Function Overriding

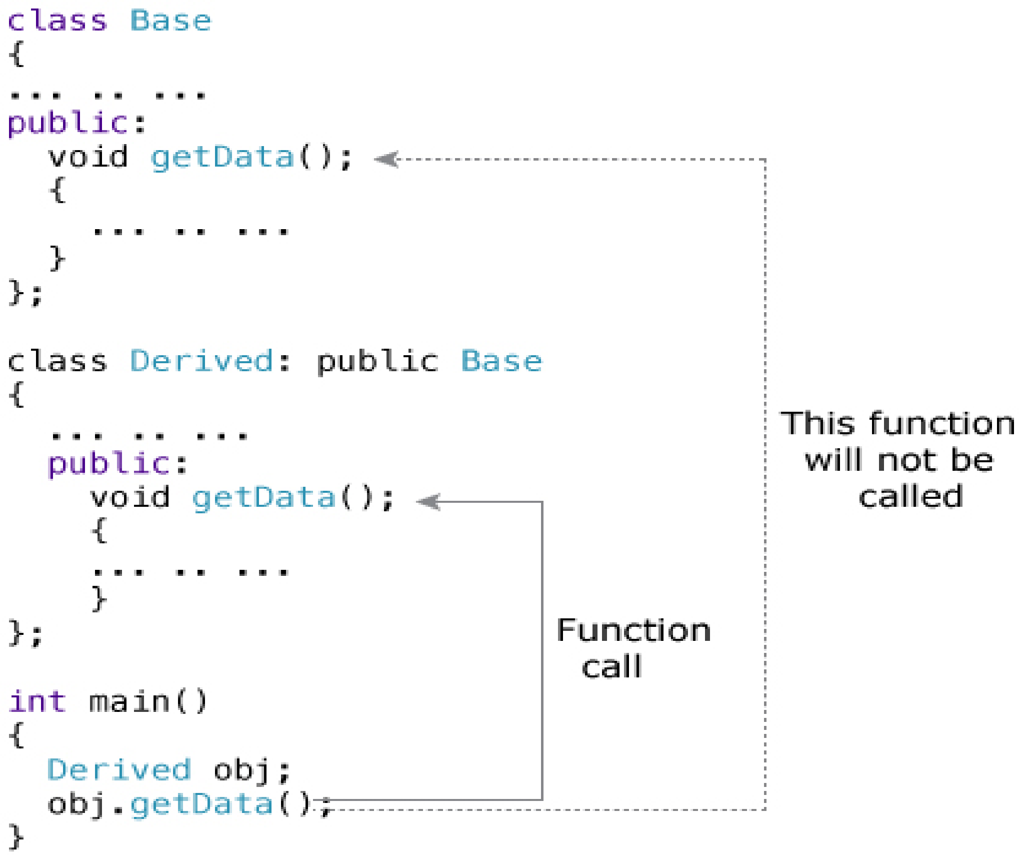
Chapter: 7, Teach Yourself C++

Prepared by: Lec Tasmiah Tamzid Anannya, CS Dept, AIUB

Function Overriding

- ❑Suppose, both base class and derived class have a member function with same name and arguments (number and type of arguments).
- ❑If you create an object of the derived class and call the member function which exists in both classes (base and derived), the member function of the derived class is invoked and the function of the base class is ignored.

