Base Classes and Derived Classes

Base class

- **"Base class** is a class which defines those **qualities common to all objects** to be derived from the base."
- The base class represents the most **general description.**
- A class that is **inherited** is referred to as a base class.

Derived Classes

- "The classes derived from the base class are usually referred to as **derived classes**."
- "A **derived class** includes **all** features of the generic base class and then adds **qualities specific** to the derived class."
- The class that does the **inheriting** is called the derived class.

Consider a base class **Shape** and its derived class **Rectangle** as follows:

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

// Base class
class Shape
{
  public:
    void setWidth(int w)
    {
      width = w;
    }
    void setHeight(int h)
    {
      height = h;
    }
  protected:
    int width;
    int height;
};

// Derived class
class Rectangle: public Shape
{
  public:
    int getArea()
    {
```

```
        return (width * height);
      }
};

int main(void)
{
  Rectangle Rect;

  Rect.setWidth(5);
  Rect.setHeight(7);

  // Print the area of the object.
  cout << "Total area: " << Rect.getArea() << endl;

  return 0;
}
```

Output:
Total area: 35

Protected Members

Data hiding is one of the important features of Object Oriented Programming which allows preventing the functions of a program to access directly the internal representation of a class type. The access restriction to the class members is specified by the labeled **public, private,** and **protected** sections within the class body. The keywords public, private, and protected are called access specifiers.

Protected members:

A **protected** member variable or function is very similar to a private member but it provided one additional benefit that they can be accessed in child classes which are called derived classes.

You will learn derived classes and inheritance in next chapter. For now you can check following example where I have derived one child class **SmallBox** from a parent class **Box**.

Following example is similar to above example and here **width** member will be accessible by any member function of its derived class SmallBox.

```
#include <iostream>
```

```
using namespace std;


class Box
{
   protected:
      double width;
};


class SmallBox:Box // SmallBox is the derived class.
{
   public:
      void setSmallWidth( double wid );
      double getSmallWidth( void );
};


// Member functions of child class
double SmallBox::getSmallWidth(void)
{
    return width ;
}


void SmallBox::setSmallWidth( double wid )
{
    width = wid;
}


// Main function for the program
int main( )
{
   SmallBox box;


   // set box width using member function
   box.setSmallWidth(5.0);
```

```
   cout << "Width of box : "<< box.getSmallWidth() << endl;


   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Width of box : 5
```

Casting Class pointers and Member Functions

C++ lets you convert between types. This is called *casting*. Sometimes this happens automatically.

For example,

```
int i=7;
float f=i;
```

Works without fuss because *implicit casting* happens. It also lets you do

```
float f=7.7;
int i=f;
```

Even though an integer can't store the value 7.7.

Casting class pointers

Pointers can be cast too. Here's an example

```
class base {
  public:
  int i;
};

class derived: public base {
   public:
   float f;
};

int main () {
  base b;
  derived d;

  base *pointer_to_base;
  derived *pointer_to_derived;
  pointer_to_base=&d; // OK. Note that you can't access d.f via
  // pointer_to_base though.
```

```
  // pointer_to_derived=&b; // Not ok. Were it allowed, then
  // pointer_to_derived->f=7; would cause trouble, because b
  // doesn't have an f field. But you can force the issue.
  pointer_to_derived=static_cast<derived*>(&b); // Allowed
}
```

Casting member funtion

Classes can not only contain variables, they can contain functions too. Conversion routines are ideal candidates for inclusion within a class. Here the derived class is being extended so that it can deal with the problems mentioned above. An extra constructor (a copy constructor) is added so that a derived object can be created as a copy of a base object, the derived object's f field being initialised to 0.

