# Software Development Fundamentals (SDF) – II
## ODD 2020

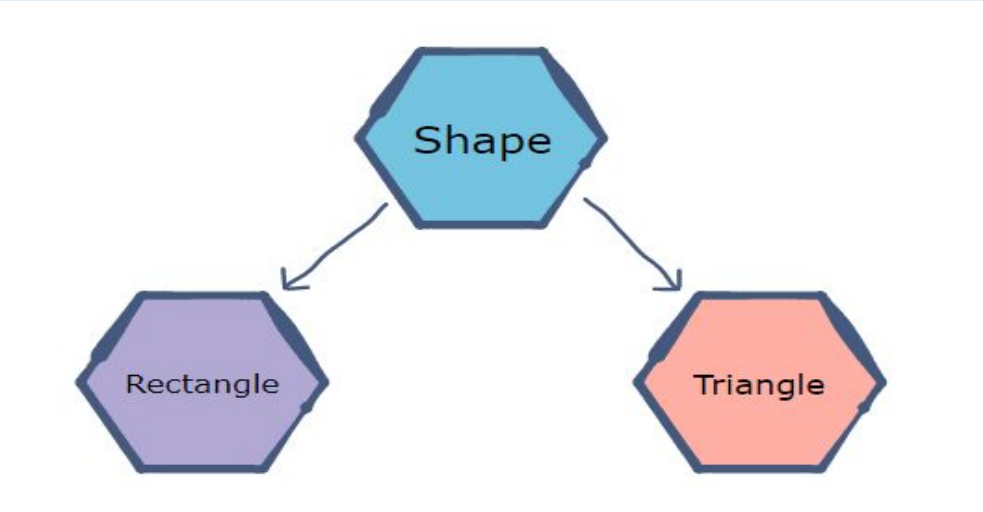

## Lecture # 16

## Abstract Class/Dynamic Dispatch/RTTI

**Jaypee Institute of Information Technology (JIIT)**
**A 10, Sector 62, Noida**

# Abstract Class

1. Often in real life, it is needed to have the base class at abstract level, *i.e.* do not want to create the objects of base class. However, the derived class objects can be created which are inheriting the properties of base class.




In real life, shape is an abstract entity. However, objects are rectangular shaped or triangular shaped, *etc.* and inherits the properties of shape.

2. In OOPS (C++), a class is made as abstract class by making at least one virtual function of that class as pure virtual function (*i.e.* undefined/without implementation in that class). *e.g.* virtual void f1() = 0;

2. The pure virtual function will/should be defined later in child class

2. We cannot create objects of abstract classes, however we can create the object of the child class in which the pure virtual function of the base class is defined.

**Example 1:** Abstract class in Single Inheritance

```
class A
{
    public:
       virtual void f1() = 0;        //Pure virtual function, i.e. A is abstract class
       virtual void f2() { cout<<“1”;}
};
class B : public A
{
     public:
       void f1() { cout<<“2”;}     // Pure virtual function of base class is defined in derived class
};

A o1;

// We can not create object of class A as it is abstract class

A *o2 = new B;
B o3;
B *o4 = new B;

// o2, o3, & o4 can be created
```

**Example 2:** Abstract class in multi-level inheritance

```
class A
{
    public:
       virtual void f1() = 0;        //Pure virtual function i.e. A is abstract class
       virtual void f2() { cout<<“1”;}
};
class B : public A
{
     public:
       void f2() { cout<<“2”;}     // Pure virtual function, f1() of base class A is undefined in class B. So B is also an abstract class
};
class C : public B
{
     public:
       void f1() { cout<<“3”;}     // Pure virtual function, f1() of base class is defined in child class, C. So C is non-abstract class
};
```

A o1; B o2; A *o3 = new B **// o1, o2, & o3 cannot be created**

A *o4 = new C; B *o5 = new C; C o6; C *o7 = new C;  // o4, o5, o6, & o7 can be created

## Virtual Table and Dynamic Dispatch

1. A virtual method table (VMT) is a mechanism used in a programming language to support dynamic dispatch (or run-time method binding).

