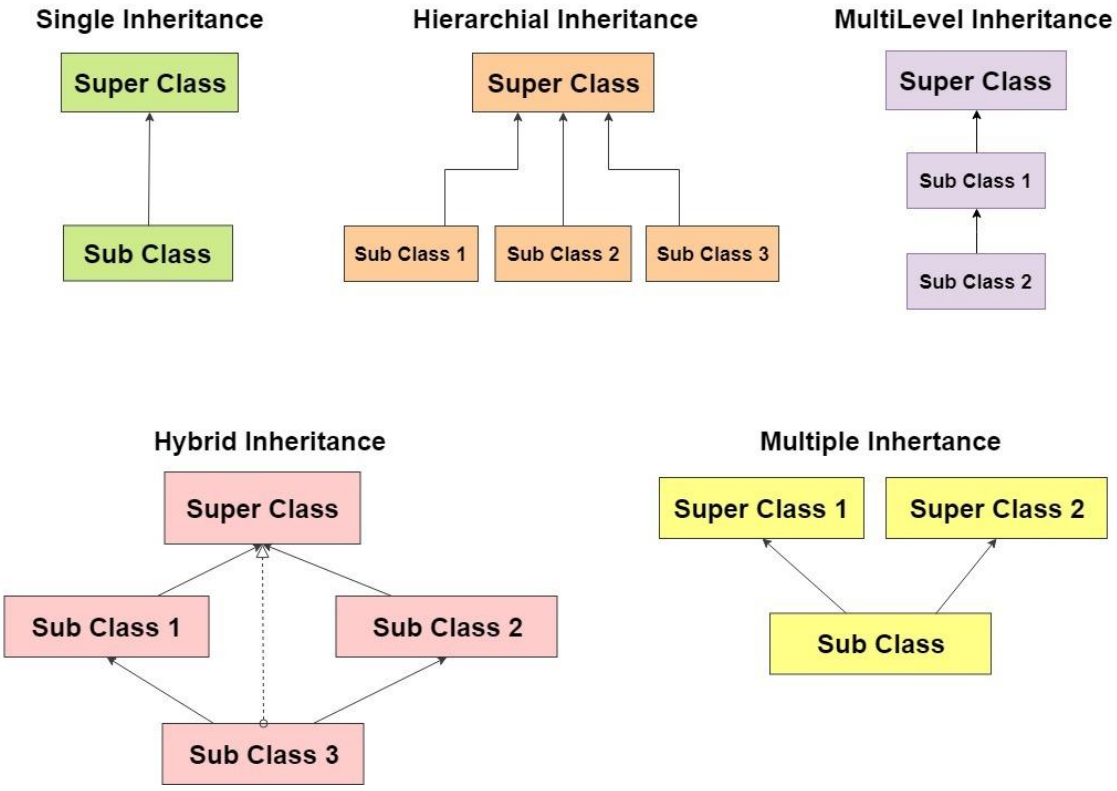
5. Maintenance:
    - Changes made to the base class are automatically reflected in the derived classes, streamlining maintenance tasks, and ensuring consistency throughout the codebase.

6. Readability:
    - Inheritance contributes to code readability by establishing a hierarchical relationship between classes, making it easier for developers to understand the structure and relationships within the software.

7. Flexibility:
    - Inheritance provides a flexible framework for adapting to changing requirements. New functionality can be added by creating derived classes without modifying existing code.

8. Hierarchy Construction:
    - Inheritance supports the creation of class hierarchies, allowing developers to model real-world relationships and dependencies in a structured and intuitive manner.

