Object Oriented Programming with C++

Polymorphism

Lecturer : Astha Sharma

CHAPTER 5

Polymorphism

(8 hrs)

Polymorphism in programming language

Polymorphism in real life

As we all know, polymorphism is taking many forms. We can relate it with the example of a man, who is a husband, an employee at an office, son, father hence, man can take different forms.

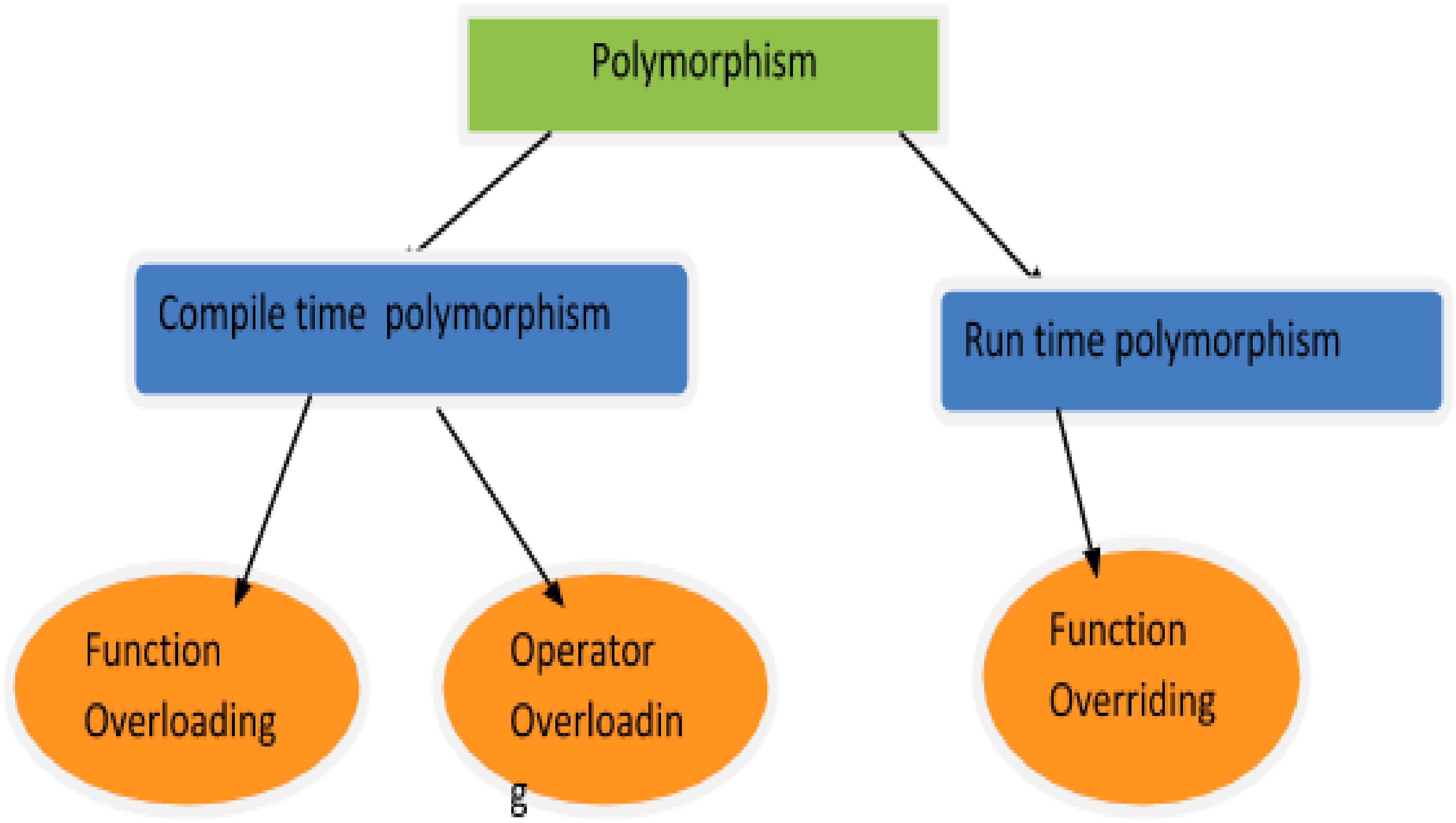
Suppose a teacher at a school at teaching different subjects in different classes like he can be an English teacher for class 9 and the same teacher can be a science class for class 10 hence taking various forms at different times.

