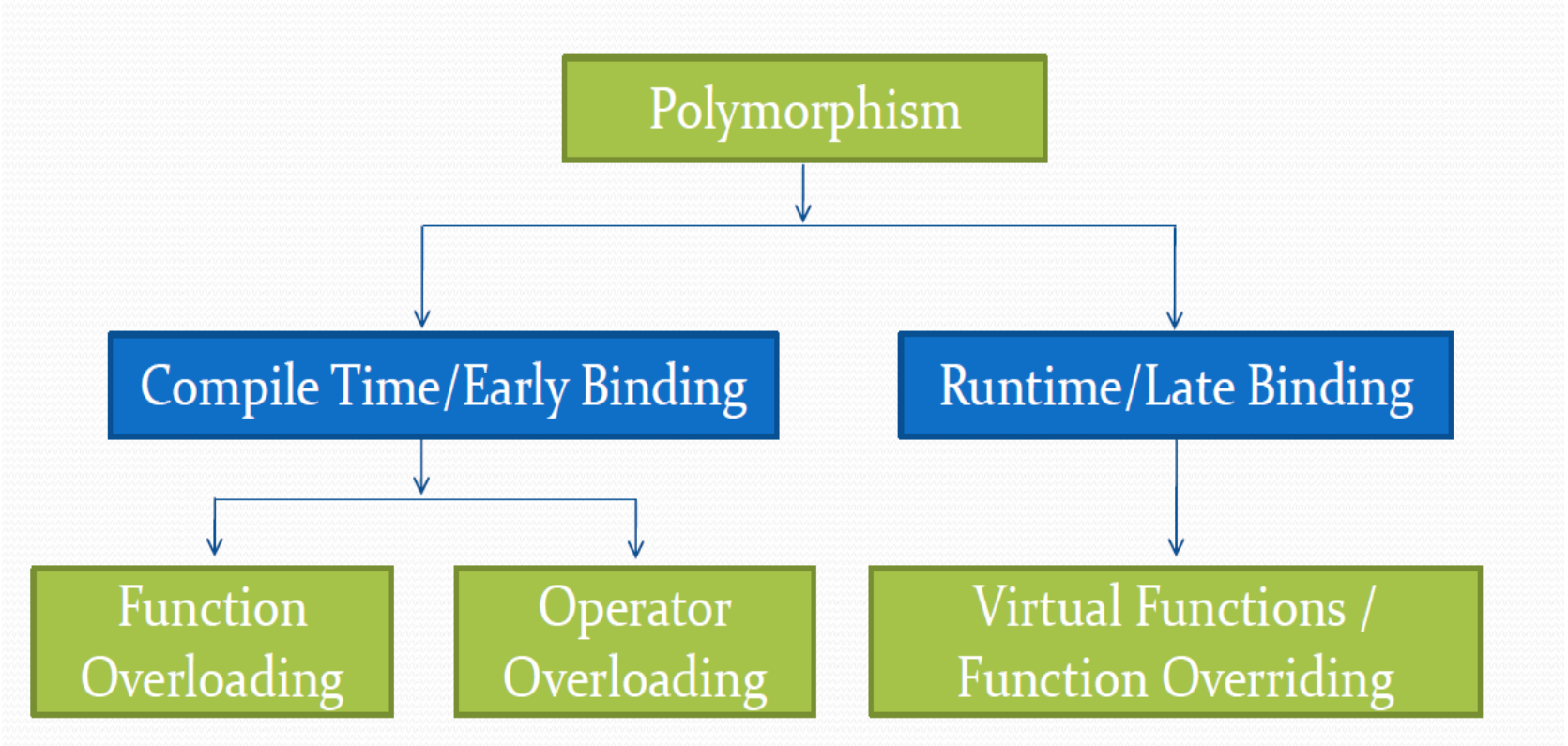
Polymorphism

Polymorphism

- The process of representing one Form in multiple forms is known as Polymorphism.
- Here one form represent original form or original method always resides in base class and multiple forms represents overridden method which resides in derived classes.
- Polymorphism is derived from 2 Greek words: poly and morphs.
- The word "poly" means many and morphs means forms.
- So polymorphism means many forms.


Polymorphism
Compile Time/Early Binding
Runtime/Late Binding
Function Overloading
Operator Overloading
Virtual Functions / Function Overriding

Runtime Binding

Typically polymorphic binding is done dynamically at runtime

Virtual functions

Compile Time Binding

Invoked through an object of the class type

Invoked through a constructor or destructor

What is Binding?

