Lecture 1 OBJECT ORIENTED PROGRAMMING I Asst. Prof. Dunia Fadheel Saffo

القسم : علوم الحاسوب

المادة : البرمجة الشيئية I

المرحلة : الثانية

الفصل الدراسي : الاول

مدرس المادة : أ.م دنيا فضيل صفو

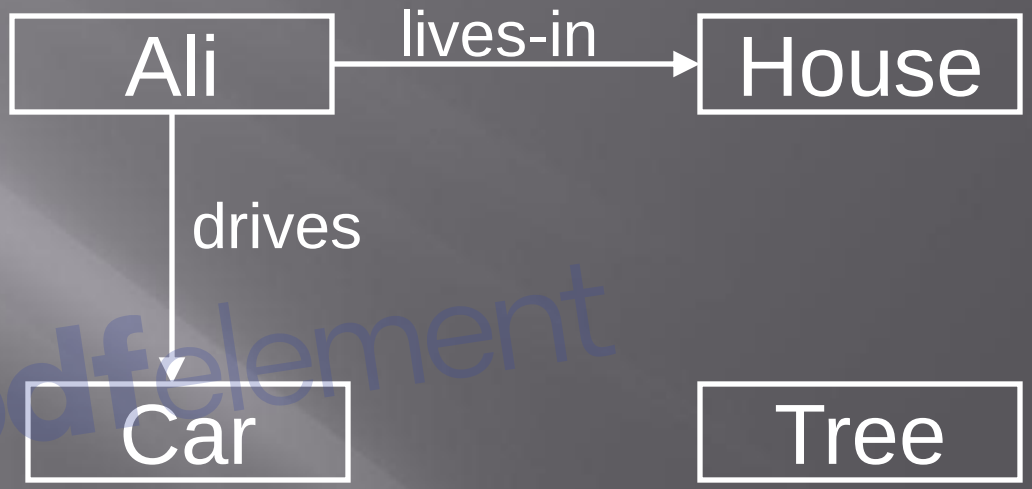
Lecture No. 1

What is Object-Orientation?

- A technique for system modeling
- OO model consists of several interacting objects

…Example – OO Model

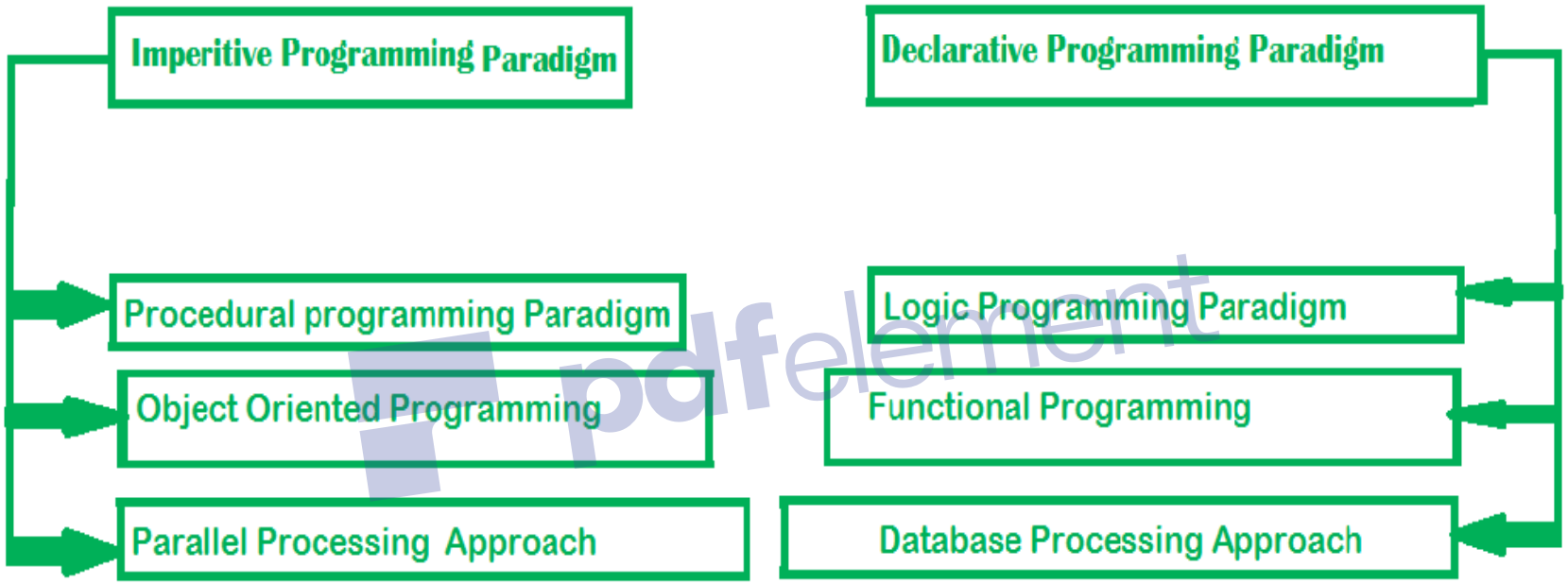
- Objects
  - Ali
  - House
  - Car
  - Tree
- Interactions
  - Ali lives in the house
  - Ali drives the car



