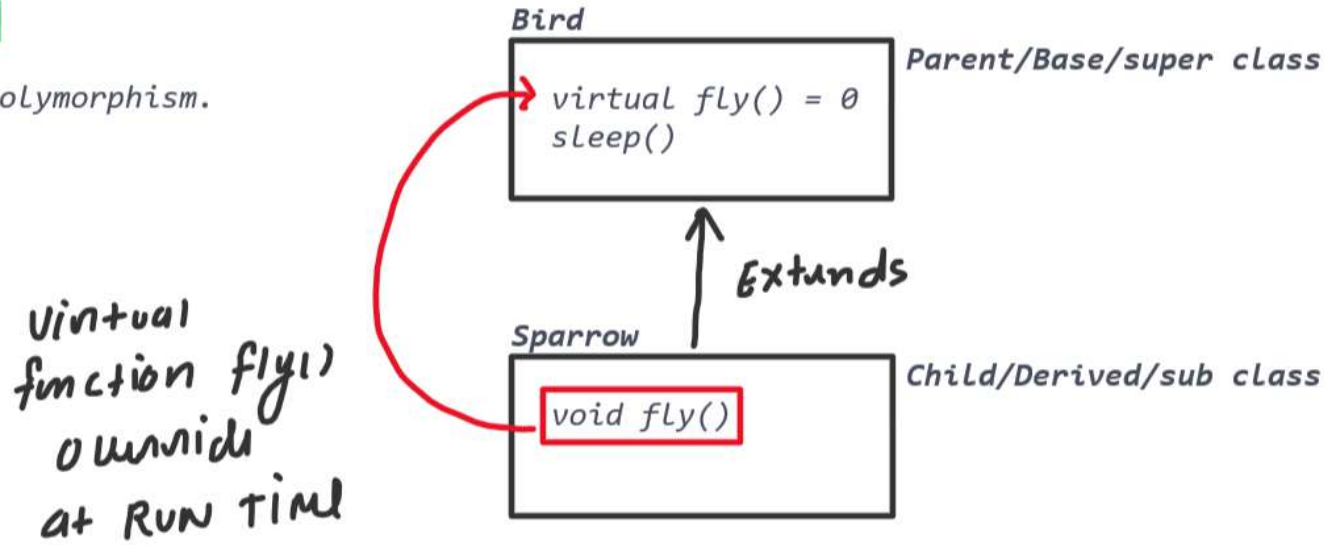
28/10/2023

Object Oriented Programming Class 03 Homework

1. Virtual CTOR Vs Virtual DTOR

What is virtual in C++?

Way to achieve Runtime polymorphism.