1. Dispatching (can be static or dynamic) just refers to the action of finding the right function to call.

1. When we define a method inside a class, the compiler remembers its definition and execute it every time a call to that method is encountered

```
class A
{
    public:
       void f1();
};
void A::f1() { cout<<“1”;}
```

4. Here, the compiler will create a routine for f1() and remember its address. This routine will be executed every time the compiler finds a call to f1() on an instance of A. Here only one routine exists per class method, and is shared by all instances of the class. This process is known as static dispatch or early binding: the compiler knows which routine to execute during compilation.

## Virtual Table and Dynamic Dispatch contd..

5. There are cases where it is not possible for the compiler to know which routine to execute at compile time. *e.g.* virtual functions which can be overridden by subclasses and compiler needs to know which routine to execute

```
class A
{
    public:
       virtual void f1();
       virtual void f2();
};
void A::f1() { cout<<“1”;}
void A::f2() { cout<<“2”;}
```

```
class B : public A
{
    public:
       void f1();  // f1() is overridden
};
void B::f1() { cout<<“3”;}
```

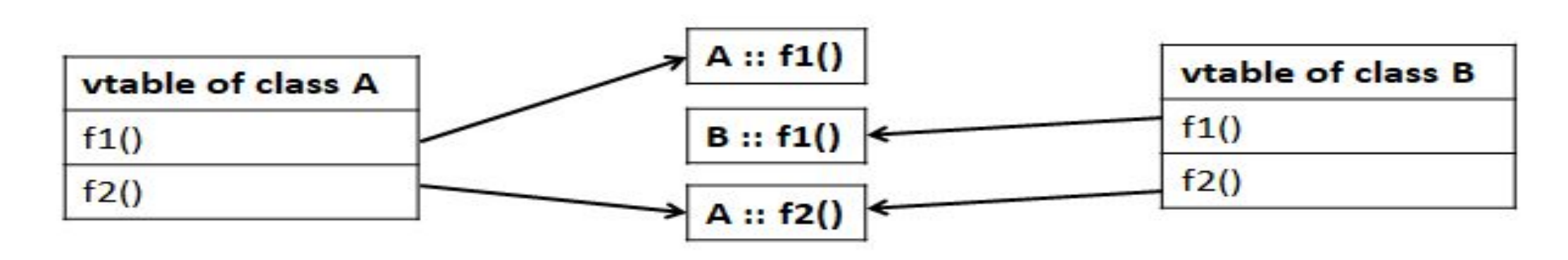
Lets call the function f1() as follows: A *o1 = new B; o1 □ f1();

6. If we use static dispatch, the call o1->f1() would execute A::f1(), since (from the point of view of the compiler) o1 points to an object of type A. This would be wrong, off course, because o1 actually points to an object of type B and B::f1() should be called instead.

6. As virtual functions can be redefined in subclasses, calls via pointers (or references) to a base type can not be dispatched at compile time. The compiler has to find the right function definition (*i.e.* the most specific one) at runtime. This process is called dynamic dispatch or late method binding.

## Virtual Table and Dynamic Dispatch contd..

8. For every class that contains virtual functions, the compiler constructs a virtual table, *i.e.* vtable. The vtable contains an entry for each virtual function accessible by the class and stores a pointer to its definition. Only the most specific function definition callable by the class is stored in the vtable. Entries in the vtable can point to either functions declared in the class itself (*e.g.* B::f1()), or virtual functions inherited from a base class (*e.g.* B::f2()).