# Types of inheritance




1. Single Inheritance:
   - A class inherits from only one base class.
   - Syntax: `class DerivedClass : public BaseClass { /* ... */ };`

2. Multiple Inheritance:
   - A class can inherit from more than one base class.
   - Syntax: `class DerivedClass : public BaseClass1, public BaseClass2 { /* ... */ };`

3. Multilevel Inheritance:
- A class is derived from another derived class, forming a chain of inheritance.

- Syntax: `class DerivedClass1 : public BaseClass { /* ... */ };`
  `class DerivedClass2 : public DerivedClass1 { /* ... */ };`

4. Hierarchical Inheritance:
- Multiple classes are derived from a single base class.
- Syntax: `class BaseClass { /* ... */ };`
  `class DerivedClass1 : public BaseClass { /* ... */ };`
  `class DerivedClass2 : public BaseClass { /* ... */ };`

5. Hybrid (Virtual) Inheritance:
- A combination of multiple and hierarchical inheritance with the use of virtual classes.
- Helps to avoid the "diamond problem" where ambiguity may arise in the presence of multiple base classes.
- Syntax: `class BaseClass { /* ... */ };`
  `class VirtualBase { /* ... */ };`
  `class DerivedClass : public BaseClass, virtual public VirtualBase { /* ... */ };`

6. Single Inheritance with Interfaces:
- In C++, where there is no explicit interface keyword, interfaces can be emulated using abstract base classes (classes with pure virtual functions).
- Syntax: `class Interface { virtual void method() = 0; };`
  `class DerivedClass : public Interface { void method() override { /* ... */ } };`

It's important to choose the appropriate type of inheritance based on the design requirements. Care should be taken to avoid excessive complexity and potential issues such as the diamond problem in multiple inheritance scenarios. In some cases, alternative design patterns or composition may be preferable to inheritance.

# Types/Modes of derivation

In the context of C++ and object-oriented programming, "types/modes of derivation" generally refers to the access control modes associated with the inheritance relationship. These modes determine how the members (attributes and methods) of the base class are accessible in the derived class. In C++, there are three access control modes for inheritance:

1. Public Derivation:
   - In public derivation, public members of the base class remain public in the derived class, protected members become protected, and private members remain inaccessible.
   - This is the most common mode of inheritance and is used to model an "is-a" relationship.
   - Syntax: `class DerivedClass : public BaseClass { /* ... */ };`

2. Protected Derivation:
   - In protected derivation, public and protected members of the base class become protected in the derived class, and private members remain inaccessible.
   - This mode is less common and is used when you want to restrict access to the base class's public interface in the derived class.
   - Syntax: `class DerivedClass : protected BaseClass { /* ... */ };`

3. Private Derivation:
   - In private derivation, both public and protected members of the base class become private in the derived class, and private members remain inaccessible.
   - This mode is used when you want to encapsulate the implementation details of the base class in the derived class.
   - Syntax: `class DerivedClass : private BaseClass { /* ... */ };`

Here's a summary:

- Public Derivation:
  - Base class public members remain public.
  - Base class protected members become protected.
  - Base class private members remain inaccessible.

- Protected Derivation:
  - Base class public and protected members become protected.
  - Base class private members remain inaccessible.

- Private Derivation:
  - Base class public and protected members become private.
  - Base class private members remain inaccessible.

# Ambiguity in Multipath Inheritance

Multiple inheritance in C++ allows a class to inherit from more than one class as we have already discussed. When there is a chain of inheritance involving multiple classes, and a derived class inherits from more than one base class, it is referred to as "**multipath inheritance**." Ambiguity in multipath inheritance arises when there are two or more paths to reach a common base class, and the compiler is uncertain about which version of the base class's member to use.

Simple example to illustrate the ambiguity in multipath inheritance:

```cpp
#include <iostream>

// Common base class with a virtual function
class BaseCommon {
public:
    virtual void display() {
        std::cout << "BaseCommon Display\n";
    }
};


class Base1 : public BaseCommon {
public:

};

class Base2 : public  BaseCommon {
public:

};


class Derived : public Base1, public Base2 {
public:

};

int main() {
    Derived obj;

    obj.display();  // No ambiguity with virtual inheritance

    return 0;
}
```

Let’s try to understand the above code:

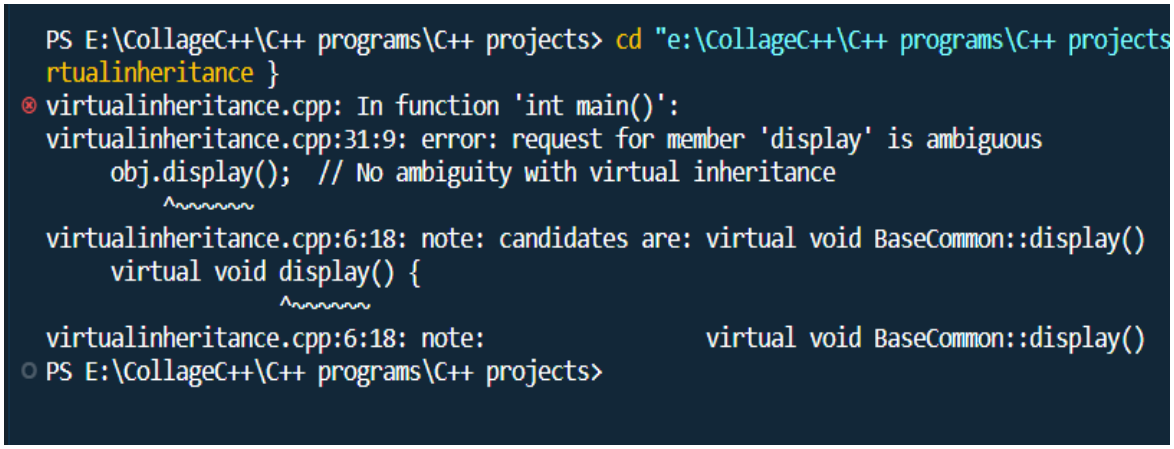
1. BaseCommon Class:
   - `BaseCommon` is a base class with a virtual function `display()`. This class serves as a common base for both `Base1` and `Base2` known as virtual base class.

2. Base1 and Base2 Classes:
   - `Base1` and `Base2` are two classes that inherit publicly from `BaseCommon`. They don't add any additional functionality but share the `display` function from the common base.

3. Derived Class:
   - `Derived` is a class that inherits from both `Base1` and `Base2`. This is an example of multiple inheritance (multipath).

4. Main Function:
   - In the `main` function, an object of the `Derived` class named `obj` is created.

5. Display Function Call:
   - `obj.display();` calls the `display` function on the `Derived` object. it would cause ambiguity error because the compiler doesn't know which version of the `display()` function to use. So the compiler throws an error. Error looks something like this.



Now how to solve this problem well to solve this problem we have the concept of virtual base class and virtual inheritance.

A virtual base class in C++ is a base class that can be used multiple times in an inheritance hierarchy without causing duplication of data members. It's marked with the virtual keyword.

Similarly virtual inheritance in C++ is a mechanism to handle ambiguity and prevent the "Diamond Problem" in multiple inheritance. It ensures that only one instance of a virtual bas class exists in the inheritance hierarchy, even if it's inherited through different paths.

Certainly! Let's delve into an example scenario to better understand how virtual base classes resolve ambiguity in the context of method inheritance.

In summary, virtual base classes ensure that there is a single instance of the common base class in the inheritance hierarchy, preventing ambiguity issues when accessing members or methods that are not overridden in the derived classes. This makes the code more predictable and avoids complications associated with the diamond problem in multipath inheritance.

Let's see how can we implement the concept of virtual base class and virtual inheritance in code:

```
#include <iostream>

// Common base class with a virtual function
class BaseCommon {
public:
    virtual void display() {
        std::cout << "BaseCommon Display\n";
```

```
    }
};


class Base1 : public virtual BaseCommon {
public:

};

class Base2 : public virtual BaseCommon {
public:

};


class Derived : public Base1, public Base2 {
public:

};

int main() {
    Derived obj;

    obj.display();  // No ambiguity with virtual inheritance

    return 0;
}
```

In the above code, there are three classes: `BaseCommon`, `Base1`, and `Base2`. `BaseCommon` is a base class with a virtual function `display()`. Both `Base1` and `Base2` inherit virtually from `BaseCommon`. This virtual inheritance ensures that there is only one shared instance of `BaseCommon` in the hierarchy, avoiding ambiguity in case of multiple inheritance. The `Derived` class inherits from both `Base1` and `Base2` with virtual inheritance. In the `main` function, an object of `Derived` named `obj` is created, and calling `obj.display()` invokes the `display` function from the virtually inherited `BaseCommon`. The output will be "BaseCommon

Display." This example demonstrates the use of virtual inheritance to maintain a clear and unambiguous class hierarchy when dealing with multiple base classes sharing a common base class.

