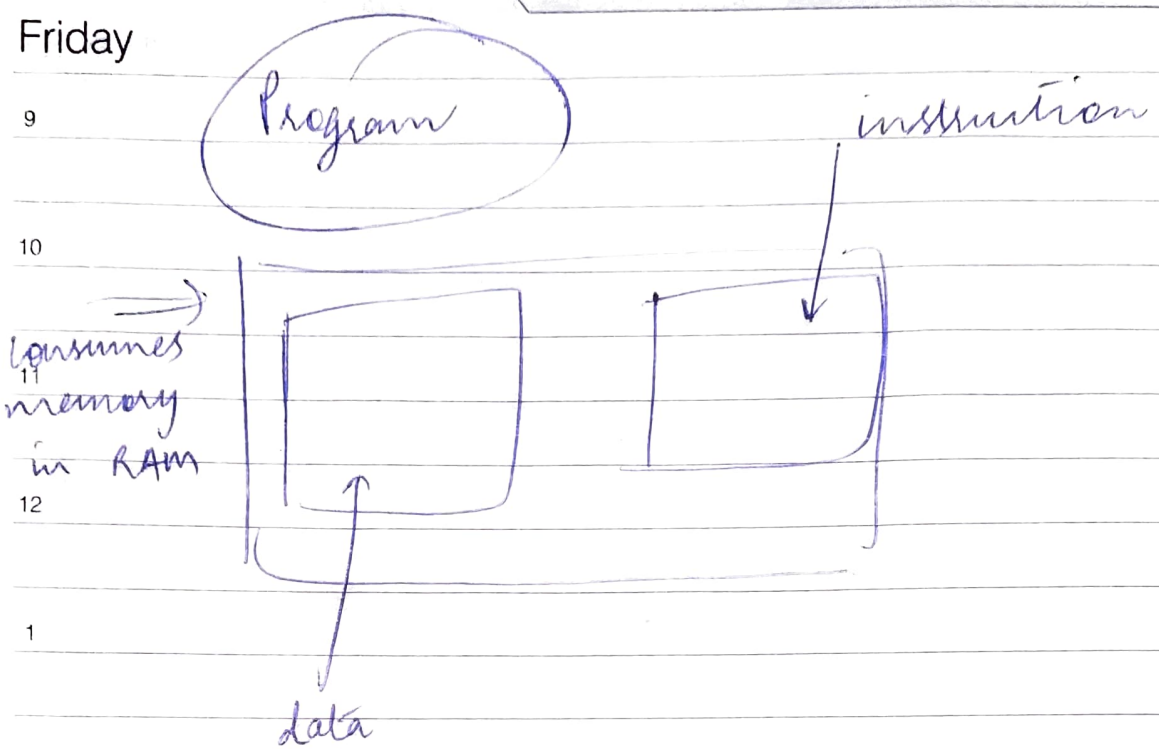
OOPS Object Oriented Programming

01 Friday

060 • 305 | Week 09

MARCH 2019



- Total Reserved words in C++ ⇒ 95

⇒ Explicit Declaration of function is recommended but not mandatory in C but it is compulsory in C++

⇒ Header files = Predefined functions are defined in header files

⇒ reference variable is an internal pointer
↳ must be initialized during declaration

C ⇒ NO-OOPS

Structure ⇒ user defined data type
⇒ way to group variables
→ collection of same or dissimilar elements
⇒ used to create data type

• Memory is not provided to a variable untill it is used. (structures)

Encapsulation ⇒ Wrapping up of data and functions into a single unit.

```
#include <iostream.h>
struct book {
    int bookid;
    char title[20];
    float price;
    void input() {
        cout << "Enter the details\n";
        cin >> bookid >> title >> price;
    }
    void output() {
        cout << bookid << title << price;
    }
```

```
void main(){
    book b1;
    b1.input();
    b1.output();
}
```

Classes

Class

members are by default private

Structure

members are by default public

Example 8-

```
#include<conio.h>
#include<iostream.h>
class complex {
  private:
    int a, b;
  public:
    void set_data(int x, int y){
      a=x; b=y;
    }
    void show_data(){
    cout<<a<<b;
    }
```

```
complex add (complex c) {
    complex temp;
    temp.a = a + c.a;
    temp.b = b + c.b;
    return temp;
  }
}

void main() {
  complex c1, c2, c3;
  c1.set_data(3,4);
  c2.set_data(5,6);
  c3 = c1.add(c2);
  c3.show_data();
}
```

- Class is the description of an object
- Object is the instance of a class.