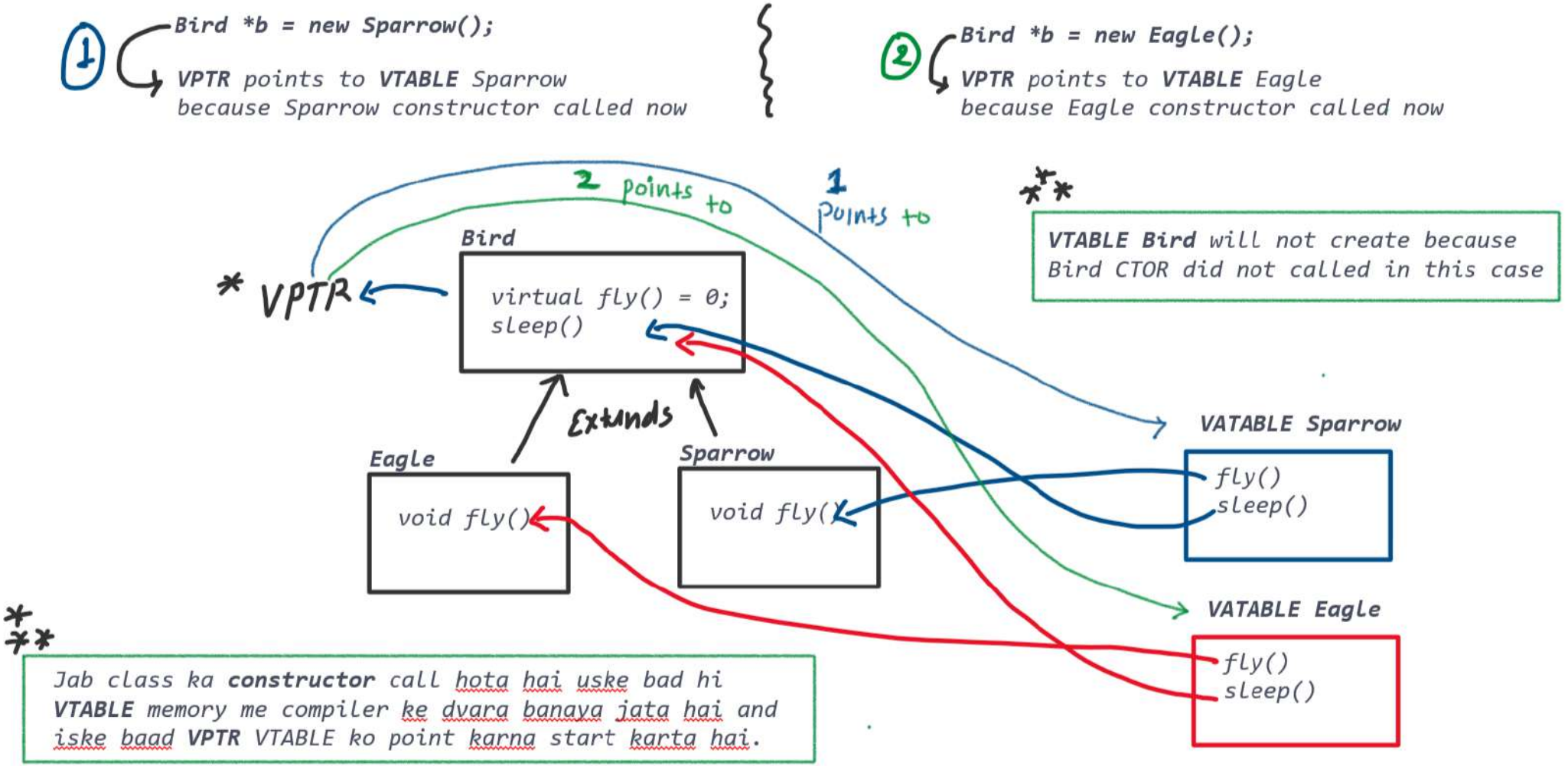


How does 'virtual' work?

1. **VTables**
    1. Formed for every class having at least one virtual function and for its derived classes.
    2. It is static arrays, hence one instance for a class.
    3. **VPtr** (a hidden member pointer) is added by compiler to classes with virtual and its derived classes.
    4. Depending upon the object type **VPtr** is bonded to a **Vtable**
    (**static array/table class name/virtual table/function Pointer table**).

2. **VTables are created at compile time.**

3. **When object of a particular type is created at runtime. There will be a VPtr which will be initialised to point to a static VTable at the time of construction.**


1
Bird *b = new Sparrow();
VPTR points to VTABLE Sparrow
because Sparrow constructor called now
2
Bird *b = new Eagle();
VPTR points to VTABLE Eagle
because Eagle constructor called now
2 points to
1 points to
* VPTR
Bird
virtual fly() = 0;
sleep()
Extends
Eagle
void fly()
Sparrow
void fly()
VATABLE Sparrow
fly()
sleep()
VATABLE Eagle
fly()
sleep()
** VTABLE Bird will not create because Bird CTOR did not called in this case
*** Jab class ka constructor call hota hai uske bad hi VTABLE memory me compiler ke dvara banaya jata hai and iske baad VPTR VTABLE ko point karna start karta hai.

Can we make virtual CTOR?

1. NO.

2. Constructor cannot be virtual, because when constructor of a class is executed there is no virtual table in the memory, means no virtual pointer defined yet. So, the constructor should always be non-virtual.

