Inheritance

Course Code: CSC1102 &1103 Course Title: Introduction to Programming

Dept. of Computer Science
Faculty of Science and Technology

Lecturer No:	**11**	**Week No:**	**8 (1X1.5)** **9(1X1.5)**	**Semester:**	
Lecturer:	*Name & email*				

Inheritance

- Inheritance provides an opportunity to reuse the code functionality and fast implementation time.
- When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class.
- The mechanism of deriving a new class from an old class/previous written class in known as inheritance. Also known as “is a” or ”kind of” or ”is a kind of” relationship.
- The class which is inherited is called base class/parent class/super class. The class that inherits the base class is known as sub class/child class/derived class.

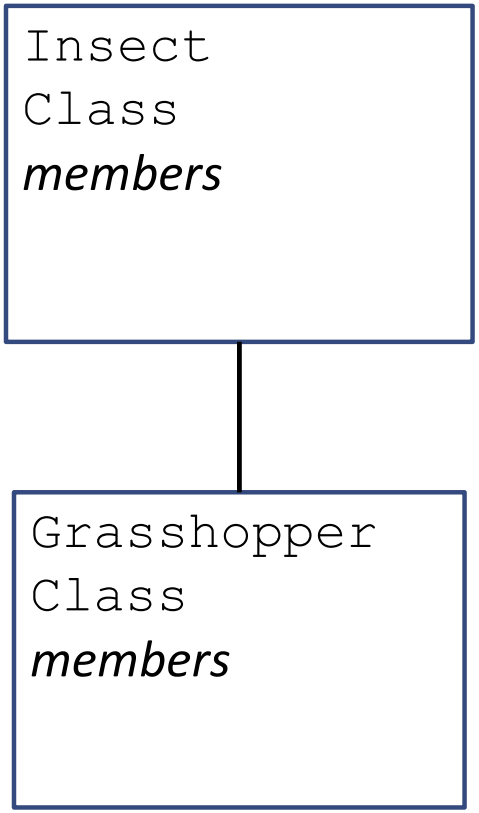
class derived-class: access-specifier base-class

Inheritance

- Inheritance allows a hierarchy of classes to be built.
- Move from the most general to the most specific
- The class that is inherited is the **base class**.
- The inheriting class is called the **derived class**.
- A derived class inherits traits of the base class
  &
  adds properties that are specific to that class.

Inheritance

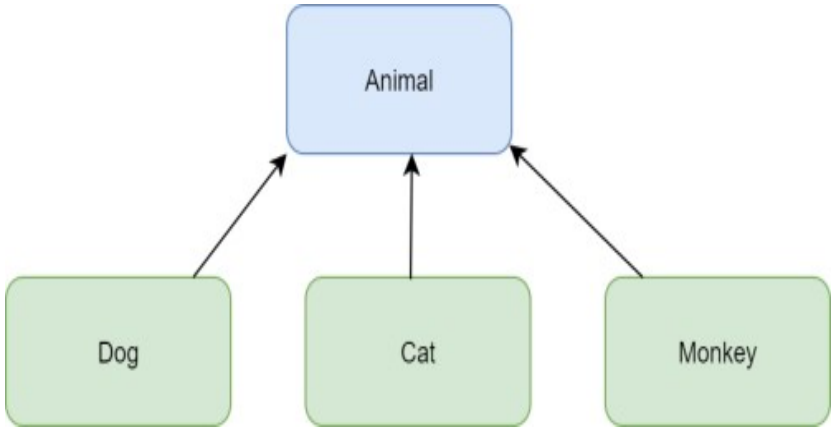
- Inheritance = the "**Is a**" Relationship
- A poodle **is a** dog
- A car **is a** vehicle
- A tree **is a** plant
- A rectangle **is a** shape
- A football player **is a** an athlete
- Base Class is the ***General Class***
- Derived Class is the ***Specialized Class***



Inheritance

In object-oriented programming, the concept of IS-A is a totally based on Inheritance, which can be of two types Class Inheritance or Interface Inheritance. It is just like saying "A is a B type of thing". For example, Apple is a Fruit, Car is a Vehicle etc. Inheritance is uni-directional. For example, House is a Building. But Building is not a House.

The idea of inheritance implements the is a relationship. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on.



Syntax

***Access Specification:* Public**

- Public members of Base are public members of Derived

- Private members of Base remain private members, but are inherited by the Derived class.

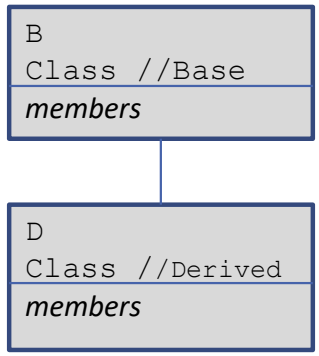
i.e. "They are invisible to the Derived class"

```
class B {
   int I;
public:
   void Set_I(int X){I=X;}
   int  Get_I() {return I;}
};
```

Base Class Access Specification

```
class D : public B {
   int J;
public:
   void Set_J(int X)
        {J = X;}
   int  Mul()
        {return J * Get_I();}
        // J * I → Compile error!
};
```

```
int main() {
   D ob;
   ob.Set_J(10);
   ob.Set_I(4);
   // ob.I = 8; Compile error!
   cout << ob.Mul() << endl;
   return 0;
} // end main
```



Inheritance

- A base class in not exclusively “owned” by a derived class. A base class can be inherited by any number of different classes.

