NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

OBJECT ORIENTED PROGRAMMING - LAB

Instructors: Muhammad Sudais

Email: muhammad.sudais.v@nu.edu.pk

LAB 09

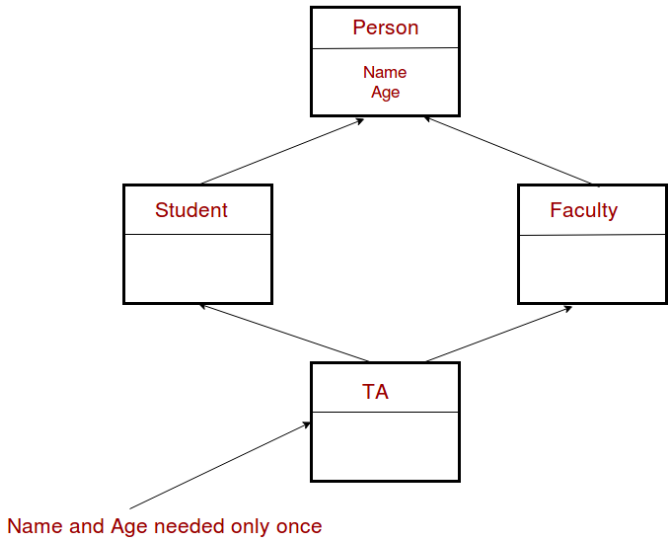
Outline

- Diamond Problem
- Virtual Function
- Abstract Classes
- Casting

Diamond Problem

In case of hybrid inheritance, a Diamond problem may arise. The “dreaded diamond” refers to a class structure in which a particular class appears more than once in a class’s inheritance hierarchy.

The diamond problem occurs when two super classes of a class have a common base class. For example, in the following diagram, the TA class gets two copies of all attributes of Person class, this causes ambiguities.



For example, consider the following program.

```
#include<iostream>
using namespace std;
class Person {
  // Data members of person
public:
   Person(int x)  { cout << "Person::Person(int ) called" << endl;  }
};
```

```
class Faculty : public Person {
  // data members of Faculty
public:
   Faculty(int x):Person(x)   {
      cout<<"Faculty::Faculty(int ) called"<< endl;
   }
};

class Student : public Person {
  // data members of Student
public:
   Student(int x):Person(x) {
      cout<<"Student::Student(int ) called"<< endl;
   }
};

class TA : public Faculty, public Student  {
public:
   TA(int x):Student(x), Faculty(x)   {
      cout<<"TA::TA(int ) called"<< endl;
   }
};

int main()  {
   TA ta1(30);
}
```

Solution:

In the above program, constructor of 'Person' is called two times. Destructor of 'Person' will also be called two times when object 'ta1' is destructed. So object 'ta1' has two copies of all members of 'Person', this causes ambiguities. The solution to this problem is 'virtual' keyword. We make the classes 'Faculty' and 'Student' as virtual base classes to avoid two copies of 'Person' in 'TA' class. For example, consider the following program.

```
#include<iostream>
using namespace std;
class Person {
public:
   Person(int x)  { cout << "Person::Person(int ) called" << endl;   }
   Person()    { cout << "Person::Person() called" << endl;   }
};

class Faculty : virtual public Person {
public:
   Faculty(int x):Person(x)   {
      cout<<"Faculty::Faculty(int ) called"<< endl;
   }
};

class Student : virtual public Person {
public:
   Student(int x):Person(x) {
      cout<<"Student::Student(int ) called"<< endl;
   }
};
```

```
class TA : public Faculty, public Student  {
public:
   TA(int x):Student(x), Faculty(x)   {
      cout<<"TA::TA(int ) called"<< endl;
   }
};

int main()  {
   TA ta1(30);
}
```

In the above program, constructor of 'Person' is called once. One important thing to note in the above output is, the default constructor of 'Person' is called. When we use 'virtual' keyword, the default constructor of grandparent class is called by default even if the parent classes explicitly call parameterized constructor.