Instance member variable ⇒ a, b
(allocated space after defining objects i.e. instances)
(attributes, data members, fields, properties)

Instance member functions ⇒ set_data, show_data
add
↳ (if there is no instance who would they perform the operation on)
(methods, procedure, actions, operation)

MARCH 2019

Su	Mo	Tu	We	Th	Fr	Sa
31						
3	4	5	6	7	1	2
10	11	12	13	14	8	9
17	18	19	20	21	15	16
24	25	26	27	28	22	23
					29	30

State of an object ⇒ collection of instances variable member values

Generally, state should be changed through a instance member function.

Behaviour of objects ⇒ instance members functions

Static members

↳ static local variable

→ Static member variables

→ Static member functions

Static local variable

→ default value = 0

→ memory allocated at the time of starting of program.

→ Lifetime ⇒ throughout program.

Static member variable

⇝ also known as class member variable.

→ declared inside class

→ defined outside class

→ doesn't belong to any object, but to whole class

→ only one copy of static member variable for whole class

→ any object can use same copy of class variable.

→ they can also be used with class name.

example :-

```
class account {
  private:
    int balance;          // Instance member variable
    static float roi;     // static member variable / class variable
};
float account :: roi = 3.5f;
void main() {
  account a1;
}
```

MARCH 2019

Su	Mo	Tu	We	Th	Fr	Sa
31					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

Static member function

→ They are qualified with the keyword static.

→ They are also called class member functions.

→ They can be invoked with or without object.

→ They can only access static members of the class.

```
class account {
    private:
        int balance
        static float roi;
    public:
    void setbalance(int b) { balance = b; }
    static void setRoi(float r) {
        roi = r; }
};
float account :: roi = 3.5f;
void main() {
    account a;
    a.SetRoi(4.5f);
    Account :: setRoi(4.5f);
```

MARCH 2019

Constructor

→ member functio

→ name same as

→ no return typ

→ must be an
can't be st

→ implicitely
created.

→ used to

```
class complex {
    private:
        int a,b
    public:
        complex(
            a=
            compl
            comp
```

Constructor overloading

MARCH 2019

Su	Mo	Tu	We	Th	Fr	Sa
31					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

Same meth
no diff

Constructor

→ member function of class

→ name same as class name

→ no return type

→ must be an instance member function, i.e., can't be static

→ implicitely ~~called~~ invoked when an object is created.

→ used to solve problem of initialization

~~3 types~~ ~~default~~

```
class complex{
private:
    int a,b;
public:                          → parameterised constructor
    complex(int x, int y){
        a=x; b=y;}
    complex(int x){
        a=x;}
    complex(){}                  → kind of default constructor
};
```

Constructor overloading

- if a user creates any of the constructor then compiles doesn't creates default constructor

Sunday 10

```
void main(){
    complex c1(3,4), c2(5), c3;
    complex c4=complex(3,5), c5=complex(7);
    complex c6=9;  (=> valid for single transfer)
}
```

Same method no diff.

MARCH 2019
Su Mo Tu We Th Fr Sa
31 1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30

- If a user creates any type of constructor then default constructor is not created by compiler.
- If a user creates copy constructor then none of default or copy constructor is created by compiler.

```
class complex{
    private:
        int a, b;
    public:
        complex (complex &c){
            a = c.a;
            b = c.b;
        }
};
void main(){
    complex c1(3,4);
    complex c2(c1);
    complex c3 = c1;
}
```

In copy constructor object is passed as reference i.e. c stores the values of c1. because if it would not be a reference variable then it would result in recursion.

Destructor

→ instance member function

→ same name as class but preceded by tilde(~)

→ never be static

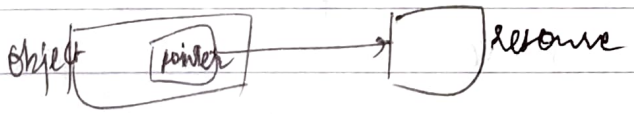
→ NO return type.

