Unit 2: Inheritance and Pointers

Syllabus :

Inheritance: Introduction, defining derived classes, forms of inheritance, ambiguity in multiple and multipath inheritance, virtual base class, object slicing, overriding member functions, object composition and delegation, order of execution of constructors and destructors.

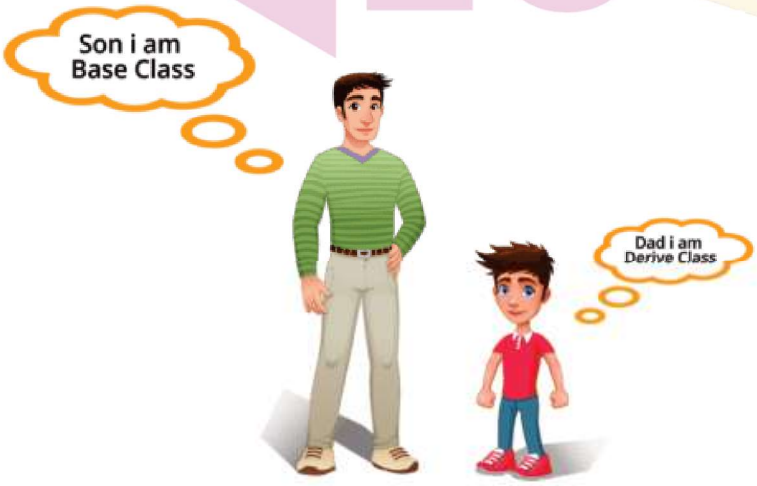
Pointers and Dynamic Memory Management: Declaring and initializing pointers, accessing data through pointers, pointer arithmetic, memory allocation (static and dynamic), dynamic memory management using new and delete operators, pointer to an object, this pointer, pointer related problems - dangling/wild pointers, null pointer assignment, memory leak and allocation failures.

Section 1: Inheritance

1.1 Introduction to Inheritance

1. Definition of Inheritance

- Inheritance is a fundamental concept in object-oriented programming (OOP) by which a new class (derived class) can inherit properties and behaviors from an existing class (base class).
- The derived class inherits the attributes and methods of the base class, enabling code reusability and establishing a hierarchical relationship between classes.
- The class that is being inherited from is known as the "parent class" or "base class" or "superclass", and the class that inherits from the base class is known as the “child class” or "derived class" or "subclass".
- The child class will inherit all the public and protected properties (attributes) and methods from the parent class. In addition, it can have its own properties and methods,this is called inheritance.



- Example: A Car class may inherit from a Vehicle class. Here, Car is a specific type of Vehicle.

2. Purpose and Benefits

1. Code Reusability: Inheritance allows classes to inherit attributes and methods from other classes, reducing redundancy and promoting code reuse.
2. Extensibility: Inheritance enables the addition of new features to a class without modifying its existing structure, thus facilitating the extension of functionality.
3. Relationship Representation: Inheritance models real-world relationships making the code more intuitive and reflective of real-world scenarios.
4. Specialization: Inheritance allows for the creation of specialized classes (derived classes) that inherit common features from a more general class (base class) and add their own unique functionalities.
5. Transitivity: If class B inherits from class A, and class C inherits from class B, then class C automatically inherits properties and behaviors from class A.
6. Hierarchical Organization: Inheritance enables the creation of a hierarchical organization of classes, where classes can be organized into a hierarchy based on their relationships.
7. Encapsulation: Inheritance encourages encapsulation, as it promotes the creation of well-defined, modular classes with clear boundaries.
8. Polymorphism: Inheritance supports polymorphism, which allows objects of derived classes to be treated as objects of their base class.

1.2 Defining Derived Classes

1. Syntax and Structure:

In C++, a derived class is defined using the following syntax:

```
class DerivedClassName : accessSpecifier BaseClassName {
    // Member variables and functions
};
```

- DerivedClassName: This is the name of the derived class.
- accessSpecifier: Specifies the access level for the base class members inherited by the derived class. It can be one of three access specifiers: public, protected, or private.
- BaseClassName: This is the name of the base class from which the derived class inherits.

