

OBJECT ORIENTED PROGRAMMING LAB



Lab Manual # 6+7

Types of Inheritance in C++

Composition in C++

Instructor: Iqra Rehman

Semester Summer, 2024

Course Code: CL1004

**Fast National University of Computer and Emerging Sciences
Peshawar**

Department of Computer Science

OBJECT ORIENTED PROGRAMMING LANGUAGE

Table of Contents

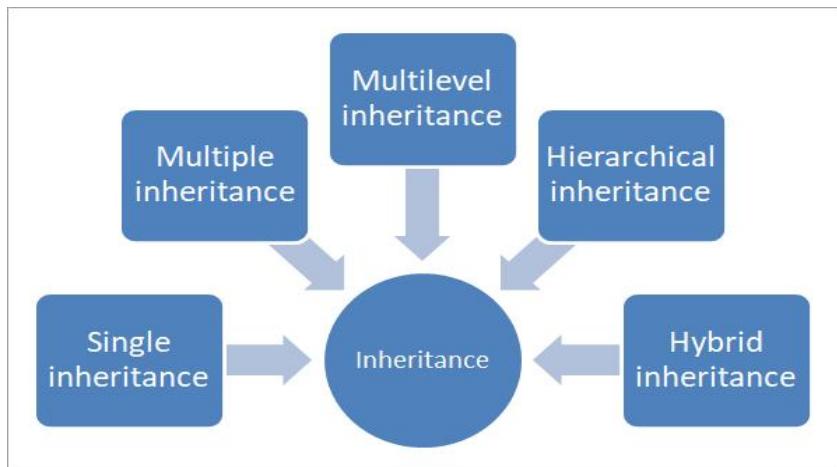
Types of Inheritance	1
1) Single Inheritance	1
2) Multilevel Inheritance	1
3) Multiple Inheritance	6
Ambiguity in Multiple Inheritance	9
4) Hierarchical Inheritance	10
5) Hybrid Inheritance	15
Derived Class Constructors	15
Constructor in Single Inheritance without Arguments	15
Constructor in Single Inheritance with Arguments	16
Constructors in Multiple Inheritance Without Arguments	20
Constructors in Multiple Inheritance with Arguments	21
Introduction to Complex Objects and Composition	23
Scope of Use	24
Program Example # 01	25
C++ Composition	26
Program Example # 02	27
Program Example # 03	28
References	29

Types of Inheritance

Inheritance is one of the core feature of an object-oriented programming language. It allows software developers to derive a new class from the existing class. The derived class inherits the features of the base class (existing class).

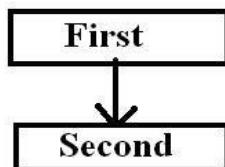
There are various models of inheritance in C++ programming.

- 1) Single Inheritance
- 2) Multilevel Inheritance
- 3) Multiple Inheritance
- 4) Hierarchical Inheritance
- 5) Hybrid Inheritance



1) Single Inheritance

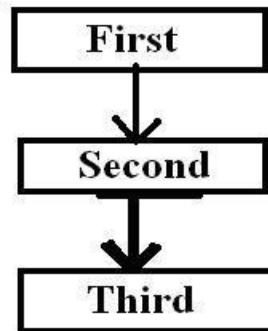
- ❖ It is the inheritance hierarchy wherein one derived class inherits from one base class.
- ❖ There is only one Super Class and Only one Sub Class Means they have one to one Communication between them.



2) Multilevel Inheritance

- ❖ A Derived class can also be inherited by another class Means in this When a Derived Class again will be inherited by another Class then it creates a Multiple Levels.

- ❖ It is the inheritance hierarchy wherein subclass acts as a base class for other classes.
- ❖ In C++ programming, not only you can derive a class from the base class but you can also derive a class from the derived class. This form of inheritance is known as multilevel inheritance.
- ❖ It is implemented by defining at least three classes. Each derived class must have a kind of relationship with its immediate base class.



Syntax of Multilevel Inheritance

```
class A {  
    ...  
}; // end of class A  
  
class B: public A {  
    ...  
}; // end of class B  
  
class C: public B {  
    ...  
}; // end of class C
```

Here, class B is derived from the base class A and the class C is derived from the derived class B.