```
class base {
  public:
  int i;
};

class derived: public base {
   public:
   float f;
   derived(const base& b) {
     i=b.i;
     f=0;
   };
};

int main () {
  base b;
  derived d=b;
}
```

Overriding

A method in child class overrides a method in parent class if they have the same name and type signature.

Overriding

- Classes in which methods are defined must be in a parent-child relationship.
- Type signatures must match.
- Dynamic binding of messages.
- Runtime mechanism based on the dynamic type of the receiver.
- Contributes to code sharing (non-overriding classes share same method).

Function Overriding

If base class and derived class have member functions with same name and arguments. If you create an object of derived class and write code to access that member function then, the member function in derived class is only invoked, i.e., the member function of derived class overrides the member function of base class. This feature in C++ programming is known as function overriding.

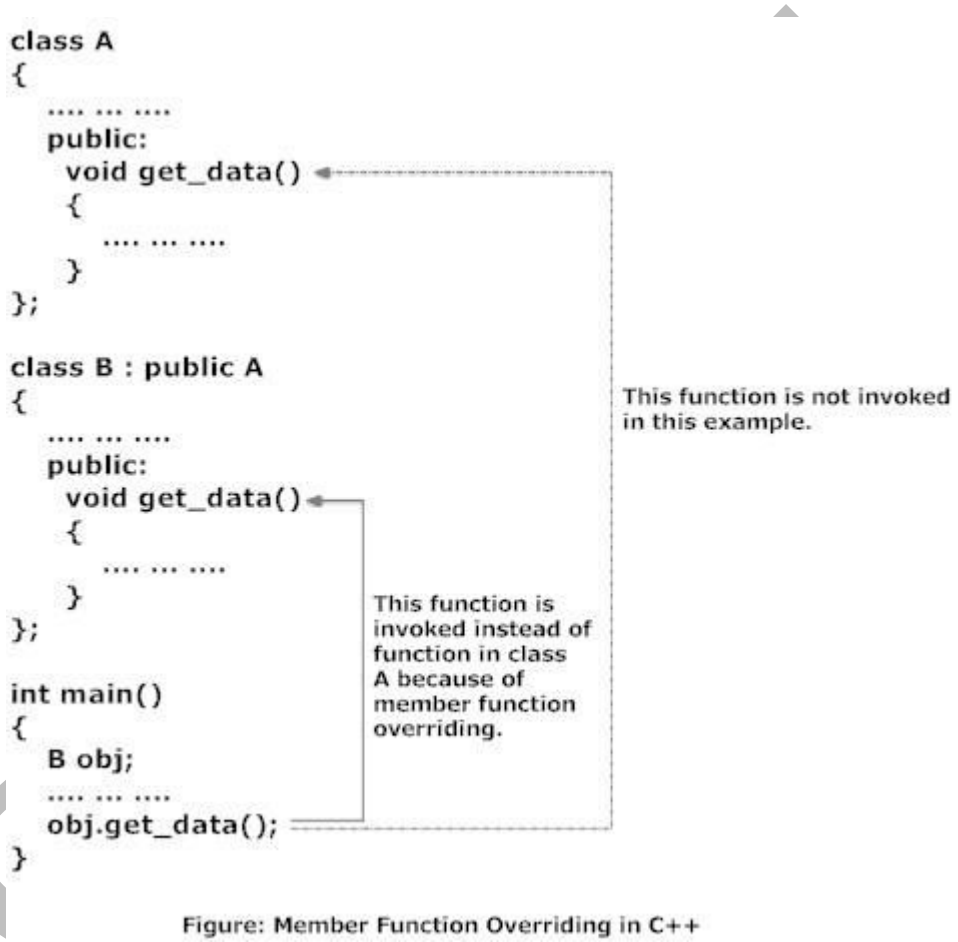


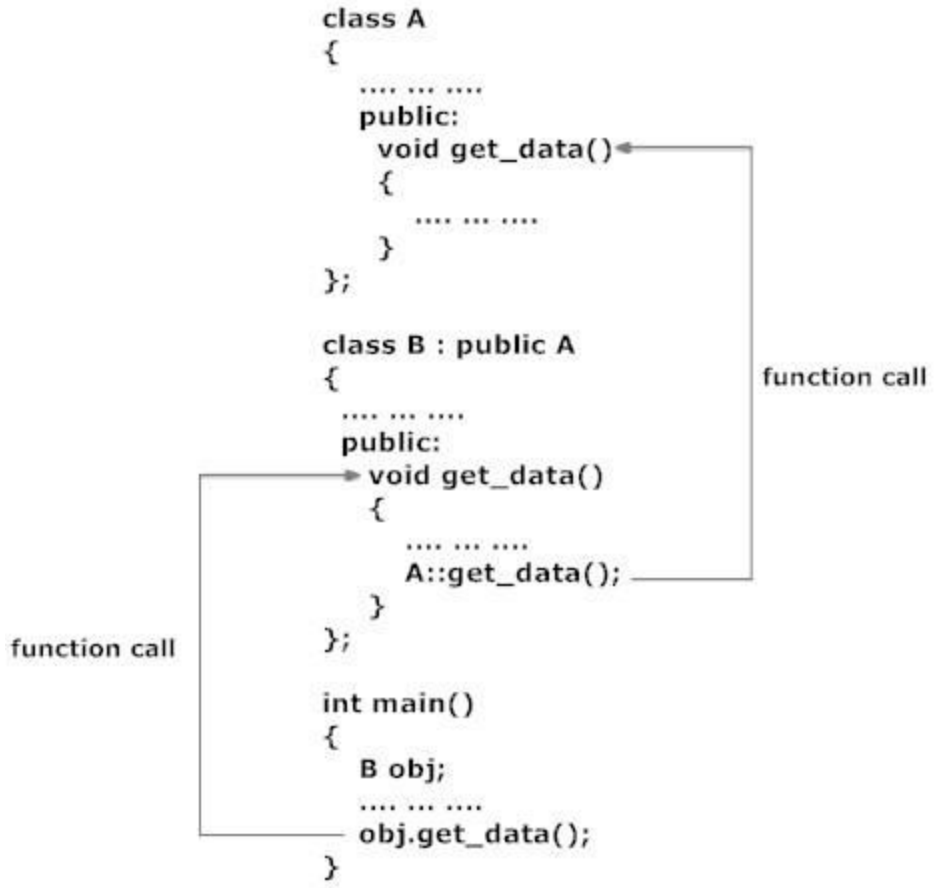
Figure: Member Function Overriding in C++

Accessing the Overridden Function in Base Class From Derived Class

To access the overridden function of base class from derived class, scope resolution operator **::**. For example: If you want to access get_data() function of base class from derived class in above example then, the following statement is used in derived class.

A::get_data; // Calling get_data() of class A.

It is because, if the name of class is not specified, the compiler thinks get_data() function is calling itself.



Public, Protected and Private Inheritance

- "Inheritance is the mechanism to provide the power of **reusability** and **extendibility**."
- "Inheritance is the process by which one **object can acquire the properties of another object**."
- "Inheritance is the process by which **new** classes called ***derived*** **classes** are created from **existing** classes called ***base*** **classes**."
- Allows the creation of **hierarchical** classifications.

Public Inheritance: When deriving a class from a **public** base class, **public** members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's **private** members are never accessible directly from a derived class, but can be accessed through calls to the **public** and**protected** members of the base class.

Protected Inheritance: When deriving from a **protected** base class, **public** and **protected**members of the base class become **protected** members of the derived class.

Private Inheritance: When deriving from a **private** base class, **public** and **protected** members of the base class become **private** members of the derived class.