Object-Orientation - Advantages

- People think in terms of objects
- OO models map to reality
- Therefore, OO models are
  - easy to develop
  - easy to understand

Programming Paradigms



Programming Fundamentals

Structured Programming

What is Structured Programming?

In structured programming,we divide the whole program into small modules, so that program become easy to understand.

Why we use Structured Programming?

- We use structured programming because it enables the programmer to understand the program easily.
- If a program consists of thousands of instructions and an error occurs than it is very difficult to find that error in the whole program but in structured programming we can easily detect the error and then go to that location and correct it.
- This saves a lot of time.

Function

- We divide the whole program in to small blocks called functions.
- Function is a block of statements that are executed to perform a task.
- Function plays an important role in structured programming.
- Using a function is like hiring a person to do a specific job.

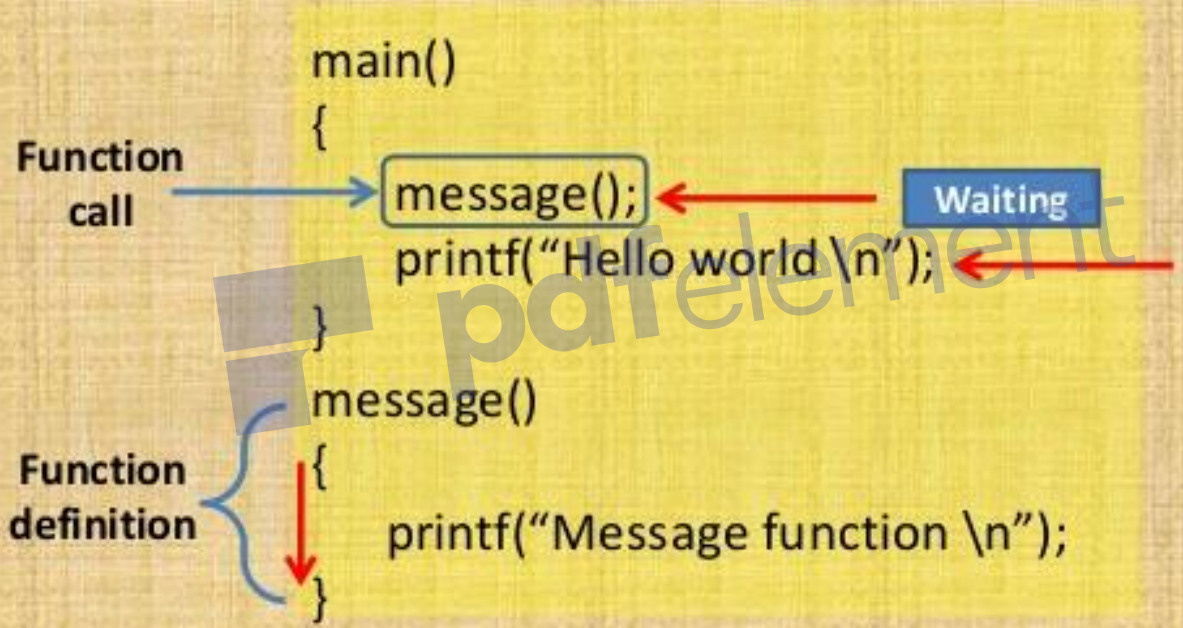
Function Call :

- When a function is called by its name the control moves to the function definition and executes all the statements written in it
- Semi colon is used after the name of function.