Function Overriding: Example



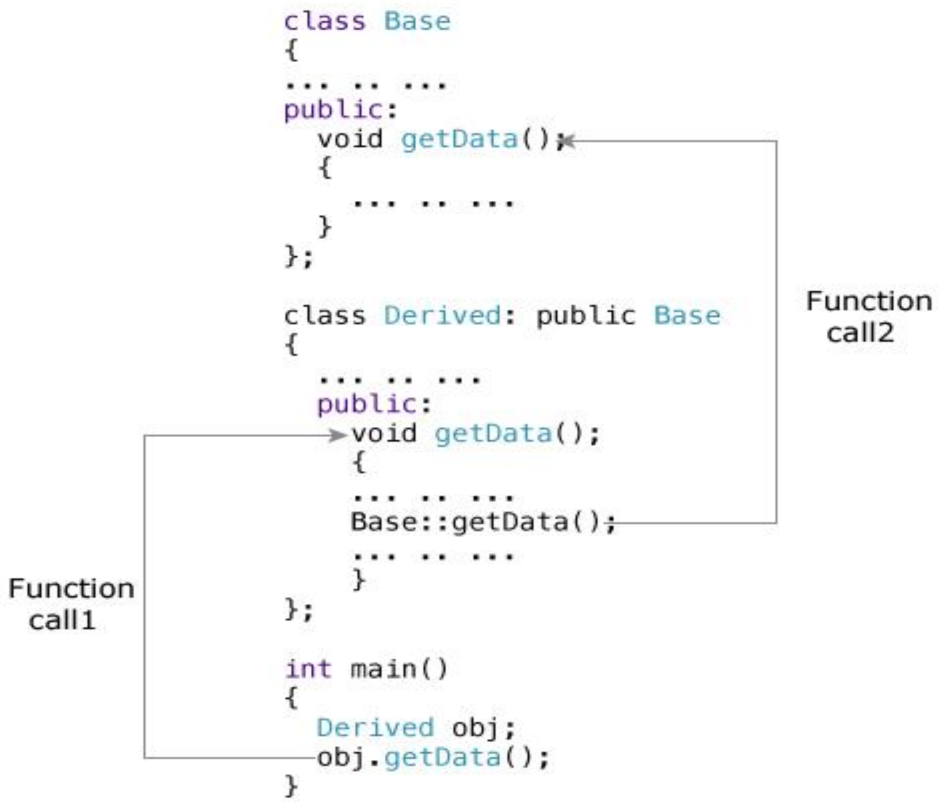
```
class Base
{
... .. ...
public:
   void getData();
   {
      ... .. ...
   }
};

class Derived: public Base
{
   ... .. ...
   public:
      void getData();
      {
      ... .. ...
      }
};

int main()
{
   Derived obj;
   obj.getData();
}
```

Function Overriding

How to access the overridden function in the base class from the derived class?- To access the overridden function of the base class from the derived class, scope resolution operator :: is used.



```
class Base
{
... .. ...
public:
  void getData();
  {
    ... .. ...
  }
};

class Derived: public Base
{
  ... .. ...
  public:
    void getData();
    {
    ... .. ...
    Base::getData();
    ... .. ...
    }
};

int main()
{
  Derived obj;
  obj.getData();
}
```

Function Overriding vs Overloading

- Function Overriding is known as run-time polymorphism.
- Function Overloading is known as compile-time polymorphism.

- **What is the difference between run-time polymorphism and compile-time polymorphism?**

Virtual Function

Chapter: 10, Teach Yourself C++

Pointers to Derived Classes

- A pointer of base class can also be used to point to any class derived from that base class.

```
Base *p;             //base class pointer
Base base_ob;        //object of type base
Derived derived_ob;  //object of derived class
p=&base_ob;          //p can, of course point to base object
p=&derived_ob;       //p can also point to derived objects
```

- A base class pointer can point to any object of any class derived from that class without generating a type mismatch error.
- But, you can only access those members of the derived object that were inherited from the base.
  - Because, base pointer has knowledge only of the base class.