Polymorphism in C++

Polymorphism is a feature of OOP that allows the object to behave differently in different conditions. In C++ we have two types of polymorphism:

1) Compile time Polymorphism – This is also known as static (or early) binding.

2) Runtime Polymorphism – This is also known as dynamic (or late) binding.


Polymorphism
Compile time polymorphism
Run time polymorphism
Function Overloading
Operator Overloadin g
Function Overriding

1) Compile time Polymorphism

Function overloading and Operator overloading are perfect example of Compile time polymorphism.

Function overloading example:

In this example, we have two functions with same name but different number of arguments. Based on how many parameters we pass during function call determines which function is to be called, this is why it is considered as an example of polymorphism because in different conditions the output is different. Since, the call is determined during compile time thats why it is called compile time polymorphism.

```
#include <iostream>
using namespace std;
class Add
{ public:
int sum(int num1, int num2)
{ return num1+num2;
}
 int sum(int num1, int num2, int num3)
{ return num1+num2+num3; }
};
int main()
{ Add obj;
//This will call the first function
cout<<"Output: "<<obj.sum(10, 20)<<endl;
//This will call the second function
cout<<"Output: "<<obj.sum(11, 22, 33)<<endl;
return 0;
}
```

Output:

```
Output: 30
Output: 66
```

Operator overloading example:

•In C++, we can make operators work for user-defined classes. This means C++ can provide the operators with a special meaning for a data type; this ability is known as operator overloading.

•We can overload most of the inbuilt operators in c++ such as +,-, and many more defined operator

•In this example we take two integer values feet and inches and negate it respectively using – operator.

•Syntax:
Class_name operator symbol (information)
Where;
Operator is keyword,
Symbol is +,-,etc,
Information is info for the operator, it can or cannot exist.

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

 class Distance
{
private:      int feet;            // 0 to infinite
              int inches;          // 0 to 12
public:
 // required constructors
            Distance() {        feet = 0;
                                 inches = 0;}
             Distance(int f, int i)
            {       feet = f;
                    inches = i;     }
// method to display distance
            void displayDistance()
            {cout << "F: " << feet << " I:" << inches<<endl;   }

// overloaded minus (-) operator
            Distance operator - ()
            {       feet = -feet;       inches = -inches;
            return Distance(feet, inches);      }
};
```

```
int main()
{
Distance D1(11, 10), D2(-5, 11);
-D1;                    // apply negation
D1.displayDistance();    // display D1
-D2;                    // apply negation
D2.displayDistance();    // display D2
return 0;
}
```

OUTPUT:

```
F: -11  I:-10
F: 5  I:-11
```

```
#include <iostream>
using namespace std;
class Distance{
int feet, inch;
public:    Distance(int a, int b)
 {        feet = a;        inch = b;    }

void printDistance()    {
cout<<"Feet:"<<feet<<endl;
cout<<"Inch:"<<inch<<endl;    }

Distance operator *()
{
feet *= feet;
inch *= inch;    }};

int main()
{ Distance d(1,2);
d.printDistance();
*d;
d.printDistance();
return 0;}
```

```
int main()
{
Distance D1(11, 10), D2(-5, 11);
*D1;                    // apply negation
D1.displayDistance();    // display D1
*D2;                    // apply negation
D2.displayDistance();    // display D2
return 0;
}
```

```
OUTPUT:
F: -11  I:-10
F: 5  I:-11
```

2) Runtime Polymorphism

Function overriding is an example of Runtime polymorphism.

Function Overriding:
When child class declares a method, which is already present in the parent class then this is called function overriding, here child class overrides the parent class.
In case of function overriding we have two definitions of the same function, one is in parent class and one in child class. The call to the function is determined at **runtime** to decide which definition of the function is to be called, that's the reason it is called runtime polymorphism.