Example 1: C++ Multilevel Inheritance

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

class A {
public:
    void display() {
        cout<<"Base class content.";
    }
};

class B : public A {};

class C : public B {};

int main() {
    C obj;
    obj.display();
    return 0;
}
/*
Output
Base class content.*/

```

- ❖ In this program, class C is derived from class B (which is derived from base class A).
- ❖ The obj object of class C is defined in the main() function.
- ❖ When the display() function is called, display() in class A is executed. It's because there is no display() function in class C and class B.
- ❖ The compiler first looks for the display() function in class C. Since the function doesn't exist there, it looks for the function in class B (as C is derived from B).
- ❖ The function also doesn't exist in class B, so the compiler looks for it in class A (as B is derived from A).
- ❖ If display() function exists in C, the compiler overrides display() of class A (because of member function overriding).

Example 2: C++ Multilevel Inheritance

```
#include<iostream>
using namespace std;
class person {
```

```
private:  
    char name[20];  
    long int phno;  
public:  
    void read() {  
        cout<<"Enter name and phno = ";  
        cin>>name>>phno;  
    }  
    void show() {  
        cout<<"\nName = "<<name;  
        cout<<"\nPhone number = "<<phno;  
    }  
}; // end of person class  
class student : public person {  
private:  
    int rollno;  
    char course[20];  
public:  
    void read() {  
        person::read(); //Access person's read()  
        cout<<"Enter rollno and course=";  
        cin>>rollno>>course;  
    }  
    void show() {  
        person::show(); //Access person's show()  
        cout<<"\nRollno = "<<rollno;  
        cout<<"\nCourse ="<<course;  
    }  
}; // end of student class  
class exam : public student {  
private:  
    int m[4];  
    double per;  
public:  
    void read();  
    void cal();  
    void show();  
}; // end of exam class  
  
// Function definitions  
void exam::read() {  
    student::read(); //Access student's read()  
    cout<<"Enter marks :";  
    for(int i=0;i<4;i++)
```

```
    cin>>m[i];
}
void exam::cal() {
    int tot_marks=0;
    for(int i=0;i<4;i++)
        tot_marks+=m[i];
    per=double(tot_marks)/4;
}
void exam::show() {
    student::show(); //Access student's show()
    cout<<"\nMarks :";
    for(int i=0;i<4;i++)
        cout<<m[i]<<"\t";
    cout<<"\nPercentage = "<<per;
}
//main function
int main() {
    exam e1;
    e1.read();
    e1.cal();
    e1.show();
    return 0;
}

/*
Output
Name = Aisha
Phone number = 57868882
Rollno = 1222
Course =Computer
Marks :88      77      66      98
Percentage = 82.25
*/
```

Explanation: The above program implements multilevel inheritance. The program inputs the students basic and academic information calculates the result and displays all the students' information. The program consists of three classes, namely person, student and exam. The class person is the base class. The class student is derived from person base class, which acts as a base class for the derived class exam. The exam class inherits the features of student class directly and features of person class indirectly. In main(), the statement

```
e1.read();
e1.show();
```

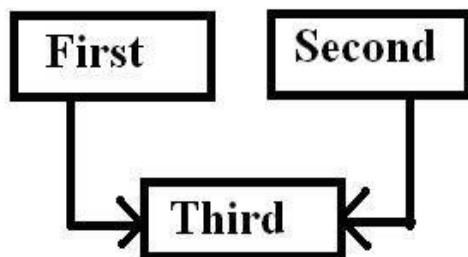
Invokes the respective member functions of exam class as e1 is the object of this class. Look carefully at the definition of read() and show() of the exam class, and you will find that both the member functions access same-named member function of the student class directly and person class indirectly. The statement,

