

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

CL 103 - COMPUTER PROGRAMMING LAB

Instructors: Mr. Basit Ali, Ms. Mahrukh Khan, Ms. Ammara Yaseen, Ms. Tooba Ali, Ms. Maham Mobin, Mr. Muhammad Irfan Ayub

Email: basit.jasani@nu.edu.pk, mahrukh.khan@nu.edu.pk, ammara.yaseen@nu.edu.pk, tooba.ali@nu.edu.pk, maham.mobin@nu.edu.pk, muhammad.irfan@nu.edu.pk

Lab # 09

Outline

- Friend Function
 - Casting
 - Runtime Polymorphism
 - Pure Virtual Functions
 - Abstract Class
 - Examples
 - Exercise
-

Friend Function

Friend Functions:

The functions which are not member functions of the class yet they can access all private members of the class are called friend functions.

Why they are needed?

They are needed in situations where we have written code for some function in one class and it need to be used by other classes as well for example, suppose we wrote the code to compute a complex mathematical formula in one class but later it was required by other classes as well, in that case we will make that function friend of all other classes.

Are friend functions against the concept of Object Oriented Programming?

It can be said that friend functions are against the principle of object oriented programming because they violate the principle of encapsulation which clearly says that each object methods and functions should be encapsulated in it. But there we are making our private member accessible to other outside functions.

Consider the following

class: class X

```
{
private:
int a, b;
public:
void MemberFunction ();
...
}
```

Suppose we have a global function DoSomething that need to access the private members of class X, when we will try to access them compiler will generate error as outside world can not access private members of a class except its member functions.

void DoSomething(X obj)

```
{
    obj.a = 3; //Error
    obj.b = 4; //Error
}
```

In order to access the member variables of the class, we must make function friend of that class, **class X**

```
{
private:
    int a, b;
public:
...
friend void DoSomething(X obj);
}
```

Now the function DoSomething can access data members of **class X**

void DoSomething(X obj)

```
{
    obj.a = 3;
    obj.b = 4;
}
```

Prototypes of friend functions appear in the class definition. But friend functions are NOT member functions.

Friend functions can be placed anywhere in the class without any effect Access specifiers don't affect friend functions or Classes.

```

class X{
    ...
private:
friend void DoSomething(X);
public:
friend void DoAnything(X);
    ...
};

```

While the definition of the friend function is:

```

void DoSomething(X obj)
{
    obj.a = 3; // No Error obj.b =
    4; // No Error
    ...
}

```

friend keyword is not given in definition.

Friend Classes:

Similarly, one class can also be made friend of another class: class X

```

{
friend class Y;
    ...
};

```

Member functions of class Y can access private data members of class X **class X**

```

{
    friend class Y;
private:
    int x_var1, x_var2;
    ...
}; class Y
{
private:
    int y_var1, y_var2; X objX;
public: void setX()
{
    objX.x_var1 = 1;
}
};

```

```

int main()
{
    Y objY; objY.setX();
return 0;
}

```

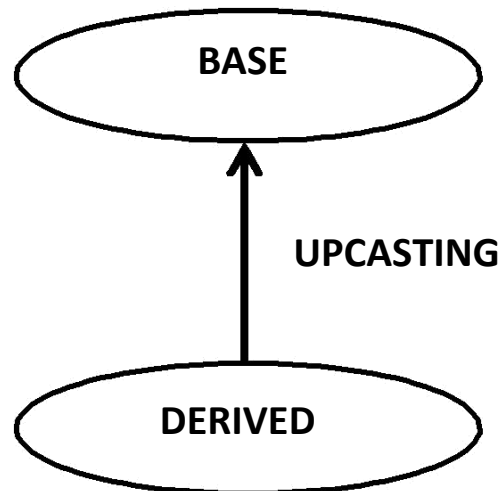
CASTING

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.