- For every function call; compiler binds or links the call to one function definition.

- This linking can happen at 2 different time
- At the time of compiling program, or
- At Runtime

Compile Time Polymorphism

- Function Overloading is an example of Compile Time Polymorphism.

- This decision of binding among several functions is taken by considering formal arguments of the function, their data type and their sequence.

- In this method object is bound to the function call at the compile time itself.

In C++ programming you can achieve compile time polymorphism

- Method Overloading

same method name is exiting multiple times in the same class with different number of parameter or different order of parameters or different types of parameters is known as method overloading

```
void MyFunction(int i)
{
        cout << “an int is passed”;
}
void MyFunction(char c)
{
        cout << “a char is passed”;
}

int main()
{
        MyFunction(10);
        MyFunction(‘x’);
}
```

Method Overriding

- Any method in both base class and derived class with same name, same parameters or signature, this concept is known as method overriding.

Requirements for Overriding

- Inheritance should be there.
- Function overriding cannot be done within a class. For this we require a derived class and a base class.
- Function that is redefined must have exactly the same declaration in both base and derived class, that means same name, same return type and same parameter list.

```
class Base
{
 public:
void show()
 {
        cout << "Base class";
 }
};

class Derived : public Base
{
 public:
void show()
{
 cout << "Derived Class";
}
 };
```

Function Call Binding with class Objects

- Connecting the function call to the function body is called Binding.

- When it is done before the program is run, its called Early Binding or Static Binding or Compile-time Binding.

Function Call Binding with class Objects

```
class Base
{ public:
void show()
 {
        cout << "Base Classt";
} };
class Derived : public Base
{ public:
void show()
{
        cout << "Derived Class";
 } };
int main()
 {
    Base b; //Base class object
    Derived d; //Derived class object
    b.show(); //Early Binding Occurs
    d.show(); }
```

Function Call Binding using Base class Pointer

But when we use a Base class's pointer or reference to hold Derived class's object, then Function call Binding gives some unexpected results.

```
class base
{
public:
void show()
{
cout << “Show from base”;
}
};
class derived : public base
{
public:
void show()
{
cout << “Show from derived”;
}
};
```

```
int main()
{
base *ptr; //Base class pointer
derived ob;
ptr = &ob;
ptr -> show(); //Early Binding
}
```

Output

```
Show from base
```

The object is of Derived class, still Base class's method is called. This happens due to Early Binding. Compiler on seeing Base class's pointer, set call to Base class's show() function, without knowing the actual object type.

Runtime Polymorphism

- In late binding; call to a function is resolved at Runtime, the compiler determines the type of object at execution time and then binds the function call to a function definition.
- Late binding is also called as Dynamic Binding or Runtime Binding.
- Virtual Functions are example of Late Binding in C++
- Runtime polymorphism is achieved using pointers.

Virtual Functions

- Virtual Function is a function in base class, which is overrided in the derived class, and which tells the compiler to perform Late Binding on this function.

- Virtual Keyword is used to make a member function of the base class Virtual.

- When you want to use same function name in both the base and derived class, then the function in base class is declared as virtual by using the virtual keyword and again re-defined this function in derived class without using virtual keyword.

```
virtual  return_type function_name()
 {
        .......
         .......
 }
```

Problem without Virtual Keyword

```
class A {
 public:
void display()
{
cout<<"Content of base class.\n";
 } };
class B : public A
 { public:
void display()
{
cout<<"Content of derived class.\n";
 } };
```

```
int main()
{
A *b; //declare pointer variables
B d; //create the object d of
b = &d; //Store Address of object d in
pointer variable
 b->display(); // try to calling display
function of Derived Class
 return 0;
}
```

Content of base class

Since early binding takes place at compile-time, therefore when the compiler saw that **b** is a **pointer of the base class**,

Using Virtual Keyword Example

```
class A {
 public:
virtual void display()
{
cout<<"Content of base class.\n";
 } };
class B : public A
 { public:
void display()
{
cout<<"Content of derived class.\n";
 } };
```

```
int main()
{
A *b; //base class pointer
B d; //derived class object
b = &d; //Store Address of object d in
pointer variable
 b->display(); // Late binding occurs
 return 0;
}
```

On using Virtual keyword with Base class's function, Late Binding takes place and the derived version of function will be called, because base class pointer pointes to Derived class object.

