An inline function in C++ is a function defined with the inline keyword, which suggests to the compiler to insert the function's code directly into each place it's called, instead of performing a standard function call.

**Key Points about Inline Functions**

* **Small Functions**: Best suited for small, simple functions, as large functions may increase the program size if inlined too frequently.
* **Defined in Header Files**: Inline functions are often defined in header files since they need to be available at every point of use.
* **No Recursion**: Inline functions shouldn’t contain recursion, as it defeats the purpose of reducing function call overhead.
* **Linkage**: Inline functions should have internal linkage, meaning they’re typically static when used in a header file, or explicitly marked inline to avoid multiple definition errors.

**1. What is a Pointer?**

A pointer is a variable that stores the memory address of another variable. Pointers are powerful tools in C++ for dynamic memory management, accessing arrays, and enabling efficient data manipulation.

**2. Declaring Pointers**

To declare a pointer, use the \* symbol. For example:

cpp

Copy code

int\* ptr; // Pointer to an integer

char\* charPtr; // Pointer to a character

This creates a pointer named ptr that can hold the address of an int variable.

**3. Assigning an Address to a Pointer**

To assign a pointer to an address, use the & operator:

cpp

Copy code

int x = 10;

int\* ptr = &x; // ptr now holds the address of x

Here, ptr points to the memory address of x.

**4. Dereferencing a Pointer**

Dereferencing a pointer accesses the value at the address the pointer is holding. This is done with the \* operator:

cpp

Copy code

std::cout << \*ptr; // Outputs the value of x (10)

**5. Null Pointer**

A pointer can be set to nullptr (or NULL in older C++) to indicate that it doesn’t currently point to a valid address.

cpp

Copy code

int\* ptr = nullptr;

**6. Pointer Arithmetic**

Pointer arithmetic allows navigation through an array or memory block. Operations include ++, --, +, and -:

cpp

Copy code

int arr[] = {10, 20, 30};

int\* ptr = arr; // Points to the first element

std::cout << \*ptr; // 10

ptr++; // Moves to the next integer

std::cout << \*ptr; // 20

**7. Arrays and Pointers**

Arrays and pointers have a close relationship in C++. The name of an array (arr) is a pointer to the first element:

cpp

Copy code

int arr[] = {10, 20, 30};

int\* ptr = arr;

std::cout << \*(ptr + 2); // Outputs 30

**8. Pointers to Pointers**

A pointer can also store the address of another pointer, creating a multi-level pointer structure:

cpp

Copy code

int x = 10;

int\* ptr = &x; // Pointer to an int

int\*\* ptr2 = &ptr; // Pointer to a pointer to an int

std::cout << \*\*ptr2; // Outputs 10

**9. Dynamic Memory Allocation**

Pointers are essential for dynamic memory allocation with new and delete:

cpp

Copy code

int\* ptr = new int(5); // Allocates memory for an int and assigns 5

delete ptr; // Deallocates memory

int\* arr = new int[5]; // Allocates memory for an array of 5 integers

delete[] arr; // Deallocates memory for the array

**10. Common Pointer Pitfalls**

* **Dangling Pointers**: When a pointer points to a deallocated memory location. Avoid by setting the pointer to nullptr after delete.
* **Memory Leaks**: If dynamically allocated memory is not deleted, it leads to memory leaks.
* **Wild Pointers**: Uninitialized pointers that may point to random locations. Always initialize pointers, even if just to nullptr.

**Example of Pointers Usage:**

cpp

Copy code

#include <iostream>

void swap(int\* a, int\* b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

int main() {

int x = 10, y = 20;

swap(&x, &y);

std::cout << "x: " << x << ", y: " << y << std::endl;

return 0;

}

In this example, swap takes pointers as parameters, allowing it to modify the original values directly.