```
e1.cal();
```

Invokes the member function cal() of exam class for calculating the result of a student.

3) Multiple Inheritance

- ❖ It is the inheritance hierarchy wherein one derived class inherits from multiple base class(es).
- ❖ So far, we have been deriving a class from a single base class. However, it is also possible to derive a class from two or more unreleased base classes. When we derive a class from two or more base classes, this form of inheritance is known as Multiple Inheritance. It allows the new class to inherit the functionality of two or more well developed and tested base classes, thus reusing predefined classes' features.
- ❖ Like Single Inheritance, each derived class in multiple inheritances must have a kind of relationship with its base classes.
- ❖ This form of inheritance is useful in those situations in which derived class represents a combination of features from two or more base classes.
- ❖ **For Example:** In an educational institute, a member can act like a teacher and a student. He may be a student of higher class as well as teaching lower classes.



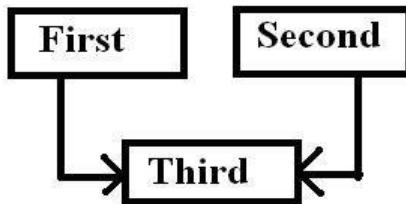
Multiple Inheritance Syntax

The syntax for declaring a derived class which inherited from more than one base classes is:

```
class der_class_name : access_specifier base1, access_specifier base2, ... {
    //Implementation
};
```

Here access_specifier may be either public, private or protected. Each base class from which class is derived must be specified to the colon's right after the access specifier and must be separated by commas.

- ❖ The Fig. shows multiple inheritances in which class Third is derived from class First and class Second, which are independent of each other.
- ❖ Here, class Third inherits all the members of its base classes First and Second. The following program segment shows how a class Third is derived from two base classes First and Second publicly.



```
class Third: public First, public Second
{
    //Implementation
};
```

- ❖ The inheritance rules and the usages of access specifier for each base class member remain the same as in single inheritance. The access to each base class member is independent of the access to the members of other base classes using which you derive your class.

Example 1: Multiple Inheritance

```
#include<iostream>
using namespace std;
class base1 {
protected:
    int x;
public:
```

```
void readx() {
    cout<<"Enter value of x : ";
    cin>>x;
}
void showx()
{
    cout<<"x= "<<x;
}
}; // end of class base1
class base2 {
protected:
    int y;
public:
    void ready() {
        cout<<"Enter value of y: ";
        cin>>y;
    }
    void showy()
    {
        cout<<"\ny= "<<y;
    }
}; // end of class base2
class der : public base1,public base2 {
private:
    int z;
public:
    void add() {
        z=x+y; //Accessing protected members of base classes
    }
    void showz ()
    {
        cout<<"\nz = "<<z;
    }
}; // end of class der1
int main() {
    der d1;
    d1.readx(); //calling readx() of base1
    d1.ready(); //calling ready() of base2
    d1.add(); //calling add() of der
    d1.showx(); //calling showx() of base1
    d1.showy(); //calling showy() of base2
    d1.showz(); //calling showz() of der
    return 0;
}

/*
Output
Enter value of x : 22
Enter value of y: 11
```

```
x= 22
y= 11
z = 33
*/
```

Explanation: In this program, we define a class base1 containing a protected data member x and public member functions readx() and showx() for inputting and displaying the value of x. Similarly, another class base2 is defined that contains a data member y and two public member function ready() and showy(). The class der is inherited from both the base classes base1 and base2 publicly through multiple inheritances. As the class der is derived publicly from both the base class base1 and base2, the derived class object can access public base el and base2 in the main(). In the main() , the statement d1.readx() ; invokes the member function readx() of the base1 class for inputting value of x. Similarly the statement, d1.ready() ; invokes the ready() of base2 for inputting value of y. The statement d1.add(); invokes the member function add() of the der class to perform addition of values of protected data members x and y as they can be accessed directly in the der class. The remaining statement displays the values of x, y and z.

Ambiguity in Multiple Inheritance

- ❖ The most obvious problem with multiple inheritance occurs during function overriding.
- ❖ Suppose, two base classes have a same function which is not overridden in derived class.
- ❖ If you try to call the function using the object of the derived class, compiler shows error. It's because compiler doesn't know which function to call.
- ❖ **For example,**

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

class base1 {
public:
    void someFunction( )
    {
        cout<<"Function of base1";
    }
}; // end of base1 class
class base2 {
public:
    void someFunction( )
    {
        cout<<"Function of base1";
    }
}
```

```

}; // end of base2 class

class derived : public base1, public base2 {
    // code of derived class
};

int main() {
    derived obj;
    obj.someFunction() // Error!
} // end of main function

```

Problem Solution:

This problem can be solved using the scope resolution function to specify which function to call either **base1** or **base2**

```

int main() {
    derived obj;

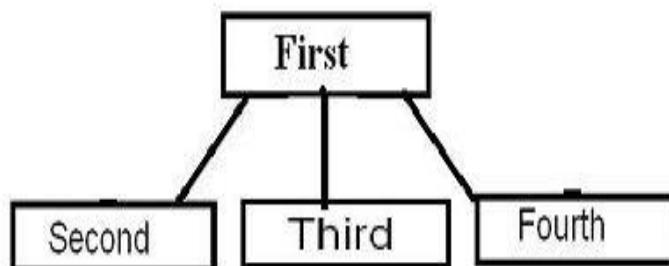
    obj.base1::someFunction( ); // Function of base1 class is called
    obj.base2::someFunction(); // Function of base2 class is called.
}

/*
Output
Function of base1
Function of base1
*/

```

4) Hierarchical Inheritance

- ❖ If more than one class is inherited from the base class, it's known as hierarchical inheritance. In hierarchical inheritance, all features that are common in child classes are included in the base class.
- ❖ For example, Physics, Chemistry, Biology are derived from Science class. Similarly, Dog, Cat, Horse are derived from Animal class.



Syntax of Hierarchical Inheritance

```
class base_class {  
    ... ... ...  
}  
  
class first_derived_class: public base_class {  
    ... ... ...  
}  
  
class second_derived_class: public base_class {  
    ... ... ...  
}  
  
class third_derived_class: public base_class {  
    ... ... ...  
}
```

Example 1: Hierarchical Inheritance in C++ Programming

```
// C++ program to demonstrate hierarchical inheritance  
#include <iostream>  
using namespace std;  
  
// base class  
class Animal {  
public:  
    void info() {  
        cout << "I am an animal." << endl;  
    }  
};  
  
// derived class 1  
class Dog : public Animal {  
public:  
    void bark() {  
        cout << "I am a Dog. Woof woof." << endl;
```

```
    }
};

// derived class 2
class Cat : public Animal {
public:
    void meow() {
        cout << "I am a Cat. Meow." << endl;
    }
};

int main() {
    // Create object of Dog class
    Dog dog1;
    cout << "Dog Class:" << endl;
    dog1.info(); // Parent Class function
    dog1.bark();

    // Create object of Cat class
    Cat cat1;
    cout << "\nCat Class:" << endl;
    cat1.info(); // Parent Class function
    cat1.meow();

    return 0;
}

/*
Output
Dog Class:
I am an animal.
I am a Dog. Woof woof.
Cat Class:
I am an animal.
I am a Cat. Meow.
*/
```

Here, both the **Dog** and **Cat** classes are derived from the **Animal** class. As such, both the derived classes can access the **info()** function belonging to the **Animal** class.

Example 2: Hierarchical Inheritance in C++ Programming

- ❖ The following program implements hierarchical Inheritance. Here, we define a base class person. Then we derive two classes teacher and student. The derived classes inherit the common member's name, phno, read(), show() from the base class person, and have its members.
- ❖ The public read() and show() of the base class redefined in each derived class to input and display teacher and student information.

```
#include<iostream>
using namespace std;
class person {
private:
    char name[20];
    long int phno;
public:
    void read() {
        cout<<"Enter name and phno = ";
        cin>>name>>phno;
    }
    void show() {
        cout<<"\nName = "<<name;
        cout<<"\nPhone number = "<<phno;
    }
}; //Base class specification ends
class student : public person {
private:
    int rollno;
    char course[20];
public:
    void read() {
        person::read();
        cout<<"Enter rollno and course=";
        cin>>rollno>>course;
    }
    void show() {
        person::show();
        cout<<"\nRollno = "<<rollno;
        cout<<"\nCourse = "<<course;
    }
}; //Derived class specification ends
class teacher : public person {
private:
    char dept_name[10];
    char qual[10];
public:
    void read() {
```

```
person::read();
cout<<"Enter dept_name and qualification = ";
cin>>dept_name>>qual;
}
void show() {
    person::show();
    cout<<"\nDepartment name = "<<dept_name;
    cout<<"\nQualification = "<<qual;
}
}; //Derived class specification ends
int main() {
    student s1;
    cout<<"Enter student Information\n";
    s1.read();
    cout<<"Displaying Student Information ";
    s1.show();
}

teacher t1;
cout<<"\nEnter teacher Information\n";
t1.read();
cout<<"Displaying teacher Information ";
t1.show();
return 0;
}

/*
Output
Enter student Information
Enter name and phno = Ausaf 0344
Enter rollno and course=44 OOP

Displaying Student Information
Name = Ausaf
Phone number = 344
Rollno = 44
Course = OOP

Enter teacher Information
Enter name and phno = Abdullah      0332
Enter dept_name and qualification = Computer MS
Displaying teacher Information

Name = Abdullah
Phone number = 332
Department name = Computer
```