Function Definition :

- Function definition contains the instructions that are executed when a function is called
- Semi colon is not used after the name of function.

Function call and definition



Points to remember

- C program must contains at least one function
- Execution of C program begins with **main()** function.
- If there are more than one function then one function must be **main()**
- There is no limit on number of functions in C program
- Functions in a program are called in sequence as mentioned in **main()** function
- After the execution of function control returns to **main()**
- A function can call itself such a process is called recursion.

Function Call

```
main()
{
printf("During Summer ");
wakeup();
milk();
Study();
Play();
Sleep();
system("pause");
}
```

Function Definition

```
study()
{
printf("I study at 2'pm");
}
wakeup()
{
printf("I wake up at 6'am");
}
milk()
{
printf("I drink milk at 8'am");
}
Sleep()
{
printf("I sleep at 9'pm");
}
Play()
{
printf("I play at 3'pm");
}
```

Function Prototype

Function prototype contains following things about the function:

- The data type returned by the function
- The number of parameters received
- The data types of the parameters
- The order of the parameters
- If there is no datatype and return value then we use void at the place of datatype and perameters

```
int m  ( int , int );
main ( )
{ int a , b , Mul;
 cout<< " Enter the values of a and b ";
 cin >> a >> b ;
 Mul=m ( a , b );
 cout << " multiplication of "<<a <<" and "<< b <<"="<<Mul;
}
int m( int x , int y )
{ int multiply;
  multiply = x * y ;
  return  multiply;
 }
```

Function prototype

Function call

Function definition

Imp Point

- A function can return only one value at a time for example these statements are invalid:

```
Return(a,b);
Return(x,12);
```

ELEMENTS OF A PROCEDURE

- A name for the declared Procedure
- A body consisting of local declaration and statements
- The formal parameters which are place holders for actuals
- An optional result type

Example

```
int square ( int x)
{
    int sq;
    sq = x * x;
    return  sq;
}
```

Variable Scope

- Variable scope determine the area in a program where variable can be accessed.
- When a variable loses its scope, it means its data value is lost
- Common types of variables in C,
  - local
  - global
- Global variable can be accessed any where in a program
- Local variable can be accessed only in that function in which it is declared

Example Program

```
#include<stdio.h>
int a,b;
main()
{
printf("Enter the values of a & b");
scanf("%d%d",&a,&b);
sum(a,b);
system("pause");
  }
sum(inta,intb)
{
int c;
c=a+b;
printf("Sum is %d",c);
  }
```

Global Variable

Local Variable

PARAMETER PASSING METHODS

- **Formal Parameter :** A variable and its type as they appear in the prototype of the function or method.

Actual Parameter : The variable or expression corresponding to a formal parameter that appears in the function or method call in the calling environment.

- If communication is desired between the caller and the callee , arrangements must be made for passing values back and forth through the procedures parameters.

