OPP Related Concepts

Encapsulation

Encapsulation in Object-Oriented Programming (OOP) is a concept that binds together the data and functions that manipulate the data, and keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of data hiding.

Data encapsulation is a mechanism where the details of the class are hidden from the user. The user can perform a restricted set of operations on the hidden members of the class by executing special functions called methods.

In the provided C++ code, the class `Person` is an example of encapsulation. The data members `name`, `ID`, and `age` are encapsulated within the `Person` class.

Abstraction

Abstraction in Object-Oriented Programming (OOP) is a process of hiding the implementation details from the user, only the functionality will be provided to the user. In other words, the user will have the information on what the object does instead of how it does it.

In the provided C++ code, the class `Person` is an example of abstraction. The user doesn't need to know how the `displayInfo()` method works internally. They only need to know that calling this method on a `Person` object will display the person's information. The internal workings of the `displayInfo()` method (i.e., how it accesses the `name`, `ID`, and `age` data members and prints them) are abstracted away from the user.

Data Hiding

Data hiding in Object-Oriented Programming (OOP) is the practice of making the data members private (or protected) so they can't be accessed directly from outside the class. Only the class's own methods can directly access and modify these data members.

In the provided C++ code, the class `Person` is an example of data hiding. The data members `name`, `ID`, and `age` are not accessible directly from outside the class, as shown by the commented-out line `//Safia.name = "Safia";` which would result in an error if uncommented.

Instead, the `displayInfo()` method is provided to interact with these data members. This method can be called on an instance of the `Person` class, like `Safia.displayInfo();` or `Obj->displayInfo();`, to display the information of the `Person` instance. This is a way of controlling access to the hidden data members.
General Topics

```
#include <iostream>
#include <string>
using namespace std;
class Person
{
        //attributes
        //Private data unaccessible outside class
        string name;
        int age;
        string ID;
        //Methods
        //Public Data Accessible outside the class
        public:
                //Default Constructor
                Person(){
                }
                void displayInfo(void)
                {
                        std::cout << "player name is " << name << std::endl;
                        std::cout << "ID is " << ID << std::endl;
                        std::cout << "Age: " << age << std::endl;
                }
};
int main()
{
        Person Safia;
        //Acess Data
        //Dot operation for accessing the member
        //Safia.name = "Safia"; //Error name is private accesifier
        Safia.displayInfo();
        //Pointer to Object of this class
        Person* Obj = new Person;
        //Arrow Operator to access the member
        Obj->displayInfo();
}
```

Implementing function outside class

```
class Person
{
        //attributes
        //Private data unaccessible outside class
        string name;
        int age;
        string ID;
        //Methods
        //Public Data Accessible outside the class
        public:
                //Default Constructor
                Person(){
                }
                void displayInfo(void);
};

void Person::displayInfo(void)
{
        std::cout << "player name is " << name << std::endl;
        std::cout << "ID is " << ID << std::endl;
        std::cout << "Age: " << age << std::endl;
}
```

The provided code snippet is a constructor implementation for the `Player` class. Constructors are special member functions that are called when an object of a class is created. They are responsible for initializing the object's data members.

In this case, the constructor is defined without any parameters, which means it is a default constructor. It is used to create a `Player` object without providing any initial values for its data members.

- The constructor is named `Player`, which matches the name of the class.
- It has an empty parameter list, indicating that it doesn't take any arguments.
- The colon (`:`) after the constructor declaration is used to initialize the base class or member variables.
- Inside the colon initializer list, we see `Player("none", 0)`. This is a constructor delegation, which means it calls another constructor of the `Player` class to perform the initialization.
- The constructor being called here is `Player(std::string source1, int source2)`, which takes a string and an integer as arguments.
- The arguments passed to the delegated constructor are `"none"` and `0`, respectively.
- After the constructor delegation, we have a `std::cout` statement that prints `"Player::Player()"` to the console. This is just a debug message to indicate that

the constructor has been called.