Example:

```
// Base class
class Base {
public:
    int baseVar;
    void baseMethod() {
        // Implementation
    }
};

// Derived class inheriting from Base publicly
```

```
class Derived : public Base {
public:
    int derivedVar;
    void derivedMethod() {
        // Implementation
    }
};
```

2. Relationship Between Base and Derived Classes:

- **Inheritance Relationship:**
  - A derived class inherits attributes and behaviors (member variables and functions) from its base class.
  - The derived class can access the public and protected members of the base class.
  - The private members of the base class are not accessible directly by the derived class; however, they can be accessed through public or protected member functions of the base class.
- **Relationship Type:**
  - The relationship between a base class and a derived class is an "is-a" relationship.
  - For example, in the above code, Derived "is-a" Base. This implies that an object of the derived class can be treated as an object of the base class.
- **Access Control:**
  - The public, protected, and private access specifiers determine how the members of the base class are accessed by the derived class:
    - public: The public members of the base class remain public in the derived class.
    - protected: The protected members of the base class remain protected in the derived class.
    - private: The private members of the base class remain inaccessible in the derived class.
  - The choice of access specifier in the derived class declaration determines the level of visibility for the inherited members.

Example:

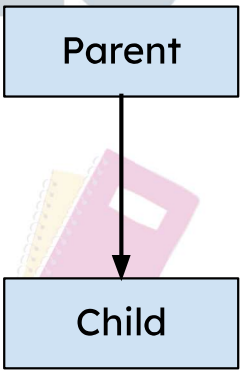
```
// Base class
class Base {
public:
    int publicVar;
protected:
    int protectedVar;
private:
    int privateVar;
};
```

```
// Derived class inheriting publicly
class DerivedPublic : public Base {
    // Accessible: publicVar, protectedVar
    // Inaccessible: privateVar
};

// Derived class inheriting protectedly
class DerivedProtected : protected Base {
    // Accessible: protectedVar
    // Inaccessible: publicVar, privateVar
};

// Derived class inheriting privately
class DerivedPrivate : private Base {
    // Accessible: None (all members are private)
};
```

1.3 Forms of Inheritance

1. Single Inheritance:

- In single inheritance, a derived class inherits from only one base class.
- It forms a simple one-to-one relationship between classes.
- It is like Father -> Child relationship.
- Example:

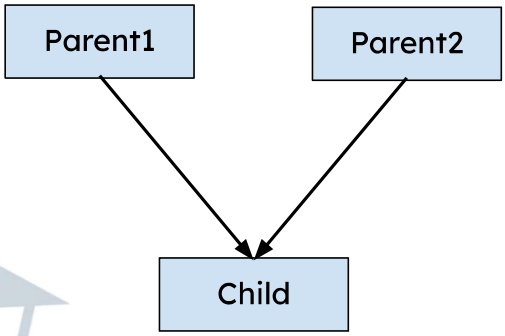



```
#include <iostream>
using namespace std;

// Base class (Parent)
class Parent {
public:
    void parentMethod() {
        cout << "Method of Parent Class" << endl;
    }
};

// Derived class (Child) inheriting from Parent
class Child : public Parent {
public:
    void childMethod() {
        cout << "Method of Child Class" << endl;
    }
};

int main() {
    Child obj;
```

```
    obj.childMethod(); // Output: Method of Child Class
    obj.parentMethod(); // Output: Method of Parent Class
    return 0;
}
```

2. Multiple Inheritance:

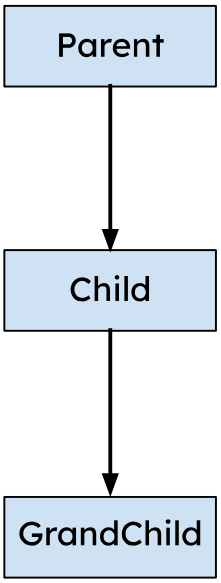
- In multiple inheritance, a derived class inherits from multiple base classes.
- It allows a class to inherit attributes and behaviors from more than one parent class.
- It is like Mother & Father -> Child relationship.
- Example:




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

// Parent1 class
class Parent1 {
public:
    void parent1Method() {
        cout << "Method of Parent1 Class" << endl;
    }
};