→ Takes no argument. (No overloading)

→ invoked implicitly when object is going to destroy. (but doesnot destroy object)

→ compiler automatically creates it but if user creates it then ~~const~~ compiler doesn't.

→ should be defined to release resources allocated to object.




```
class complex {
    private:
        int a, b;
    public:
        ~complex() {
            cout<<"Destruction"; }
};
void fun() {
    complex obj;
}
void main() {
    fun();
}
```

Operator overloading

→ When an operator is overloaded with multiple jobs it is known as operator overloading

→ It is a way to implement compile time polymorphism.

→ Any ~~Oper~~ symbol can be used as function name

- if it is a valid operator in C.
- if it is preceded by operator keyword

→ sizeof and ?: Operators can't be overloaded

```
class complex{
    private:
        int a, b;
    public:
        void setdata(int x, int y){ a=x, b=y;}
    complex operator +(complex c){
        complex temp;
        temp.a = a+c.a;
        temp.b = b+c.b;
        return temp;
    }
};
```

```
void main(){

complex c1, c2, c3, c4, c5;

c1.setdata(3,4);
c2.setdata(4,5);
c3 = c1.operator+(c2);
c4 = c1 + c2;
}
```

→ Both methods are valid and will give same result.

→ left operand is caller object for binary operators

Operator overloading of unary operators

```
class complex{
private:
    int a, b;
public:
void setdata(int x, int y){
    a=x; b=y;}

void showdata(){ cout << a<<b;}

complex operator-(){
    complex temp;
    temp.a = -a; temp.b = -b;
    return temp;
};}
```

```
void main(){
  complex c1, c2, c3;
  c1.setdata(2,3);
  c2 = -c1;          → Both are same
  c3 = c1.operator-();
}
```

Overloading of increment/decrement operator

```
class Integer {
  private:
    int x;
  public:
    void setdata(int a){ x=a; }
    void showdata(){ cout<<x; }

    Integer operator++(){          //preincrement
      Integer i;
      i.x = ++x;
      return(i);
    }

    Integer operator++(int){       //post increment
      Integer i;
      i.x = x++;
      return i;
    }
};
```

→ syntax for distinguishing pre from post.

```
void main(){

Integer i1,i2,i;
i1.setdata(5);

i2 = ++i1;
i3 = i1++;
i1.showdata();      ⟶ 7
i2.showdata();      ⟶ 6
i3.showdata();      ⟶ 6
}
```

Sunday 17

Friend function

→ Friend function is not a member function of a class to which it is a friend.

→ Friend function is declared in class with friend keyword.

→ It must be defined outside the class to which it is friend.

→ It can access any member of the class to which it is a friend.

→ It can't access members of the class directly.

→ no caller object

→ shouldn't be defined with membership label.

```
class complex {
    private :
    int a, b;
    public :
    void setdata (int x, int y) { a=x, b=y; }
    void showdata() { cout<<a<<b; }
```

6

```
    friend void fun(complex);
};
void fun(complex C){
    cout<<C.a+C.b;
}
void main(){ complex C1,C2;
    C1.setdata(2,3);
    fun(C1);
}
```

→ Imp. use of friend function

→ It can become friend to more than one class.

example: class A; class B;

```
class A{
    private:
        int a;
    public:
        friend void fun(A,B);
};
class B{
    private:
        int b;
    public:
        friend void fun(A,B);
```

MARCH 2019
Su Mo Tu We Th Fr Sa
31 1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30

```
void fun ( A o1, B o2) {
    cout << o1.a + o2.b;
}
void main() {
    A ob1;
    B ob2;
    fun (ob1, ob2);
}
```

// use setdata for classes

→ Overloading of operators as a friend function

```
class complex {
    private:
     int a, b;
    public:
     void setdata( int x, int y) { a=x, b=y; }
     void showdata () { cout<<a<<b; }
     friend operator+ (complex, complex);
};
```

```
complex operator +(complex X , complex Y) {
    complex temp;
    temp.a = X.a + Y.a;
    temp.b = X.b + Y.b;
    return temp;
}

void main()
{
    complex C1, C2, C3;
    C1.setdata(2,3);
    C2.setdata(3,4);
    C3=C1+C2
}
```

Now operator + is a friend function and hence since there is no caller object therefore 2 arguments are passed for binary operator. Hence for unary there should be 1 operator

