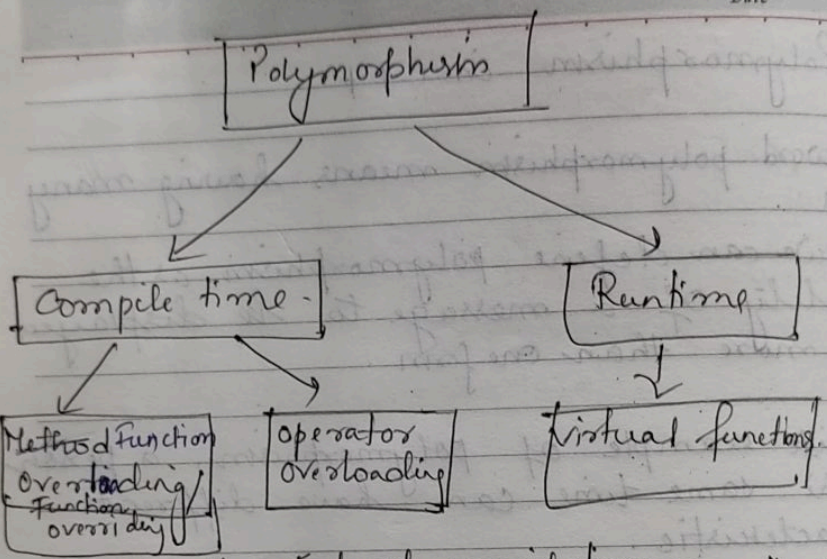
Date



memory allocated at compile time. sometimes called as early binding

1. Compile time polymorphism

This type of polymorphism is achieved by function overloading or operator overloading

(i) Function overloading :

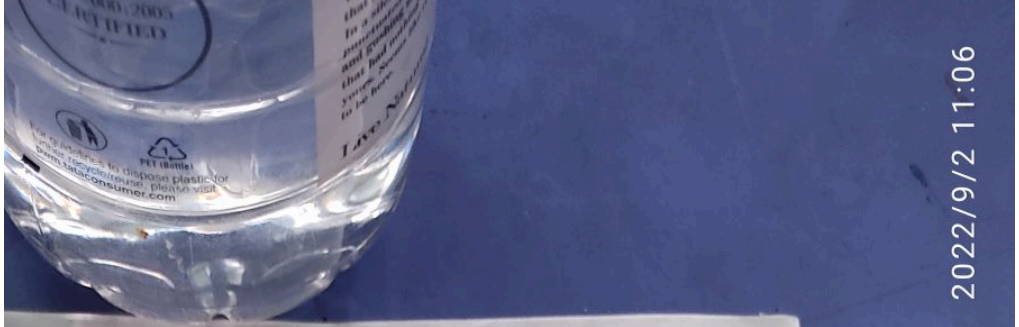
When there are multiple functions with same name but different parameters then these functions are said to be overloaded.

Functions can be overloaded by change in number of arguments or/and change in type of arguments.

Date

Rules of Function Overloading.

```
#include <iostream.h>
class Geeks
{ public: void funct (int x)
          { cout << " value of x is " <<x<<endl;
          }
          void funct (double x)
          {
            cout << "value of x is "<<x<<endl;
          }
          void funct (int x, int y)
            { cout << "value of x and y is"
                 <<x<<" , "<<y<<endl;
            }
};
```



Date

```
int main ()
{
    Geeks obj1;
    obj1.funct(7);
    obj1.funct(9.132);
    obj1.funct(85,64);
    return 0;
}
```

output:
value of x is 7

value of x is 9.132

value of x and y is 85, 64.

Function overiding -

same name with same parameters.

Inheritance concept should be there if we are using function overiding.

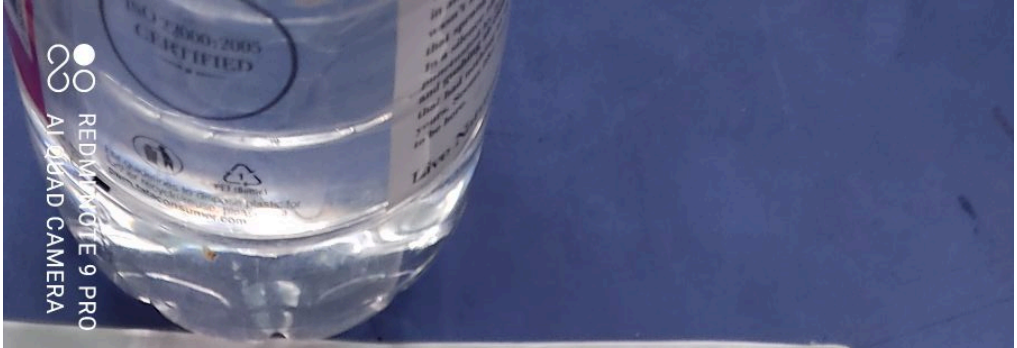
It cannot be done within a class.