Content of derived class.

One More Example of Virtual Keyword-1

```
class A
{
 public: virtual void show()
 {
 cout<<"Hello base class\n"; } };
class B : public A
 {
public: void show()
{
cout<<"Hello derive class"; } };
```

```
int main()
 {
    A aobj;
    B bobj;
    A *bptr;
    bptr=&aobj;
    bptr->show(); // call base class function
    bptr=&bobj;
    bptr->show(); // call derive class function
}
```

Hello base class

Hello derive class

Another use of Virtual Keyword

- Using Virtual Keyword and Accessing Private Method of Derived class.
- We can call private function of derived class from the base class pointer with the help of virtual keyword.
- Compiler checks for access specifier only at compile time.
- So at run time when late binding occurs it does not check whether we are calling the private function or public function.

Accessing Private Method of Derived class

```
class A
{
public: virtual void show()
{
cout << "Base class Function \n"; } };
class B: public A
{
private:
virtual void show()
{
cout << "Derived class Function\n"; } };
 int main()
{
A *a; //Base class pointer
B b; //Derived class object
 a = &b;
a -> show(); //Late Binding Occurs
 }
```

Derived class Function

Pure Virtual Function

Pure virtual function is a virtual function which has no definition. Pure virtual functions are also called **abstract functions**.

```
virtual void sound() = 0;
```

Abstract Class

An **abstract class** is a class whose instances (objects) can't be made. We can only make objects of its subclass (if they are not abstract). Abstract class is also known as **abstract base class**.

An abstract class has at least one abstract function (pure virtual function).

```
class Employee // abstract base class
{
   virtual int getSalary() = 0;
};
class Developer : public Employee
 {
        int salary;
        public: Developer(int s)
        {
                salary = s;
        }
        int getSalary()
        {
                return salary;  } };
```

```
class Driver : public Employee
 {
        int salary;
        public: Driver(int t)
        {
            salary = t;
         }
        int getSalary()
        {
            return salary;
        } };
```

```
#include<iostream>
using namespace std;
 class Base
{
   int x;
public:
    virtual void fun() = 0;
    int getX() { return x; }
};

// This class inherits from Base and implements fun()
class Derived: public Base
{
    int y;
public:
    void fun() { cout << "fun() called"; }
};

int main(void)
{
    Derived d;
    d.fun();

    return 0; }
```

Output: fun() called

We cant create object for abstract class

We can have pointers and references of abstract class type.

```
class Base
{
public:
    virtual void show() = 0;
};

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

int main(void)
{
    Base *bp = new Derived();
    bp->show();
    return 0;

}
```

In Derived

If we do not override the pure virtual function in derived class, then derived class also becomes abstract class.