→ Overloading of unary operator as a friend function

```
class complex {
    friend complex operator- (complex);
};
```

MARCH 2019
Su Mo Tu We Th Fr Sa
31 1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30

```
complex operator - (complex x) {
    complex temp;
    temp.a = -x.a;
    temp b = - x. b;
    return temp;
}

void main(){
    complex c1, c2;
    c1. setdata(3,4);
    c2 = -c1.
    c1. showdata();      → 3, 4
    c2. showdata();      → -3, -4
}
```

→ Overloading of insertion and extraction operators:

```
class complex {
    private:
        ==
    public:
        ==
        ==
```

```
friend ostream& operator<<(ostream&, complex);
friend istream& operator>>(istream&, complex);
};

ostream& operator<<(ostream &dout, complex C){
    cout<< C.a <<C.b;
    return dout;
}
```

→ can be cout too

```
istream& operator>>(istream &din, complex C){
    cin >> C.a >> C.b;
    return din;
}
```

→ can be cin too.

```
void main(){
    complex C1;
    cin >> C1;      // operator>>(cin, C1);
    cout<<C1;       // operator<<(cout, C1);
}
```

Sunday 24

⇒ insertion and extraction operators can only be overloaded through a friend function because calling function is ostream & istream which are at left and hence cannot be used.

→ Member function of one class can become friend to another class.

```
class A {
public:
    void fun() { }
    void foo() { }
};
class B {
    friend void A::fun();
    friend void A::foo();
};
```

(For making all functions of A as friend of B then just write
friend class A;
inside class B)

Inheritance

→ process of inhereting properties and behaviours of existing class into a new class

→ existing class = old class = parent class = base class

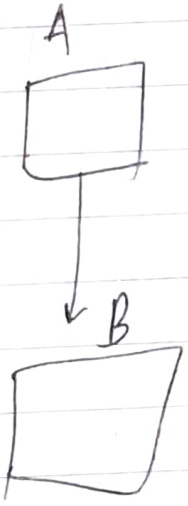
→ new class = child class = derived class

syntax :-

```
class base_class { };
class derived_class : visibility_mode base_class { };
```

single inheritance

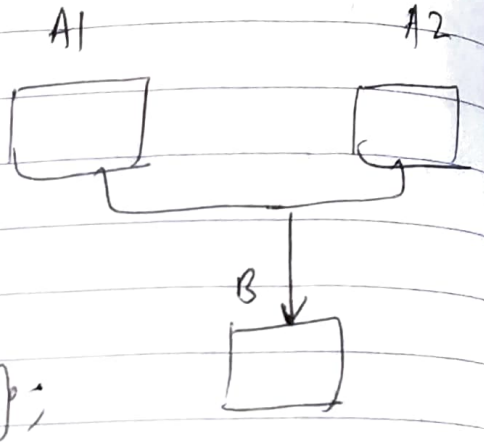
```
class A { };
class B : public A { };
```



Multilevel inheritance

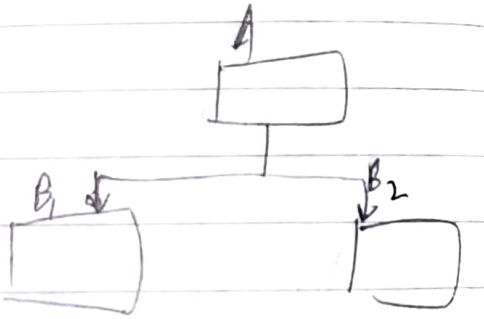
```
class A { };
class B : public A { };
class C : public B { };
```

Multiple inheritance




```
class A1 { };
class A2 { };
class B : public A1, public A2 { };
```