- There may be times when you want to keep a member of a base class private but still permit a derived class access to it.
  SOLUTION: Designate the data as **protected.**

Protected Data Inherited as Public

Private members of the base class are always private to the derived class regardless of the access specifier.

```
class Base {
   protected:
       int a, b;
   public:
        void Setab(int n, int m)
             { a = n; b = m;}
};
```

```
class Derived: public Base {
     int c;
   public:
     void Setc(int x) { c = x;}
     void Showabc() {
        cout << a << " " << b << " " << c << endl;
      }
};
```

```
int main() {
   Derived ob;

   ob.Setab(1,2);
   ob.Setc(3);
   ob.Showabc();
   //ob.a = 5 NO! NO!

   return 0;

} // end main
```

Inheritance

- *Private members of Base remain private members and are inaccessible to the derived class.*
- *Public members of Base are public members of Derived*

BUT

- **Protected members of a base class are accessible to members of any class derived from that base.**

  Protected members, like private members, are not accessible outside the base or derived classes.

Private members of the base class are always private to the derived class regardless of the access specifier

- But when a base class is inherited as **protected**, ***public and protected*** members of the base class become protected members of the derived class.

```
class Base {
   protected:
      int a, b;
   public:
       void Setab(int n, int m)
            { a = n; b = m;}
};
```

```
class Derived: protected Base {
      int c;
   public:
     void Setc(int x) { c = x;}
     void Showabc() {
        cout << a << " " << b << " " << c << endl;
      }
};
```

```
int main() {
   Derived ob;

   //ob.Setab(1,2); ERROR
   //ob.a = 5;     NO! NO!

   ob.Setc(3);
   ob.Showabc();

   return 0;
} // end main
```

Inheritance

Private members of the base class are always private to the derived class regardless of the access specifier

- **Protected Access Specifier**
  - Private members of the base class are inaccessible to the derived class.
  - Public members of the base class become protected members of the derived class.
  - Protected members of the base class become protected members of the derived class.

i.e. only the public members of the derived class are accessible by the user application.

Inheritance

- **Constructors & Destructors**
  - When a base class and a derived class both have constructor and destructor functions
    - Constructor functions are executed in order of derivation – base class before derived class.
    - Destructor functions are executed in reverse order – the derived class's destructor is executed before the base class's destructor.
  - A derived class does not inherit the constructors of its base class.

Inheritance

```
class Base {
   public:
      Base() { cout << "Constructor Base Class\n";}
      ~Base() {cout << "Destructing Base Class\n";}
};
class Derived : public Base {
   public:
      Derived() { cout << Constructor Derived Class\n";}
      ~Derived(){ cout << Destructing Derived Class\n";}
};
```

```
int main() {
   Derived ob;
   return o;
}
```

```
---- OUTPUT ----
Constructor Base Class
Constructor Derived Class
Destructing Derived Class
Destructing Base Class
```

Inheritance

- **Passing an argument to a derived class's constructor**

```
Class Base {
   public:
      Base() {cout << "Constructor Base Class\n";}
      ~Base(){cout << "Destructing Base Class\n";}
};
Class Derived : public Base {
      int J;
   public:
      Derived(int X) {
          cout << Constructor Derived Class\n";
          J = X;
      }
      ~Derived(){ cout << Destructing Derived Class\n";}
      void ShowJ() { cout << "J: " << J << "\n"; }
};
```

```
int main() {
   Derived Ob(10);
   Ob.ShowJ();
   return 0;
} // end main
```

Inheritance

- **Arguments to both Derived and Base Constructors**

```
Class Base {
   int I;
   public:
      Base(int Y) {
         cout << "Constructor Base Class\n";
         I = Y;}
      ~Base(){cout << "Destructing Base Class\n";}
      void ShowI() { cout << "I: " << I << endl; }
};
Class Derived : public Base {
      int J;
   public:
      Derived(int X) : Base (X) {
         cout << Constructor Derived Class\n";
         J = X;
      }
      ~Derived(){ cout << Destructing Derived Class\n";}
      void ShowJ() { cout << << "J:" << J << "\n"; }
};
```

```
int main() {
   Derived Ob(10);

   Ob.ShowI();
   Ob.ShowJ();
   return 0;
} // end main
```

Different arguments to the Base – All arguments to the Derived.

```
Class Base {
   int I;
   public:
      Base(int Y) {
         cout << "Constructor Base Class\n";
         I = Y;}
      ~Base(){cout << "Destructing Base Class\n";}
      void ShowI() { cout << "I: " << I << endl; }
};
Class Derived : public Base {
      int J;
   public:
      Derived(int X, int Y) : Base (Y) {
         cout << Constructor Derived Class\n";
         J = X;
      }
      ~Derived(){ cout << Destructing Derived Class\n";}
      void ShowJ() { cout << << "J:" << J << "\n"; }
};
```

```
int main() {
   Derived Ob(5,8);

   Ob.ShowI();
   Ob.ShowJ();
   return 0;
} // end main
```