By using constructor delegation, the default constructor is able to reuse the initialization logic from another constructor. In this case, it delegates to the constructor that takes a string and an integer, passing default values of `"none"` and `0`.

```
Player():Player("none",0){
        std::cout << "Player::Player()" << std::endl;
}
```

This is the implementation of the `Player` class constructor with an `int` parameter. This constructor is called when a `Player` object is created with an `int` argument.

The constructor uses member initializer syntax to delegate the construction to another constructor of the `Player` class. In this case, it delegates to the constructor that takes a `std::string` and an `int` as parameters.

By using delegation, the constructor with the `int` parameter avoids duplicating code and ensures that the common initialization logic is executed. This promotes code reuse and helps maintain consistency within the class.

After delegating to the appropriate constructor, the constructor with the `int` parameter prints a message to the console, indicating that it has been called.

Overall, this constructor provides a way to create a `Player` object with an `int` value, while still ensuring that the necessary initialization steps are performed.

```
Player(int source) : Player("none",source){
        std::cout << "player::player(int)" << std::endl;
}
```

In this case, the constructor has a single parameter of type `std::string` named `source`.

1. The constructor is defined with the name `Player` and a single parameter of type `std::string` named `source`.
2. The colon (`:`) after the parameter list is used to initialize the base class of the `Player` class. In this case, it is calling another constructor of the `Player` class with two arguments: `source` and `0`. This is known as constructor delegation or constructor chaining.
3. Inside the constructor body, a message is printed to the console using `std::cout`. This line is executed when the constructor is called.

The purpose of this constructor is to initialize a `Player` object with a given `source` string and a default value of `0` for the `health` attribute. It also prints a message to indicate that this specific constructor has been called.

It's worth noting that there are other constructor implementations provided in the code snippet as well. These constructors allow creating `Player` objects with different combinations of parameters, such as no parameters, an integer parameter, or a string and an integer parameter. This provides flexibility when creating `Player` objects based on different scenarios or requirements.

```
Player (std::string source) : Player(source,0){
        std::cout << "Player::player(string)" << std::endl;
}
```

This constructor takes a `std::string` parameter called `source1` and an `int` parameter called `source2`. It directly initializes the `name` member with the value of the `source1` parameter and the `health` member with the value of the `source2` parameter. It does not call any other constructor.

```
 Player(std::string source1,int source2) : name{source1},health{source2}{
        std::cout << "player::player(string,int)" << std::endl;
}
```

The provided code is a destructor for a class named `Player` in C++. A destructor is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the `delete` expression is applied to a pointer to the object of that class.

The destructor has the same name as the class, preceded by a tilde (`~`). It doesn't return a value or take any parameters. Destructors can't be inherited or overloaded.

```
~Player(){
std::cout << "Player::~Player()" << std::endl;
}
```

