

```
mirror_mod = modifier_ob.  
set mirror object to mirror.  
mirror_mod.mirror_object  
operation == "MIRROR_X":  
mirror_mod.use_x = True  
mirror_mod.use_y = False  
mirror_mod.use_z = False  
operation == "MIRROR_Y":  
mirror_mod.use_x = False  
mirror_mod.use_y = True  
mirror_mod.use_z = False  
operation == "MIRROR_Z":  
mirror_mod.use_x = False  
mirror_mod.use_y = False  
mirror_mod.use_z = True  
  
selection at the end -add  
mirror_ob.select= 1  
modifier_ob.select=1  
context.scene.objects.active  
("Selected" + str(modifier_ob.  
mirror_ob.select = 0  
= bpy.context.selected_object  
data.objects[one.name].select  
  
print("please select exactly  
  
-- OPERATOR CLASSES ----  
  
types.Operator):  
on X mirror to the selected  
object.mirror_mirror_x"  
mirror X"  
  
context):  
context.active_object is not
```

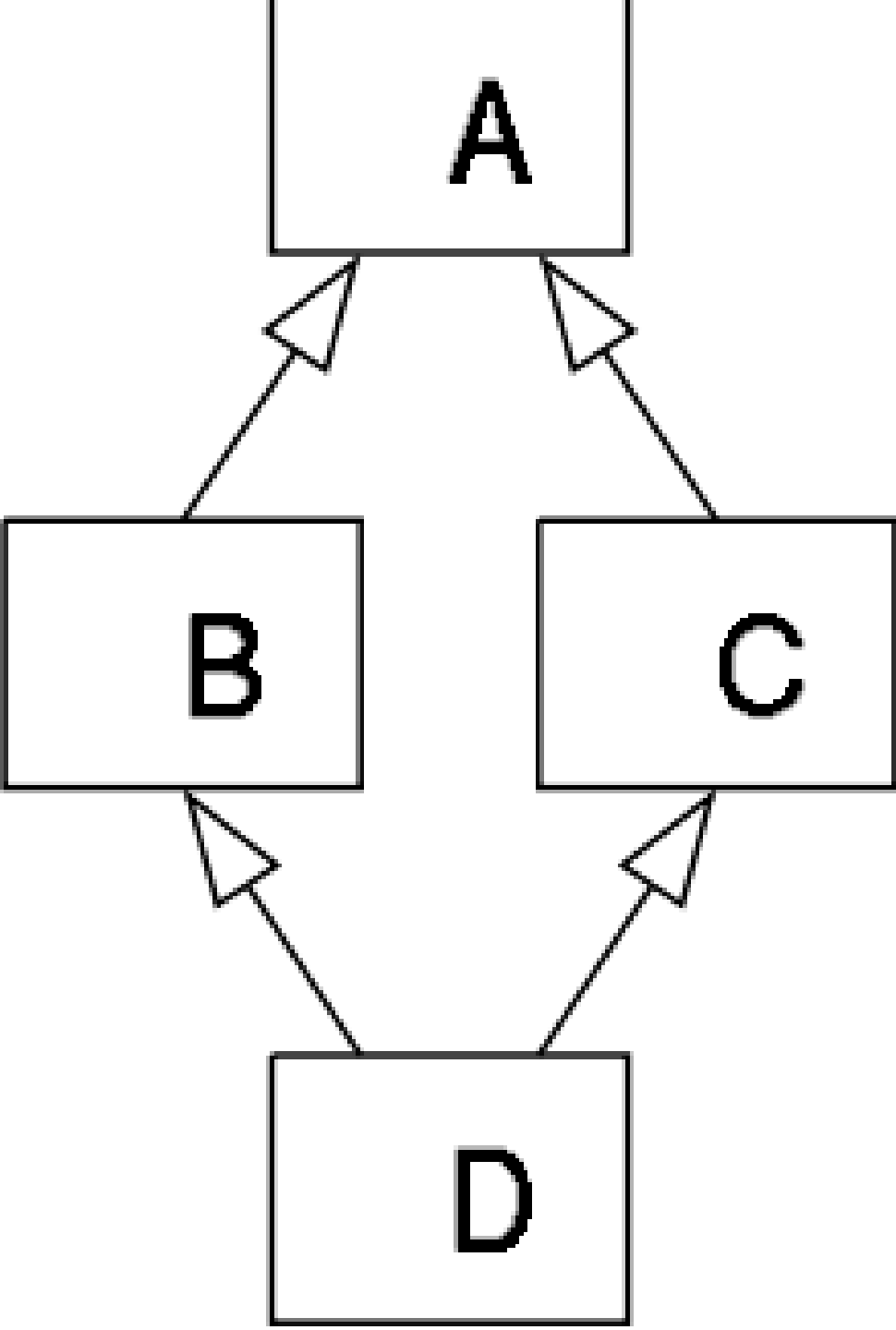
Object Oriented Programming

Bilal Khalid Dar



Inheritance II





Why Multiple Inheritance is dangerous

We will answer it soon!!

Method Overriding



If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in C++.

In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Usage of C++ Method Overriding

Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.

Method overriding is used for runtime polymorphism

Rules for C++ Method Overriding



The method must have the same name as in the parent class



The method must have the same parameter as in the parent class.



There must be an IS-A relationship (inheritance).

Example – Without Overriding

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

class Base {
public:
    void print() {
        cout << "Base Function" << endl;
    }
};

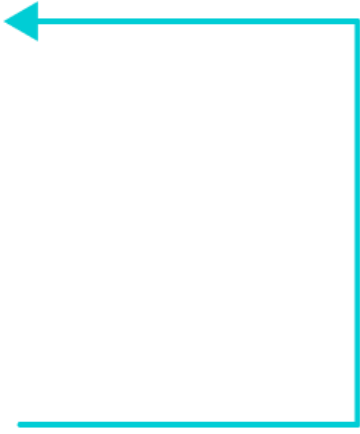
class Derived : public Base {
}

int main() {
    Derived derived1;
    derived1.print();
    return 0;
}
```

- Base Function

Example – Overriding

```
class Base {  
    public:  
        void print() {  
            // code  
        }  
};  
  
class Derived : public Base {  
    public:  