Pointers to Derived Classes

- The reverse is not true.
- A pointer of derived type cannot be used to access an object of the base class.

```
Derived *p;                     //base class pointer
Base base_ob;               //object of type base
Derived derived_ob;   //object of derived class
p=&base_ob;                 //ERROR!!!
p=&derived_ob;           //p can also point to derived objects
```

Example

```
class base
{
   int x;
public:
   void setx(int a){x=a;}
   int getx(){return x;}
};

class derived : public base
{
   int y;
public:
   void sety(int b){y=b;}
   int gety(){return y;}
};
```

```
int main()
{
   base *p;
   base base_ob;
   derived derived_ob;










}
```

Example

```
class base
{
   int x;
public:
   void setx(int a){x=a;}
   int getx(){return x;}
};

class derived : public base
{
   int y;
public:
   void sety(int b){y=b;}
   int gety(){return y;}
};
```

```
int main()
{
   base *p;
   base base_ob;
   derived derived_ob;

   p=&base_ob;
   p->setx(10);
   cout<<"base object x:"<<p->getx()<<endl;
}
```

Example

```
class base
{
    int x;
public:
    void setx(int a){x=a;}
    int getx(){return x;}
};

class derived : public base
{
    int y;
public:
    void sety(int b){y=b;}
    int gety(){return y;}
};
```

```
int main()
{
    base *p;
    base base_ob;
    derived derived_ob;

    p=&base_ob;
    p->setx(10);
    cout<<"base object x:"<<p->getx()<<endl;

    p=&derived_ob;
    p->setx(100);
    cout<<"derived object x"<<p->getx()<<endl;
}
```

Example

```
class base
{
   int x;
public:
   void setx(int a){x=a;}
   int getx(){return x;}
};

class derived : public base
{
   int y;
public:
   void sety(int b){y=b;}
   int gety(){return y;}
};
```

```
int main()
{
   base *p;
   base base_ob;
   derived derived_ob;

   p=&base_ob;
   p->setx(10);
   cout<<"base object x:"<<p->getx()<<endl;

   p=&derived_ob;
   p->setx(100);
   cout<<"derived object x"<<p->getx()<<endl;



   derived_ob.sety(1000);
   cout<<"derived object y: "<<derived_ob.gety()<<endl;
}
```

Example

```
class base
{
    int x;
public:
    void setx(int a){x=a;}
    int getx(){return x;}
};

class derived : public base
{
    int y;
public:
    void sety(int b){y=b;}
    int gety(){return y;}
};
```

```
int main()
{
    base *p;
    base base_ob;
    derived derived_ob;

    p=&base_ob;
    p->setx(10);
    cout<<"base object x:"<<p->getx()<<endl;

    p=&derived_ob;
    p->setx(100);
    cout<<"derived object x"<<p->getx()<<endl;

//cannot use pointer p to set y, so doing it using derived object
    derived_ob.sety(1000);
    cout<<"derived object y: "<<derived_ob.gety()<<endl;
}
```

Example

```
class base
{
    int x;
public:
    void setx(int a){x=a;}
    int getx(){return x;}
};

class derived : public base
{
    int y;
public:
    void sety(int b){y=b;}
    int gety(){return y;}
};
```

```
base object x:10
derived object x100
derived object y:1000
```

```
int main()
{
    base *p;
    base base_ob;
    derived derived_ob;

    p=&base_ob;
    p->setx(10);
    cout<<"base object x:"<<p->getx()<<endl;

    p=&derived_ob;
    p->setx(100);
    cout<<"derived object x"<<p->getx()<<endl;

//cannot use pointer p to set y, so doing it using derived object
    derived_ob.sety(1000);
    cout<<"derived object y: "<<derived_ob.gety()<<endl;
}
```

Virtual Functions

- ❑A *virtual* function is a member function that is declared within a base class and redefined by a derived class.
- ❑When virtual function is redefined by the derived class, the keyword *virtual* is not needed.
- ❑A virtual function can be called just like any other member function.
- ❑A function that contains a virtual function is referred to as a *polymorphic* class.