```
#include <iostream>
using namespace std;
class A
{
 public:
void disp()
{ cout<<"Super Class Function"<<endl; }
 };
 class B: public A
{
 public:
void disp()
{ cout<<"Sub Class Function"<<endl;}
 };
 int main()
{ //Parent class object
A obj;
obj.disp();
 //Child class object
B obj2;
obj2.disp();
return 0; }
```

Output:

```
Super Class Function
Sub Class Function
```

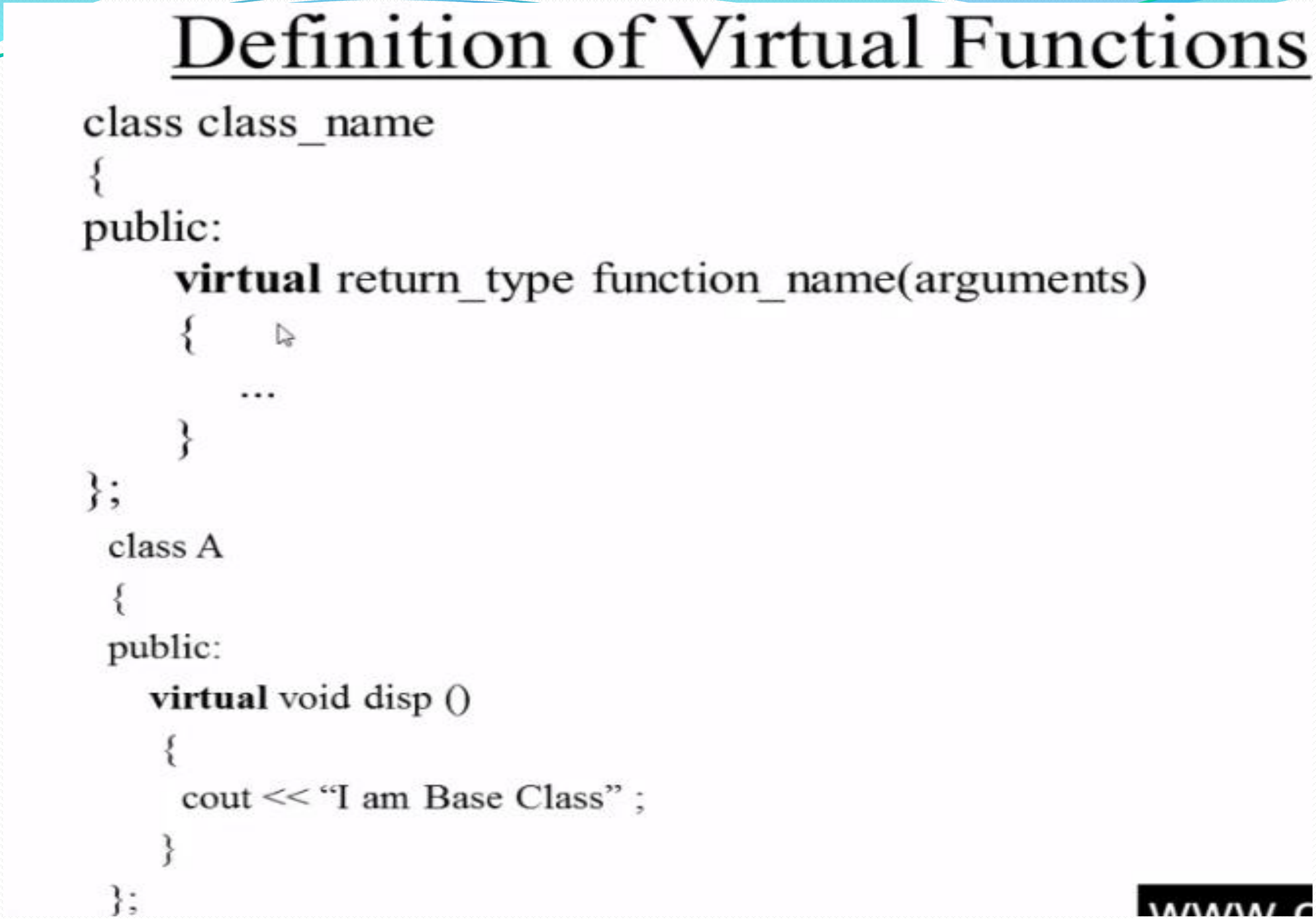
```cpp
class Base {
    public:
      void print() {
          // code
      }
};

class Derived : public Base {
    public:
      void print() {
          // code
      }
};

int main() {
    Derived derived1, derived2;

    derived1.print();

    derived2.Base::print();

    return 0;
}
```

Virtual Function:

- It is a member function which is declared within a base class and is redefined(overridden) by a derived class.
- It ensures that the correct function is called for an object , regardless of the type of reference(or pointer) used for function call.
- They are mainly used to achieve “run-time polymorphism”.
- Functions are declared using a “**virtual**” keyword in base class.
- The resolving of function call is done at run-time.

Rules for virtual functions:

1. Virtual functions cannot be static.
2. A virtual function can be a friend function of another class.
3. Virtual functions should be accessed using pointer or reference of base class type to achieve run-time polymorphism.
4. The prototype of virtual functions should be the same in the base as well as derived class.
5. They are always defined in the base class and overridden in a derived class.
6. A class may have virtual destructor but it cannot have a virtual constructor.



Definition of Virtual Functions

