Contents:

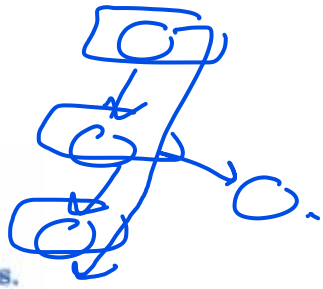
- Before OOPS
- Flaws of POP (Procedural oriented programming)
- What is OOPS?
- Benefits of OOPS?
- Little perquisite before starting classes
- What are classes

What this video will cover: E. Balaguruswamy by chapter 5

Next video -> Constructor

What is Procedural Oriented Programming?

Program -> sequence of things to be done



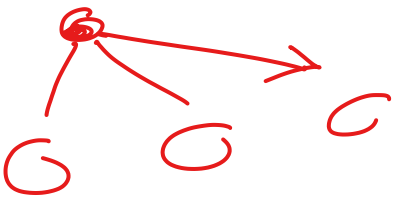
Some characteristics exhibited by procedure-oriented programming are:

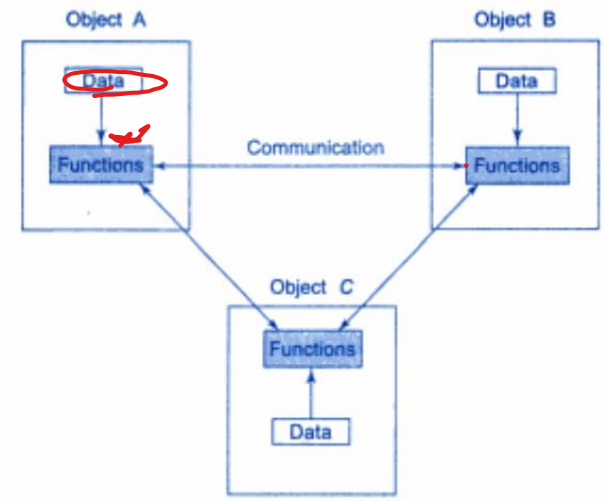
- Emphasis is on doing things (algorithms).
- Large programs are divided into smaller programs known as functions.
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.
- Employs *top-down* approach in program design.

What is OOPS?

- Data (Critical element)
- Data (Doesn't move freely)
- Binds data and functions (which operate on data together)
- (Data + function) encapsulate

Working of OOPS





OOPS VS POP

- ➔ Data hidden
- ➔ Program divided into function
- ➔ Objects communicate through each other with functions
- ➔ Follows bottom-up approach

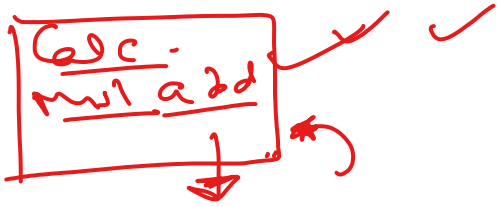
What OOPS brings?

1) Objects
2) Classes
3) Data abstraction & encapsulation
4) Inheritance
5) Polymorphism

What is Abstraction and Encapsulation?

Encapsulation	Abstraction
Wrapping (data+function) in single unit	Representing only essential features only
Insulates Data from Outside	Only give small access to outside world

Example:

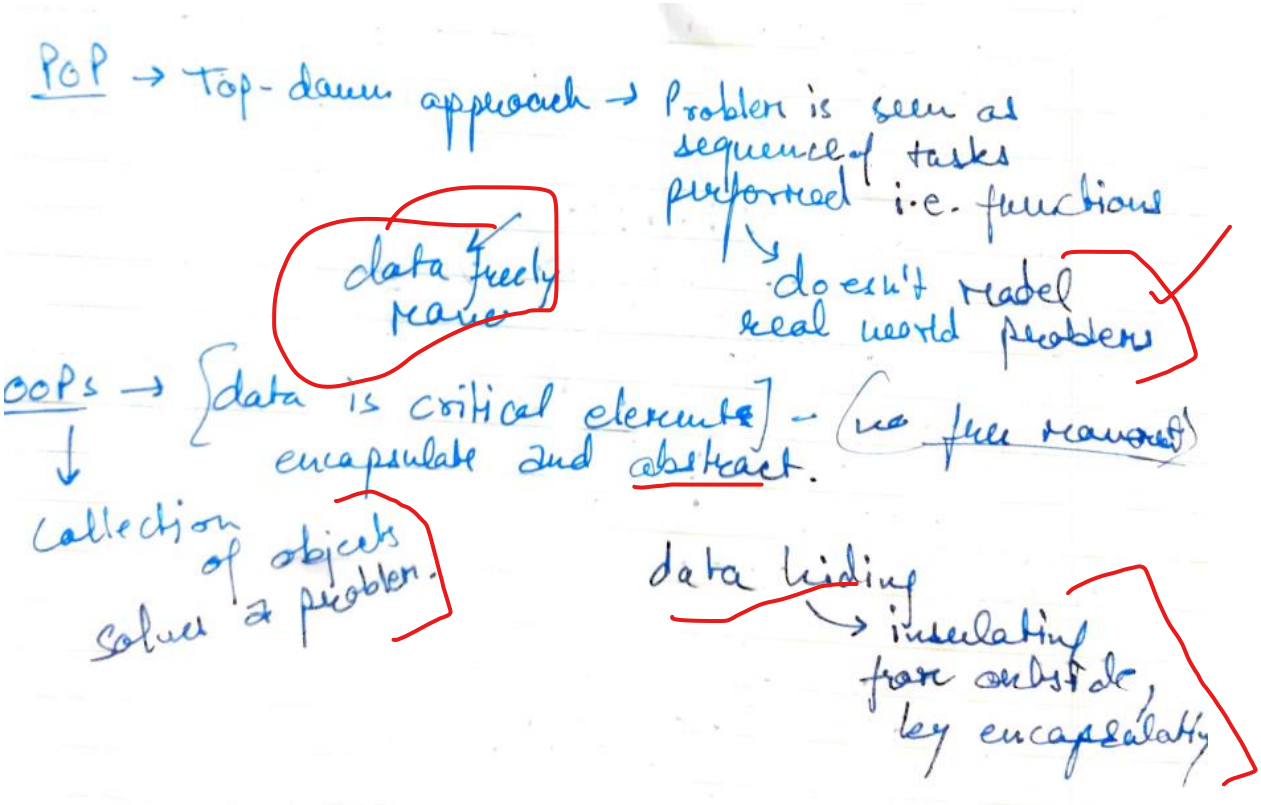


As Classes use Data Abstraction hence are called Abstract Data Type (ADT)

Benefits of OOPS

- Through inheritance, we can eliminate redundant code and extend the use of existing classes.
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
- The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
- It is possible to have multiple instances of an object to co-exist without any interference.
- It is possible to map objects in the problem domain to those in the program.
- It is easy to partition the work in a project based on objects.
- The data-centered design approach enables us to capture more details of a model in implementable form.
- Object-oriented systems can be easily upgraded from small to large systems.
- Message passing techniques for communication between objects makes the interface descriptions with external systems much simpler.
- Software complexity can be easily managed.

Revision:



Storage Classes:

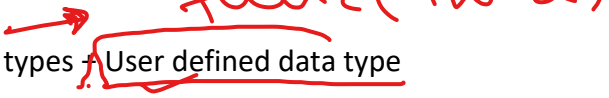
Tells 2 things -> **Lifetime and Visibility**

- Auto
- External
- Static
- Register
- Mutate (ignore)

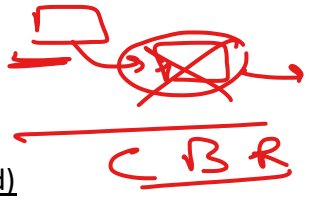
C++ Storage Class

Storage Class	Keyword	Lifetime	Visibility	Initial Value
Automatic	auto	Function Block	Local	Garbage
External	extem	Whole Program	Global	Zero
Static	static	Whole Program	Local	Zero
Register	register	Function Block	Local	Garbage
Mutable	mutable	Class	Local	Garbage

Reference Variable



Works for: Build in data types + User defined data type



IMP: Must be declared and initialized in single line (No other declaration is allowed)

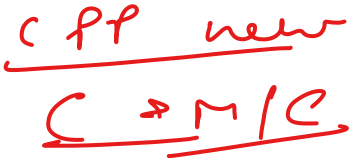
Eg: int y = 5;

Int &x = y; // Declaration and initialization in single line

Memory Allocation

How to get runtime memory from Heap area ?

Using NEW keyword in cpp, and Malloc/Calloc in C.