# Abstract base class

abstract base classes are the type of base class via which abstraction can be achieved. It a type of a class which contains at least one pure virtual function and cannot be instantiated. An abstract base class in C++ is a class that serves as a template or blueprint for a group of related classes. It defines common behavior and characteristics that are shared by all derived classes, but it cannot be instantiated directly. Instead, it serves as a foundation for creating concrete subclasses that implement the abstract class's defined behavior.
Abstract base classes play a crucial role in object-oriented programming by promoting code organization, reusability, and type safety. They ensure that all derived classes share a common set of functionalities and adhere to specific requirements, enhancing the consistency and predictability of code.

Key Characteristics of Abstract Base Classes:

1. Abstractness: Abstract base classes cannot be instantiated directly. They define the essential features and behaviour for their derived classes.

2. Pure Virtual Functions: Abstract base classes often contain at least one pure virtual function. These functions lack an implementation in the base class and must be overridden by derived classes to provide their specific implementations.

3. Common Behavior: Abstract base classes establish a common foundation of behavior for derived classes, ensuring consistent functionality across the hierarchy.
4. Code Organization: They promote code organization by grouping related classes together and defining shared behavior, making the code more structured and maintainable.

5. Code Reusability: They enhance code reusability by providing a common codebase that can be shared across derived classes, reducing redundancy and increasing development efficiency.

6. Type Safety: They enforce type safety by ensuring that all derived classes adhere to the abstract class's requirements, preventing misuse and unexpected behavior.

Simple Example of Abstract Base Class in C++:

Consider a scenario where you want to create a hierarchy of shapes, including circles, squares, and triangles. You can define an abstract base class called Shape that encapsulates the common properties and behavior of all shapes.

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

};
```

Since Shape contains at least one pure virtual function, it is an abstract class and cannot be instantiated directly. Derived classes, such as Circle, Square, and Triangle, must provide implementations for these abstract virtual functions.

Let’s see the implementation of derived classes.

```
class Circle : public Shape {
public:
  double radius;

  Circle(double radius) {
    this->radius = radius;
  }

  double getArea() override {
    return 3.14159 * radius * radius;
  }

  void draw() override {
    std::cout << "Drawing a circle" << std::endl;
  }
};

class Square : public Shape {
public:
  double sideLength;

  Square(double sideLength) {
    this->sideLength = sideLength;
  }

  double getArea() override {
    return sideLength * sideLength;
  }

  void draw() override {
    std::cout << "Drawing a square" << std::endl;
  }
};
```

Each derived class provides its specific implementation for the abstract virtual functions getArea() and draw(), tailored to its unique characteristics. This exemplifies how abstract base classes establish a common framework for derived classes while allowing them to adapt and implement specific behaviors.

# Constructors and destructors in inheritance

In object-oriented programming, constructors and destructors play a crucial role in inheritance, ensuring proper initialization and destruction of objects across class hierarchies. When an object of a derived class is created, the constructors of both the base class and the derived class are invoked in a specific order, ensuring that the base class's members are initialized before the derived class's members.

Constructor Invocation in Inheritance:

1. Base Class Constructor: The constructor of the base class is always called first, regardless of the depth of the inheritance hierarchy. This ensures that all base class members are properly initialized before the derived class's members are accessed.

2. Derived Class Constructor: After the base class constructor completes, the constructor of the derived class is called. This allows the derived class to initialize its members and perform any additional initialization tasks specific to its type.

Destructor Invocation in Inheritance:

1. Derived Class Destructor: The destructor of the derived class is called first, allowing it to perform any cleanup tasks specific to

its type. This includes releasing any resources allocated by the derived class.
2. Base Class Destructor: Finally, the destructor of the base class is called, ensuring that any resources allocated by the base class are properly released. This ensures that all resources are cleaned up in the reverse order of their allocation.

Implications of Constructors and Destructors in Inheritance:

1. Proper Initialization: Constructors ensure that both base class and derived class members are properly initialized, preventing unexpected behaviour and ensuring correct object state.

2. Resource Management: Destructors ensure that resources allocated by both base class and derived class are properly released, preventing memory leaks and ensuring efficient resource utilization.

3. Type Safety: Constructors and destructors enforce type safety by ensuring that objects are constructed and destroyed in a consistent and predictable manner, preventing errors, and maintaining the integrity of the class hierarchy.

In summary, constructors and destructors play a critical role in inheritance by ensuring proper initialization, resource management, and type safety across class hierarchies. They are essential for creating robust and maintainable object-oriented designs.

# Virtual function and polymorphism

## Early vs Late Binding

In C++, binding refers to the association of a function call with the code that implements that function. There are two types of binding in C++: early binding (also known as static binding) and late binding (also known as dynamic binding or runtime binding). Let's explore each one with code examples.

1.Early Binding (Static Binding): Early binding occurs at compile-time, and it is resolved before the program is executed. The compiler determines the address of the function to be called during the compilation phase.