        void print() {  
            // code  
        }  
};  
  
int main() {  
    Derived derived1;  
    derived1.print();  
    return 0;  
}
```



A diagram consisting of a teal line that starts from the `derived1.print();` line in the `main()` function, extends horizontally to the right, then vertically upwards, and finally horizontally to the left, ending with a teal arrowhead pointing to the `void print() {` line inside the `Derived` class definition. This illustrates that the call to `print()` on a `Derived` object is resolved to the `print()` method defined in the `Derived` class, demonstrating method overriding.

Example – Overriding

// C++ program to demonstrate function overriding

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

```
class Base {
public:
    void print() {
        cout << "Base Function" << endl;
    }
};
```

```
class Derived : public Base {
public:
    void print() {
        cout << "Derived Function" << endl;
    }
};
```

```
int main() {
    Derived derived1;
    derived1.print();
    return 0;
}
```

- Derived Function

Example – Access Overridden Function in C++

```
class Base {  
    public:  
        void print() {  
            // code  
        }  
};  
  
class Derived : public Base {  
    public:  
        void print() {  
            // code  
        }  
};  
  
int main() {  
    Derived derived1, derived2;  
  
    derived1.print();  
  
    derived2.Base::print();  
  
    return 0;  
}
```

The diagram illustrates the function calls in the provided C++ code. Two teal arrows originate from the `main()` function. The first arrow starts at the `derived1.print();` line and points to the `void print() {` line within the `Derived` class definition. The second arrow starts at the `derived2.Base::print();` line and points to the `void print() {` line within the `Base` class definition.

Example – Access Overridden Function in C++

// C++ program to demonstrate function overriding

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