// Parent2 class
class Parent2 {
public:
    void parent2Method() {
        cout << "Method of Parent2 Class" << endl;
    }
};

// Child class inheriting from both Parent1 and Parent2
class Child : public Parent1, public Parent2 {
public:
    void childMethod() {
        cout << "Method of Child Class" << endl;
    }
};

int main() {
    Child obj;
    obj.childMethod(); // Output: Method of Child Class
    obj.parent1Method(); // Output: Method of Parent1 Class
    obj.parent2Method(); // Output: Method of Parent2 Class
    return 0;
}
```

3. Multilevel Inheritance:

- In multilevel inheritance, a derived class becomes the base class for another derived class, forming a chain of inheritance.
- It allows for creating a hierarchy of classes with each level adding additional features.
- It is like Father -> Child -> Grandchild relationship.



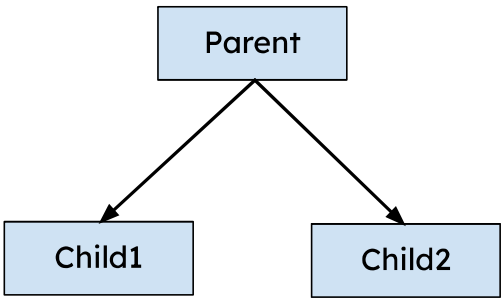
- Example:

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

// Parent class
class Parent {
public:
    void parentMethod() {
        cout << "Method of Parent Class" << endl;
    }
};

// Child class inheriting from Parent
class Child : public Parent {
public:
    void childMethod() {
        cout << "Method of Child Class" << endl;
    }
};

// GrandChild class inheriting from Child
class GrandChild : public Child {
public:
    void grandChildMethod() {
        cout << "Method of Grand Child Class" << endl;
    }
};

int main() {
    GrandChild obj;
    obj.grandChildMethod(); // Output: Method of Grand Child Class
    obj.childMethod(); // Output: Method of Child Class
    obj.parentMethod(); // Output: Method of Parent Class
    return 0;
}
```

4. Hierarchical Inheritance:

- In hierarchical inheritance, multiple derived classes inherit from a single base class.
- It allows for creating a hierarchy of classes with a common base class.

- It is like Father -> Son & Daughter relationship.



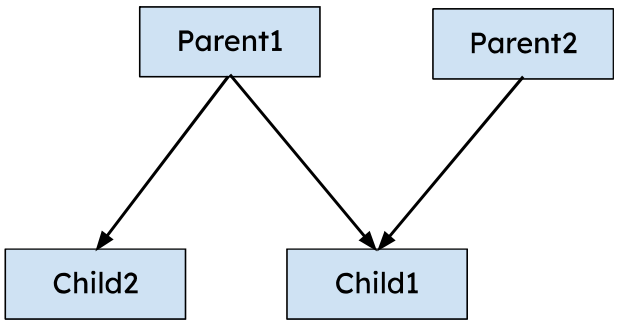
- Example:

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

// Parent class
class Parent {
public:
    void parentMethod() {
        cout << "Method of Parent Class" << endl;
    }
};

// Child1 class inheriting from Parent
class Child1 : public Parent {
public:
    void child1Method() {
        cout << "Method of Child1 Class" << endl;
    }
};

// Child2 class inheriting from Parent
class Child2 : public Parent {
public:
    void child2Method() {
        cout << "Method of Child2 Class" << endl;
    }
};

int main() {
    Child1 obj1;
    obj1.child1Method(); // Output: Method of Child1 Class
    obj1.parentMethod(); // Output: Method of Parent Class

