Chapter 5.
Derived classes (1 week)

- Inheritance and instantiation
- Friend « classes »
- Virtual <u>Base</u> Class
- Multiple inheritance

Inheritance

- Inheritance is a mechanism that allows a class to inherit properties and behaviors from another class.
- The class that is being inherited from is called the base class or parent class, and the class that inherits is called the derived class or child class.
- Inheritance facilitates code reuse and the creation of a hierarchy of classes.
- There are several types of inheritance in C++:

Single Inheritance: A class can inherit from only one base class.

```
class Base
{    // Base class members};

class Derived : public Base
 {    // Derived class members};
```

Multiple Inheritance: A class can inherit from more than one base class.

```
class Base1
{    // Base1 class members};
class Base2
{    // Base2 class members};

class Derived : public Base1, public Base2
 {    // Derived class members};
```

Multilevel Inheritance: A class can be derived from another derived class, creating a hierarchy of classes.

```
class Base
{    // Base class members};

class Derived1 : public Base
 {    // Derived1 class members};

class Derived2 : public Derived1
 {    // Derived2 class members};
```

It's worth noting that C++ supports different access specifiers (public, private, protected) for class members, which control the visibility and accessibility of those members in derived classes and client code.

```
// base class
class Animal {
   public:
    void eat() {
        cout << "I can eat!" << endl;
    }

    void sleep() {
        cout << "I can sleep!" << endl;
    }
};

// derived class
class Dog : public Animal {
  public:
    void bark() {
        cout << "I can bark! Woof woof!!" <<
endl;
    }
};
```

```
int main() {
    // Create object of the Dog class
    Dog dog1;

    // Calling members of the base class
    dog1.eat();
    dog1.sleep();

    // Calling member of the derived class
    dog1.bark();

    return 0;
}
```

Output:

```
I can eat!
 I can sleep!
I can bark! Woof woof!!
```

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

class Animal {           // base class
  private:
    string color;
   protected:
    string type;
   public:
    void eat() {
        cout << "I can eat!" << endl; }

    void sleep() {
        cout << "I can sleep!" << endl; }

    void setColor(string clr) {  color = clr;  }

    string getColor() { return color;}
};
```

```
class Dog : public Animal {// derived class
   public:
    void setType(string tp) {
        type = tp; }

    void displayInfo(string c) {
        cout << "I am a " << type << endl;
        cout << "My color is " << c << endl;}

 void bark() {
<< "I can bark! Woof woof!!" << endl; }
};
```

```
int main() {
   Dog dog1; // Create object of the Dog class

    // Calling members of the base class
    dog1.eat();
    dog1.sleep();
    dog1.setColor("black");

    // Calling member of the derived class
    dog1.bark();
    dog1.setType("mammal");

    // Using getColor() of dog1 as argument
    // getColor() returns string data
    dog1.displayInfo(dog1.getColor());

    return 0;
}
```

Output:

```
I can eat!
 I can sleep!
I can bark! Woof woof!!
 I am a mammal
 My color is black
```

<u>**IMPORTANT:**</u>

1. the variable *type* is protected and is thus accessible from the derived class Dog. We can see this as we have initialized type in the Dog class using the function setType().

2. the private variable *color* cannot be initialized in Dog

3. since the protected keyword hides data, we cannot access *type* directly from an object of Dog or Animal class.

Friend « classes »

- The friend keyword is used to designate a function or an entire class as a friend of another class.
- When a class or function is declared as a friend of another class, it is granted special access privileges to the private and protected members of that class.

```
class FriendClass; // Forward declaration of FriendClass
class MyClass
{private:
      int privateData;
  public:
     MyClass(int data) : privateData(data)
       {}

  friend class FriendClass; // Declaration of FriendClass as a friend
};

class FriendClass       // Definition of FriendClass
{public:
    void accessPrivateData(MyClass& obj)
 { // FriendClass can access private members of MyClass
 cout << "Accessing private data: " << obj.privateData << endl;    }
};
int main()
 {    MyClass obj(42);    // FriendClass can access private members of MyClass
FriendClass friendObj;
 friendObj.accessPrivateData(obj);
return 0;}
```

In this example, FriendClass is declared as a friend of MyClass using the friend keyword.

As a result, FriendClass can access the private member privateData of MyClass directly.

The main function demonstrates how an instance of FriendClass can access and print the private data of an object of MyClass.

Virtual Base Class

- A virtual base class is a class that is intended to be used as a common base class for multiple derived classes.
- It helps to avoid certain problems that can arise in multiple inheritance scenarios.
- When a base class is declared as virtual, it ensures that only a single instance of the base class is shared among the derived classes.

```
class Base
 {public:
    int data;
    Base(int value) : data(value) {}
  void display() const
{ cout << "Base::display() - data: " << data
<< std::endl;    }
};

class Derived1 : public virtual Base
 {public:
   Derived1(int value) : Base(value) {}
};

class Derived2 : public virtual Base
 {public:    Derived2(int value) : Base(value)
{}
};
```

```
class MultipleDerived : public Derived1,
public Derived2
 {public:    MultipleDerived(int value) :
Base(value), Derived1(value),
Derived2(value) {}
};

int main()
 { MultipleDerived obj(42);
 // Access data member from the base class
obj.display();
return 0;
}
```

In this example, both Derived1 and Derived2 virtually inherit from the Base class, and MultipleDerived inherits from both Derived1 and Derived2.

By using the virtual keyword in the inheritance, you ensure that there is only one instance of the Base class within MultipleDerived.

This helps to avoid a problem that can occur in multiple inheritance, where ambiguity arises due to the presence of multiple instances of a common base class.

The display function from the Base class can be called through an instance of MultipleDerived, and it will work correctly without ambiguity.