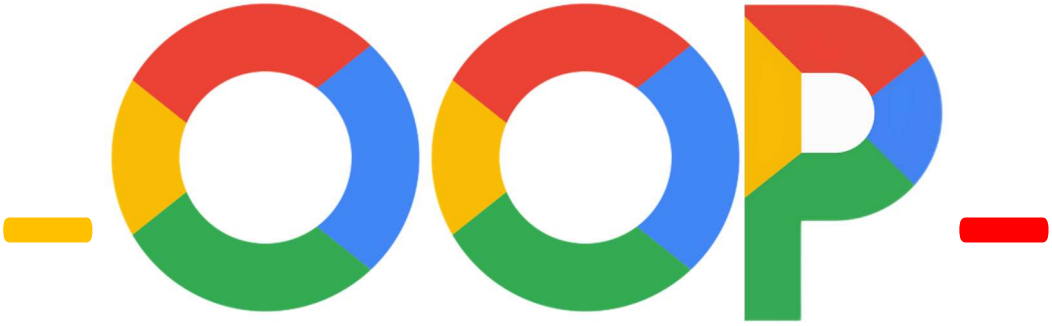

Course: CS1004 Object Oriented Programming.

Instructor: Lecturer Sara Rehmat.

Submitted by: Muhammad Rehan

Roll no: 22P-9106

Class: BSE-2A

Date: April 20th 2023.

Assignment no 02

Department of Computer Science

Question no 02:

A. Code:

main.cpp:

```
#include <iostream>
#include "AType.h"
#include "BType.h"
#include "DType.h"

int main() {
//    AType a;
/* Can't make object of A class as
it's Abstract class. */
    BType b(7);
    DType d;
    d.print();
    b.print();
    AType *a1 = new CType;
    a1->print();
}
```

AType.h:

```
class AType {
protected:
    int y;
    int x;
public:
    AType();
    virtual void print() const = 0;
};
```

AType.cpp:

```
#include"AType.h"
AType::AType():x( {
    x = 0;
    y = 0;  }
```

BType.h:

```
class BType : virtual protected
AType {
protected:
    int z;
public:
    explicit BType(int a = 0);
    void print() const override; };
```

BType.cpp:

```
#include <iostream>
#include "BType.h"

BType::BType(int a) {
    z = a;  }
void BType::print() const {
    std::cout << y << " " << z <<
std::endl;

 }
```

CType.h:

```
#include "AType.h"
class CType : virtual public AType
{
private:
    int w{};
public:
    void print() const override; };
```

CType.cpp:

```
#include <iostream>
#include "CType.h"

void CType::print() const {
    std::cout << y << " " << x << "
" << w << std::endl;

}
```

DType.h:	DType.cpp:

`#include "CType.h"`
`#include "BType.h"`
 	
`class DType : public BType, public CType {`
 	
`public:`
`  void print() const override;`
`};`
	`#include "DType.h"`
	
	`void DType::print() const {`
	`  BType::print();`
	`}`

B. Output:

0 0

0 7

0 0 0

Question no 03:

main.cpp

```
#include <iostream>
#include "DeckOfCards.h"

int main() {
    // Creating our programme starts with creating an object of type
DeckOfCard (a deck of 52 cards).
    DeckOfCards deck;

    // Shuffling the cards.
    deck.shuffle();

    // A loop to deal all cards to user.
    while (deck.moreCards()) {
        std::unique_ptr<Card> card = deck.dealCard();
        std::cout << card->toString() << std::endl;
    }
    return 0;   }
```

Card.h

```
#include <string>

class Card {

    // Two integers for face and suit.
    int face;
    int suit;
    // Two arrays to store the faces and suits strings.
    static const std::string faces[];
    static const std::string suits[];

public:

    /*
     * Constructor to initialize a Card object with the given face and
suit.
     * Face and suit default to 0. 'explicit' keyword is used to avoid
implicit conversions.
     * */
    explicit Card(int face = 0 , int suit =0 );

    void setFace(int face);

    void setSuit(int suit);

    std::string toString ();
};
```