For this, we required derived class & base class.

Date

eg.
```
class base
{ public : void show()
           { cout << " Base class";
           }
};
class derived : public base
{ public : void show()
           { cout << " derived class";
           }
};
int main()
{ Base b;
  derived d;
  b.show();
  d.show();
}
```

For function overriding inheritance concept is used



Date

(11) operator overloading :

Type of polymorphism in which an operator is overloaded to give user defined meaning to it.

① to add two integer we will use '+' operator

② want to add concatenating two strings we use '+' operator strings.

How to overload the operator.

⇒ to overload operator, a operator function is defined inside a class.

```
class class_name
  { public : return type operator sign (args)
              { . . .
              }
  };
```

Date

eg. class Rectangle

```
{ int l, b;
  public: Rectangle()
          { l=0;
            b=0;
          }
  void operator ++()
          { l+=2;
            b+=2;
          }
  void display()
          { cout << "\n\t length" << l;
            cout << "\n\t Breadth " << b;
          }
};
int main()
  { Rectangle R;
    cout << " length & breadth before
            increment";
    R.display;
    ++R;
    cout << "length & breadth after increment
    R.display; return 0;
}
```

output: length & breadth before increment
length = 0
breadth = 0.
length & breadth after increment
length = 2
breadth = 2.

Rules :

(1) only existing member can be overloaded, we can't created our own operator to overload.

(2) the overload operator must have atleast one operand of user defined type.

(3) It follows syntax rules of original operator. This means we can't change the basic meaning of operator.

(4) Some operators can't be overloaded.
member access operator (.)
ptr. to member access operator (.*)
scope resolution operator (::)
size operator (sizeof)
ternary operator (?:)

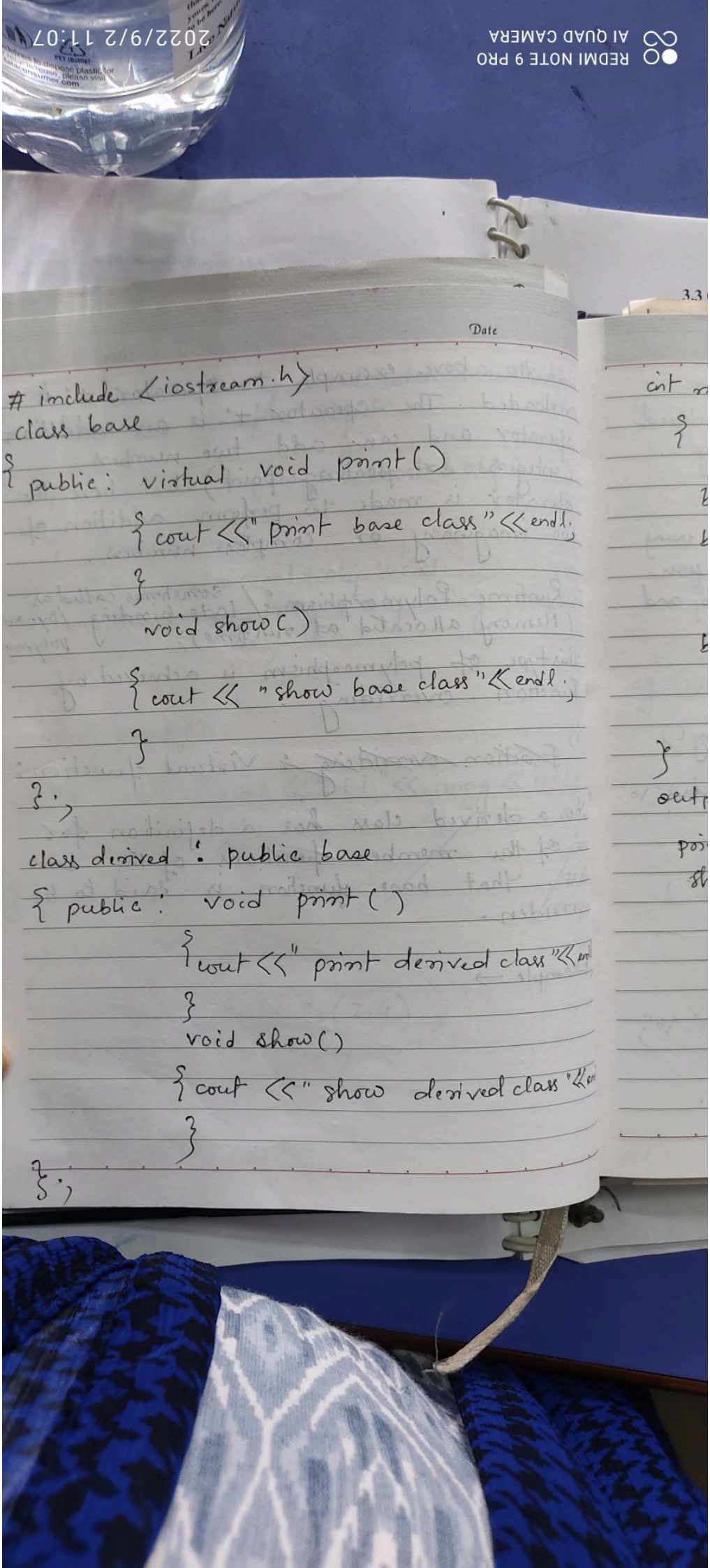
2. Runtime Polymorphism :/ sometimes called as late binding. (Dynamic polymorph
(Memory allocated at runtime).