```
class Base {
public:
    void print() {
        cout << "Base Function" << endl;
    }
};
```

```
class Derived : public Base {
public:
    void print() {
        cout << "Derived Function" << endl;
    }
};
```

```
int main() {
    Derived derived1;
    derived1.print();
    derived1.Base::print();
    return 0;
}
```

- Derived Function
- Base Function

Example: C++
Call
Overridden
Function From
Derived Class

```
class Base {  
    public:  
        void print() {  
            // code  
        }  
};  
  
class Derived : public Base {  
    public:  
        void print() {  
            // code  
            Base::print();  
        }  
};  
  
int main() {  
    Derived derived1;  
    derived1.print();  
    return 0;  
}
```

```
graph TD; Main["int main() {  
    Derived derived1;  
    derived1.print();  
    return 0;  
}"] --> Derived["class Derived : public Base {  
    public:  
        void print() {  
            // code  
            Base::print();  
        }  
};"]; Derived --> Base["class Base {  
    public:  
        void print() {  
            // code  
        }  
};"];
```

Example: C++ Call Overridden Function From Derived Class

```
// C++ program to call the overridden function  
// from a member function of the derived class
```

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

```
class Base {  
    public:  
    void print() {  
        cout << "Base Function" << endl;  
    }  
};
```

```
class Derived : public Base {  
    public:  
    void print() {  
        cout << "Derived Function" << endl;  
  
        // call overridden function  
        Base::print();  
    }  
};
```

```
int main() {  
    Derived derived1;  
    derived1.print();  
    return 0;  
}
```

- Derived Function
- Base Function

Overriding and Multiple Inheritance

```
class base1 {  
    public:  
        void someFunction( ) {....}  
};  
class base2 {  
    void someFunction( ) {....}  
};  
class derived : public base1, public base2 {};
```

```
int main() {  
    derived obj;  
    obj.someFunction()  
}
```

Function Overloading	Function Overriding
Function Overloading provides multiple definitions of the function by changing signature.	Function Overriding is the redefinition of base class function in its derived class with same signature.
An example of compile time polymorphism.	An example of run time polymorphism.
Function signatures should be different.	Function signatures should be the same.
Overloaded functions are in same scope.	Overridden functions are in different scopes.
Overloading is used when the same function has to behave differently depending upon parameters passed to them.	Overriding is needed when derived class function has to do some different job than the base class function.
A function has the ability to load multiple times.	A function can be overridden only a single time.
In function overloading, we don't need inheritance.	In function overriding, we need an inheritance concept.

Try this example

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

class ClassA {
public:
    int a;
};

class ClassB : public ClassA {
public:
    int b;
};

class ClassC : public ClassA {
public:
    int c;
};

class ClassD : public ClassB, public ClassC {
public:
    int d;
};
```

```
int main()
{
    ClassD obj;

    obj.a = 10;
    obj.b = 20;
    obj.c = 30;
    obj.d = 40;

    cout << "\n b : " << obj.b;
    cout << "\n c : " << obj.c;
    cout << "\n d : " << obj.d << '\n';
}
```

Try this example

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

class ClassA {
public:
    int a;
};

class ClassB : public ClassA {
public:
    int b;
};

class ClassC : public ClassA {
public:
    int c;
};