**Summary**

* **Pointer Basics**: Hold memory addresses.
* **Dereferencing**: Accesses the value at the address.
* **Dynamic Allocation**: new and delete manage memory at runtime.
* **Pointer Arithmetic**: Navigate arrays or memory blocks.
* **Safety**: Use nullptr, avoid memory leaks, and handle deallocation carefully.

## 1. **Encapsulation**

Encapsulation is the concept of bundling data (attributes) and methods (functions) that operate on that data within a class. This keeps the data safe from outside interference and misuse.

### In C++:

* Use **access specifiers** (private, protected, public) to control access to members (variables and methods).
* private members are accessible only within the class, protected members are accessible within the class and derived classes, and public members are accessible from anywhere.

### Key Points:

* **Private members**: Control data access.
* **Getters/Setters**: Control and validate access to private data.

### Example in C++:

cpp

Copy code

class Account {

private:

double balance;

public:

Account(double initial\_balance) : balance(initial\_balance) {}

double getBalance() const { return balance; }

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

};

### Comparison with Java:

* In Java, the syntax is similar with private, protected, and public.
* Both Java and C++ use **getters** and **setters** for controlled access to private data.

## 2. **Inheritance**

Inheritance is the mechanism by which one class (derived/child) inherits attributes and methods from another class (base/parent). This promotes code reusability and a hierarchical structure.

### In C++:

* Use the syntax class Derived : public Base to define a derived class that inherits from a base class.
* C++ supports **multiple inheritance** (a class can inherit from more than one base class), whereas Java only supports single inheritance (a class can extend only one other class).

### Key Points:

* **Base Class (Parent)**: The class being inherited from.
* **Derived Class (Child)**: The class that inherits.

### Example in C++:

cpp

Copy code

class Animal {

public:

void eat() { cout << "Eating..." << endl; }

};

class Dog : public Animal {

public:

void bark() { cout << "Woof!" << endl; }

};

### Comparison with Java:

* In Java, you would use extends instead of : public.
* Java supports single inheritance, but multiple inheritance can be achieved through **interfaces**.

## 3. **Polymorphism**

Polymorphism allows methods to do different things based on the object they are acting upon. It enables one interface to be used for a general class of actions, typically through **method overriding**.

### In C++:

* Achieved by marking methods in the base class as virtual, allowing derived classes to override them.
* When a function is marked as virtual, C++ uses **dynamic binding** to determine which function to call at runtime based on the object type.

### Key Points:

* **Virtual Functions**: Enable runtime polymorphism.
* **Override Functions**: Customize behavior in derived classes.

### Example in C++:

cpp

Copy code

class Animal {

public:

virtual void sound() { cout << "Some generic animal sound" << endl; }

};

class Dog : public Animal {

public:

void sound() override { cout << "Woof!" << endl; }

};

### Comparison with Java:

* Java automatically uses **dynamic binding** with overridden methods. In C++, you need to explicitly mark functions as virtual in the base class to achieve the same.
* In Java, you use @Override to indicate an overridden method.

Polymorphism is a fundamental concept in object-oriented programming (OOP) that allows objects to be treated as instances of their parent class, even if they belong to a subclass. Polymorphism enables the same function or operator to behave differently based on the object calling it. In C++, polymorphism can occur in two forms: **compile-time (or static polymorphism)** and **run-time (or dynamic polymorphism)**.

### 1. Compile-Time Polymorphism (Static Polymorphism)

Compile-time polymorphism, also known as static polymorphism, is resolved during the compilation process. This type of polymorphism includes function overloading and operator overloading. The compiler determines which function or operator to use based on the arguments provided.

#### a) **Function Overloading**

* This allows multiple functions with the same name but different parameter lists.
* The compiler determines which function to call based on the argument types and count at compile time.