This type of polymorphism is achieved by Function overriding.

(i) ~~Function overriding~~ : Virtual function:

when a derived class has a definition for one of the member functions of the base class. That base function is said to be overriden.

Example →

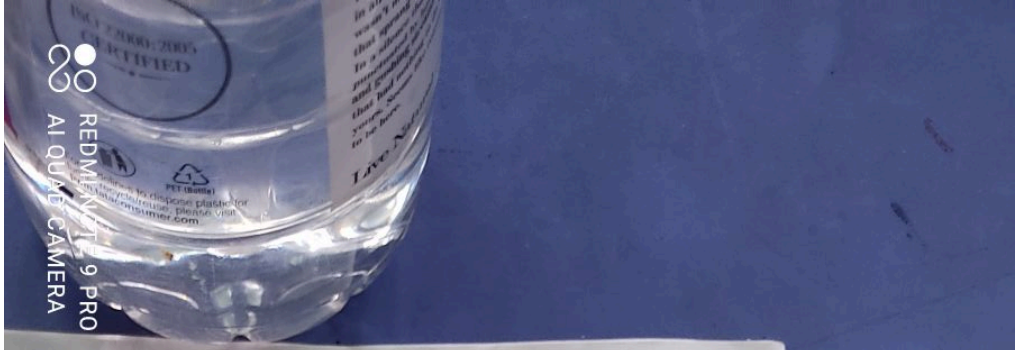

2022/9/2 11:07
REDMI NOTE 9 PRO
AI QUAD CAMERA
Date
include <iostream.h>
class base
{ public: virtual void print()
{ cout << " print base class " << endl;
}
void show()
{ cout << " show base class " << endl;
}
};
class derived : public base
{ public: void print()
{ cout << " print derived class "
}
void show()
{ cout << " show derived class "
}
};

Date

```
int main()
{
    base *bptr;
    derived d;
    bptr = &d;
    bptr->print();    // virtual function,
                      binded at runtime
                      (Runtime polymorphism)
    bptr->show();     // non-virtual function
                      binded at compile time
    return 0;
}
```

output:

print derived class.
show base class.



Date

Virtual function:

① It is a member function which is declared within base class and is redefined by derived class.

② when you refer to a derived class obj using a pointer of a reference to base class, you can call a virtual function for that obj and execute the derived class function.

eg ①

```
class A
  { public: virtual void display()
        { cout << " content of baseclass" \n";
        }
  };
class B: public A
 { public: void display ()
    { cout << " content of derived class" \n";
    }
 };
```

Date

```
int main()
{ A *b;   // base class ptr
  B d;    // derived class obj
  b = &d;
  b -> display(); // late binding occurs.
  return 0;
}
```

eg. (2)

```
class A
{ public: virtual void show()
          {
            cout << " hello base class \n";
          }
};
class B: public A
{ public: void show()
          { cout << " hello derived class";
          }
};
```

Date

```
int main()
{
    A aobj;
    B bobj;
    A * bptr;
    bptr = &aobj;
    bptr->show();
    bptr = &bobj;
    bptr->show();
    return 0;
}
```

Output: hello base class
hello derived class.

Date

Rules for virtual function :

1. virtual function must be member of same class.
2. They cannot be static member.
3. They are accessed by using object pointers. (A * bptr)
4. Virtual function can be friend of other function.
5. Virtual function in a base class must be defined, even though it may not be used.
6. We cannot have virtual constructors, but we have virtual destructors.