29-06-2024

Differences between Method overloading & Method overrifing

METHOD OVERLOADING	METHOD OVERRIDING
-> Re-using same name for Super & sub-class methods with different number of arguments. [OR] with different type of arguments with [OR] without same return type within a class is called method overloading	-> Re-using same name for Super & sub-class methods with same number of arguments, with same type of arguments & with same return type is called Method overriding
-> No of arguments / type of arguments must be different	-> No of arguments / type of arguments must be same
-> It doesn't signify return() type	-> It signifies return() type
-> No method hides no another method	-> Sub class method hides Super class method
-> Inheritence is not involved	-> Inheritence is involved
-> It is a compile time / static Polymorphism	-> It leads to Dynamic binding

```
-> using namespace std;
   # include <iostream>
   class abc                                    .cpp
   {
     private: int a, b;
     public: void read ()
              {
                 cout << "enter a, b";
                 cin >> a >> b;
              }
              void write ()
              {
                 cout << a << "   " << b;
              }
   };
   class xyz : public abc
   {
     private: int x, y;
     public: void read ()
              {
                 cout << "enter x, y";
                 cin >> x >> y;
              }
              void write ()
              {   cout << x << "   " << y;
              }
   };
   main()
   {  xyz p;
    p.read(); p.write(); p.abc::read(); p.abc::write();
   }
```

COMPILE TIME BINDING:-

```
-> Using namespace std;
   # include <iostream>
   class abc
    {
     public: void rose ()
               {
                 cout << " abc rose ";
               }
   };

   class xyz : public abc
   {
    public:  void rose ()
               {
                 cout << "xyz rose";
               }
    };

   main ()
   {
        abc *p, r;
        xyz  q;
        p = & q;
        p -> rose ();
        p = & r;
        p -> rose ();
   }.
```

o/p :-

```
abc rose
abc rose
```

o/p :-

```
abc rose
abc rose
```

If we run above program abc rose is called, because the pointer is abc class pointer & the binding is COMPILE TIME binding.

virtual (): postpones the decision at binding at compile time & decision is taken at Run-time of the program.

VIRTUAL() SYNTAX:-

```
    virtual name (----)
             {
                 body
             }
```

RUNTIME BINDING :-

```
-> using namespace std;
   # include <iostream>
   class abc
   {
       public: virtual void rose ()
               {
                   cout << "abc rose";
               }
   };
   class xyz : public abc
    {
       public: void rose ()
               {
                   cout << "xyz rose";
               }
    };
```

```
main ( )
{
    abc  *p, r;
    xyz  q;
    p = &q;
    p -> rose ( );
    p = &r;
    p -> rose ( );
}
```

O/P :-

```
xyz rose
abc rose
```