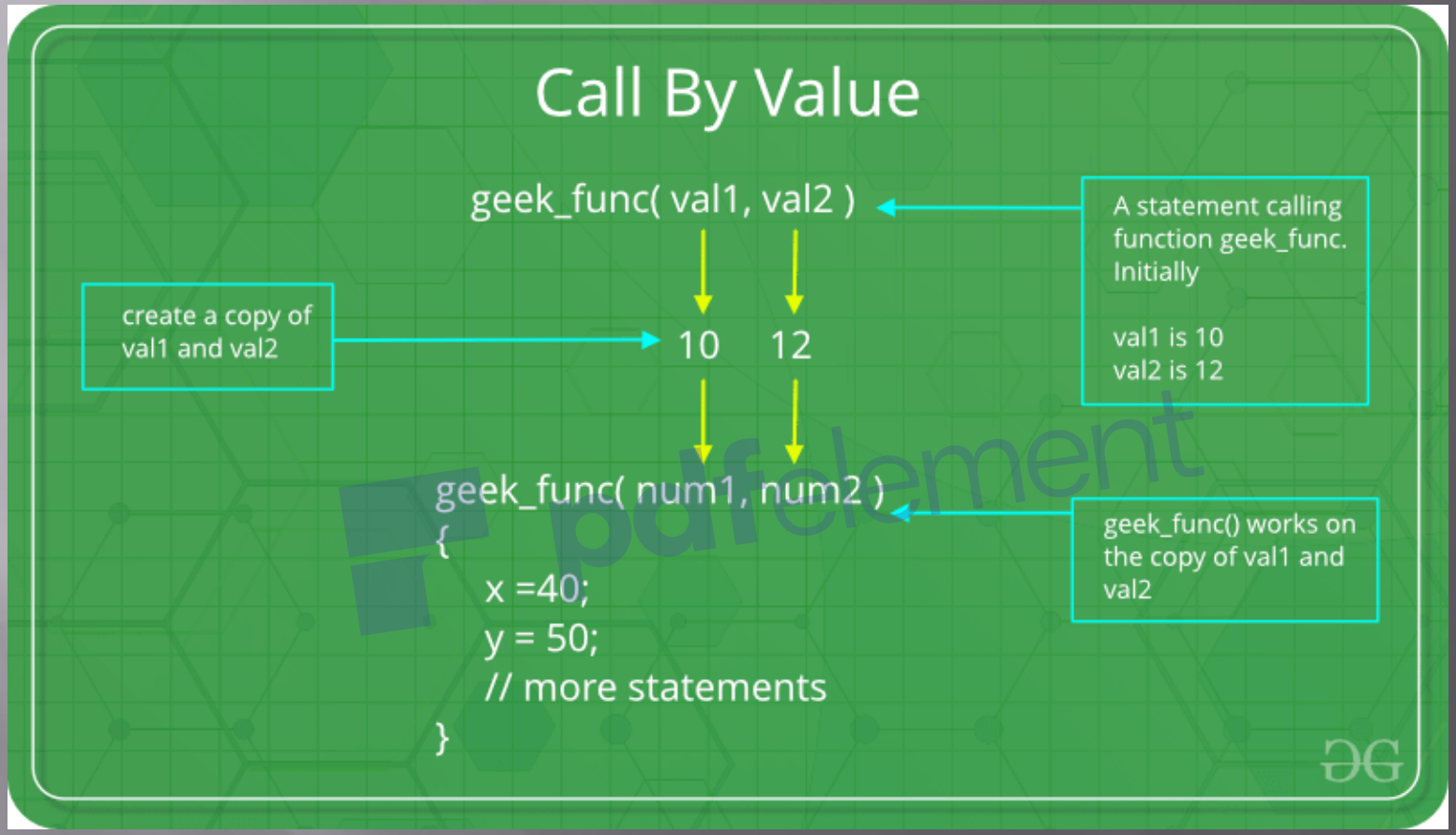
- Parameter passing refers to the matching of actuals with formals when a Procedure call occurs

- Different interpretations of what a parameter stands for leads to different parameter passing methods.

Call by Value
Call by Reference

VALUE PARAMETER

- This method uses *in-mode* semantics.
- Gets Own Memory location
- Gets initial value from corresponding actual position
- Uses but does not change the actual parameter
- Actual parameter s can be variables or expressions
- Changes made to formal parameter do not get transmitted back to the caller. Any modifications to the formal parameter variable inside the called function or method affect only the separate storage location and will not be reflected in the actual parameter in the calling environment. This method is also called as *call by value*



Call by value

```
void sum(int a, int b)
{
    a += b;
    cout<<“in function sum :“<<“a=“<<a<<“b=“<<b;
}
 main(void)
{
    int x = 5, y = 7;

    // Passing parameters
    sum(x, y);
    cout<<"In main, x = <<x<<“y=“<<y;
     }
```

Output:

In function sum: a = 12 b = 7

In main, x = 5 y = 7

Pass by reference

- This technique uses *in/out-mode* semantics.
- Changes made to formal parameter do get transmitted back to the caller through parameter passing.
- Any changes to the formal parameter are reflected in the actual parameter in the calling environment as formal parameter receives a reference (or pointer) to the actual data.
- This method is also called as **call by reference**.
- Shares the memory location of the actual Parameter
- The Actual Reference Parameter must have Location

Pass by reference

```
// C++ program to swap two numbers using
// pass by reference.
  #include <iostream>
using namespace std;
void swap(int& x, int& y)
{
    int z = x;
    x = y;
    y = z;
}
int main()
{
    int a = 45, b = 35;
    cout << "Before Swap\n";
    cout << "a = " << a << " b = " << b << "\n";
      swap(a, b);
      cout << "After Swap with pass by reference\n";
    cout << "a = " << a << " b = " << b << "\n";
}
```

Output:

Before Swap a = 45 b = 35

After Swap with pass by reference a = 35 b = 45

OBSERVATIONS

- Program execution always begins in the main
- Formal Parameters(function definition) and actual Parameters (function call) are matched by position. Their names need not agree
- Data types of parameters do not appear in the function call
- When a function completes the flow of control returns to the place that called it.