Example

```
class base
{
   int x;
public:
   base(int a){x=a;}
   virtual void func()
   {
      cout<<"In function of base:";
      cout<<x<<endl;
   }
};
class derived :public base
{
   int y;
public:
   derived(int a, int b):base(a)
   {y=b;}
   void func()
   {
      cout<<"In function of derived:";
      cout<<y;
   }
};
```

```
int main()
{
   base *p;
   base base_ob(10);
   derived derived_ob(100,200);




}
```

Example

```
class base
{
   int x;
public:
   base(int a){x=a;}
   virtual void func()
   {
      cout<<"In function of base:";
      cout<<x<<endl;
   }
};
class derived :public base
{
   int y;
public:
   derived(int a, int b):base(a)
   {y=b;}
   void func()
   {
      cout<<"In function of derived:";
      cout<<y;
   }
};
```

```
int main()
{
   base *p;
   base base_ob(10);
   derived derived_ob(100,200);

   p=&base_ob;
   p->func();

}
```

Example

```
class base
{
   int x;
public:
   base(int a){x=a;}
   virtual void func()
   {
      cout<<"In function of base:";
      cout<<x<<endl;
   }
};
class derived :public base
{
   int y;
public:
   derived(int a, int b):base(a)
   {y=b;}
   void func()
   {
      cout<<"In function of derived:";
      cout<<y;
   }
};
```

```
int main()
{
   base *p;
   base base_ob(10);
   derived derived_ob(100,200);

   p=&base_ob;
   p->func();

}
```

```
In function of base:10
```

Example

```
class base
{
   int x;
public:
   base(int a){x=a;}
   virtual void func()
   {
      cout<<"In function of base:";
      cout<<x<<endl;
   }
};
class derived :public base
{
   int y;
public:
   derived(int a, int b):base(a)
   {y=b;}
   void func()
   {
      cout<<"In function of derived:";
      cout<<y;
   }
};
```

```
int main()
{
   base *p;
   base base_ob(10);
   derived derived_ob(100,200);

   p=&base_ob;
   p->func();

   p=&derived_ob;
   p->func();
}
```

```
In function of base:10
```

Example

```
class base
{
   int x;
public:
   base(int a){x=a;}
   virtual void func()
   {
      cout<<"In function of base:";
      cout<<x<<endl;
   }
};
class derived :public base
{
   int y;
public:
   derived(int a, int b):base(a)
   {y=b;}
   void func()
   {
      cout<<"In function of derived:";
      cout<<y;
   }
};
```

```
int main()
{
   base *p;
   base base_ob(10);
   derived derived_ob(100,200);

   p=&base_ob;
   p->func();

   p=&derived_ob;
   p->func();
}
```

```
In function of base:10
In function of derived:200
```

Practice

Now, create another class called derived2 and make it a child class of base. Rewrite the virtual function of base class in the derived2 class. Use the base pointer to point the object of the class and show what will be the output.

Example

In the previous example, if we do not use the keyword virtual, then the output will be:

```
In function of base:10

In function of base:100
```

Example

When a derived class does not override a virtual function, the base class version is used.

```
class base
{
    int x;
public:
    base(int a){x=a;}
    virtual void func()
    {
        cout<<"In function of base:";
        cout<<x<<endl;
    }
};

class derived :public base
{
    int y;
public:
    derived(int a, int b):base(a)
    {y=b;}

};
```

```
int main()

{

    base *p;

    base base_ob(10);

    derived derived_ob(100,200);









}
```

Example

When a derived class does not override a virtual function, the base class version is used.

```
class base
{
   int x;
public:
   base(int a){x=a;}
   virtual void func()
   {
      cout<<"In function of base:";
      cout<<x<<endl;
   }
};

class derived :public base
{
   int y;
public:
   derived(int a, int b):base(a)
   {y=b;}

};
```

```
int main()
{
   base *p;
   base base_ob(10);
   derived derived_ob(100,200);


   p=&base_ob;
   p->func();




}
```