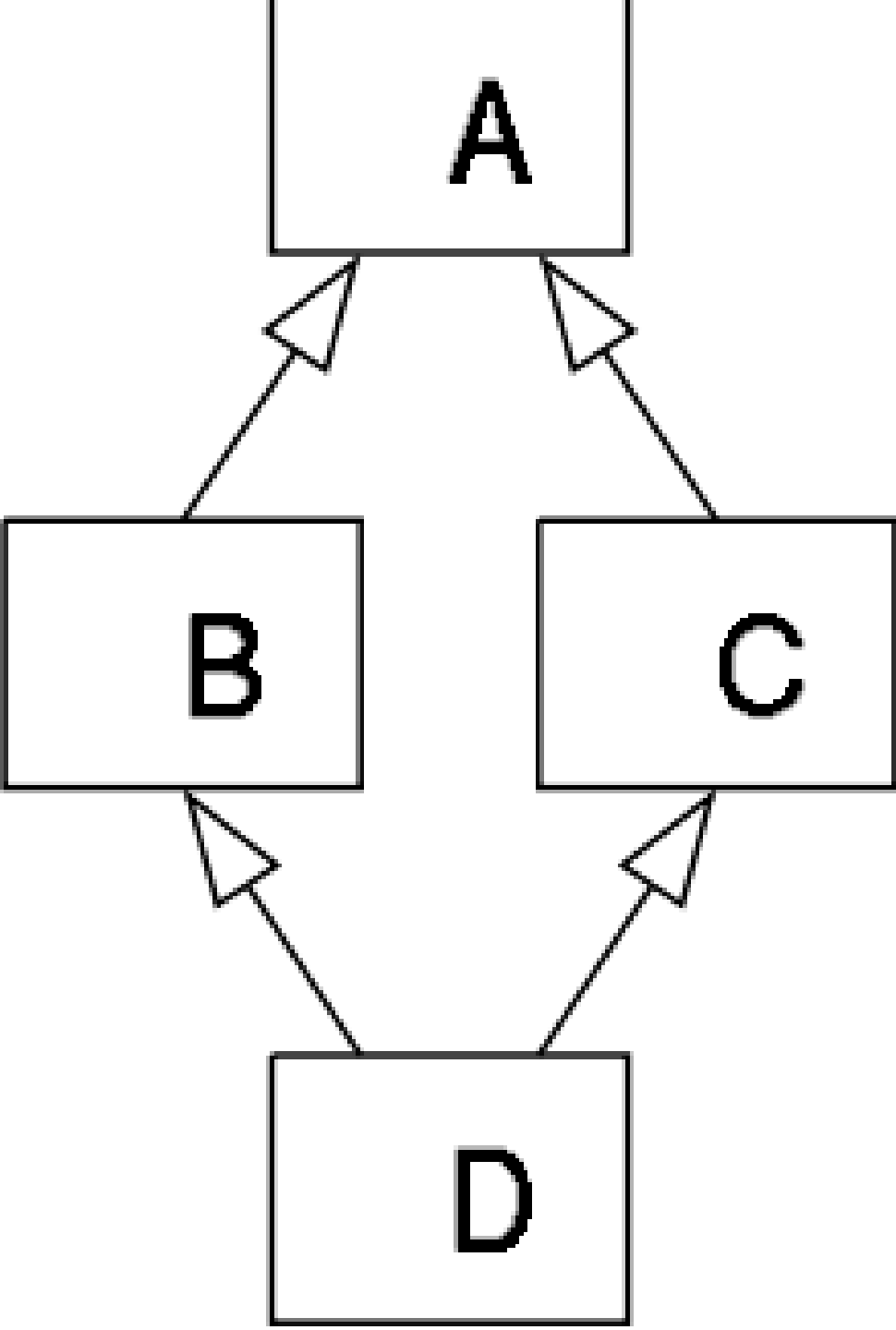
class ClassD : public ClassB, public ClassC {
public:
    int d;
};
```

```
int main()
{
    ClassD obj;

    obj.a = 10;
    obj.b = 20;
    obj.c = 30;
    obj.d = 40;

    cout << "\n b : " << obj.b;
    cout << "\n c : " << obj.c;
    cout << "\n d : " << obj.d << '\n';
}
```

Ambiguity



Why Multiple Inheritance is not Supported

The "**diamond problem**" (sometimes referred to as the "Deadly Diamond of Death") is an ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C. If there is a method in A that B and C have overridden, and D does not override it, then which version of the method does D inherit: that of B, or that of C?