Card.cpp

```
#include "Card.h"

const std::string Card::faces[] = {"Ace", "2", "3", "4", "5", "6", "7",
"8", "9", "10", "Jack", "Queen", "King"};
const std::string Card::suits[] = {"Spades", "Hearts", "Diamonds",
"Clubs"};


Card::Card(int face, int suit) {
    setFace(face);
    setSuit(suit);

}

void Card::setFace(int face) {
    Card::face = face;
```

```
}

void Card::setSuit(int suit) {
    Card::suit = suit;
}

std::string Card::toString() {

    return faces[face]+" of "+ suits[suit];

}
```

DeckOfCard.h

```
#define DECKSIZE 52
#include <memory>
#include <vector>
#include "Card.h"


class DeckOfCards {
    // A unique pointer to the vector of cards
    std::vector<std::unique_ptr<Card>> deck;
    // A flag integer to keep track of the count, initialized with zero.
    int nextCard{0};

public:

    void shuffle ();
    std::unique_ptr<Card> dealCard();

    // nodiscard so compiler does not discard the return value.
    [[nodiscard]] bool  moreCards() const;

    DeckOfCards();
};
```

DeckOfCard.cpp

```
#include <bits/stdc++.h>
#include "DeckOfCards.h"
#include "Card.h"

DeckOfCards::DeckOfCards() {

    // Initializing our vector of decks
    for (int i = 0; i < DECKSIZE; i++) {
```

```
            short int face = i % 13;
            short int suit = i / 13;
            // pushing back the objects of card till DECK-SIZE of 52.
            deck.push_back(std::make_unique<Card>(face, suit));
        }
}

void DeckOfCards::shuffle() {
    // Shuffling the cards.
    std::srand(time(nullptr));
    for (int i = 0; i < deck.size(); i++) {
        int j = std::rand() % deck.size();
        // Using the function swap, instead of a temp object.
        swap(deck[i], deck[j]);
    }
}

std::unique_ptr<Card> DeckOfCards::dealCard() {
    if (moreCards())
        /*
         * If there are more cards, it moves the unique pointer to
the next card in the deck,
         * using std::move() to transfer ownership to the calling
code.
         * */
        return std::move(deck[nextCard++]);
        //else it returns the nullptr
    return nullptr;
}

bool DeckOfCards::moreCards() const {
    // if the next card is less than 52, till will return true else
false.
    return nextCard < 52;
}
```

Question no 01:

a. When an object is passed to a function by value and returned by value, the copy is made using the copy constructor in both conditions. When an object is initialized with another object, the

copy constructor is called to create a copy of the original object again.

b. In general, an object of one class cannot be assigned to an object of another class in C++ as it owns specific data members and member functions. However, there are certain situations where an object of one class can be assigned to an object of another class. One example is when the two classes have a common parent class, and the assignment is made through a reference or pointer to the parent class. This is known as polymorphism and allows objects of different classes to be treated as if they are of the same class.

c. Including **#ifndef** (or "header guards") in a program prevents multiple definitions of the same header file when the file is included in various source files. The #ifndef statement checks if a macro has already been defined and, if so, skips the code that follows. This ensures that the header file is included only once in a given compilation unit, preventing errors such as the redefinition of classes or functions and reducing compile times.

d. Method overloading is where multiple methods with the same name but different parameters can be defined within a class. This allows for different versions of a method to be used based on the arguments passed to it.

Method overriding, on the other hand, is a feature in inheritance where a derived class provides its own implementation of a method that is already defined in the base class. This allows for a more specialized behavior of the method in the derived class.

In summary, method overloading is about creating multiple versions of the same method within a class, while method overriding is about modifying the behavior of a method already defined in a base class within a derived class.

e. In inheritance, the constructors are executed in the order of inheritance, starting with the base class constructor and followed by the derived class constructor. On the other hand, the destructors are executed in the reverse order of inheritance, starting with the derived class destructor and then the base class destructor. This ensures that the resources the classes allocate are managed and released correctly.