    Child2 obj2;
    obj2.child2Method(); // Output: Method of Child2 Class
    obj2.parentMethod(); // Output: Method of Parent Class

    return 0;
}
```

5. Hybrid Inheritance:

- Hybrid inheritance is a combination of multiple types of inheritance, such as single, multiple, multilevel or hierarchical inheritance.

- It allows for creating complex class hierarchies by combining features of different types of inheritance.
- Example:

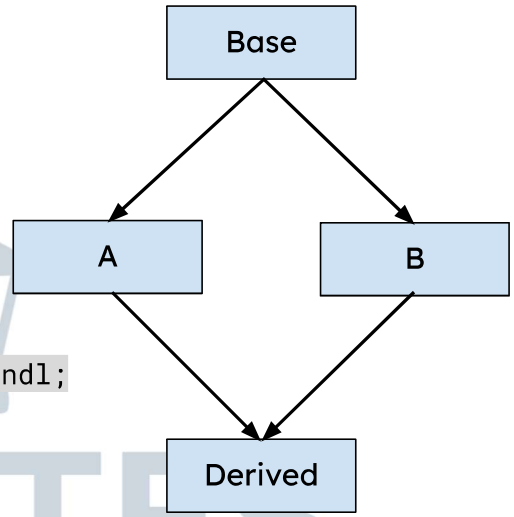



```
#include <iostream>
using namespace std;

// Parent1 class
class Parent1 {
public:
    void parent1Method() {
        cout << "Method of Parent1 Class" << endl;
    }
};

// Parent2 class
class Parent2 {
public:
    void parent2Method() {
        cout << "Method of Parent2 Class" << endl;
    }
};

// Child1 class inheriting from both Parent1 and Parent2
class Child1 : public Parent1, public Parent2 {
public:
    void child1Method() {
        cout << "Method of Child1 Class" << endl;
    }
};

// Child2 class inheriting from Parent1
class Child2 : public Parent1 {
public:
    void child2Method() {
        cout << "Method of Child2 Class" << endl;
    }
};

int main() {
    Child1 obj1;
    obj1.child1Method(); // Output: Method of Child1 Class
    obj1.parent1Method(); // Output: Method of Parent1 Class
    obj1.parent2Method(); // Output: Method of Parent2 Class

    Child2 obj2;
```

```
    obj2.child2Method(); // Output: Method of Child2 Class
    obj2.parent1Method(); // Output: Method of Parent1 Class

    return 0;
}
```

6. Multipath Inheritance:

- Multipath inheritance is a form of hybrid inheritance which occurs when there are multiple paths to reach a common base class through intermediate classes.
- This can happen when a class inherits from another class that indirectly inherits from the same base class.
- Example:




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

// Base class
class Base {
public:
    void baseMethod() {
        cout << "Method of Base Class" << endl;
    }
};

// Intermediate class A
class A : public Base {
public:
    void methodA() {
        cout << "Method of Class A" << endl;
    }
};

// Intermediate class B
class B : public Base {
public:
    void methodB() {
        cout << "Method of Class B" << endl;
    }
};

// Derived class inheriting from both A and B
class Derived : public A, public B {
public:
    void derivedMethod() {
        cout << "Method of Derived Class" << endl;
    }
```

```
};

int main() {
    Derived obj;
    // obj.baseMethod();  // Compile-time error due to ambiguity

    // To resolve ambiguity, explicitly call the base method through a
specific path using scope resolution operator
    obj.A::baseMethod();  // Output: Method of Base Class
    obj.B::baseMethod();  // Output: Method of Base Class
    obj.methodA();        // Output: Method of Class A
    obj.methodB();        // Output: Method of Class B
    obj.derivedMethod();  // Output: Method of Derived Class
    return 0;
}
```

In the above example, the Derived class inherits from both A and B, which in turn inherit from Base. This creates a multipath inheritance situation where there are multiple paths to reach the Base class from the Derived class.

1.4 Ambiguity in Multiple and Multipath Inheritance

Ambiguity in inheritance occurs when the derived class has more than one path to a base class. This situation can arise in both multiple inheritance and multipath inheritance.

1. **Multiple Inheritance:** Ambiguity arises when multiple base classes have methods with the same name. It can be resolved using the scope resolution operator.
2. **Multipath Inheritance:** Ambiguity arises due to multiple paths to a common base class. It can be resolved using the scope resolution operator or using virtual inheritance, ensuring only one instance of the base class is shared.

1. Ambiguity in Multiple Inheritance

In multiple inheritance, a derived class inherits from multiple base classes. Ambiguity occurs when multiple base classes have methods with the same name. The derived class does not know which method to inherit, leading to ambiguity.

Example:

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

// Base class 1
class Base1 {
public:
    void display() {
        cout << "Display from Base1" << endl;
    }
```