Whole Code

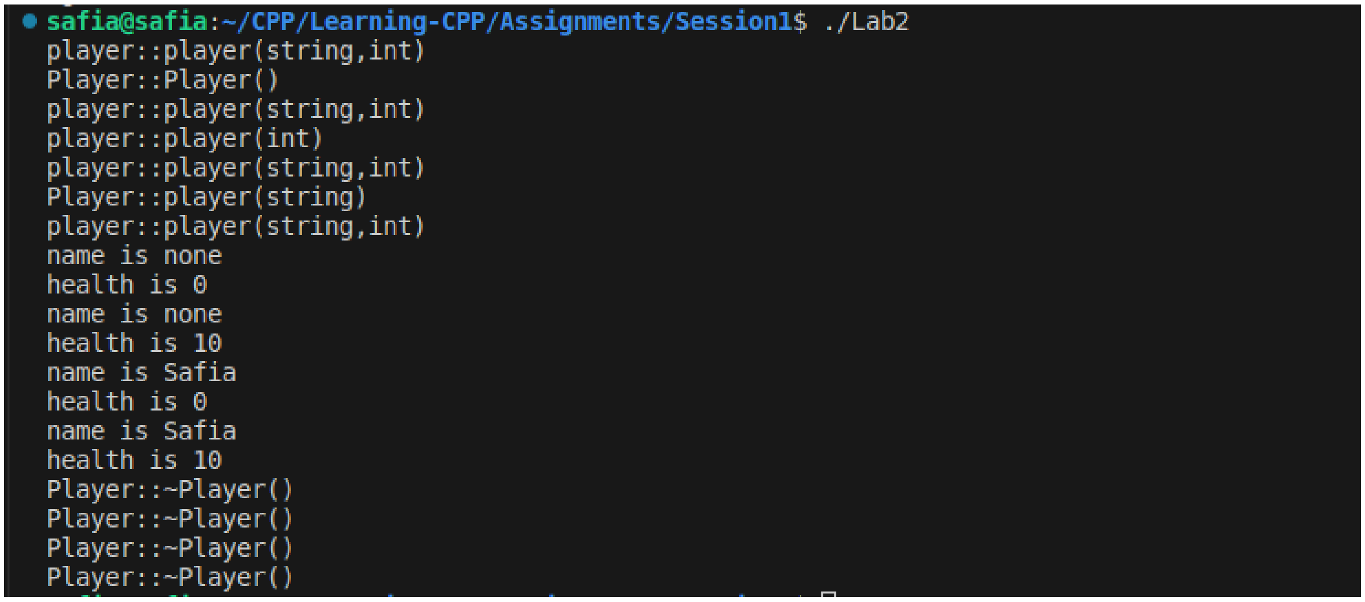
```
#include <iostream>
#include <string>
```

```cpp
using namespace std;

class Person
{
    // private members
    std::string name;
    std::string ID;
    int *ptr;
    // public members
    public:
        // Default Constructor
        Person() : name{"None"},ID {"None"}{
        std::cout << "Constructor has been called" << std::endl;
    }
    // parameterized constructor
    Person (std::string source, std::string source2) :
name{source},ID{source2}{
        std::cout << "Person::Person(string,string)" << std::endl;
    }
    // Destructor
    ~Person(){
        std::cout <<  "Person::~Person()"   << std::endl;
    }
    void displayInfo(void);

};
void Person::displayInfo(void)
{
    std::cout << "name is " << name << std::endl;
    std::cout << "ID is " << ID << std::endl;
}
class Player {
    // Class members and methods go here
    std::string name;
    int health;
    double jh;
    public:
        Player():Player("none",0){
            std::cout << "Player::Player()" << std::endl;
        }
        Player(int source) : Player("none",source){
            std::cout << "player::player(int)" << std::endl;
        }
        Player (std::string source) : Player(source,0){
            std::cout << "Player::player(string)" << std::endl;
        }
        Player(std::string source1,int source2) :
name{source1},health{source2}{
            std::cout << "player::player(string,int)" << std::endl;
        }
        ~Player(){
            std::cout << "Player::~Player()" << std::endl;
        }
        void displayInfo(void)
        {
            std::cout << "name is " << name << std::endl;
            std::cout << "health is " << health << std::endl;
        }
};
int main()
{
    Player Obj;
    Player Obj1(10);
    Player Obj2("Safia");
    Player Obj3("Safia",10);
    Obj.displayInfo();
    Obj1.displayInfo();
    Obj2.displayInfo();
    Obj3.displayInfo();
}
```

Output



```
safia@safia:~/CPP/Learning-CPP/Assignments/Session1$ ./Lab2
player::player(string,int)
Player::Player()
player::player(string,int)
player::player(int)
player::player(string,int)
Player::player(string)
player::player(string,int)
name is none
health is 0
name is none
health is 10
name is Safia
health is 0
name is Safia
health is 10
Player::~Player()
Player::~Player()
Player::~Player()
Player::~Player()
```