LECTURE 2
FUNCTIONS

EXAMPLE #1:

Define a function that prints the maximum value of two integers, your main should call the function.

```
void    max ( int x , int y ) { if (x > y ) cout <<x ;
                                else if (x < y ) cout << y ; }
main ( ) { int  m , n ;
           cout << "input two integers ";
           cin>> m >> n ;
           cout<<" the maximum value  of " << m <<
           "and"<< n <<"is ";
            max ( m , n );
            cout <<endl;
            cout <<" the maximam value of "<<  m+5 <<
            "and 25  is ";
            max ( m+5 , 25 );
           }
```


EXAMPLE #2:

Define each of the following functions:

a. A function that returns the maximum of two integers.

b. A function that returns the minimum of two integers.

Let your main call each defined function.

```
int   max  ( int x , int y ){ if (x >= y )  return x ;
                              else if (x < y ) return y ;
                                 }
   int   min  ( int x , int y ) { if  ( x <= y )   return x ;
                                  else  if (  y < x )  return  y ;  }
   main ( ) { int m , n ;
              cout <<" input two integer value ";
              cin >> m >> n ;
              cout << " The maximum of "<< m <<"and " <<n
<<"="<<  max ( m , n );
              cout << " The minimum of " << m << "and " << n
<<"="<< min ( m , n );
              if ( max ( m , 3 )  < min ( n , 2 ) )  cout<< m ;
                                                     else  cout << n ;
            }
```


EXAMPLE #3:

Define a function that returns the maximum and minimum of two integers.

```
void   min_max_1 ( int   x, int y ,int &L,  int  & s )
{ if (x>y ) {L=x ; s=y;}
   else {L=y ; s=x;}}
 int min_max_2( int x , int y ,int &s)
  { int L;
     if (x>y)  {L=x; s=y;}
      else (L=y ; s=x;}
       return L;}
main ( )  { int  x , y ,  small , larg;
              cout <<" input two integers";
                cin >>  x >> y ;
                 min_max_1(x,y,larg,small);
                 cout<<“max value=“<<larg<<“min value=“<<small;
                  larg = min_max_2 ( x,y , small);
                cout << " max. value in the array = "<< larg <<
                "and min. value = "<< small ;
          }
```

EXAMPLE #4:

Define a function that returns the maximum and minimum value in an array of 10 integers.

```
int   array_min_max ( int   r [10],  int  & s )
{ int L = r [0] ;
   for   ( int  i = 1; i< 10 ; i ++ )
   if ( r[i] > L  )  L=r [i];
  else if  ( r[i] < s )   s = r [i] ;
   return   L ; }

main ( )  { int a [10],  small;
             for  ( int i = 0 ; i<10 ; i++)
              { cout <<" input an array element ";
                 cin >>  a[i];}
                 small = a[0];
                 int  larg= array_min_max ( a , small);
                 cout << " max. value in the array = "<< larg <<
                 "and min. value = " << small ;
            }
```


EXAMPLE #4:

Define a function that returns the maximum and minimum value in an array of 10 integers.

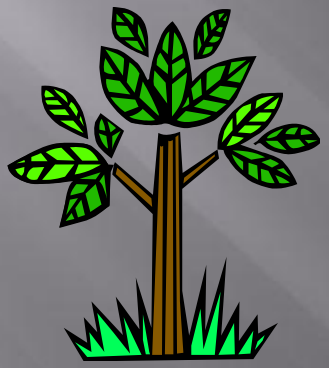
```
int   array_min_max ( int   r [10],  int  & s )
{ int L = r [0] ;
   for   ( int  i = 1; i< 10 ; i ++ )
   if ( r[i] > L  )  L=r [i];
  else if  ( r[i] < s )   s = r [i] ;
   return   L ; }

main ( )  { int a [10],  small;
             for  ( int i = 0 ; i<10 ; i++)
              { cout <<" input an array element ";
                 cin >>  a[i];}
                 small = a[0];
                 int  larg= array_min_max ( a , small);
                 cout << " max. value in the array = "<< larg <<
                 "and min. value = " << small ;
            }
```