3. A virtual call is a mechanism to get work done given partial information.
In particular, "virtual" allows us to call a function knowing only any interfaces and not the exact type of the object.
To create an object you need complete information.

In particular, you need to know the exact type of what you want to create.
Consequently, a "call to a constructor" cannot be virtual.

Can we make virtual CTOR?

1. YES.
2. It is important to handle proper destruction of Derived class.

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

class Base
{
    public:
        Base()
        {
            cout<<"Base CTOR Called\n";
        }

        ~Base()
        {
            cout<<"Base DTOR Called\n";
        }
};

class Derived: public Base
{
    public:
        Derived()
        {
            cout<<"Derived CTOR Called\n";
        }

        ~Derived()
        {
            cout<<"Derived DTOR Called\n";
        }
};

int main()
{
    Base *b = new Derived();
    delete b;
    return 0;
}
```

→ Leakage memory

Non virtual

→ Called

→ Did Not Called

OUTPUT:

- Base CTOR Called
- Derived CTOR Called
- Base DTOR Called

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

class Base
{
    public:
        Base()
        {
            cout<<"Base CTOR Called\n";
        }

        virtual ~Base()
        {
            cout<<"Base DTOR Called\n";
        }
};

class Derived: public Base
{
    public:
        Derived()
        {
            cout<<"Derived CTOR Called\n";
        }

        ~Derived()
        {
            cout<<"Derived DTOR Called\n";
        }
};

int main()
{
    Base *b = new Derived();
    delete b;
    return 0;
}
```

→ FREE ✓ memory

Virtual

→ Called

→ Called

OUTPUT:

- Base CTOR Called
- Derived CTOR Called
- Derived DTOR Called
- Base DTOR Called

2. Abstraction in C++

1. Delivering only essential information to the outer world while masking the background details.
2. It is a design and programming method that separates the interface from the implementation.
3. Real life e.g., various functionalities of **AirPods** but don't know the actual implementation/working.

Example: To drive a car, one only needs to know the driving process and not the mechanics of the car engine.

Abstraction in Header files:

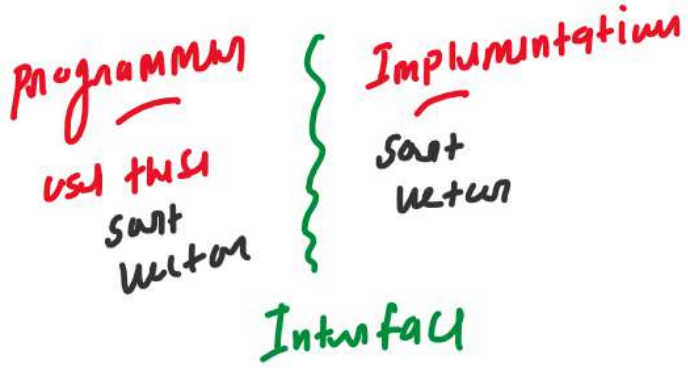
1. Function's implementation is hidden in header files.
2. We could use the same program without knowing its inside working.
3. **E.g.,** Sort(), for example, is used to sort an array, a list, or a collection of items, and we know that if we give a container to sort, it will sort it, but we don't know which sorting algorithm it uses to sort that container.

```cpp
#include<iostream> // cout
#include<algorithm> // sort
#include<vector> // vector
using namespace std;

int main(){
    vector v = {3, 4, 1, 2};
    sort (v.begin(), v. end());
    for (auto i : v){
        cout << i << " ";
    }
    return 0;
}
```

HEADER FILES

OUTPUT:
1 2 3 4



Abstraction using classes:

1. *Grouping data members and member functions into classes using access specifiers.*
2. *A class can choose which data members are visible to the outside world and which are hidden.*

Like ENCAPSULATION

Abstraction in C++

enjoy algorithms

```
class AbstractionExample{
  private:
     int num;
     char ch;

  public:
    void setMyValues(int n, char c) {
      num = n; ch = c;
    }

    void getMyValues() {
      cout<<"Numbers is: "<<num<< endl;
      cout<<"Char is: "<<ch<<endl;
    }
};
```

By making these data members private, we have hidden them from outside world.