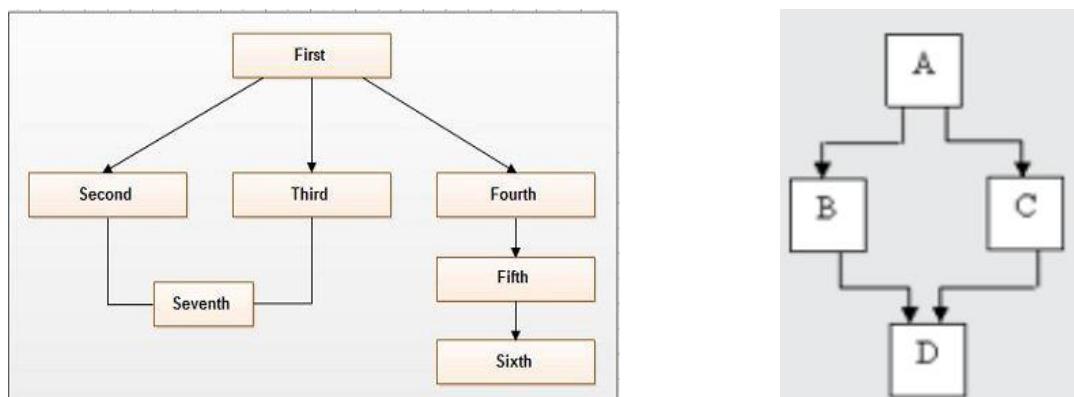
Qualification = MS
*/

Explanation:

- ❖ In main (), we first creates an object s1 of class student through which we call the read() and show() member functions of student's class for inputting and displaying student information.
- ❖ Then similarly, we create an object t1 of teacher class for inputting and displaying teacher information.

5) Hybrid Inheritance

- ❖ The inheritance hierarchy that reflects any legal combination of other four types of inheritance.
- ❖ This is a Mixture of two or More Inheritance and in this Inheritance a Code May Contains two or Three types of inheritance in Single Code.



Derived Class Constructors

In inheritance, a constructor of the derived class as well as the constructor functions of the base class are automatically executed when an object of the derived class is created.

Constructor in Single Inheritance without Arguments

The following example explains the concept of execution of constructor functions in single inheritance.

Example:

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

class Parent
```

```
{  
    public:  
    Parent(void)  
    {  
        cout<<"Constructor of the Parent Class Called"\  
    }  
}; // end of Parent class  
  
class Child : public Parent  
{  
    public:  
    Child(void)  
    {  
        cout<<"Constructor of the Child Class Called"\  
    }  
}; // end of child class  
  
int main() {  
  
    Child obj; // object of the child class  
  
    return 0;  
}  
  
/*Sample Output  
  
Constructor of the Parent Class Called  
Constructor of the Child Class Called */
```

In the above program, when an object of the derived class Child is created, the constructor of both the derived class as well the base class are executed.

Constructor in Single Inheritance with Arguments

In case of constructor functions with arguments, the syntax to define constructor of the derived class is different. To execute a constructor of the base class that has arguments through the derived class constructor, the derived class constructor is defined as:

- Write the name of the constructor function of the derived class with parameters.
- Place a colon (:) immediately after this and then write the name of the constructor function of the base class with parameters.

The following example explains the concept of the constructor functions with arguments in single inheritance.