**Example:**

cpp

Copy code

class Printer {

public:

void print(int i) {

std::cout << "Printing integer: " << i << std::endl;

}

void print(double d) {

std::cout << "Printing double: " << d << std::endl;

}

void print(const std::string &s) {

std::cout << "Printing string: " << s << std::endl;

}

};

int main() {

Printer p;

p.print(5); // Calls print(int)

p.print(3.14); // Calls print(double)

p.print("Hello"); // Calls print(string)

return 0;

}

In this example, the compiler decides which print function to use based on the argument type, making it compile-time polymorphism.

#### b) **Operator Overloading**

* Operator overloading allows you to redefine the behavior of operators for user-defined types (like +, -, \*, etc.) based on the operands.
* This also occurs at compile time because the compiler determines which operator function to invoke based on the types of operands.

**Example:**

cpp

Copy code

class Complex {

private:

int real, imag;

public:

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

// Overloading the '+' operator

Complex operator + (const Complex &c) {

return Complex(real + c.real, imag + c.imag);

}

void print() const {

std::cout << real << " + " << imag << "i" << std::endl;

}

};

int main() {

Complex c1(3, 4), c2(1, 2);

Complex c3 = c1 + c2; // Calls the overloaded '+' operator

c3.print(); // Output: 4 + 6i

return 0;

}

Here, the + operator is overloaded to add two Complex numbers, and the decision on which + to use is made at compile time.

### 2. Run-Time Polymorphism (Dynamic Polymorphism)

Run-time polymorphism, also known as dynamic polymorphism, is resolved during program execution rather than at compile time. This type of polymorphism is achieved using **inheritance and virtual functions**. The compiler can’t determine which function to call at compile time; instead, the correct function is chosen at run time based on the type of the object.

#### Virtual Functions and Method Overriding

* In run-time polymorphism, a base class defines a virtual function, and derived classes override this function to provide specific implementations.
* The function to call is determined at run time, based on the actual type of the object (not the type of the pointer/reference pointing to it).

**Example:**

cpp

Copy code

class Animal {

public:

virtual void sound() {

std::cout << "Some generic animal sound" << std::endl;

}

};

class Dog : public Animal {

public:

void sound() override {

std::cout << "Woof!" << std::endl;

}

};

class Cat : public Animal {

public:

void sound() override {

std::cout << "Meow!" << std::endl;

}

};

int main() {

Animal \*a;

Dog d;

Cat c;

a = &d;

a->sound(); // Output: Woof! (calls Dog's version of sound)

a = &c;

a->sound(); // Output: Meow! (calls Cat's version of sound)

return 0;

}

In this example:

* sound is a virtual function in the base class Animal.
* Dog and Cat classes override sound.
* The actual function called depends on the object type (Dog or Cat), even though we’re using a pointer to Animal.

#### How Run-Time Polymorphism Works

* The compiler creates a **virtual table (vtable)** for classes with virtual functions. This table holds pointers to the actual function implementations for each class.
* When a virtual function is called through a pointer or reference, the program looks up the function’s address in the vtable at run time, thus determining which overridden function to execute.

### Key Differences between Compile-Time and Run-Time Polymorphism

| **Feature** | **Compile-Time Polymorphism** | **Run-Time Polymorphism** |
| --- | --- | --- |
| How It’s Achieved | Function overloading, operator overloading | Virtual functions, inheritance |
| When It’s Resolved | Compile time | Run time |
| Performance | Faster (no look-up needed) | Slightly slower (vtable look-up) |
| Use Cases | Multiple functions with similar names, e.g., print() for different data types | Overriding behavior in derived classes, e.g., sound() for specific animals |
| Flexibility | Limited flexibility | More flexible, can handle dynamic types |

In summary:

* **Compile-time polymorphism** is used when different behaviors can be determined at compile time through overloading.
* **Run-time polymorphism** is useful when you want to override behaviors dynamically based on the derived object type at runtime.

## 4. **Abstraction**

Abstraction is about simplifying complex systems by modeling classes appropriate to the problem, focusing only on relevant details. Abstract classes and interfaces allow us to define methods that must be implemented by subclasses.