Hierarchical inheritance



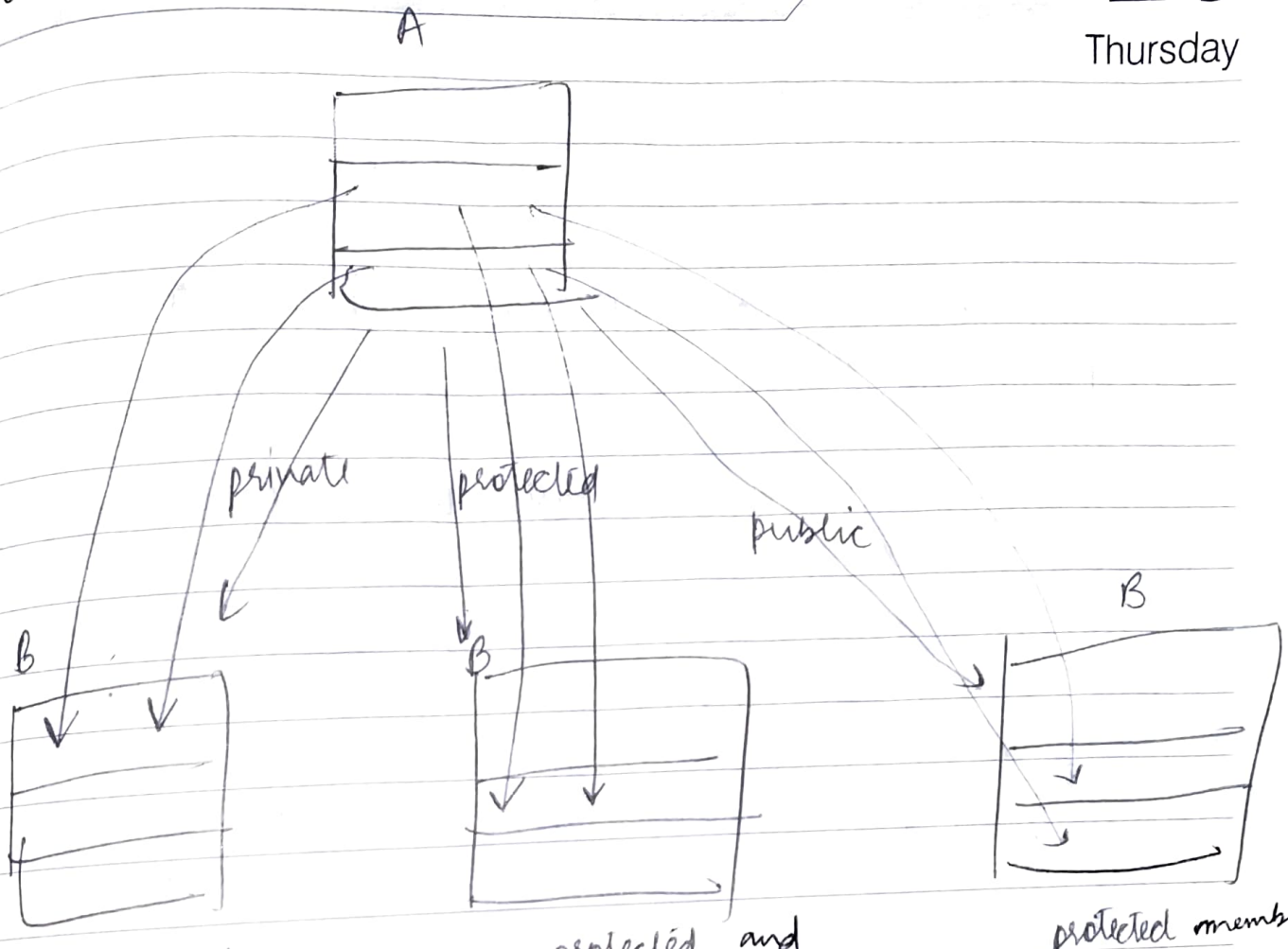

```
class A { };
class B1 : public A { };
class B2 : public A { };
```

Visibility modes

- private
- protected
- public

Availability and Accessability

private members are available to all objects but not accessible by any



protected and public members of class A will be private members of class B

protected and public members of class A will be protected members of class B.

protected members of class A will be protected members of class B and public of A will be public of B.

Constructor and Destructor in inheritance

→ order of calling of constructors ⇒ from base class to derived class

→ order of execution of constructors ⇒ from parent class to child class

```
class A {
public:
    A() { }
};
class B : public A {
public:
    B() : A() { }
};
```

⇒ default constructors (compiler automatically does this)

```
class A{
    int a;
public:
    A(int k){ a=k; }
};

class B: public A{
    int b;
public:
    B(int x, int y): A(x){
        b=y;
    }
};

void main(){
    B obj(2,3);
}
```

If user makes constructor of A then user should also makes constructor of B and call constructor of A otherwise default one will be called automatically which would given an error.