```
#include <iostream>

class Base {
public:
    void display() {
        std::cout << "Base class display() called." << std::endl;
    }
};

class Derived : public Base {
public:
    void display() {
        std::cout << "Derived class display() called." << std::endl;
    }
};

int main() {
    Base baseObj;
    Derived derivedObj;
```

```
    // Early binding - resolved at compile-time
    baseObj.display();   // Calls Base::display()
    derivedObj.display(); // Calls Derived::display()

    return 0;
}
```

In this example, the function `display()` is resolved at compile-time based on the type of the object. If the object is of type `Base`, it calls `Base::display()`. If the object is of type `Derived`, it calls `Derived::display()`.

2. Late Binding (Dynamic Binding or Runtime Binding): Late binding occurs at runtime, and the actual function call is resolved dynamically during program execution. This is achieved through the use of virtual functions and the `virtual` keyword.

```
 #include <iostream>

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

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

```
int main() {
    Base baseObj;
    Derived derivedObj;

    // Late binding - resolved at runtime
    Base* ptr = &baseObj;
    ptr->display(); // Calls Base::display()

    ptr = &derivedObj;
    ptr->display(); // Calls Derived::display()

    return 0;
}
```

In this example, the `display()` function is declared as `virtual` in the `Base` class. When a pointer to the base class is used to point to an object of a derived class, the actual function to be called is determined at runtime based on the type of the object being pointed to. This is late binding in action.

In summary, early binding is resolved at compile-time, while late binding is resolved at runtime, allowing for more flexibility and polymorphic behaviour in the code.

# Overriding

Overriding is a concept related to polymorphism, and it allows a derived class to provide a specific implementation for a function that is already defined in its base class. Overriding enables you to define a function in the derived class with the same signature as a function in the base class, effectively replacing or extending the behavior of the base class function.

Here's a simple explanation and example of function overriding in C++:

Example of Overriding:

```cpp
#include <iostream>

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

class Derived : public Base {
public:
    // Overrides the display() function in the Base class
    void display() override {
        std::cout << "Derived class display() called." << std::endl;
    }
};

int main() {
    Base baseObj;
    Derived derivedObj;
```

```
    // Calls the display() function of the Base class
    baseObj.display();

    // Calls the overridden display() function of the Derived class
    derivedObj.display();

    // Using a pointer to demonstrate polymorphism
    Base* ptr = &derivedObj;
    // Calls the overridden display() function of the Derived class
    ptr->display();

    return 0;
}
```

In this example:

- The `Base` class has a virtual function `display()`.

- The `Derived` class inherits from the `Base` class and overrides the `display()` function with its own implementation.

- In the `main()` function, an object of each class is created (`baseObj` and `derivedObj`).

- The `display()` function is called on both objects. When calling the function on `derivedObj`, the overridden version in the `Derived` class is executed.

- A pointer of type `Base*` is used to point to an object of the `Derived` class. When the `display()` function is called through this pointer, it dynamically dispatches to the overridden function in the `Derived` class, demonstrating polymorphism.