```
class class_name
{
public:
      virtual return_type function_name(arguments)
      {
            ...
      }
};
 class A
 {
 public:
    virtual void disp ()
     {
      cout << "I am Base Class" ;
     }
 };
```

www.c

Example:

```
#include<iostream>
using namespace std;
class Base
{public:
virtual void print()
{cout<<“Print base class”<<endl;
}
void show()
{cout<<“Show base class”<<endl;
}};
class Derived : public Base{
public:
void print(){
cout<<“Print derived class”<<endl;}
void show()
{cout<<“Show derived class”<<endl;}
};
```

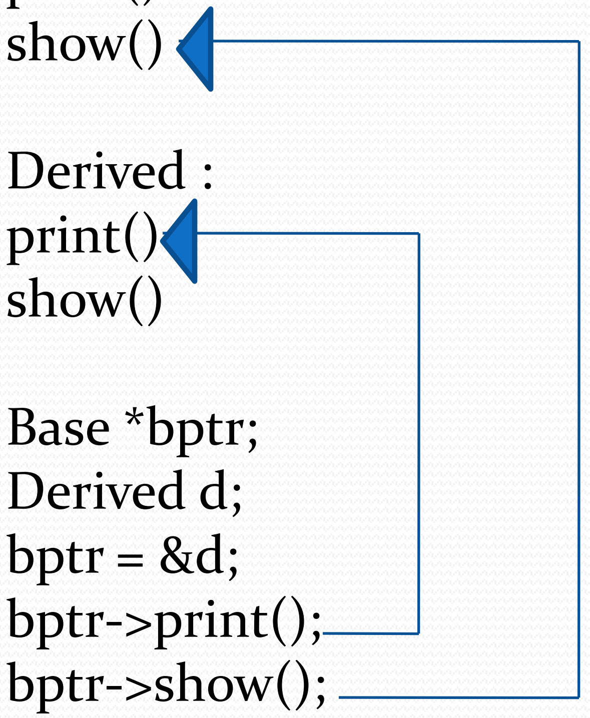
```
int main()
{Base *bptr;
Derived d;
bptr = &d;

bptr->print();
//virtual function bind at runtime
bptr->show();
//non virtual function , bind at compile
      time
}
```

```
o/p:
Print derived class
Show base class
```

Base:
print()
show()

Derived :
print()
show()

Base *bptr;
Derived d;
bptr = &d;
bptr->print();
bptr->show();



Example 2:

```
#include<iostream>
using namespace std;
class Base
{
public:
virtual void func1()
{cout<<"base1"<<endl;}

void func2()
{cout<<"base2"<<endl;}

virtual void func3()
{cout<<"base3"<<endl;}

 void func4()
{cout<<"base4"<<endl;}
};
```

```
class Derived: public Base
{
public:
void func1()
{cout<<"derived1"<<endl;}

void func2()
{cout<<"derived2"<<endl;
        }

void func3()
{cout<<"derived3"<<endl;
        }

 void func4()
{cout<<"derived4"<<endl;
        }

};
```

```
int main()
{Base *Bptr;
Derived d;
Bptr = &d;

Bptr->func1();
Bptr->func2();
Bptr->func3();
Bptr->func4();