Sunday 31

⇒ order of calling of destructor same as of constructor but of execution is opposite.

MARCH 2019

Su	Mo	Tu	We	Th	Fr	Sa
31					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

⇒ No need to call parents destructor; it automatically gets called after execution of child's destructor

This pointer

Object pointer ⇒ it contains address of an object

this pointer ⇒ • this is a local object pointer in eve
instance member function containing
address of caller object
• this pointer can not modify
• used to refer caller object in member
function

```
class Box {
  private:
    int l, b, h;
  public:
    void setdimensions (int l, int b, int h){
      this → l = l ; this → b = b ; this → h = h ;}
};
void main(){
  Box smallBox;
  smallBox. setDimensions (12, 10, 5);
}
```

New and Delete

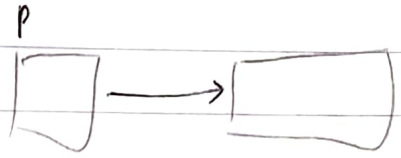
SMA : Static Memory Allocation

DMA : Dynamic " "

- memory to be allocated is decided at compile time (SMA)
- " " " " " " " runtime (DMA)

ex :-

```
int *p = new int;
float *q = new float;
complex *ptr = new complex;
```




```
float *q = new float[5];
```

- Memory of DMA variables is kept throughout the program and hence for releasing memory delete is used.

ex :-

```
delete p;
delete []p;
```


APRIL 2019

Su	Mo	Tu	We	Th	Fr	Sa
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30				

Method overriding, overloading - hiding

```
                                    → ex - car
class A{
public:                                  → changegear
    void f1(){ }
    void f2(){ }
};                                  → sports car
class B: public A{
public:                                  → changegear
    void f1(){ }                    // method overriding
    void f2(int x){ }               // method hiding
};
void main(){
    B obj;
    obj.f1();       //B
    obj.f2();       //error
    obj.f2(4);      //B
}
```

overloading refers if the scope is same of all function with same name and different parameters.

- A pointer of parent class can contain address of child class (an exception) but child's class's pointer can't contain the address of parent's class.

```
class A {
public:
    void f1() { }
};
class B : public A {
public:
    void f1() { }
    void f2() { }
};
void main()
{
A *p, O1;
B O2;
p = &O2;
p->f1(); // A
}
```

// overriding

```
class A {
public:
virtual void f1() { }
};
class B : public A {
public:
    void f1() { }
    void f2() { }
};
void main() {
A *p, O1;
B O2;
p = &O2;
p->f2(); // B
}
```

→ enables late binding and thus pointer's content can be identified of class & hence f1 of B called

APRIL 2019
Su Mo Tu We Th Fr Sa
7 1 2 3 4 5 6
14 8 9 10 11 12 13
21 15 16 17 18 19 20
28 22 23 24 25 26 27
29 30

↓ Early binding (at compile time)

↓ Late binding (at runtime)

compiler creates this pointer if it sees a virtual function in any class but it is not made in descendant classes.

*-vptr → this stores address of vtable of its own class.

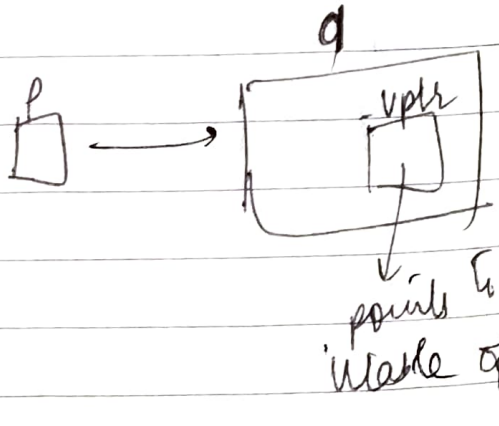
```
class A {
public:
    void f1() { }
    virtual void f2() { }
    virtual void f3() { }
    virtual void f4() { }
};
class B : public A {
public:
    void f1() { }
    void f2() { }
    void f4(int x) { }
};
```

vtable of A: | f2 | f3 | f4 |

vtable of B: | f2 | f3 | f4 |

vtable => static array of function pointer. (of only virtual functions)

```
main() {
    A *p, o1;
    p = &o1;
    p -> f1(); // early binding
    p -> f2(); // late "
    p -> f3(); // late "
    p -> f4(); // late "
    p -> f4(5); // early " -> error
}
```