```
#include <iostream>
using namespace std;
#include<string.h>

class Student
{
    private:
        string name ;
        string address;

    public:

        Student(string nm, string add)
        {
            this->name = nm;
            this->address = add;

        }
        void show()
        {
            cout<<"Name is: "<<name<<endl;
            cout<<"Address is: "<<address<<endl;
        }
}; // end of Student class

class Marks : public Student
{
    private:
        int sub1, sub2, sub3, total;

    public:

        Marks(string nm, string add, int a, int b, int c):Student(nm, add)
        {s
            sub1=a;
            sub2=b;
            sub3=c;
            total=a+b+c;
        }
        void show_detail();
}; // end of Marks class

void Marks:: show_detail()
{
    show();
```

```
cout<<"Marks of 1st subject: "<<sub1<<endl;
cout<<"Marks of 1st subject: "<<sub2<<endl;
cout<<"Marks of 1st subject: "<<sub2<<endl;
cout<<"Total Marks      : "<<total<<endl;
}

int main() {

    Marks marks("Aiman", "Hangu", 66,77,88);
    marks.show_detail();
    return 0;
}

/*Sample Output
Name is: Aiman
Address is: Hangu
Marks of 1st subject: 66
Marks of 1st subject: 77
Marks of 1st subject: 88
Total Marks      : 231
*/
```

The compiler automatically calls a base class constructor before executing the derived class constructor. The compiler's default action is to call the default constructor in the base class. You can specify which of several base class constructors should be called during the creation of a derived class object.

```
#include<iostream>
using namespace std;
class Rectangle
{
private:
    float length;
    float width;
public:
    Rectangle ()
    {
        length = 0;
        width = 0;
    }

    Rectangle (float len, float wid)
    {
        length = len;
        width = wid;
    }
}
```

```
float area()
{
    return length * width;
}

class Box: public Rectangle
{
private:
    float height;
public:
    Box ()
    {
        height = 0;
    }

    Box (float len, float wid, float ht) : Rectangle(len, wid)
    {
        height = ht;
    }

    float volume()
    {
        return area() * height;
    }
};

int main ()
{
    Box bx;
    Box cx(4,8,5);
    cout << bx.volume() << endl;
    cout << cx.volume() << endl;
    return 0;
}
/*
output
0
160
*/
```

Constructors in Multiple Inheritance Without Arguments

When a class is derived from more than one base classes, the constructor of the derived class as well as of all its base classes are executed when an object of the derived class is created.

If the constructor functions have no parameters then first the constructor functions of the base classes are executed and then the constructor functions of the derived class are executed.

A program is given below to explain the execution of the constructor without parameters.

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

class A
{
public:
A(void)
{
    cout<<"Constructor of class A"<<endl;
}
}; // end of A class

class B
{
public:
B(void)
{
    cout<<"Constructor of class B"<<endl;
}
}; // end of B class

class C : public A, public B
{
public:
C(void)
{
    cout<<"Constructor of class C"<<endl;
}
}; // end of C class

int main() {
    C objC;
    return 0;
}
```

```
/*
Constructor of class A
Constructor of class B
Constructor of class C
*/
```

In the above program, three classes are defined. The **C** class is publicly derived from two classes **A** and **B**. All the classes have the constructor functions. When an object of the derived class **C** is created, the constructor functions are executed in the following sequence:

- Constructor of the class **A** is executed first.
- Constructor of the class **B** is executed second.
- Constructor of the derived class **C** is executed in the end.

Constructors in Multiple Inheritance with Arguments

To execute constructors of base classes that have arguments through the derived constructor functions, the derived class constructor is defined as:

- Write the constructor function of the derived class with parameters.
- Place a colon (:) immediately after this and then write the constructor functions names of the base classes with parameters separated by commas.

An example is given below to explain the constructor functions arguments in multiple inheritance.

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

class Student
{
    private:
        string name ;
        string address;

    public:

        Student(string nm, string add)
        {
            this->name = nm;
            this->address = add;

        }
        void show()
```

```
{  
    cout<<"Name is: "<<name<<endl;  
    cout<<"Address is: "<<address<<endl;  
}  
}; // end of Student class  
  
class Marks  
{  
    private:  
    int sub1, sub2, sub3, total;  
  
    public:  
  
    Marks( int a, int b, int c)  
    {  
        sub1=a;  
        sub2=b;  
        sub3=c;  
        total=a+b+c;  
    }  
    void show_detail(){  
        cout<<"Marks of 1st subject: "<<sub1<<endl;  
        cout<<"Marks of 1st subject: "<<sub2<<endl;  
        cout<<"Marks of 1st subject: "<<sub2<<endl;  
        cout<<"Total Marks : "<<total<<endl;  
    }  
}; // end of Marks class  
  
class ShowStudent : public Student, public Marks  
{  
    public:  
  
