```
    Meter(double val) : value(val) {}

    void display() {
        cout << "Meter value: " << value << endl;
    }
};

class Feet {
private:
    double value;

public:
    Feet(double val) : value(val) {}

    // Conversion operator for Meter
    operator Meter() const {
        return Meter(value * 0.3048); // Convert feet to meter
    }
};

int main() {
    Feet feet(10);
    Meter meter = feet; // Conversion from Feet to Meter

    meter.display();
    return 0;
}
```

Section 3 : Virtual Functions & Polymorphism

3.1 Concept of Binding

Binding in C++ refers to the association between function calls and function definitions. It determines which function definition gets executed when a function is called. Binding can be of two types:

1. Static (or Early) Binding
2. Dynamic (or Late) Binding

A. Early Binding (Static or Compile-time Binding)

Early binding, also known as static binding or compile-time binding, occurs when the function call is resolved at compile time. In early binding, the compiler determines which function implementation to call based on the static type of the object or pointer.
Static binding is used for normal function calls, function overloading and operator overloading.

Example:

```
include <iostream>
using namespace std;

class Base {
public:
    void display() {
        cout << "Display method of Base class" << endl;
    }
};

class Derived : public Base {
public:
    void display() {
        cout << "Display method of Derived class" << endl;
    }
};

int main() {
    Derived d;
    Base* ptr = &d; // Pointer of Base class pointing to Derived object

    ptr->display(); // Early binding, calls Base::display() at compile time
    return 0;
}
```

B. Late Binding (Dynamic or Runtime Binding)

Late binding, also known as dynamic binding or runtime binding, occurs when the function call is resolved at runtime. In late binding, the actual function implementation to be called is determined based on the dynamic type of the object pointed to. Dynamic binding is achieved through the use of virtual functions or method overriding and is used in scenarios involving polymorphism.

Example:

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

class Base {
public:
    virtual void display() {
        cout << "Display method of Base class" << endl;
    }
};

class Derived : public Base {
```

```
public:
    void display() {
        cout << "Display method of Derived class" << endl;
    }
};

int main() {
    Derived d;
    Base* ptr = &d; // Pointer of Base class pointing to Derived object

    ptr->display(); // Late binding, calls Derived::display() at runtime
    return 0;
}
```

When to Use Dynamic Binding

Dynamic binding is particularly useful in situations where you need polymorphic behavior. This allows you to write more generic and flexible code. For instance, if you have a base class Shape and derived classes Circle, Square, etc., you can use dynamic binding to call the appropriate draw() function for each shape at runtime.

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

class Shape {
public:
    virtual void draw() = 0;  // Pure virtual function
    virtual ~Shape() {}
};

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

class Square : public Shape {
public:
    void draw() override {
        cout << "Drawing Square" << endl;
    }
};

void displayShape(Shape* shape) {
    shape->draw();
```

```
}

int main() {
    Circle circle;
    Square square;

    displayShape(&circle);  // Drawing Circle
    displayShape(&square);  // Drawing Square

    return 0;
}
```

Here, the draw() function call is dynamically bound to the correct derived class implementation at runtime, demonstrating polymorphic behavior.

3.2 Virtual Functions

In C++, a virtual function is a member function of a class that is declared with the virtual keyword. Virtual functions enable polymorphic behavior, allowing derived classes to provide their own implementation of the function by method overriding.

Declaration and Syntax:

- The virtual keyword is used to declare a function as virtual in the base class.

```
class Base {
public:
    virtual void myVirtualFunction() {
        // Base class implementation
    }
};
```

- In the derived class, the override keyword is used to explicitly indicate that the function is overriding a virtual function from the base class.

```
class Derived : public Base {
public:
    void myVirtualFunction() override {
        // Derived class implementation
    }
};
```

Purpose and Benefits:

1. Polymorphism: Virtual functions enable polymorphism, allowing different classes to provide different implementations of the same function.
2. Dynamic Binding: Virtual functions are resolved at runtime based on the actual type of the object, enabling dynamic method dispatch.
3. Base Class Pointers: Virtual functions are often used with base class pointers to achieve runtime polymorphism.