### In C++:

* Use **abstract classes** by defining at least one **pure virtual function** (a function with = 0 at the end), forcing derived classes to implement this function.
* C++ does not have interfaces like Java; instead, it achieves abstraction through abstract classes.

### Key Points:

* **Pure Virtual Functions**: Functions that must be implemented by derived classes.
* **Abstract Class**: Cannot be instantiated, used only as a base class.

### Key Points about Dynamic Binding:

1. **Polymorphism**: Dynamic binding allows polymorphic behavior. A base class pointer or reference can be used to refer to objects of derived classes, and the correct function is chosen at runtime based on the actual type of the object.
2. **Virtual Functions**: To enable dynamic binding, the functions in the base class should be marked as virtual. This tells the compiler to set up a virtual table (vtable), which is used at runtime to decide which function to call.
3. **Vtable and Vptr**: Each class with virtual functions has a virtual table, which stores pointers to virtual functions. Each object of such a class has a hidden pointer (vptr) to its class’s vtable, ensuring the correct function is called for the specific object type.

### Example in C++:

cpp

Copy code

class Shape {

public:

virtual void draw() = 0; // Pure virtual function

};

class Circle : public Shape {

public:

void draw() override { cout << "Drawing a Circle" << endl; }

};

### Comparison with Java:

* Java uses **interfaces** to achieve full abstraction. In C++, an abstract class with only pure virtual functions acts like a Java interface.
* Java classes can implement multiple interfaces, while in C++, a class can inherit from multiple abstract classes.

## 5. **Other Key OOP Concepts**

### Constructors and Destructors

* **Constructor**: Special function that initializes an object when it is created. In C++, the constructor has the same name as the class.
* **Destructor**: Special function that cleans up when an object is destroyed. In C++, it is denoted by a ~ before the class name.

cpp

Copy code

class MyClass {

public:

MyClass() { cout << "Constructor called!" << endl; }

~MyClass() { cout << "Destructor called!" << endl; }

};

### Comparison with Java:

* Java has a garbage collector, so it doesn’t need destructors. Java also uses finalize() (now deprecated) for cleanup.

### Overloading and Overriding

* **Method Overloading**: Having multiple functions with the same name but different parameters. C++ and Java both support overloading.
* **Method Overriding**: Re-defining a base class function in a derived class. In C++, use virtual for polymorphism, and in Java, use @Override.

### Access Specifiers

* **public**: Accessible from anywhere.
* **protected**: Accessible within the class, derived classes, and classes in the same package in Java.
* **private**: Accessible only within the class.

In Java, protected members are also accessible to other classes in the same package, while in C++ this only applies to derived classes.

### Summary of Key Differences

| **Concept** | **C++** | **Java** |
| --- | --- | --- |
| Inheritance | Supports multiple inheritance | Single inheritance, multiple interfaces |
| Virtual Functions | Requires virtual keyword | Polymorphism is implicit |
| Abstract Classes | Pure virtual functions (no interfaces) | Interfaces and abstract classes |
| Destructor | Explicitly defined using ~ | Managed by garbage collection |
| Access Specifiers | public, protected, private | public, protected, private |

These notes cover the essentials of OOP concepts in C++ and compare them with Java, helping you transition smoothly and understand the key principles and differences.

4o

**Properties of Friend Functions in C++**

1. **Access to Private and Protected Members**:
   * Friend functions can access private and protected members of the class they are friends with, providing a controlled way to access these members from outside the class.
2. **Non-Member Function**:
   * A friend function is not a member of the class, so it does not have a this pointer. It’s typically defined outside the class scope but is given special access privileges.
3. **Declared Inside the Class, Defined Outside**:
   * The function is declared with the friend keyword inside the class definition but can be defined outside the class, similar to regular functions.
4. **No Inheritance of Friendship**:
   * Friend status is not inherited. If a class B is derived from class A, a friend of A does not automatically become a friend of B. Friendship is strictly between the friend function and the specific class.
5. **Friendship is Not Mutual**:
   * If Class A declares a function from Class B as a friend, it doesn’t mean Class B has access to Class A’s private members unless explicitly declared.
6. **Used to Overload Operators**:
   * Friend functions are often used to overload operators that cannot be defined as member functions, such as the << and >> operators for input/output.