    ShowStudent(string nm, string add, int a, int b, int c) :  
    Student(nm, add), Marks(a, b, c){  
  
        show();          // calling show() method of student class  
        show_detail();   // calling show_detail() of marks class  
    }  
  
}; // end of ShowStudent class  
/*
```

output

```
Name is: Amir  
Address is: Kohat  
Marks of 1st subject: 66  
Marks of 1st subject: 99  
Marks of 1st subject: 88  
Total Marks      : 253 */
```

Introduction to Complex Objects and Composition

An object is a basic unit of Object-Oriented Programming and represents the real-life entities. **Complex objects are the objects that are built from smaller or a collection of objects. For example**, a mobile phone is made up of various objects like a camera, battery, screen, sensors, etc. In this article, we will understand the use and implementation of a complex object.

In object-oriented programming languages, object composition is used for objects that have a “**has-a**” relationship with each other. **For example**, a mobile has-a battery, has-a sensor, has-a screen, etc. Therefore, the complex object is called the whole or a parent object whereas a simpler object is often referred to as a child object. In this case, all the objects or components are the child objects which together make up the complex object(mobile).

Classes that have data members of built-in type work really well for simple classes. However, in real-world programming, the product or software comprises of many different smaller objects and classes. In the above example, the mobile phone consisted of various different objects that on a whole made up a complete mobile phone. Since each of those objects performs a different task, they all are maintained in different classes. Therefore, this concept of complex objects is being used in most of the real-world scenarios. The advantages of using this concept are:

- Each individual class can be simple and straightforward.
- One class can focus on performing one specific task and obtain one behavior.
- The class is easier to write, debug, understand, and usable by other programmers.
- While simpler classes can perform all the operations, the complex class can be designed to coordinate the data flow between the simpler classes.
- It lowers the overall complexity of the complex object because the main, task of the complex object would then be to delegate tasks to the sub-objects, who already know how to do them.

The most important advantage is if any changes have to be made in a child class, only the child class can be changed rather than changing the entire parent class.

- For example, if we want to change the battery class in the mobile object, with the help of composition, the changes are only made in the battery class and the entire mobile object works fine with the updated changes.

Scope of Use

Although there are no well-defined rules to state when a programmer must use the composition, as a rule of thumb, each class should be built to accomplish a single task. The task should be to either perform some part of manipulation or be responsible for coordinating other classes but cannot perform both tasks. This immensely increases the maintenance of the code and future updatations because, when we wish to update a feature or an object, only the class pertaining to that specific functionality needs to be updated. And also, it's very easy to keep track of the errors in the program. For example:

```
// C++ program to implement a composition

// A point class
class Point {
private:
    int x;
    int y;
};

// Every location has a point.
// Every point has two coordinates.
// The above class's functionality
// is only to store the coordinates
// in two variables. Any functionality
// using the points is implemented
// here
class Location {
    Private:
    Point Source;
    Point Destination
};
```

In the classes given here, **Location** uses objects of class **Point** as its data members. Hence, **Location** is a complex class which uses a simple class **Point**. Let us have a look at the program that makes use of the composition.

Below is the implementation of a composite class:

Program Example # 01

```
// C++ program to implement a composite class
using namespace std;
#include <iostream>

// Class with a private parameter
// and the getters and setters
class One {

    // Private parameter
private:
    int num;

    // Public setter and getter
public:
    void set(int i)
    {
        num = i;
    }
    int get()
    {
        return num;
    }
};

// Another class
class Two {

    // Public method and the object
    // of the previous class
public:
    One O;
    void show()
    {
        cout << "\n Number = "
            << O.get();
    }
};

// Driver code
int main()
{
    // Creating the object of
    // class Two
```

```
Two T;  
  