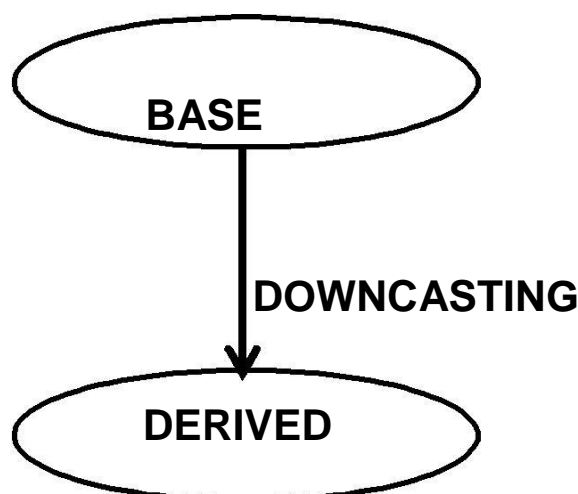
- A base class pointer can only access the public interface of the base class.
- The additional members defined in the derived class are therefore inaccessible.
- Upcasting is not needed manually. We just need to assign derived class pointer (or reference) to base class pointer.



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 this case.
- After downcasting a pointer or a reference, the entire public interface of the derived class is accessible.



STATIC CAST

Syntax: static_cast<type>(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*.

VIRTUAL FUNCTION

Virtual Function is a function in base class, which is overridden 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;
}
```

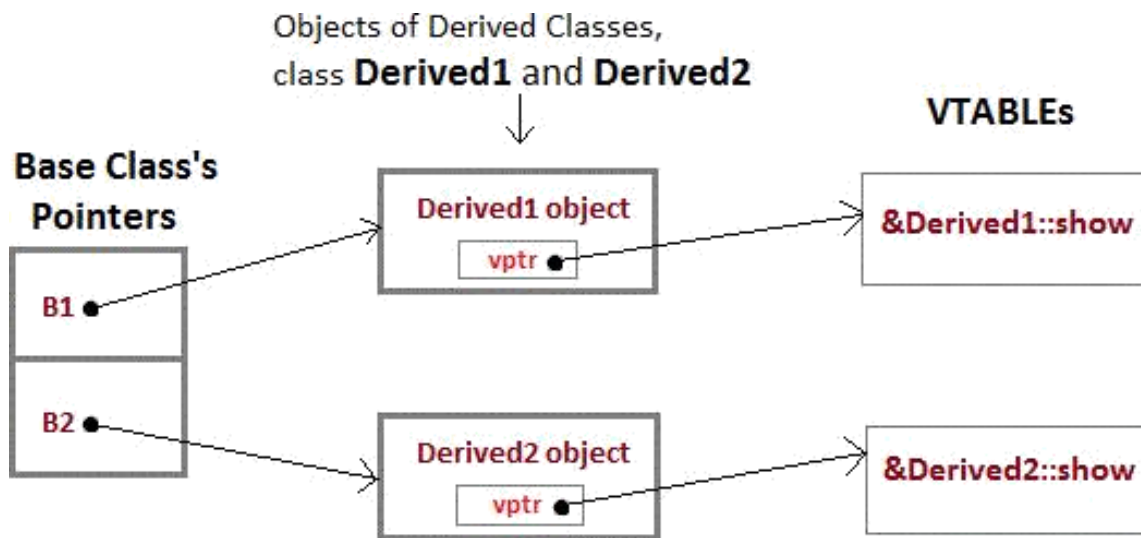
INTERESTING FACTS

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 VirtualFunction.



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 default behaviour
- The keyword `virtual` allows the use of dynamic binding.

//Header.h

```
//Polygon.h
#ifndef __POLYGON_H_INCLUDED__ // if polygon.h hasn't been included yet...
#define __POLYGON_H_INCLUDED__ // #define this so the compiler knows it has been included
class Polygon {
protected:
    int width, height;
public:
    void set_values (int a, int b);
    virtual int area ();
};