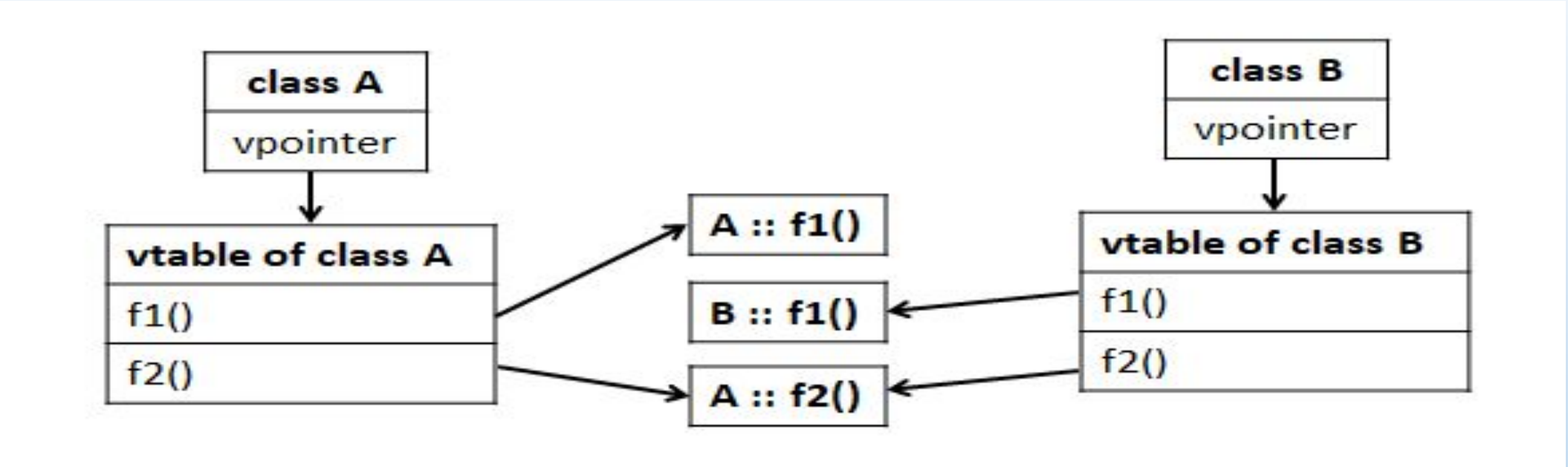



9. vtables exist at the class level, i.e. there exists a single vtable per class, and is shared by all instances.

9. Every time the compiler creates a *vtable* for a class, it adds an extra argument to it: a pointer to the corresponding virtual table, called the *vpointer*.

9. vpointer is just another class member added by the compiler and increases the size of every object that has a vtable by sizeof(vpointer)

## Virtual Table and Dynamic Dispatch contd..




12. When a call to a virtual function on an object is performed, the *vpointer* of the object is used to find the corresponding *vtable* of the class. The function name is used as index to the *vtable* to find the correct (most specific) routine to be executed.

# RTTI (Run-time type Information) in C++

1. It provides information about an object's data type at runtime
2. It is available only for the classes which have at least one virtual function
3. It allows the type of an object to be determined during program execution
4. The RTTI is provided using two operators:

**typeid** □ it returns the actual type of the object referred to by a pointer/reference

**dynamic_cast** □ it converts from a pointer/reference to a base type to a pointer/ reference to a derived type

# RTTI (Run-time type Information) in C++ contd..

```
#include<iostream>
#include <typeinfo>
using namespace std;
class A
{
public:
      virtual void f1() {}
};
class B: public A
{
};
int main()
{
    A *o1 = new B;
    B *o3, *o2 = new B;
    o3 = dynamic_cast<B*>(o1);
    if(typeid(o3) == typeid(o2))
        cout << "Dynamically casted the same type of object";
    else
        cout << "casting from A* to B* cannot be done";
    getchar();
    return 0;
}
```

Here, o1 which is a pointer of A type and pointing to B type object is casted to B type and assigned into o3

Typeids of o2 and o3 are B type.

Hence, OUTPUT:

Dynamically casted the same type of object

** typeinfo (header file) to be included in the program to use typeid function

# End of Lecture # 16