Eg:

// For single variable

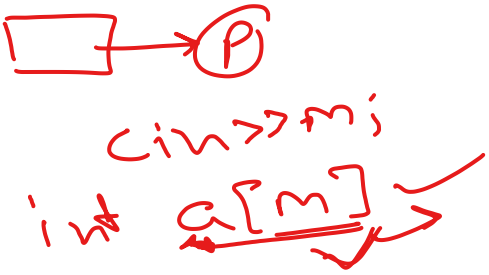
int *x = new int;

int *x = new int(5);

// For Array

int *x = new int[10];

int *x = new int[size]; // variable size is inputted during runtime by user



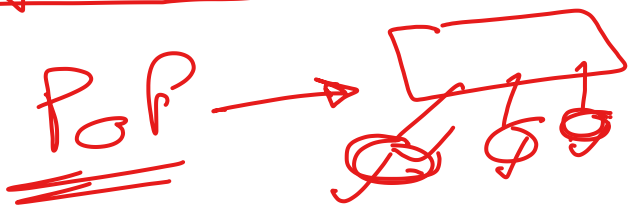
Using DELETE keyword to free the memory

delete ptr; // for single variable

delete []ptr; // for array

Functions:

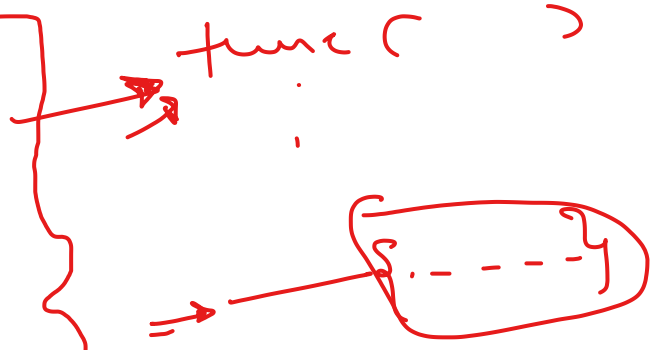
- Structured programming
- Reduce size
- Reusability of code



Inline Function: Expands in line where it is invoked.

Eg: inline int add(int x, int y)

{ return x+y; }



- Only used for small function call;
- Not memory sufficient
- Cant use it in -> Loops, Goto, Switch and recursion

Function Overloading:

Using same thing for different purpose

Also called Function Polymorphism

Perform different operations on basis of -> Argument list.

- Number of arguments
- Type of argument

Eg:- add(int a, int b)

add(int a, int b, int c)

add(string a, string b)

MAJOR PART

Classes/Objects:

Earlier we had Structures: Provides a way to bind data together

Eg: struct student{

char name[10];

int roll_number;

};

Struct creates a user-data type of name student

Limitations:

- Data Hiding
- Not like build in data type!

Whereas C++ supports Structures

But brings **Classes**

- A attempt to bridge gap between User datatypes and built-in datatypes
- Can hide data [Using access specifiers -> private, public, protected]
- Can inherit characteristics from other classes
- Binds [Function and data] together
- So many things more

It has 2 things mainly:

- Class declaration
- Class Function definition

Eg:

```
class Student{
private:
    int roll_no;
    char name[10];
public:
    void get(){
       cout<<roll_no<<" "<<name<<"\n";
}
```

```
class class_name
{
        private:
             variable declarations;
             function declarations;
        public:
             variable declarations;
             function declaration;
};
```

A class is like family,
Variables which store data are called -> Data Members
Functions that operate on that data -> Member Function

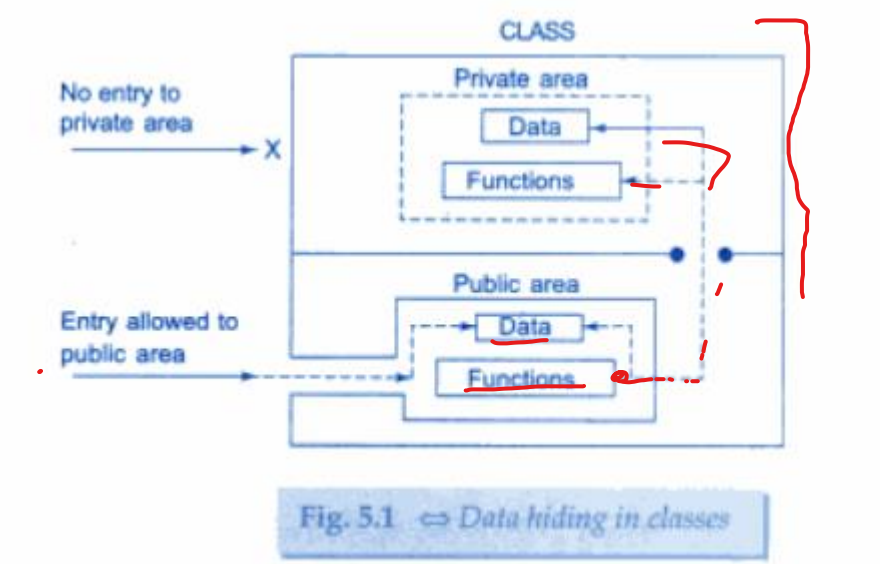


Fig. 5.1 ⇔ Data hiding in classes

WHAT DOES CLASS DO? -> They make User defined datatype

And Objects are instances of this class
eg:
int x; // x is instance of datatype int
student arjun; // similarly arjun is instance of datatype student

How to access Class members?
➔ object.data
➔ object.func(...arguments...)