// Perform some operation using  
// the object of One in this class  
T.O.set(100);  
T.show();  
}
```

Output:

Number = 100

Explanation: Note that in the program, class Two has an object of class One. To access a member of One, we must use the object of Two as in T.O.set(100). Moreover, since num is a private member of One, we must use a public function to access its value.

C++ Composition

In real-life complex objects are often built from smaller and simpler objects. For example, a car is built using a metal frame, an engine some tires, a transmission system, a steering wheel, and a large number of other parts. A personal computer is built from a CPU, a motherboard, memory unit, input and output units, etc. even human beings are building from smaller parts such as a head, a body, legs, arms, and so on. This process of building complex objects from simpler ones is called c++ composition. It is also known as object composition.

So far, all of the classes that we have used had member variables that are built-in data type (e.g. int, float, double, char). While this is generally sufficient for designing and implementing small, simple classes, but it gradually becomes difficult to carry out from more complex classes, especially for those built from many sub-parts. In order to facilitate the building of complex classes from simpler ones, C++ allows us to do object C++ Composition in a very simple way by using classes as member variables in other classes.

A class can have one or more objects of other classes as members. A class is written in such a way that the object of another existing class becomes a member of the new class. this relationship between classes is known as C++ Composition. It is also known as containment, part-whole, or has-a relationship. A common form of software reusability is C++ Composition.

In C++ Composition, an object is a part of another object. The object that is a part of another object is known as a sub-object. When a C++ Composition is destroyed, then all of its subobjects are destroyed as well. Such as when a car is destroyed, then its motor, frame, and other parts are also destroyed with it. It has a do and die relationship.

Program Example # 02

How to use C++ Composition in programming

```
#include <iostream>
using namespace std;
class X
{
    private:
        int d;
    public:
        void set_value(int k)
        {
            d=k;

        }
        void show_sum(int n)
        {
            cout<<"sum of "<<d<<" and "<<n<<" = "<<d+n<<endl;

        }
};

class Y
{
    public:
        X a;
        void print_result()
        {
            a.show_sum(5);

        }
};

int main()
{
    Y b;
    b.a.set_value(20);
    b.a.show_sum(100);
    b.print_result();

}
/*
Output
```

```
sum of 20 and 100 = 120
sum of 20 and 5 = 25
*/
```

Programming Explanation:

- In this program, class X has one data member ‘d’ and two member functions ‘set_value()’ and ‘show_sum()’. The set_value() function is used to assign value to ‘d’. the show_sum() function uses an integer type parameter. It adds the value of parameter with the value of ‘d’ and displays the result on the screen.
- Another class Y is defined after the class x. the class Y has an object of class x that is the C++ Composition relationship between classes x and y. this class has its own member function print_result().
- In the main() function, an object ‘b’ of class y is created. The member function set_value() of object ‘a’ that is the sub-object of object ‘b’ is called by using two dot operators. One dot operator is used to access the member of the object ‘b’ that is object ‘a’, and second is used to access the member function set_value() of sub-object ‘a’ and ‘d’ is assigned a value 20.
- In the same way, the show_sum() member function is called by using two dot operators. The value 100 is also passed as a parameter. The member function print_result of object ‘b’ of class Y is also called for execution. In the body of this function, the show_sum() function of object ‘a’ of class X is called for execution by passing value 5.

Program Example # 03

```
#include <iostream>
#include <string>

using namespace std;

class Birthday{

public:
    Birthday(int cmonth, int cday, int cyear){
        cmonth = month;
        cday = day;
        cyear = year;

    }
    void printDate(){
        cout<<month <<"/" <<day <<"/" <<year <<endl;
    }
}
```

```
private:  
    int month;  
    int day;  
    int year;  
  
};  
  
class People{  
  
public:  
    People(string cname, Birthday cdateOfBirth)  
    :name(cname),  
    dateOfBirth(cdateOfBirth)  
    {  
  
    }  
    void printInfo(){  
        cout<<name <<" was born on: "  
        dateOfBirth.printDate();  
    }  
  
private:  
    string name;  
    Birthday dateOfBirth;  
  
};  
  
int main() {  
  
    Birthday birthObject(7,9,97);  
    People infoObject("Lenny the Cowboy", birthObject);  
    infoObject.printInfo();  
  
}
```

References

<https://beginnersbook.com/2017/08/cpp-data-types/>

http://www.cplusplus.com/doc/tutorial/basic_io/

<https://www.w3schools.com/cpp/default.asp>

<https://www.javatpoint.com/cpp-tutorial>

<https://www.geeksforgeeks.org/object-oriented-programming-in-cpp/?ref=lbp>

<https://www.programiz.com/>

<https://ecomputernotes.com/cpp/>