Objectives:

to understand type conversion between basic types, between basic and user-defined types, and between user-defined types, as well as to learn inheritance concepts including base and derived classes, protected access, derived class declaration, Is-a and Has-a relationships, different forms of inheritance, member overriding.

Theory:

Type Conversion

Type conversion is the process of converting one data type to another. In C++, this can happen automatically (implicit conversion) or manually (explicit conversion). Conversion happens between basic types (like int to float), between basic and user-defined types (using constructors or conversion operators), and between user-defined types (via conversion functions).

1. Conversion Between Basic Types

Syntax:

```
int a = 10;
float b = a;  // implicit conversion from int to float
```

2. Conversion Between Basic and User-Defined Types

A constructor or conversion operator allows conversion between basic and user-defined types.

Syntax:

```
class MyClass {
public:
   MyClass(int x) { /*...*/ }  // Constructor converts int to MyClass
   operator int() { return value; }  // Converts MyClass to int
};
```

Example:

```
class MyClass {
   int value;
public:
   MyClass(int x) { value = x; }
   operator int() { return value; }
};

MyClass obj = 10;  // int to MyClass
int n = obj;       // MyClass to int
```

3. Conversion Between User-Defined Types

Conversion constructors or conversion operators can convert objects of one class to another.

Syntax:

```
class B;

class A {
public:
    operator B();  // Conversion from A to B
};

class B {
public:
    B(const A& a) { /*...*/ }  // Conversion constructor from A
};
```

Example:

```
class B;

class A {
public:
    int x;
    A(int val) : x(val) {}
    operator B();  // Declare conversion operator
};

class B {
public:
    int y;
    B(const A& a) { y = a.x; }
};

A::operator B() {
    return B(*this);
}

int main() {
    A a(5);
    B b = a;  // Conversion from A to B
}
```

Inheritance

Inheritance allows a class (derived class) to inherit properties and behavior from another class (base class). It supports code reuse and models “Is-a” relationships. Access specifiers (public, protected, private) control member visibility. Derived classes can override base class members to change behavior.

1. Base and Derived Class & Protected Access Specifier

Syntax:

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

class Derived : public Base {
    // Inherits protectedVar and publicVar
};
```

Example:

```
class Base {
protected:
    int prot;
public:
    int pub;
};

class Derived : public Base {
public:
    void show() {
        prot = 10;  // Accessible because protected
        pub = 20;
    }
};
```

2. Derived Class Declaration

```
Syntax:
class Derived : access_specifier Base {
    // ...
};
```

3. Is-a and Has-a Relations

Is-a: Derived class is a type of base class (inheritance).
Has-a: One class contains an object of another (composition).

Example:

```
class Engine {
   // Engine details
};

class Car {
   Engine engine;  // Car has-a Engine
};

class SportsCar : public Car {
   // SportsCar is-a Car
};
```

4. Member Overriding

Member overriding occurs when a derived class provides its own version of a function that is already defined in its base class. This allows the derived class to change or extend the behavior of the base class function. In C++, to enable overriding and achieve runtime polymorphism, the base class function is usually declared with the virtual keyword. When a base class pointer or reference calls a virtual function, the version of the function in the derived class is executed if it exists.

Syntax:

```
class Base {
public:
   virtual void display() { cout << "Base"; }
};

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