Inheritance & Access Specifier

Access	public	protected	private
Members of the same class	Yes	Yes	Yes
Members of derived classes	Yes	Yes	No
Non-members	Yes	No	No

Constructors and Destructors in derived Classes

CONSTRUCTORS AND DESTRUCTORS IN DERIVED CLASSES:

A derived class inherits its base class members, when an object of a derived class is instantiated, the base class's constructor must be called to initialize the base class members of the derived class object.

Instantiating a derived-class object begins a chain of constructor calls in which the derived-class constructor, before performing its own tasks, invokes its direct base class's constructor either explicitly (via a base-class member initializer) or implicitly (calling the base class's default constructor). Similarly, if the base class is derived from another class, the base-class constructor is required to invoke the constructor of the next class up in the hierarchy, and so on. The last constructor called in this chain is the one of the class at the base of the hierarchy, whose body actually finishes executing *first*. The original derived-class constructor's body

finishes executing *last*. Each base-class constructor initializes the base-class data members that the derived-class object inherits.

When a derived-class object is destroyed, the program calls that object's destructor. This begins a chain (or cascade) of destructor calls in which the derived-class destructor and the destructors of the direct and indirect base classes and the classes' members execute in *reverse* of the order in which the constructors executed. When a derived-class object's destructor is called, the destructor performs its task, and then invokes the destructor of the next base class up the hierarchy. This process repeats until the destructor of the final base class at the top of the hierarchy is called. Then the object is removed from memory. Base-class constructors, destructors and overloaded assignment operators are *not* inherited by derived classes. Derived-class constructors, destructors and overloaded assignment operators, however, can call base-class versions.

Implicit Derived – Class Object To Base

BaseClassObject = derivedClassObject; „

- This will work

Remember, the derived class object has more members than the base class object

- Extra data is not given to the base class

DerivedClassObject = baseClassObject; „

- May not work properly

Unless an assignment operator is overloaded in the derived class, data members exclusive to the derived class will be unassigned „

- Base class has less data members than the derived class

– Some data members missing in the derived class object

Example:

```
#include <iostream>

class Base {
    public:
        void func() { std::cout << "Base::func\n"; }
};

class Derived : public Base {
    public:
        void otherfunc() { std::cout << "Derived::otherfunc\n"; }
};

int main() {
    Derived* d = new Derived;
    d->otherfunc(); // Derived::otherfunc
    d->func(); // Base::func
```

```
    Base* b = new Base;
    b->func(); // Base::func
//  b->otherfunc(); -- syntax error, no otherfunc in Base

    Base* db = new Derived;
    db->func(); // Base::func
//  db->otherfunc(); -- syntax error, it doesn't know db is a Derived
}
```

Composition Vs. Inheritance

In inheritance the superclass is created when the subclass is created. In Composition, the object is created when the coder wants it to.

This is inheritance, when the Child class is created the parent is created because the child inherits from parent.

```
class Parent {

    //Some code
}

class Child extends Parent{

//Some code
}
```

This is composition, the object is not created when the child class is created, instead it is created whenever it is need.

```
class Parent{

    //Some code
}

class Child{

    private Parent parent = new Parent();
    //Some code
}
```

In this case, the Parent class is also created when the Child class is created. Below is another example of Composition without the object being created when the child class is created

```
class Parent{

    //Some code
}

class Child{

    private Parent parent;
```

```
    public Child()
    {
    }
    public void createParent()
    {
         parent = new Parent();
    }
}
```

Note how the parent is not created until a call is made to the createParent.

Virtual functions

A Virtual function is a member function which is declared in the base class and redefined by the derived class. A Virtual function is one that exists, does not really exist but appears real to some or the other part of the program.

The "virtual" keyword literally means existing in effect but not in reality. It tells the compiler (more specifically the linker) that it should defer its decision on what function is implied by the call until run time.

DECLARING A VIRTUAL FUNCTION

Once declared a function virtual in base class, it will be virtual for all the subsequent derived classes.

We don't need to specify the virtual keyword in the derived classes, but the function will still be virtual nonetheless.

Example

(a) struct point

{ …….

virtual float area (void);

// virtual keyword

};

(b) struct circle: point

{ …….

```
    void area(void); // virtual keyword not required here.
};
```

Pure Virtual Function

For understanding the concept of pure virtual function, we have to be familiar with the term 'Abstract classes'. In most of the situations the base class of hierarchy should be very general. It should contain very little code at all. Instead, it should leave all of the specifics to the derived classes. In fact, in some cases, it makes sense for the base class to be so general that it shouldn't even be used to directly create objects.

This pointer

Every object in C++ has access to its own address through an important pointer called **this** pointer. The **this** pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used torefer to the invoking object.

Friend functions do not have a **this** pointer, because friends are not members of a class. Only member functions have a **this** pointer.