In source code you can only access to public members of class.

Various methods to define Members of class

Defining member functions

- Outside class defination

```
return-type class-name :: function-name (argument declaration)
  {
        Function body
  }
```

-

```
void item :: getdata(int a, float b)
{
     number = a;
     cost = b;
}
```

-
- Important point to note:
  - Different classes can have same function name
  - Member functions can access private data
  - Member function can call other member function directly without '.' Operator

Inside Class definition;

```
class item
{
    int number;
    float cost;
  public:
    void getdata(int a, float b);   // declaration
        // inline function
    void putdata(void)              // definition inside the class
    {
        cout << number  <<  "\n";
        cout << cost    <<  "\n";
    }
};
```

IMPORTANT

Functions defined inside class are treated as INLINE function

Hence Same limitations are applied

Big Functions are always defined outside

Also we can make outside functions inline too

```
class item
{
        .....
        .....
   public:
        void getdata(int a, float b);          // declaration
};
inline void item :: getdata(int a, float b)   // definition
{
       number = a;
       cost = b;
}
```

Nested Member functions

A function calling other member function inside it

Private Member functions

- Deleting account of customer
- Changing salary of employees

How is memory allocated for a object

- Memory space for member functions is used immediately when a class is declared.
  (No extra space is used while declaring a object)
- Memory space for Data Members is created when object is declared
  (for each object separate memory region for data members is assigned.)

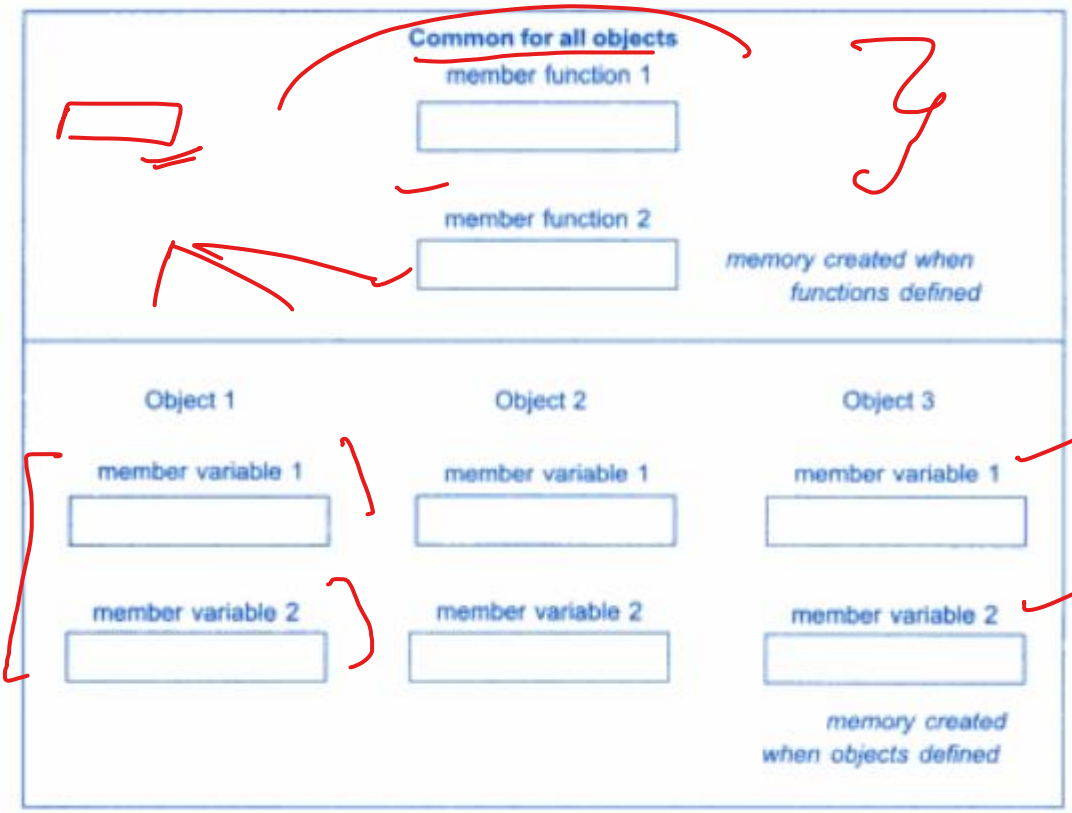


Fig. 5.3 ⇔ Object of memory

Static data members:

Applications: Counter for number of objects created per class

- Always only initialized with zero. (No other initialization allowed)
- Only one copy for per class.
- Visibility -> Inside Class, Lifetime-> Entire Program
- Always Defined outside class as they have to be stored other then class memory
- Similar to non-inline member function as they are declared in class but defined outside the class

```
class item
{
        static int count;
        int number;
  public:
        void getdata(int a)
        {
              number = a;
              count ++;
        }
        void getcount(void)
        {
              cout << "count: ";
              cout << count  << "\n";
        }
};

int item :: count;
```

Static Member Functions

Applications: To get static data members

➔ Can have access to static members only of same class only

➔ Can be called directly with class name

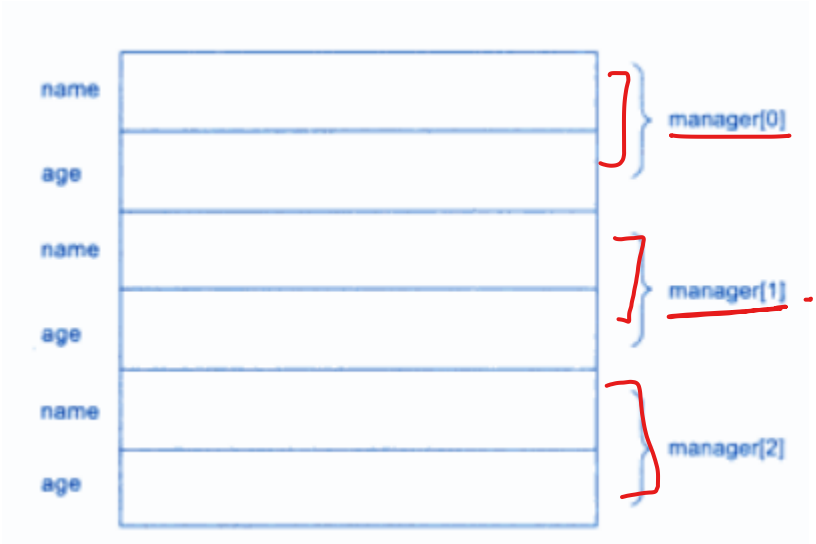
```
class test
{
        int code;
        static int count;          // static member variable
public:
        void setcode(void)
        {
        code = ++count;
        }
        void showcode(void)
        {
        cout << "object number: " << code << "\n";
        }
        static void showcount(void)    // static member function
        {
        cout << "count: " << count << "\n";
        }
};

int test :: count;
```

Remember, the following function definition will not work:

```
static void showcount()
{
     cout << code;      // code is not static
}
```

Arrays of object



Friend Function:

Applications -> Swapping data members of 2 classes, where a function is friend of both

A non-member (outside) function just declared as friend. Now this function is friend, hence now have full right access all members of a class.

```
class ABC
{
      .....
      .....
  public:
      .....
      .....
      friend void xyz(void); // declaration
};
```

Scope -> Where the function is declared (not in class)

- Cannot be called using object of class
- Its just a normal function, we used to use member of class using object name and '.' operator.
- Can be declared in private/ public area, doesn't affect its work.

We can have friend class to just like friend function

```
class Z
{
      .....
      friend class X;    // all member functions of X are
                         // friends to Z
};
```

Pointers to members of class: (different from pointer to a class)

int A:: *ip = &A::m;

ip is having address of m, which is data member of A.

ip acts like a class member now, we need a object to call ip.

```
int *ip = &m;   // won't work
```

As **m** is not a simple int datatype, it's a part of family, a class

USAGE:

```
cout << a.*ip;   // display
cout << a.m;     // same as above
```

Now, look at the following code:

```
ap = &a;                // ap is pointer to object a
cout << ap ->*ip;       // display m
cout << ap -> m; // same as above
```

See the second line in above code!! Important!

Local Classes

- A class defined inside a function or a block
- No static data members/ member function
- Enclosing function cannot access the private data members of class
  (Just make the enclosing function FRIEND for the access to private data members)

Example:

```
void test(int a)          // function
{
      .....
      .....
      class student       // local class
      {
            .....
            .....         // class definition

            .....
      };
      .....
      .....
      student s1(a);            // create student object
      .....                     // use student object
}
```