Example

When a derived class does not override a virtual function, the base class version is used.

```
class base
{
   int x;
public:
   base(int a){x=a;}
   virtual void func()
   {
      cout<<"In function of base:";
      cout<<x<<endl;
   }
};

class derived :public base
{
   int y;
public:
   derived(int a, int b):base(a)
   {y=b;}

};
```

```
int main()
{
   base *p;
   base base_ob(10);
   derived derived_ob(100,200);


   p=&base_ob;
   p->func();



}
```

```
In function of base:10
```

Example

When a derived class does not override a virtual function, the base class version is used.

```
class base
{
   int x;
public:
   base(int a){x=a;}
   virtual void func()
   {
      cout<<"In function of base:";
      cout<<x<<endl;
   }
};

class derived :public base
{
   int y;
public:
   derived(int a, int b):base(a)
   {y=b;}

};
```

```
int main()
{
   base *p;
   base base_ob(10);
   derived derived_ob(100,200);


   p=&base_ob;
   p->func();


   p=&derived_ob;
   p->func();
}
```

```
In function of base:10
```

Example

When a derived class does not override a virtual function, the base class version is used.

```
class base
{
   int x;
public:
   base(int a){x=a;}
   virtual void func()
   {
      cout<<"In function of base:";
      cout<<x<<endl;
   }
};

class derived :public base
{
   int y;
public:
   derived(int a, int b):base(a)
   {y=b;}

};
```

```
int main()
{
   base *p;
   base base_ob(10);
   derived derived_ob(100,200);


   p=&base_ob;
   p->func();


   p=&derived_ob;
   p->func();
}
```

```
In function of base:10
In function of base:100
```

Example

```
class area
{
   double dim1, dim2;
public:
   void setarea(double x, double y)
   {dim1=x; dim2=y;}
   double getdim1(){return dim1;}
   double getdim2(){return dim2;}
    virtual double getarea(){
      cout<<"you must override this
 function."<<endl;
   }
};
class rectangle :public area
{
public:
   double getarea()
   {
      return getdim1()*getdim2();
   }

};
```

```
int main()
{
   area *p;
   area base_ob;
   rectangle derived_ob;
   derived_ob.setarea(10,10);



}
```

Example

```
class area
{
   double dim1, dim2;
public:
   void setarea(double x, double y)
   {dim1=x; dim2=y;}
   double getdim1(){return dim1;}
   double getdim2(){return dim2;}
    virtual double getarea(){
      cout<<"you must override this
 function."<<endl;
   }
};
class rectangle :public area
{
public:
   double getarea()
   {
      return getdim1()*getdim2();
   }

};
```

```
int main()
{
   area *p;
   area base_ob;
   rectangle derived_ob;
   derived_ob.setarea(10,10);

   p=&base_ob;
   p->getarea();




}
```

Example

```
class area
{
   double dim1, dim2;
public:
   void setarea(double x, double y)
   {dim1=x; dim2=y;}
   double getdim1(){return dim1;}
   double getdim2(){return dim2;}
    virtual double getarea(){
      cout<<"you must override this
 function."<<endl;
   }
};
class rectangle :public area
{
public:
   double getarea()
   {
      return getdim1()*getdim2();
   }

};
```

```
int main()
{
   area *p;
   area base_ob;
   rectangle derived_ob;
   derived_ob.setarea(10,10);

   p=&base_ob;
   p->getarea();




}
```

```
  you must override this function.
```

Example

```
class area
{
   double dim1, dim2;
public:
   void setarea(double x, double y)
   {dim1=x; dim2=y;}
   double getdim1(){return dim1;}
   double getdim2(){return dim2;}
    virtual double getarea(){
      cout<<"you must override this
 function."<<endl;
   }
};
class rectangle :public area
{
public:
   double getarea()
   {
      return getdim1()*getdim2();
   }

};
```

```
int main()
{
   area *p;
   area base_ob;
   rectangle derived_ob;
   derived_ob.setarea(10,10);

   p=&base_ob;
   p->getarea();

   p=&derived_ob;
   cout<<p->getarea()<<endl;
}
```

```
you must override this function.
```