Note: The `override` keyword is optional but recommended when overriding functions. It helps catch errors at compile-time if the function being declared in the derived class doesn't actually override a virtual function in the base class.

# Virtual vs Pure Virtual function

We already know that virtual functions are the type of functions that are declared using virtual keyword in the base class with some logic and definition and can be overridden in derived class. On the other hand, Pure virtual functions are those which does not have any definition and logic in base class and is equal to 0 i.e., '= 0'.

Let's see in more detail:

**Virtual Functions**: A virtual function is a function declared in a base class that can be overridden by a function with the same signature in a derived class. When a virtual function is called through a pointer or reference to the base class, the actual function that gets executed is determined at runtime based on the type of the object.

Example:

```
#include <iostream>

class Base {
public:
    // Declare a virtual function
    virtual void display() {
        std::cout << "Base class display() called." << std::endl;
    }
};

class Derived : public Base {
public:
    // Override the virtual function in the derived class
    void display() override {
```

```
        std::cout << "Derived class display() called." << std::endl;
    }
};

int main() {
    Base baseObj;
    Derived derivedObj;

    // Call the virtual function through base class pointers
    Base* ptr = &baseObj;
    ptr->display(); // Calls Base::display()

    ptr = &derivedObj;
    ptr->display(); // Calls Derived::display()

    return 0;
}
```

In this example, the `display()` function in the `Base` class is declared as virtual, and it's overridden in the `Derived` class. When the function is called through a pointer to the base class, the actual function executed is determined at runtime based on the object type.

**Pure Virtual Functions**: A pure virtual function is a virtual function in a base class that is declared but has no implementation in the base class. It is marked with `= 0` at the end of its declaration. A class containing at least one pure virtual function is called an abstract class, and you cannot create an instance of an abstract class. Derived classes must provide implementations for all pure virtual functions to become concrete (instantiable) classes.

Example:

```
#include <iostream>

class Shape {
public:
    // Pure virtual function(No implementation)
    virtual void draw() = 0;
};

class Circle : public Shape {
public:
    // Provide an implementation for the pure virtual function
    void draw() override {
        std::cout << "Drawing a circle." << std::endl;
    }
};

class Square : public Shape {
public:
    // Provide an implementation for the pure virtual function
    void draw() override {
        std::cout << "Drawing a square." << std::endl;
    }
};

int main() {
    // Shape shape; // Error: Cannot instantiate an abstract class

    Circle circle;
    Square square;

    // Call the draw() function through derived class objects
```

```
    circle.draw(); // Draws a circle
    square.draw(); // Draws a square

    return 0;
}
```

In this example, the `Shape` class has a pure virtual function `draw()`. Classes `Circle` and `Square` inherit from `Shape` and provide concrete implementations for the `draw()` function. You cannot create an instance of `Shape` directly due to the pure virtual function, but you can create instances of its derived classes.

# File Handling

File handling in C++ involves performing operations on files, such as reading data from files or writing data to files. C++ provides a set of classes in the standard library for file handling, primarily found in the `<fstream>` header. Here's an overview of file handling in C++:

File Streams:

1. ofstream (Output File Stream): Used for writing data to files.

```
#include <fstream>
ofstream outputFile("example.txt");
```

2. ifstream (Input File Stream): Used for reading data from files.

```
#include <fstream>
ifstream inputFile("example.txt");
```

3. fstream (File Stream): Combined input/output stream for both reading and writing data.

```
#include <fstream>
fstream file("example.txt");
```

Opening a File:

Files must be opened before performing any operations on them. The `open` method is used for this purpose.

```
#include <fstream>
#include <iostream>

int main() {
    std::ofstream outFile;
    outFile.open("example.txt");

    if (outFile.is_open()) {
        // File operations...
        outFile << "Hello, File!";
        outFile.close();
    } else {
        std::cerr << "Unable to open the file." << std::endl;
    }

    return 0;
}
```

Writing to a File:

```cpp
#include <fstream>
#include <iostream>

int main() {
    std::ofstream outFile("example.txt");

    if (outFile.is_open()) {
        // Writing to the file
        outFile << "Hello, File!" << std::endl;
        outFile.close();
    } else {
        std::cerr << "Unable to open the file." << std::endl;
    }

    return 0;
}
```