These data members are not accessible outside the class.

The only way to set and get their values is through the public functions.

What is Abstract Class?

1. Class that contains **at least one pure virtual function**, and these classes cannot be instantiated.
2. It has come from the idea of Abstraction.

S.No.	Virtual Function	Pure Virtual Function
1.	A virtual function is a member function in a parent class that can be further defined in a child class.	A pure virtual function is a member function in a parent class, and declaration is given in a parent class and defined in a child class.
2.	The classes that contain virtual functions are not abstract.	The classes that contain pure virtual functions are abstract.
3.	In the child classes, they may or may not redefine the virtual function.	The child classes must define the pure virtual function.
4.	Instantiation can be done from the parent class with a virtual function.	It Cannot be instantiated as it becomes an abstract class.
5.	Definition of function is provided in the parent class.	Definition of function is not provided in the parent class.

```
virtual void fun()
{
    // definition
}
```

only virtual function

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

pure virtual function

Abstract Class is also known as interface class.

bird.h

Abstract Class

Interface

Implement

```bird.h
#if !defined(BIRD_H)
#define BIRD_H

// Starting Coding.....
#include<iostream>
using namespace std;
class Bird
{
    public:
        virtual void eat() = 0;
        virtual void fly() = 0;
        // Classes that inherits this class
        // has to implement pur virtual function.
};

class Sparrow: public Bird
{
    public:
        void eat()
        {
            cout<<"Sparrow is eating\n";
        }
        void fly()
        {
            cout<<"Sparrow is flying\n";
        }
};
// Ending Coding.....

#endif
```

main.cpp

```main.cpp
#include<iostream>
#include "bird.h" // Own header file
using namespace std;

void birdDoesSomething(Bird *&bird)
{
    bird->eat();
    bird->fly();
}

int main()
{
    Bird *bird = new Sparrow();
    birdDoesSomething(bird);
    return 0;
}
```

OUTPUT:

Sparrow is eating
Sparrow is flying

Design Strategy:

1. Abstraction divides code into two categories: interface *and* ***implementation****. So, when creating your component, keep the interface separate from the implementation so that if the underlying implementation changes, the interface stays the same.*

2. In this instance, *any program that uses these interfaces would remain unaffected and would require recompilation with the most recent implementation.*

```
// An abstract class
class Test
{
    // Data members of class
public:
    // Pure Virtual Function
    virtual void show() = 0;

  /* Other members */
};
```

interface

implementation

```
class A
{
public:
  virtual void Func() = 0;
};
```

Abstract class

```
class B : public A
{
public:
  void Func() override {
    cout << "B::Func()" << endl;
  }
};
```

```
class C : public B
{
public:
  void Func() override {
    cout << "C::Func()" << endl;
  }
};
```

Non-abstract classes

📂 3. Inline Function in C++

1. An inline function is a regular function that is defined by the inline keyword.

2. The code for an inline function is inserted directly into the code of the calling function by compiler while compiling, which can result in ***faster execution and less overhead compared to regular function calls.***

3. Instead of calling function the statements of functions are pasted in calling function.

4. Used with ***small sized functions.*** So that executables are small ***(handled automatically by compiler optimisation levels).***

inline function working process

```
inline void displayNum(int num) {
  cout << num << endl;
}

int main() {

  displayNum(5);

  displayNum(8);

  displayNum(666);
}
```

Compilation →

```
inline void displayNum(int num) {
  cout << num << endl;
}

int main() {

  cout << 5 << endl;

  cout << 8 << endl;

  cout << 666 << endl;
}
```

less overhead to regular function because ***dispalyNum()*** is not loading in call stack

Regular function working process

```
void displayNum(int num) {
  cout << num << endl;
}

int main() {

  displayNum(5);

  displayNum(8);

  displayNum(666);
}
```

Compilation →

*overhead to regular function because **dispalyNum()** is loading in call stack*