Example

```
class area
{
   double dim1, dim2;
public:
   void setarea(double x, double y)
   {dim1=x; dim2=y;}
   double getdim1(){return dim1;}
   double getdim2(){return dim2;}
    virtual double getarea(){
      cout<<"you must override this
 function."<<endl;
   }
};
class rectangle :public area
{
public:
   double getarea()
   {
      return getdim1()*getdim2();
   }

};
```

```
int main()
{
   area *p;
   area base_ob;
   rectangle derived_ob;
   derived_ob.setarea(10,10);

   p=&base_ob;
   p->getarea();


   p=&derived_ob;
   cout<<p->getarea()<<endl;
}
```

```
you must override this function.
100
```

Practice

Write a program that creates a base class called **dist** and that stores the distance between two points in a **double** variable.

In **dist** class, create a virtual function called **trav_time()** that outputs the time it takes to travel the distance, assuming that the distance is in miles and the speed is 60 miles per hour.

In a derived class called **metric**, override **trav_time()** so that it outputs the travel time assuming that the distance is in miles and the speed is 100 miles per hour.

$$speed = \frac{distance\ travelled}{time\ taken}$$

Pure virtual function

- When there is no meaningful action for a base class virtual function to perform, the implication is that any derived class must override the function.

- To ensure that this will occur, C++ support *pure virtual function.*

- A *pure virtual functi*on has no definition relative to the base class. Only the function's prototype is included.

virtual ret-type func-name(parameter list)=0;

- If a derived class does not override pure virtual function, a compile-time error occurs.

Abstract class

- When a class contains at least one pure virtual function, it is called *abstract class*.
- It is technically, an incomplete type and no objects of this class can be created.
- Thus, abstract classes are created only to be inherited.
- You can still create an pointer of abstract class.

Example

```
class area
{
    double dim1, dim2;
public:
    void setarea(double x, double y)
    {dim1=x; dim2=y;}
    double getdim1(){return dim1;}
    double getdim2(){return dim2;}
     //pure virtual function
     virtual double getarea()=0;
};

class rectangle :public area
{
public:
    double getarea()
    {
        return getdim1()*getdim2();
    }

};
```

```
int main()
{
    area *p;
    //area base_ob; This is not allowed
    rectangle derived_ob;
    derived_ob.setarea(10,10);




}
```

Example

```
class area
{
   double dim1, dim2;
public:
   void setarea(double x, double y)
   {dim1=x; dim2=y;}
   double getdim1(){return dim1;}
   double getdim2(){return dim2;}
    //pure virtual function
    virtual double getarea()=0;
};

class rectangle :public area
{
public:
   double getarea()
   {
      return getdim1()*getdim2();
   }

};
```

```
int main()
{
   area *p;
   //area base_ob; This is not allowed
   rectangle derived_ob;
   derived_ob.setarea(10,10);


   p=&derived_ob;
   cout<<p->getarea()<<endl;
}
```

Example

```
class area
{
   double dim1, dim2;
public:
   void setarea(double x, double y)
   {dim1=x; dim2=y;}
   double getdim1(){return dim1;}
   double getdim2(){return dim2;}
    //pure virtual function
    virtual double getarea()=0;
};

class rectangle :public area
{
public:
   double getarea()
   {
      return getdim1()*getdim2();
   }
};
```

```
int main()
{
   area *p;
   //area base_ob; This is not allowed
   rectangle derived_ob;
   derived_ob.setarea(10,10);


   p=&derived_ob;
   cout<<p->getarea()<<endl;
}
```

```
100
```

Practice

Write a program that contains a class named as **Area**. Define a **pure virtual function** named as **calculation** as a member of that class.

calculation function will be redefined by the other child classes of that class named as **triangle**, **rectangle**, and **square** to calculate the area of triangle, rectangle, and square.

Define necessary parameters as private data members of the class to write the error-free code and to produce the correct results