Reading from a File:

```cpp
#include <fstream>
#include <iostream>
#include <string>

int main() {
    std::ifstream inFile("example.txt");

    if (inFile.is_open()) {
        // Reading from the file
        std::string line;
        while (std::getline(inFile, line)) {
            std::cout << line << std::endl;
        }

        inFile.close();
    } else {
        std::cerr << "Unable to open the file." << std::endl;
    }

    return 0;
}
```

Binary File Handling:

When working with binary files, you can use the `ios::binary` flag.

```
#include <fstream>

int main() {
    std::ofstream binaryOutFile("data.bin", std::ios::binary);

    if (binaryOutFile.is_open()) {
        // Writing binary data to the file
        int data = 42;
        binaryOutFile.write(reinterpret_cast<char*>(&data),
sizeof(data));
        binaryOutFile.close();
    } else {
        std::cerr << "Unable to open the file." << std::endl;
    }

    return 0;
}
```

Checking File Status:

It's essential to check whether a file is successfully opened before performing operations on it.

```cpp
#include <fstream>
#include <iostream>

int main() {
    std::ofstream file("example.txt");

    if (file.is_open()) {
        // File operations...
        file << "Hello, File!";
        file.close();
    } else {
        std::cerr << "Unable to open the file." << std::endl;
    }

    return 0;
}
```

These are some fundamental concepts of file handling in C++. Depending on your specific requirements, you may need to explore more advanced features, error handling, and file formats. This just gives basic overview of file handeling.

# Exception handling mechanism in C++

Exceptions are sudden runtime errors that occurs unexpectedly in program and crashes the program, hence, alters the normal flow of code execution. So due to this reason handling exception becomes very important. In C++ we use exception handling mechanism to handle exceptions.

Exception handling is a mechanism in C++ that allows you to handle runtime errors and exceptional situations in a controlled manner. The primary components of C++ exception handling are:

1. **try**:

   The `try` block contains the code that might throw an exception. If an exception occurs within this block, the program looks for a matching `catch` block to handle the exception.

```
try {
    // Code that may throw an exception
}
```

2. **throw**:

   The `throw` statement is used to raise an exception explicitly (To throw an exception). It is followed by an object, typically of a class type (but it can also be of fundamental types like int, char, etc.).

```
throw SomeException("An error occurred");
```

3. **catch**:

The `catch` block is used to catch and handle exceptions thrown in the corresponding `try` block. It specifies the type of exception that it can handle.

```
try {
    // Code that may throw an exception
}
catch (const SomeException& ex) {
    // Handle the exception
    cerr << "Exception caught: " << ex.what() << endl;
}
```

You can have multiple `catch` blocks to handle different types of exceptions.

4. **throw inside catch**:

You can also use the `throw` statement within a `catch` block to rethrow an exception after handling it.

```
  try {
    // Code that may throw an exception
}
catch (const SomeException& ex) {
    // Handle the exception
    cerr << "Exception caught: " << ex.what() << endl;
    throw; // Rethrow the same exception
}
```

Here's a simple code example:

```cpp
#include<iostream>
#include<string>

using namespace std;

int main() {
    int numerator , denominator , result;
    try
    {
        cout<<"Enter numerator: ";
        cin>>numerator;
        cout<<"Enter denominator: ";
        cin>>denominator;

        if(denominator <= 0) {
            throw string("Denominator cannot be 0");
        }

        result = (numerator / denominator);
        cout<<"Result : " << result <<endl;

    }
    catch(const string& e)
    {
        cout<<"Error: " <<e <<endl;
    }

    return 0;
}
```

This C++ program employs exception handling to handle potential errors in a division operation. It prompts the user to input values for the numerator and denominator, and within a `try` block, checks if the denominator is greater than zero. If the denominator is valid, it

performs the division and outputs the result; otherwise, it throws an exception of type `string` with the message "Denominator cannot be 0." The `catch` block catches exceptions of type `string` and prints an error message to the standard error stream. The program concludes by returning 0, providing a controlled approach to manage errors during runtime.