```
#include <iostream>

using namespace std;

class Box
{
   public:
      // Constructor definition
      Box(double l=2.0, double b=2.0, double h=2.0)
      {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;
```

```
      }
      double Volume()
      {
         return length * breadth * height;
      }
      int compare(Box box)
      {
         return this->Volume() > box.Volume();
      }
   private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};

int main(void)
{
   Box Box1(3.3, 1.2, 1.5);    // Declare box1
   Box Box2(8.5, 6.0, 2.0);    // Declare box2

   if(Box1.compare(Box2))
   {
      cout << "Box2 is smaller than Box1" <<endl;
   }
   else
   {
      cout << "Box2 is equal to or larger than Box1" <<endl;
   }
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Constructor called.
Constructor called.
Box2 is equal to or larger than Box1
```

Abstract Base Classes and Concrete Classes

In C++ an *abstract class* is one which defines an interface, but does not necessarily provide implementations for all its member functions. An abstract class is meant to be used as the base class from which other classes are derived. The derived class is expected to provide implementations for the member functions that are not implemented in the base class. A derived class that implements all the missing functionality is called a *concrete class* .

E.g., the GraphicalObject class defined in Programs ❑ and ❑ is an abstract class. In particular, no implementation is given for the virtual member function Draw. The fact that no implementation is given is indicated by the =0 attached to the Draw function prototype in the class definition (Program ❑, line 19). Recall that an object carries a pointer to the appropriate routine for every virtual member function it supports. The =0says that the pointer for the Draw function is not defined in the GraphicalObject class and, therefore, it must be defined in a derived class.

A virtual member function for which no implementation is given is called a *pure virtual function* . If a C++ class contains a pure virtual function, it is an *abstract class*. In C++ it is not possible to instantiate an abstract class. E.g., the following declaration is illegal:

GraphicalObject g (Point (0,0)); // Wrong.

If we were allowed to declare g in this way, then we could attempt to invoke the non-existent member function g.Draw().

In fact, there is a second reason why the declaration of g is an error: TheGraphicalObject constructor is not public (see Program ❑, lines 15-16). So, even if the GraphicalObject class was a concrete class, we are still prevented from instantiating it. Since the constructor is protected, it can be called by a derived class. Therefore, it is possible to instantiate Circle, Rectangle and Square.

Virtual Destructors and Dynamic Binding

Virtual Destructors

The destructor in the base class is not made virtual, then an object that might have been declared of type base class and instance of child class would simply call the base class destructor without calling the derived class destructor.

Hence, by making the destructor in the base class virtual, we ensure that the derived class destructor gets called before the base class destructor.

```
class a
{
        public:
        a(){printf("\nBase Constructor\n");}
        ~a(){printf("\nBase Destructor\n");}
};

class b : public a
{
      public:
      b(){printf("\nDerived Constructor\n");}
      ~b(){printf("\nDerived Destructor\n");}
};
int main()
{
      a* obj=new b;
      delete obj;
      return 0;
}
```

Output:
Base Constructor
Derived Constructor
Base Destructor

Dynamic binding

In OOPs **Dynamic Binding** refers to linking a procedure call to the code that will be executed only at run time. The code associated with the procedure in not known until the program is executed, which is also known as late binding.

Example:

```
#include <iostream.h>
int Square(int x)
{ return x*x; }
int Cube(int x)
{ return x*x*x; }
```

```
int main()
{
int x =10;
int choice;
do
{
cout << "Enter 0 for square value, 1 for cube value: ";
cin >> choice;
} while (choice < 0 || choice > 1);
int (*ptr) (int); switch (choice)
{
case 0: ptr = Square; break;
case 1: ptr = Cube; break;
} cout << "The result is: " << ptr(x) << endl;
return 0; }
```

Result:

```
Enter 0 for square value, 1 for cube value:0
The result is:100
```