How to call the parameterized constructor of the 'Person' class? The constructor has to be called in 'TA' class. For example, see the following program.

```
#include<iostream>
using namespace std;
class Person {
public:
   Person(int x)  { cout << "Person::Person(int ) called" << endl;   }
   Person()    { cout << "Person::Person() called" << endl;   }
};

class Faculty : virtual public Person {
public:
   Faculty(int x):Person(x)   {
     cout<<"Faculty::Faculty(int ) called"<< endl;
   }
};

class Student : virtual public Person {
public:
   Student(int x):Person(x) {
      cout<<"Student::Student(int ) called"<< endl;
   }
};

class TA : public Faculty, public Student  {
public:
   TA(int x):Student(x), Faculty(x), Person(x)   {
      cout<<"TA::TA(int ) called"<< endl;
   }
};

int main()  {
   TA ta1(30);
}
```

In the above program, constructor of 'Person' is called two times. Destructor of 'Person' will also be called two times when object 'ta1' is destructed. So object 'ta1' has two copies of all members of 'Person', this causes ambiguities. The solution to this problem is 'virtual' keyword. We make the classes

'Faculty' and 'Student' as virtual base classes to avoid two copies of 'Person' in 'TA' class. For example, consider the following program.

Virtual Function

Virtual Function is a function in base class, which is overrided in the derived class, and which tells the compiler to perform Late Binding on this function.

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

class Shape
{
        public:
         virtual void Draw()
                { cout<<"Shape
                drawn!"<<endl;
        }
};

class Square : public Shape
{
        public: void
        Draw()
                { cout<<"Square
                drawn!"<<endl;
        }
};

int main()
{
        Squareobj;    //Derived ClassObject

        Shape* shape = &obj; /* derived class object being referenced by a pointer
                                to the base class */

        shape->Draw(); /* outputs "Square drawn!" if Draw() is virtual in base class
                          else outputs "Shape drawn!"*/
        return 0;
}
```

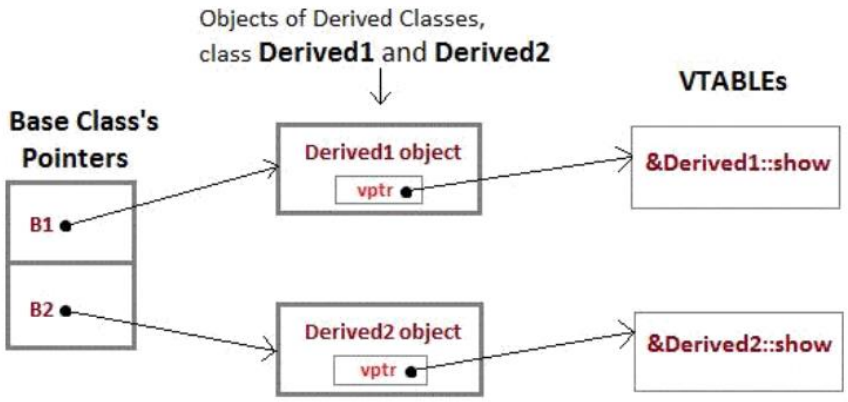
INTERESTING FACTS

We can call private function of derived class from the base class pointer with the help of virtual keyword. Compiler checks for access specifier only at compile time. So at run time when late binding occurs it does not check whether we are calling the private function or public function.

MECHANISM OF LATE BINDING

To accomplish late binding, Compiler creates VTABLEs, for each class with virtual function. The address of virtual functions is inserted into these tables. Whenever an object of such class is created the compiler secretly inserts a pointer called vpointer, pointing to VTABLE for that object. Hence when function is called, compiler is able to resolve the call by binding the correct function using the vpointer.

- Only the Base class Method's declaration needs the Virtual Keyword, not the definition.
- If a function is declared as virtual in the base class, it will be virtual in all its derived classes.
- The address of the virtual Function is placed in the VTABLE and the compiler uses VPTR (vpointer) to point to the Virtual Function.



vptr, is the vpointer, which points to the Virtual Function for that object.

VTABLE, is the table containing address of Virtual Functions of each class.

Overriding Methods – virtual keyword

- Methods in the parent class can be redefined in the child class
- In C++, static binding is the defaults behaviour
- The keyword virtual allows the use of dynamic binding.

Virtual Destructor

A virtual destructor is used to free up the memory space allocated by the derived class object or instance while deleting instances of the derived class using a base class pointer object. A base or parent class destructor use the virtual keyword that ensures both base class and the derived class destructor will be called at run time, but it called the derived class first and then base class to release the space occupied by both destructors.

Why we use virtual destructor in C++?

When an object in the class goes out of scope or the execution of the main() function is about to end, a destructor is automatically called into the program to free up the space occupied by the class' destructor function. When a pointer object of the base class is deleted that points to the derived class, only the parent class destructor is called due to the early bind by the compiler. In this way, it skips calling the derived class' destructor, which leads to memory leaks issue in the program. And when we use virtual keyword preceded by the destructor tilde (~) sign inside the base class, it guarantees that first the derived class' destructor is called. Then the base class' destructor is called to release the space occupied by both destructors in the inheritance class.

Syntax for Virtual Destructor