return 0;
}
```

o/p:

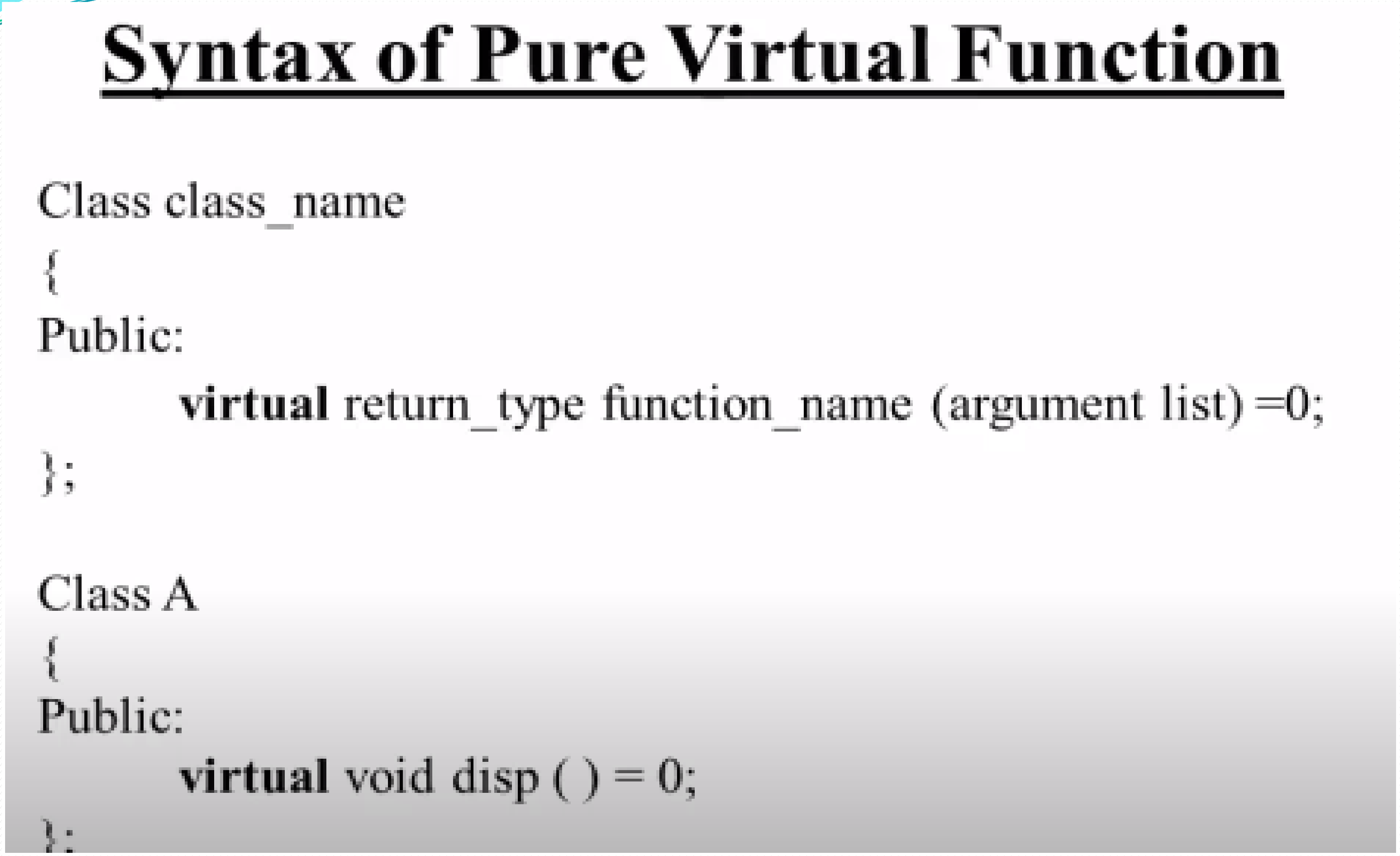
```
derived1
base2
derived3
base4
```

Pure Virtual Function:

- It is a specific type of virtual function.
- A virtual function with no body is called pure virtual function.
- It exists purely in base class only to be overridden by the derived class function.
- A base class only specifies that there is a function with its name and parameters which will be implemented by any of its derived classes or some other classes in the same inheritance hierarchy.

Rules for pure virtual functions:

1. It can exist only in base class.
2. It must have no body (or it has no definition in abstract base class).
3. The derived class must have the same function in its body.
4. The function in derived class replaces the pure virtual function in base class.



Syntax of Pure Virtual Function

```
Class class_name
{
Public:
        virtual return_type function_name (argument list) =0;
};

Class A
{
Public:
        virtual void disp ( ) = 0;
};
```

Example of pure virtual function:

```
#include <iostream>
using namespace std;
class A{
    public:    virtual void disp() =0;
};
class B:public A{
public:    void disp()
{        cout<<"\nNamaste";}
}; class C:public A{
public:    void disp()
{        cout<<"\nhi";}
};
int main()
{   A *objptr;
B objb;
objptr = &objb;
objptr->disp();
return 0;
}
```