#endif

//Rectangle.h
#include "Polygon.h"
class Rectangle: public
Polygon { public:
    int area();
};

//Triangle.h
#include "Polygon.h"
class Triangle: public
Polygon { public:
    int area ();
};

//Impl
//Polygon.cpp
#include "Polygon.h"
```



```

void Polygon::set_values (int a, int b)
{
    width=a;
    height=b;
}
int Polygon::area()
{
    return 0;
}

```

```

//Rectangle.cpp
#include "Rectangle.h"

```

```

int Rectangle::area()
{
    return width*height;
}

```

```

//Triangle.cpp
#include "Triangle.h"

```

```

int Triangle::area()
{
    return (width * height / 2);
}

```

//Main.cpp

```

#include <iostream>
#include "Rectangle.h"
#include "Triangle.h"
using namespace std;

```

```

int main () {
    Rectangle rect;
    Triangle trgl;
    Polygon poly;
    Polygon * ppoly1 = &rect;
    Polygon * ppoly2 = &trgl;
    Polygon * ppoly3 = &poly;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    ppoly3->set_values (4,5);
    cout<< ppoly1->area() << '\n';
    cout<< ppoly2->area() << '\n';
    cout<< ppoly3->area() << '\n';

    return 0;
}

```

Pure Virtual Functions

virtual Type identifier(parameters) = 0;

// Pure Virtual Functions

virtual void price() = 0;

virtual void type() = 0;

Abstract Base Class

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

```
#include <string>
#include<iostream>
using namespace std;
//Header files
//Animal.h
class Animal
{
protected:
    string m_strName;

public:
    Animal(string strName); string GetName();
    virtual string Speak() = 0; // pure virtual function
};
//Cow.h
class Cow: public Animal
{
public:
    Cow(string strName);
    virtual string Speak();

};
//Impl
Cow:: Cow(string strName): Animal(strName)
{
}
string Cow::Speak() { return "Meow";
}
Animal::Animal(string strName): m_strName(strName)
{
}
string Animal::GetName()
{
    return m_strName;
}
int main()
{
    Cow cCow("Betsy");
    Cout<<cCow.GetName()<<"says"<<cCow.Speak();
}
```

QUESTION#1

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.

QUESTION#2

A company pays its employees weekly. The employees are of four types: **Salaried employees** are paid a fixed weekly salary regardless of the number of hours worked, **hourly employees** are paid by the hour and receive overtime pay for all hours worked in excess of 40 hours, **commission employees** are paid a percentage of their sales and **base-salary-plus-commission employees** receive a base salary plus a percentage of their sales. For the current pay period, the company has decided to reward base-salary-plus-commission employees by adding 10 percent to their base salaries. The company wants you to draw UML diagram of the given scenario and do implementation on c++ that performs its payroll calculations polymorphically.

Your Employee class must have:

- First name, last name and social security number as private data members. Use accessor and mutators to set and get these values.
- Constructor with first name, last name and ssn number as parameter.
- Pure virtual function named `earning` with return type `double`.
- Virtual function named `print` with return type `void`, which employee first name, last name and ssn number.

Your salaried Employee must have:

- Earning method which return the salary
- Print method that print the employee detail and employee salary.

Your hourly Employee must have:

- You must take the wage and hours as a parameter using base class initializer.
- Earning methods which returns the salary the employed has worked.
- Print method that print the employee detail and employee salary.

Your commission employee must have:

- You must take the commission rate and gross sale rate as a parameter using base class initializer.
- Earning methods which returns the commission of the employee.
- Print method that print the employee detail and employee commission.

Your base-salary plus-commission employees must have:

- You must take the base salary as a parameter using base class initializer.
- Earning methods which return the base commission of the employee.
- Print method that print the employee detail and employee base salary.

You have to perform upcasting and downcasting, and print the details of each employee type.