```
#include<iostream>
using namespace std;
class Base
{
   public:
   Base() // Constructor member function.
{
   cout << "\n Constructor Base class";  // It prints first.
}
 virtual ~Base() // Define the virtual destructor function to call the Destructor Derived function.
{
   cout << "\n Destructor Base class";  /
}
};
// Inheritance concept
class Derived: public Base
{
   public:
   Derived() // Constructor function.
{
   cout << "\n Constructor Derived class" ; /* After print the Constructor Base, now it will prints. */
}
 ~Derived() // Destructor function
{
   cout << "\n Destructor Derived class"; /* The virtual Base Class? Destructor calls it before calling the Base Class
 Destructor. */
}
};
int main()
{
   Base *bptr = new Derived; // A pointer object reference the Base class.
   delete bptr; // Delete the pointer object.
}
```

Pure Virtual Function:

A pure virtual function is a virtual function in C++ for which we need not to write any function definition and only we have to declare it. It is declared by assigning 0 in the declaration.

- An abstract class is a class in C++ which have at least one pure virtual function.
- Abstract class can have normal functions and variables along with a pure virtual function.
- Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.
- Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.
- If an Abstract Class has derived class, they must implement all pure virtual functions, or else they will become Abstract too.
- We can't create object of abstract class as we reserve a slot for a pure virtual function in Vtable, but we don't put any address, so Vtable will remain incomplete.

```
#include<iostream>
using namespace std;
class B {
  public:
    virtual void s() = 0; // Pure Virtual Function
};

class D:public B {
  public:
    void s() {
      cout << "Virtual Function in Derived class\n";
    }
};

int main() {
  B *b;
  D dobj;
  b = &dobj;
  b->s();
}
```

Abstract Class

An abstract base class is a class that includes or inherits at least one pure virtual function that has not been defined.

The C++ interfaces are implemented using abstract classes and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data.

A class is made abstract by declaring at least one of its functions as pure virtual function. A pure virtual function is specified by placing "= 0" in its declaration as follows

The purpose of an abstract class (often referred to as an ABC) is to provide an appropriate base class from which other classes can inherit. Abstract classes cannot be used to instantiate objects and serves only as an interface. Attempting to instantiate an object of an abstract class causes a compilation error.

Thus, if a subclass of an ABC needs to be instantiated, it has to implement each of the virtual functions, which means that it supports the interface declared by the ABC. Failure to override a pure virtual function in a derived class, then attempting to instantiate objects of that class, is a compilation error.

Classes that can be used to instantiate objects are called concrete classes.

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

// Base class
class Shape {
  public:
    // pure virtual function providing interface framework.
    virtual int getArea() = 0;
    void setWidth(int w) {
      width = w;
    }
    void setHeight(int h) {
      height = h;
    }
  protected:
    int width;
    int height;
};
// Derived classes
class Rectangle: public Shape {
  public:
    int getArea() {
      return (width * height);
    }
};
class Triangle: public Shape {
  public:
    int getArea() {
      return (width * height)/2;
    }
};

int main(void) {
  Rectangle Rect;
  Triangle  Tri;

  Rect.setWidth(5);
  Rect.setHeight(7);

  // Print the area of the object.
  cout << "Total Rectangle area: " << Rect.getArea() << endl;

  Tri.setWidth(5);
  Tri.setHeight(7);

  // Print the area of the object.
  cout << "Total Triangle area: " << Tri.getArea() << endl;

  return 0;
}
```

CASTING

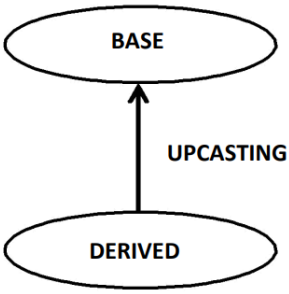
A cast is a special operator that forces one type to be converted into another.

UPCASTING

Upcasting is a process of treating a pointer or a reference of derived class object as a base class pointer.

- A base class pointer can only access the public interface of the base class.
- The additional members defined in the derived class are therefore inaccessible.
- Upcasting is not needed manually.

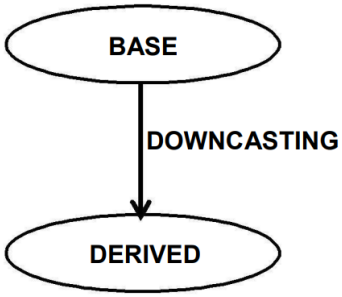
We just need to assign derived class pointer (or reference) to base class pointer.



DOWNCASTING

The opposite of Upcasting is Downcasting, It converts base class pointer to derived class pointer.