```
o/p:
Namaste
```

Abstract Class

- A special type of class which contains at least one pure virtual function is called abstract base class or simply abstract class.

- In the above example Class A is an **abstract class** because it has one pure virtual function called disp().

- Class A is also called **Abstract Base class**.

Pure polymorphism

- A class hierarchy that is defined by inheritance creates a related set of user-defined types, all of whose objects may be pointed at by a base class pointer.

- By accessing the virtual function through this pointer, C++ selects the appropriate function definition at runtime.

- This is a form of polymorphism called **pure polymorphism**.

Deferred Method

- A pure virtual function present in base class must be overridden in derived class using the same function name with some body.
- Such pure virtual function is also termed as “**Deferred method**”.
- In the above example, **disp()** is a deferred method.

Polymorphic Variable

- To make it possible for accessing the overridden derived class functions via the base class pointer, we have to declare such functions as virtual.
- Then, we can create or point to objects of different derived classes at run-time by the help of base class pointer.
- This allows a pointer of base class to act as a "Polymorphic Variable" allowing it to behave differently based upon which type of derived class object it points to.

Example:

```
A *objptr;
B objb;
objptr = &objb;
objptr->disp();
```

Here , objptr is a polymorphic variable(pointing to objb object of class B).
It can also point to any derived class object when required.

Type conversion

Type conversion refers to changing an entity of one class type, expression, function argument or return value into another.

There are two types of type conversion:

1. Implicit Type conversion
2. Explicit Type conversion

Implicit Type conversion:

- The type of data to the right of an assignment operator(=) is automatically converted to the type of variable to the left.
- Done by the compiler on its own , without any external triggers from the user.

For example 1:

```
int a=5;
char b = ‘c’;
int result = a + b;
cout<<“result is : ”<<result<<endl;

//output is : result is : 104 as ‘c ’ is converted to its ASCII value 99 and added with 5
```

For example 2:

```
#include<iostream>
using namespace std;
int main()
{
int x= 10;
char y ='a';
x=x+y;
//y is implicitly converted into int ASCII value of 'a' =97


float z=x+1.5;
//x is implicitly converted into float}
cout<<"x="<<x<<endl;
cout<<"y="<<y<<endl;
cout<<"z="<<z<<endl;
return 0;
}
```

```
o/p:
x=107
y=a
z=108.5
```

Explicit Type conversion:

- The process is also called type casting and it is user defined.
- Here the user can typecast the result to make it of a particular datatype.
- In C++ , it can be done in two ways:

1. Converting by assignment
2. Converting using cast operator

1. Converting by assignment:

- This is done by explicitly defining the required type in front of the expression in parenthesis.
- This can be also considered as forceful casting.

- **Syntax:**

(type) expression
where,
 type indicates the data types to which the final result is converted.

For example:

```
#include<iostream>
using namespace std;
int main()
{
double x=1.2;
int sum =int (x)+1;
//explicit conversion of x from double to int
cout<<"sum="<<sum;
return 0;
}
```

```
o/p
sum = 2
```

2. Converting using cast operator:

- A cast operator is an unary operator which forces ont datatype to be converted into another datatype.
- C++ supports 4 types of casting:

1. Static casting
2. Dynamic casting
3. Constant casting
4. Reinterpret casting

For example:

```
#include<iostream>
using namespace std;
int main()
{
float f=3.5;
int b = static_cast<int>(f);//static casting
cout<<b<<endl;}
```

```
o/p
3
```

Advantages of type conversion:

1. This is done to take advantage of certain features of type hierarchies or type representation.
2. It helps to compute expressions containing variables of different datatypes.

This pointer:

- C++ uses a unique keyword called “this” to represent an object that invokes a member function.

- “this” is a pointer that points to the object for which “this” function was called.

- For example, the function call A.max() will set the pointer “this ” to the address of the object A.

Example:

```
class ABC
{
int a;
-----
};
```

- The private variable ‘a’ can be used directly inside a member function like a=123;

- We can also use the following statemet to do the same job:

```
this->a=123;
```

Example:

```
#include<iostream>
using namespace std;
class A
{
int a;

public:
void input(int a)
{this->a=a;}
void output()
{cout<<this->a;}
};
int main()
{A obj;
obj.input(2);
obj.output();
}
```

```
o/p
2
```

Example:

```
#include<iostream>
using namespace std;
class X
{int a,b;
public:
void input()
{this->a=10; this->b=20;}
void output()
{cout<<a<<endl;
cout<<b<<endl;}};
int main()
{X obj;
obj.input();
obj.output();
}
```

```
o/p
10
20
```