OK – If Only Base has Argument

```
Class Base {
   int I;
   public:
      Base(int Y) {
         cout << "Constructor Base Class\n";
         I = Y;}
      ~Base(){cout << "Destructing Base Class\n";}
      void ShowI() { cout << "I: " << I << endl; }
};
Class Derived : public Base {
      int J;
   public:
      Derived(int X) : Base (X) {
         cout << Constructor Derived Class\n";
         J = 0;            // X not used here
      }
      ~Derived(){ cout << Destructing Derived Class\n";}
      void ShowJ() { cout << << "J:" << J << "\n"; }
};
```

```
int main() {
   Derived Ob(10);

   Ob.ShowI();
   Ob.ShowJ();
   return 0;
} // end main
```

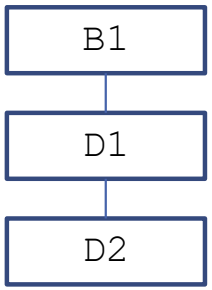
Inheritance

- **Multiple Inheritance – Inheriting more than one base class**

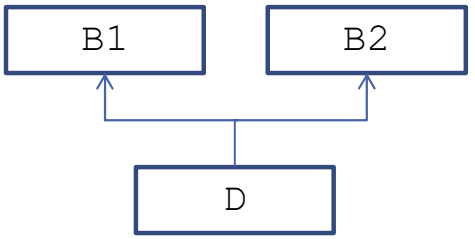
1. Derived class can be used as a base class for another derived class (***multilevel class hierarchy***)
2. A derived class can directly inherit more than one base class. 2 or more base classes are combined to help create the derived class

Inheritance

1. Multilevel Class Hierarchy
   - Constructor functions of all classes are called in order of derivation: B1, D1, D2
   - Destructor functions are called in reverse order



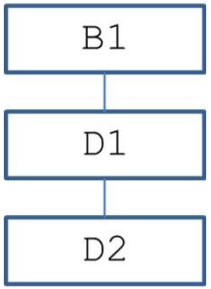
2. When a derived class directly inherits multiple base classes…
   - Access_Specifiers { public, private, protected} can be different
   - Constructors are executed in the order left to right, that the base classes are specified.
   - Destructors are executed in the opposite order.




```
class Derived_Class_Name: access Base1,
            access Base2,… access BaseN
{
   //.. body of class
} end Derived_Class_Name
```

Derived class inherits a class derived from another class.

```
class B1 {
   int A;
   public:
      B1(int Z) { A = Z;}
      int GetA() { return A; }
};
class D1 : public B1 {
      int B;
   public:
      D1(int Y, int Z) : B1 (Z) { B = Y; }
      void GetB() { return B; }
};
Class D2 : public D1 {
      int C;
    public:
      D2 (int X, int Y, int Z) : D1 ( Y, Z)) { C = X; }
      void ShowAll () {
         cout << GetA() << " " << GetB() << " " << C << endl; }
};
```



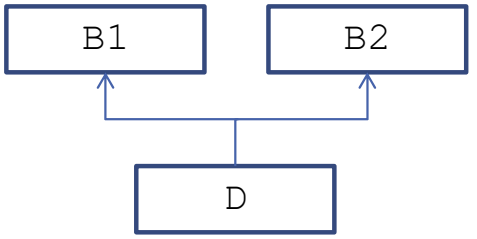
Because bases are inherited as public,
D2 has access to public elements of both B1 and D1

```
int main() {
   D2 Ob(5,7,9);

   Ob.ShowAll();

   // GetA & GetB are still public here
   cout << Ob.GetA() << " "
        << Ob.GetB() << endl;

   return 0;
} // end main
```

Derived Class Inherits Two Base Classes

```
class B1 {

   int A;

  public:

      B1(int Z) { A = Z;}

      int GetA() { return A; }};

class B2 {

      int B;

public:

      B2 (int Y) { B = Y; }

      void GetB() { return B; }};
```

```
class D : public B1, public B2
{

int C;

public:

    D (int X, int Y, int Z) :
B1(Z), B2 (Y) { C = X; }

     void ShowAll () {

    cout << GetA() << " " <<
GetB() << " " << C << endl; }

};
```




```
int main() {
   D Ob(5,7,9);

   Ob.ShowAll();

   return 0;
} // end main
```

Inheritance

- A Derived class does not inherit the constructors of its base class.
- Good Advice: You can and should include a call to one of the base class constructors when you define a constructor for a derived class.
- If you do not include a call to a base class constructor, then the default (zero argument) constructor of the base class is called automatically.
- If there is no default constructor for the base class, an error occurs.

Inheritance

- If the programmer does not define a ***copy constructor*** in a derived class (or any class), C++ will auto-generate a copy constructor for you. (Bit-wise copy)
- Overloaded assignment operators are not inherited, but can be used.
- When the destructor for the derived class is invoked, it auto-invokes the destructor of the base class. No need to explicitly call the base class destructor.

Thank You