- Type conversions that involve walking down the tree, or downcasts, can only be performed explicitly by means of a cast construction. The cast operator (type) or the static_cast<>operator are available for this task, and are equivalent in thiscase.
- After downcasting a pointer or a reference, the entire public interface of the derived class isaccessible.



STATIC_CAST

Syntax: static_cast(expression)

The operator static_cast<> converts the expression to the target type type.

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

```
public:
Employee(string fName, string lName, double sal)
{
FirstName = fName;
LastName = lName;
salary = sal;
}
string FirstName;
string LastName;
double salary; void
show()
{
cout<< "First Name: " <<FirstName<< " Last Name: " <<LastName<< " Salary: " << salary<<endl;
}
void addBonus(double bonus)
{
salary += bonus;
} };
class Manager :public Employee
{
public:
Manager(string fName, string lName, double sal, double comm) :Employee(fName, lName, sal)
{
Commision = comm;
}
double Commision; double
getComm()
{
return Commision; }
 };
```

FOR UPCASTING

```
int main()
{
Employee* emp; //pointer to base class object
Manager m1("Ali", "Khan", 5000, 0.2); //object of derived class
emp = &m1; //implicit upcasting
emp->show(); //okay because show() is a base class function
return 0;
}
```

FOR DOWNCASTING USING (type)

```
int main()
{
Employee e1("Ali", "Khan", 5000); //object of base class
//try to cast an employee to Manager
Manager* m3 = (Manager*)(&e1); //explicit downcasting
cout<< m3->getComm() <<endl;
return 0;
}
```

FOR DOWNCASTING USING (static_cast)

```
int main()
{
Employee e1("Ali", "Khan", 5000); //object of base class
//try to cast an employee to Manager
Manager* m3 = static_cast<Manager*>(&e1); //explicit downcasting
cout<< m3->getComm() <<endl;
return 0;
}
```

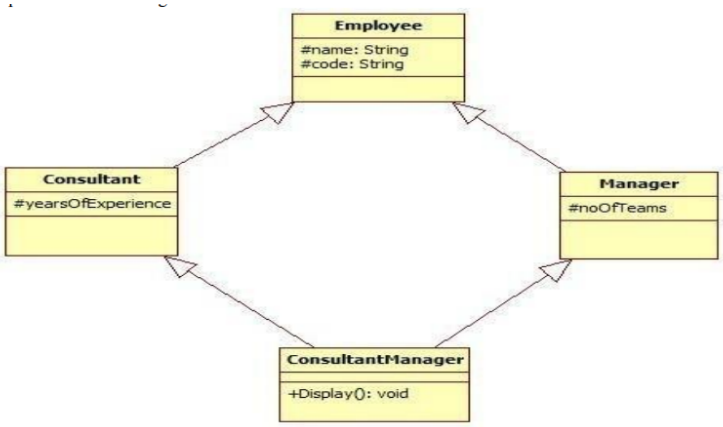
DOWNCASTING IS UNSAFE

- Since, e1 object is not an object of Manager class so, it does not contain any information about commission.
- That's why such an operation can produce unexpected results.
- Downcasting is only safe when the object referenced by the base class pointer really is a derived class type.
- To allow safe downcasting C++ introduces the concept of dynamic casting.

EXERCISE

Question 1

Implement the following scenario in C++:



1. No accessors and mutators are allowed to be used.
2. The Display() function in “ConsultantManager” should be capable of displaying the values of all the data members declared in the scenario (name,code,yearsOfExperience,noOfTeams) without being able to alter the values.
3. The “int main()” function should contain only three program statements which are as follows:
    a) In the first statement, create object of “ConsultantManager” and pass the values for all the data members:
        ConsultantManagerobj("Ali",”S-123”,17,5);
    b) In the second statement, call the Display()function.
    c) In the third statement, return0.
4. Perform upcasting in main.
5. Perform downcasting in main.

All the values are required to be set through constructors parameter.

Question 2

Design and implement a program that shows the relationship between person, student and professor. Your person class must contain two pure virtual functions named getData() of type void and isOutstanding() of type bool and as well getName() and putName() that will read and print the person name. Class student must consist of function name getData (), which reads the GPA of specific person and isOutstanding() function which returns true if the person GPA is greater than 3.5 else should return false. Class professor should take the respective persons publications in getData() and will return true in Outstanding() if publications are greater than 100 else will return false . Your main function should ask the user either you want to insert the data in professor or student until and unless user so no to add more data. Clearly mention using comments if there is any abstract class.

Question 3

Implement your own scenario of Diamond Problem. Write the code to solve the problem also.