### 1. What is a Friend in C++?

* A friend in C++ is a function or a class that is given access to the private and protected members of another class.
* Declaring a function or another class as a friend allows it to access otherwise inaccessible data members and member functions.

### 2. Friend Functions

* A **friend function** is a regular function that has access to the private and protected members of a class.
* It is declared with the keyword friend inside the class whose private members it needs to access.
* Friend functions are not member functions, so they are not called with an object of the class.

**Example:**

cpp

Copy code

#include <iostream>

using namespace std;

class MyClass {

private:

int secretNumber;

public:

MyClass(int num) : secretNumber(num) {}

// Declaring 'displaySecret' as a friend function of MyClass

friend void displaySecret(MyClass);

};

// Definition of the friend function

void displaySecret(MyClass obj) {

cout << "Secret Number is: " << obj.secretNumber << endl;

}

int main() {

MyClass obj(42);

displaySecret(obj); // Accesses private member 'secretNumber' via friend function

return 0;

}

#### Characteristics of Friend Functions

* They are not members of the class they’re friends with, so they do not have a this pointer.
* They can be declared in one or more classes and can access private/protected members of multiple classes.
* They are called like regular functions, not with the . or -> operators.

### 3. Friend Classes

* A **friend class** is a class that is given access to the private and protected members of another class.
* If class B is declared as a friend of class A, all member functions of B can access the private and protected members of A.

**Example:**

cpp

Copy code

class B; // Forward declaration

class A {

private:

int data;

public:

A() : data(10) {}

friend class B; // Declaring class B as a friend of class A

};

class B {

public:

void display(A& obj) {

// Accessing private member 'data' of class A

cout << "Value of data in A: " << obj.data << endl;

}

};

int main() {

A a;

B b;

b.display(a); // Accesses private data of A via friend class B

return 0;

}

### 4. Uses of Friend Functions and Friend Classes

* **Operator Overloading**: Friend functions are often used for overloading operators that need access to the private members of a class (e.g., +, <<).
* **External Access**: They allow external functions to interact with class internals without adding new methods to the class itself.
* **Efficiency**: Sometimes, friend functions are used for efficiency, as they don't have the overhead of member function calls.

### 5. Things to Remember

* **Friendship is not inherited**: If class B is a friend of class A, subclasses of B do not automatically gain access to A's private members.
* **Friendship is not mutual**: If class B is a friend of class A, A does not automatically become a friend of B.
* **Use cautiously**: Overusing friend functions can lead to poor encapsulation and should be done only when necessary.

### 6. Friend Functions vs. Member Functions

| **Feature** | **Friend Function** | **Member Function** |
| --- | --- | --- |
| Access to Private Data | Yes | Yes |
| Belongs to Class | No | Yes |
| Called by Object | No, called like a normal function | Yes, called by object using . |
| Access to this pointer | No | Yes |

Using friends can enhance flexibility but should be balanced with good design principles to maintain encapsulation.

**Notes on Access Specifiers in Inheritance**

Inheritance can also vary depending on the access specifier (public, protected, private) used:

* **Public inheritance**: The public and protected members of the base class remain public and protected in the derived class.
* **Protected inheritance**: The public and protected members of the base class become protected in the derived class.
* **Private inheritance**: The public and protected members of the base class become private in the derived class.