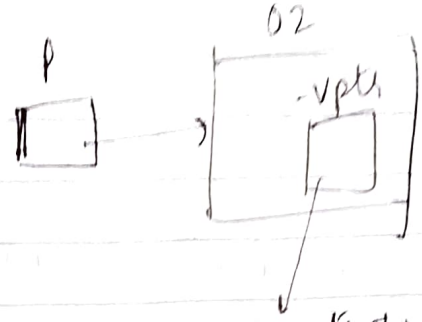


```
void main() {

    A *p;
    B o2;

    p = &o2;

A ←  p → f1();  // early binding
B ←  p → f2();  // late    "
A ←  p → f3();  // late    "
A ←  p → f4();  // late    "
     p → f4(5); // early   "
error
}
```



points to vtable of B and hence at the time of late binding vtable of class B will be seen.

Pure virtual function and Abstract Class

- A do nothing function is pure virtual function
  ↓
  with no definition

Object of an abstract class can't be created

Since it contains atleast one pure virtual function hence it is abstract class

```
class Person {
public:
    virtual void fun() = 0;
};
```

→ do nothing function

→ do nothing function

object of Person can't be created due to a do nothing function

```
class student : public Person {
public :
    void fun() { }
};
```

- do nothing function of parent class needs to be overridden in child class

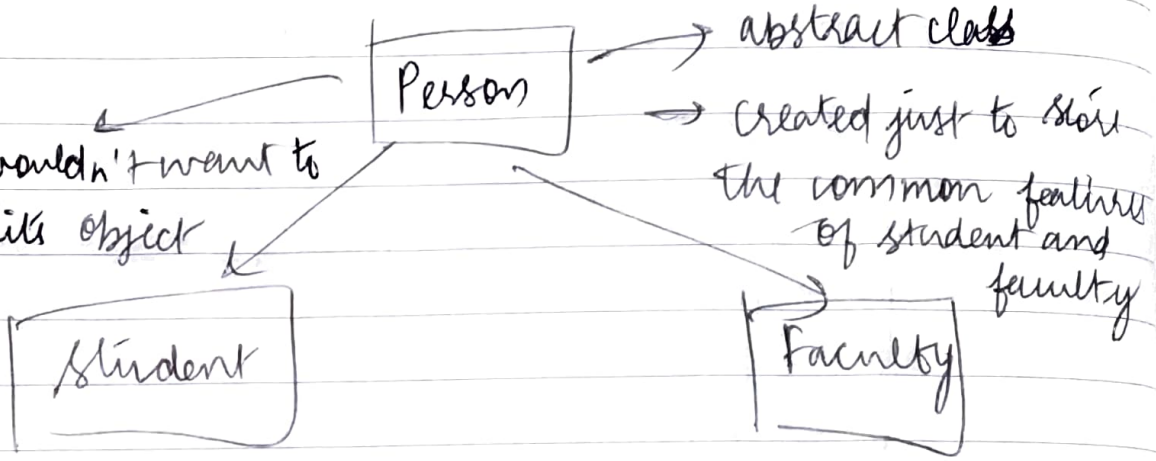
Sunday 07

APRIL 2019

Su	Mo	Tu	We	Th	Fr	Sa
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30				

~~should be~~ virtual ~~to~~ relevant early binding

example of abstract class



Function Template → Generalized function

syntax:

template <class type> type func_name(type arg1,...)

→ can be more than one types of placeholders
ex ⇒ <class t1, class t2>

ex ⇒

```
template <class X>  X big(X a, X b){
    if (a>b) return a;
     else return b;
}
void main(){
    cout << big(4,5);          → 5
    cout << big(5.6, 4.5);     ⇒ 5.6
```

Class template

- also known as generic class.

template <class type> class class_name(...);

type ↓ placeholder

Exception handling

- try - throw - catch
- try and catch work hand in hand.
- if throw is written then
- throw should throw a value.

If no value is thrown then runtime error is occured not compile time.

- catch(...) ⇒ valid
- if anything in thrown then there should be a a catching statement for such else runtime error.

APRIL 2019

Su	Mo	Tu	We	Th	Fr	Sa
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30				