```

#include <iostream>
using namespace std;

class ClassA {
public:
    int a;
};

class ClassB : public ClassA {
public:
    int b;
};

class ClassC : public ClassA {
public:
    int c;
};

class ClassD : public ClassB, public ClassC {
public:
    int d;
};

```

```

int main()
{
    ClassD obj;

    // obj.a = 10;           // Statement 1, Error
    // obj.a = 100;          // Statement 2, Error

    obj.ClassB::a = 10; // Statement 3
    obj.ClassC::a = 100; // Statement 4

    obj.b = 20;
    obj.c = 30;
    obj.d = 40;

    cout << " a from ClassB : " << obj.ClassB::a;
    cout << "\n a from ClassC : " << obj.ClassC::a;

    cout << "\n b : " << obj.b;
    cout << "\n c : " << obj.c;
    cout << "\n d : " << obj.d << '\n';
}

```

Solution 1: Use scope resolution operator

Solution 2: Virtual Base Class

- Virtual base classes are used in virtual inheritance in a way of preventing multiple “instances” of a given class appearing in an inheritance hierarchy when using multiple inheritances.

```

class ClassA
{
    public:
        int a;
};

class ClassB : virtual public ClassA
{
    public:
        int b;
};

class ClassC : virtual public ClassA
{
    public:
        int c;
};

class ClassD : public ClassB, public ClassC
{
    public:
        int d;
};

```

```

int main()
{
    ClassD obj;

    obj.a = 10;
    obj.b = 20;
    obj.c = 30;
    obj.d = 40;

    cout << "\n b : " << obj.b;
    cout << "\n c : " << obj.c;
    cout << "\n d : " << obj.d << '\n';
}

```

Solution 2: Virtual Base Class

Solution 2: Virtual Base Class

- Virtual can be written before or after the public. Now only one copy of data/function member will be copied to class C and class B and class A becomes the virtual base class.
- Virtual base classes offer a way to save space and avoid ambiguities in class hierarchies that use multiple inheritances.
- When a base class is specified as a virtual base, it can act as an indirect base more than once without duplication of its data members. A single copy of its data members is shared by all the base classes that use virtual base.



Protected
modifier in
Inheritance



The protected Modifier

- Visibility modifiers affect the way that class members can be used in a child class
- Variables and methods declared with private visibility cannot be referenced by name in a child class
- They can be referenced in the child class if they are declared with public visibility -- but public variables violate the principle of encapsulation
- There is a third visibility modifier that helps in inheritance situations: `protected`

Protected modifier in Inheritance

- Protected access modifier is similar to that of private access modifiers, the difference is that the class member declared as Protected are inaccessible outside the class but they can be accessed by any subclass(derived class) of that class.

The protected Modifier

- The `protected` modifier allows a child class to reference a variable or method directly in the child class
- It provides more encapsulation than public visibility, but is not as tightly encapsulated as private visibility
- A protected variable is visible to any class in the same package as the parent class



Exercise



Shapes

- Shape:
 - color, fields
 - draw() draw itself on the screen
 - calcArea() calculates its own area.

Kinds of Shapes

- Rectangle
- Triangle
- Circle

Each could be a kind of shape (could be specializations of the shape class).

Each knows how to draw itself, etc.

Exercise



Coding Task

- A grocery shop want to make a software that enables them to save information of their **Customers**. The Customer class keeps tracks of the names and addresses of the customers. (all data members are private). The customer information printInformation() method that prints out their names along with their addresses.
- We have special **OnlineCustomers** (that inherits from the Customer), it adds a new instance variable for the email address of a online customer. Also, the online customers can add their contact number. The class has a function that send notification message to the customers.
- Override the printInformation() method in the OnlineCustomer class to print all his/her information.

- The main of the programs as follows

```
main()
{
    Customer c("Ahmed Khan", "Murree
Road Rawalpindi");
    c.printInformation();
    OnlineCustomer c2("Memmona Khan",
    "i9 Markaz Islamabad", "mk6@gmail.com",
    "03245010000");
    c2.printInformation();
}
}
```