C++ Access Modifiers

The access modifiers of C++ are public, private, and protected.

One of the main features of object-oriented programming languages such as C++ is **data hiding**.
Data hiding refers to restricting access to data members of a class. This is to prevent other functions and classes from tampering with the class data.

However, it is also important to make some member functions and member data accessible so that the hidden data can be manipulated indirectly.

The access modifiers of C++ allows us to determine which class members are accessible to other classes and functions, and which are not.

For example,

```
class Patient {

    private:
        int patientNumber;
        string diagnosis;

    public:

      void billing() {
          // code
      }

      void makeAppointment() {
          // code
      }
};
```

Here, the variables `patientNumber` and `diagnosis` of the `Patient` class are hidden using the `private` keyword, while the member functions are made accessible using the `public` keyword.

1.1 Types of C++ Access Modifiers

In C++, there are 3 access modifiers:

- `public`
- `private`
- `protected`

1.2 public Access Modifier

- The `public` keyword is used to create public members (data and functions).
- The public members are accessible from any part of the program.

1.2.1 Example 1: C++ public Access Modifier

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

// define a class
class Sample {
```

```
    // public elements
   public:
    int age;

    void displayAge() {
        cout << "Age = " << age << endl;
    }
};

int main() {

    // declare a class object
    Sample obj1;

    cout << "Enter your age: ";

    // store input in age of the obj1 object
    cin >> obj1.age;

    // call class function
    obj1.displayAge();

    return 0;
}
```

Output

```
Enter your age: 20
Age = 20
```

In this program, we have created a class named `Sample`, which contains a `public` variable `age` and a `public` function `displayAge()`.

In `main()`, we have created an object of the `Sample` class named `obj1`. We then access the `public` elements directly by using the codes `obj1.age` and `obj1.displayAge()`.

Notice that the `public` elements are accessible from `main()`. This is because `public` elements are accessible from all parts of the program.

1.3 private Access Modifier

- The `private` keyword is used to create private members (data and functions).
- The private members can only be accessed from within the class.
- However, friend classes and friend functions can access private members.

1.3.1 Example 2: C++ private Access Specifier

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

// define a class
class Sample {

    // private elements
   private:
    int age;

    // public elements
   public:
    void displayAge(int a) {
        age = a;
        cout << "Age = " << age << endl;
    }
};

int main() {

    int ageInput;

    // declare an object
    Sample obj1;
```

```
    cout << "Enter your age: ";
    cin >> ageInput;

    // call function and pass ageInput as argument
    obj1.displayAge(ageInput);

    return 0;
}
```

Output

```
Enter your age: 20
Age = 20
```

In `main()`, the object `obj1` cannot directly access the class variable `age`.

```
// error
cin >> obj1.age;
```

We can only indirectly manipulate `age` through the public function `displayAge()`, since this function initializes `age` with the value of the argument passed to it i.e. the function parameter `int a`.

Example 2: Using public and private in C++ Class

```
// Program to illustrate the working of
// public and private in C++ Class

#include <iostream>
using namespace std;

class Room {

   private:
```

```
    double length;
    double breadth;
    double height;

  public:

    // function to initialize private variables
    void initData(double len, double brth, double hgt) {
        length = len;
        breadth = brth;
        height = hgt;
    }

    double calculateArea() {
        return length * breadth;
    }

    double calculateVolume() {
        return length * breadth * height;
    }
};

int main() {

    // create object of Room class
    Room room1;

    // pass the values of private variables as arguments
    room1.initData(42.5, 30.8, 19.2);

    cout << "Area of Room =  " << room1.calculateArea() << endl;
    cout << "Volume of Room =  " << room1.calculateVolume() << endl;

    return 0;
}
```

Output

```
Area of Room =  1309
Volume of Room =  25132.8
```

The above example is nearly identical to the first example, except that the class variables are now private.

Since the variables are now private, we cannot access them directly from `main()`. Hence, using the following code would be invalid:

```
// invalid code
obj.length = 42.5;
obj.breadth = 30.8;
obj.height = 19.2;
```

Instead, we use the public function `initData()` to initialize the private variables via the function parameters `double len`, `double brth`, and `double hgt`.

1.4 protected Access Modifier

Before we learn about the `protected` access specifier, make sure you know about inheritance in C++.

- The `protected` keyword is used to create protected members (data and function).
- The protected members can be accessed within the class and from the derived class.

1.4.1 Example 3: C++ protected Access Specifier

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

```
// declare parent class
class Sample {
    // protected elements
   protected:
    int age;
};

// declare child class
class SampleChild : public Sample {

   public:
    void displayAge(int a) {
        age = a;
        cout << "Age = " << age << endl;
    }

};

int main() {
    int ageInput;

    // declare object of child class
    SampleChild child;

    cout << "Enter your age: ";
    cin >> ageInput;

    // call child class function
    // pass ageInput as argument
    child.displayAge(ageInput);

    return 0;
}
```

Output

```
Enter your age: 20
Age = 20
```

Here, `SampleChild` is an inherited class that is derived from `Sample`. The variable `age` is declared in `Sample` with the `protected` keyword.

This means that `SampleChild` can access `age` since `Sample` is its parent class. We see this as we have assigned the value of `age` in `SampleChild` even though `age` is declared in the `Sample` class.