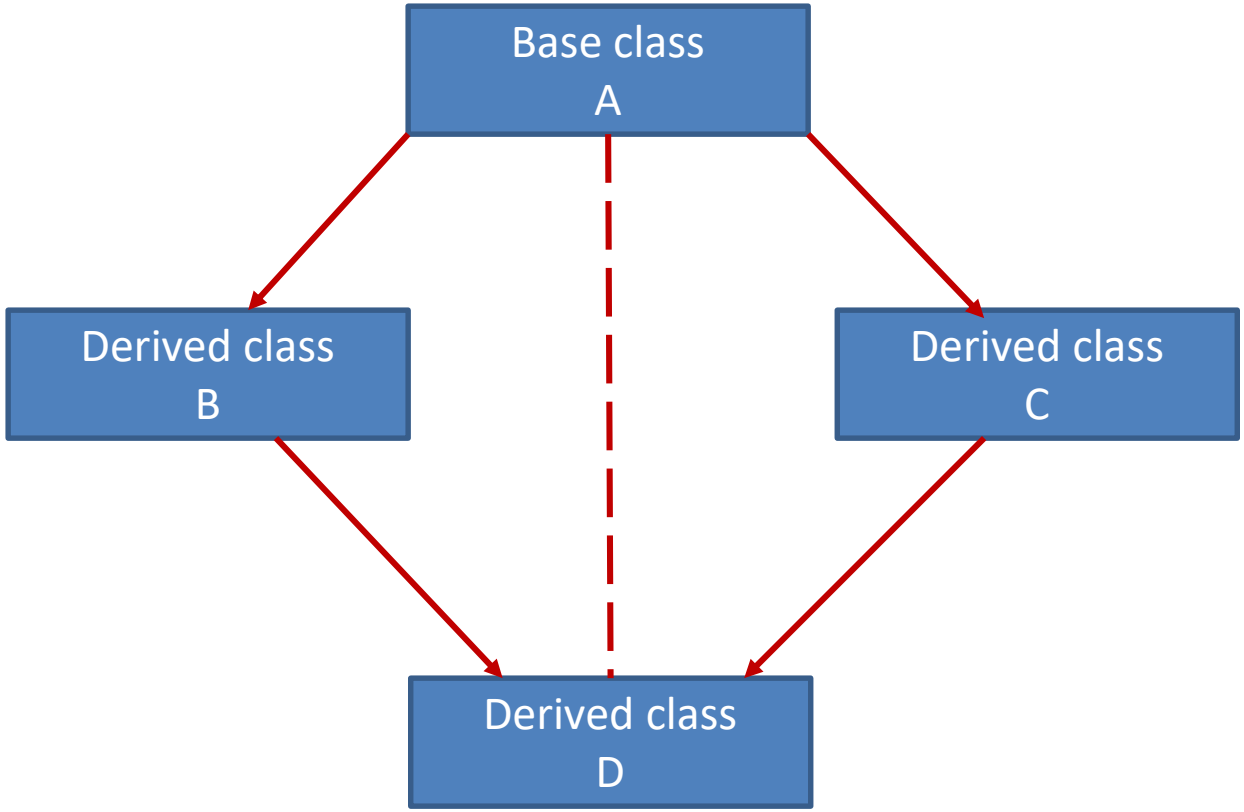
```
class Base
{
public:
    virtual void show() = 0;
};

class Derived : public Base { };

int main(void)
{
  Derived d;
  return 0;

}
```

Compiler Error: cannot declare variable 'd' to be of abstract type 'Derived' because the following virtual functions are pure within 'Derived': virtual void Base::show()

```
int main()
 {
      Developer d1(5000);
      Driver d2(3000);
      int a, b;
      a = d1.getSalary();
      b = d2.getSalary();
     cout << "Salary of Developer : " << a << endl;
     cout << "Salary of Driver : " << b << endl;
     return 0;
     }
```

Subclasses of an abstract base class must define the abstract method, otherwise, they will also become abstract classes.

In an abstract class, we can also have other functions and variables apart from pure virtual function.



Two copies of *Base* are included in *D*.
This causes ambiguity when a member of *Base* is directly used by *D*.

```
class ClassA
{
        public: int a;
 };
class ClassB : public ClassA
{
         public: int b;
};
 class ClassC : public ClassA
{
        public: int c;
 };
class ClassD : public ClassB, public ClassC
 {
        public: int d;
};
```

```
void main()
{
ClassD obj;
obj.a = 10;
obj.a = 100;
obj.b = 20;
obj.c = 30;
obj.d = 40;
cout<< "\n A : "<< obj.a;
cout<< "\n B : "<< obj.b;
cout<< "\n C : "<< obj.c;
cout<< "\n D : "<< obj.d;

}
```

```
class Base {
public:
   int i;
};
class D1 : public Base {
public:
   int j;
};
class D2 : public Base {
public:
   int k;
};
```

```
class D3 : public D1, public D2 {
  // contains two copies of ‘i’
};
void main() {
   D3 obj;
   obj.i = 10; // ambiguous, compiler error
   obj.j = 20; // no problem
   obj.k = 30; // no problem
   obj.D1::i = 100; // no problem
   obj.D2::i = 200; // no problem
}
```

```
class Base {
public:
  int i;
};
class D1 : virtual public Base {
public:
  int j;
}; // activity of D1 not affected
class D2 : virtual public Base {
public:
  int k;
}; // activity of D2 not affected
```

```
class D3 : public D1, public D2 {
  // contains only one copy of 'i'
}; // no change in this class definition
void main() {
  D3 obj;
  obj.i = 10; // no problem
  obj.j = 20; // no problem
  obj.k = 30; // no problem
  obj.D1::i = 100; // no problem, overwrites '10'
  obj.D2::i = 200; // no problem, overwrites '100'
}
```