**Summary of Types of Inheritance**

| **Type of Inheritance** | **Description** |
| --- | --- |
| Single Inheritance | Derived class inherits from one base class. |
| Multiple Inheritance | Derived class inherits from more than one base class. |
| Multilevel Inheritance | Derived class inherits from a class that is already derived from another base class. |
| Hierarchical Inheritance | Multiple derived classes inherit from a single base class. |
| Hybrid Inheritance | Combination of two or more types of inheritance. |

### Summary of Differences

| **Feature** | **Private** | **Protected** |
| --- | --- | --- |
| Access within Class | Yes | Yes |
| Access in Derived Classes | No | Yes |
| Access by Objects | No | No |
| Purpose | Hide implementation details entirely | Share certain details with derived classes |

**Key Points**

* **Use private** when you want to completely encapsulate a member, making it accessible only within its own class.
* **Use protected** when you want to allow derived classes to access a member while keeping it hidden from the outside world.
* members of the base class when inherited in different ways (public, protected, or private inheritance).

| **Base Class Member Access** | **Public Inheritance** | **Protected Inheritance** | **Private Inheritance** |
| --- | --- | --- | --- |
| **Public Member** | Remains public in derived class | Becomes protected in derived class | Becomes private in derived class |
| **Protected Member** | Remains protected in derived class | Remains protected in derived class | Becomes private in derived class |
| **Private Member** | Not accessible in derived class | Not accessible in derived class | Not accessible in derived class |

**Virtual Inheritance**

1. **Problem of Diamond Inheritance**:
   * When a derived class inherits from two classes that both inherit from a common base class, it creates ambiguity and duplication, known as the **diamond problem**.
   * Example:

cpp

Copy code

class A {}; // Base class

class B : public A {}; // Derived from A

class C : public A {}; // Derived from A

class D : public B, public C {}; // Diamond problem

* + In the above example, D has two copies of A (one from B and one from C), leading to ambiguity when accessing members of A.

1. **Virtual Inheritance Solution**:
   * **Virtual inheritance** tells the compiler to create only one instance of the base class, even if it is inherited multiple times.
   * **Syntax**:

cpp

Copy code

class B : public virtual A {};

class C : public virtual A {};

class D : public B, public C {};

* + With virtual inheritance, D will have a single instance of A, solving the diamond problem.

1. **Benefits of Virtual Inheritance**:
   * Resolves ambiguity in cases of multiple inheritance by ensuring only one copy of the base class exists.
   * Prevents duplicate copies of base class data members, saving memory.
2. **How Virtual Inheritance Works**:
   * The compiler internally manages the single instance of the base class, ensuring it is shared by all derived classes.
3. **When to Use Virtual Inheritance**:
   * Use virtual inheritance if you need multiple classes to share a common base class but want to avoid duplication and ambiguity (e.g., complex hierarchies involving multiple inheritance paths).
   * Often used in frameworks and libraries with large inheritance trees.
4. **Code Example of Virtual Inheritance**:

cpp

Copy code

#include <iostream>

using namespace std;

class A {

public:

int value;

};

class B : public virtual A {}; // Virtual inheritance

class C : public virtual A {}; // Virtual inheritance

class D : public B, public C {}; // Resolves diamond problem

int main() {

D obj;

obj.value = 5; // No ambiguity here

cout << "Value in D: " << obj.value << endl;

return 0;

}

In this example, D has a single instance of A, allowing access to value without ambiguity.

**Key Points for Exam:**

1. **Understand class syntax and basic class components** (constructors, destructors, methods, access modifiers).
2. **Know types of inheritance** and how they allow code reuse.
3. **Identify the diamond problem in multiple inheritance** and explain why it occurs.
4. **Explain virtual inheritance syntax and benefits**:
   * Reduces ambiguity and memory usage by ensuring only one instance of the base class.
5. **Remember when to use virtual inheritance** – complex hierarchies or situations where a single base class instance is needed across multiple derived paths.
6. **Practice with examples** of diamond inheritance, adding virtual inheritance to resolve issues, and understand compiler handling of a single base instance.