Example #5:
Define a function that reads an array of 10 characters

```
void read_array( char ch[10])
{ for (int i=0;i<10;i++) { cout<<"input a character";
                           cin>>ch[i];}
void print_array(char ch[10])
{for (int i=0; i<10;i++) cout<<ch[i];}

main( ){ char  x[10],y[10],z[10];
          read_array(x);  read_array(y);
          read_array(z);
          print_array(x);print_array(y);print_array(z);
        }
```


LECTURE 3

BASICS OF OOP



Information Hiding

- Information is stored within the object
- It is hidden from the outside world
- It can only be manipulated by the object itself
- Ali's name is stored within his brain
- We can't access his name directly
- Rather we can ask him to tell his name
- A phone stores several phone numbers
- We can't read the numbers directly from the SIM card Rather phone-set
- Simplifies the model by hiding implementation details
- It is a barrier against change propagation

Encapsulation

- Data and behaviour are tightly coupled inside an object
- Both the information structure and implementation details of its operations are hidden from the outer world
- EXAMPLE:
- Ali stores his personal information and knows how to translate it to the desired language
- We don't know
  - How the data is stored
  - How Ali translates this information

ADVANTAGES:

- Simplicity and clarity
- Low complexity
- Better understanding

Object has an Interface

- An object encapsulates data and behaviour
- So how objects interact with each other?
- Each object provides an interface (operations)
- Other objects communicate through this interface

IMPLEMENTATION

- Data Structure
  - SIM card
- Functionality
  - Read/write circuitry

Example – Separation of Interface & Implementation

- A driver can apply brakes independent of brakes type (simple, disk)
- Again, reason is the same interface

Messages

- Objects communicate through messages
- They send messages (stimuli) by invoking appropriate operations on the target object
- The number and kind of messages that can be sent to an object depends upon its interface

Example

- A Person sends message "stop" to a Car by applying brakes
- A Person sends message "place call" to a Phone by pressing appropriate button

Abstraction

- Abstraction is a way to cope with complexity.
- Principle of abstraction:

"Capture only those details about an object that are relevant t

advantages

- Simplifies the model by hiding irrelevant details

- Abstraction provides the freedom to defer implementation decisions by avoiding commitment to details

- Abstraction provides the freedom to defer implementation decisions by avoiding commitment to details

o current perspective"

Example – Abstraction

Driver's View

Engineer's View

Classes

- In an OO model, some of the objects exhibit identical characteristics (information structure and behaviour)

- We say that they belong to the same class

Example

- Ali studies mathematics
- Inam studies physics
- Sohail studies chemistry

- Each one is a Student
- We say these objects are *instances* of the Student class
- Ihsan teaches mathematics
- Amir teaches computer science
- Atif teaches physics

- Each one is a teacher
- We say these objects are *instances* of the Teacher class

Class

- Class is a tool to realize objects
- Class is a tool for defining a new type

- **EXAMPLE**
- Lion is an object
- Student is an object
- Both has some attributes and some behaviors
- **USES**
- The problem becomes easy to understand
- Interactions can be easily modeled

Type in C++

- Mechanism for user defined types are
  - Structures
  - Classes
- Built-in types are like int, float and double
- User defined type can be
  - Student in student management system
  - Circle in a drawing software

Defining a New User Defined Type


```
class ClassName
{
    ...
    DataType MemberVariable;
    ReturnType MemberFunction();
    ...
};
```

Syntax

Syntax

Example

```
class Student
{
   int no;
   char name[20];
   float CGPA;
   char address[20];
…
   void setName(char newName[20]);
   void setRollNo(int newNo);
…
};
```

Member variables

Member Functions

Why Member Function

- They model the behaviors of an object
- Objects can make their data invisible
- Object remains in consistent state

Object is an instantiation of a user defined type or a class

Declaring class variables

Variables of classes (objects) are declared just like variables of structures and built-in data types

TypeName VaraibaleName;

int *var;*

Student st;

Declaring class variables

- Variables of classes (objects) are declared just like variables of structures and built-in data types

TypeName VaraibaleName;

int *var;*

Student st;

LECTURE 4

INTRODUCTION

- Class
  - Concept
  - Definition
- Data members
- Member Functions
- Access specifier

CLASSES AND OBJECTS

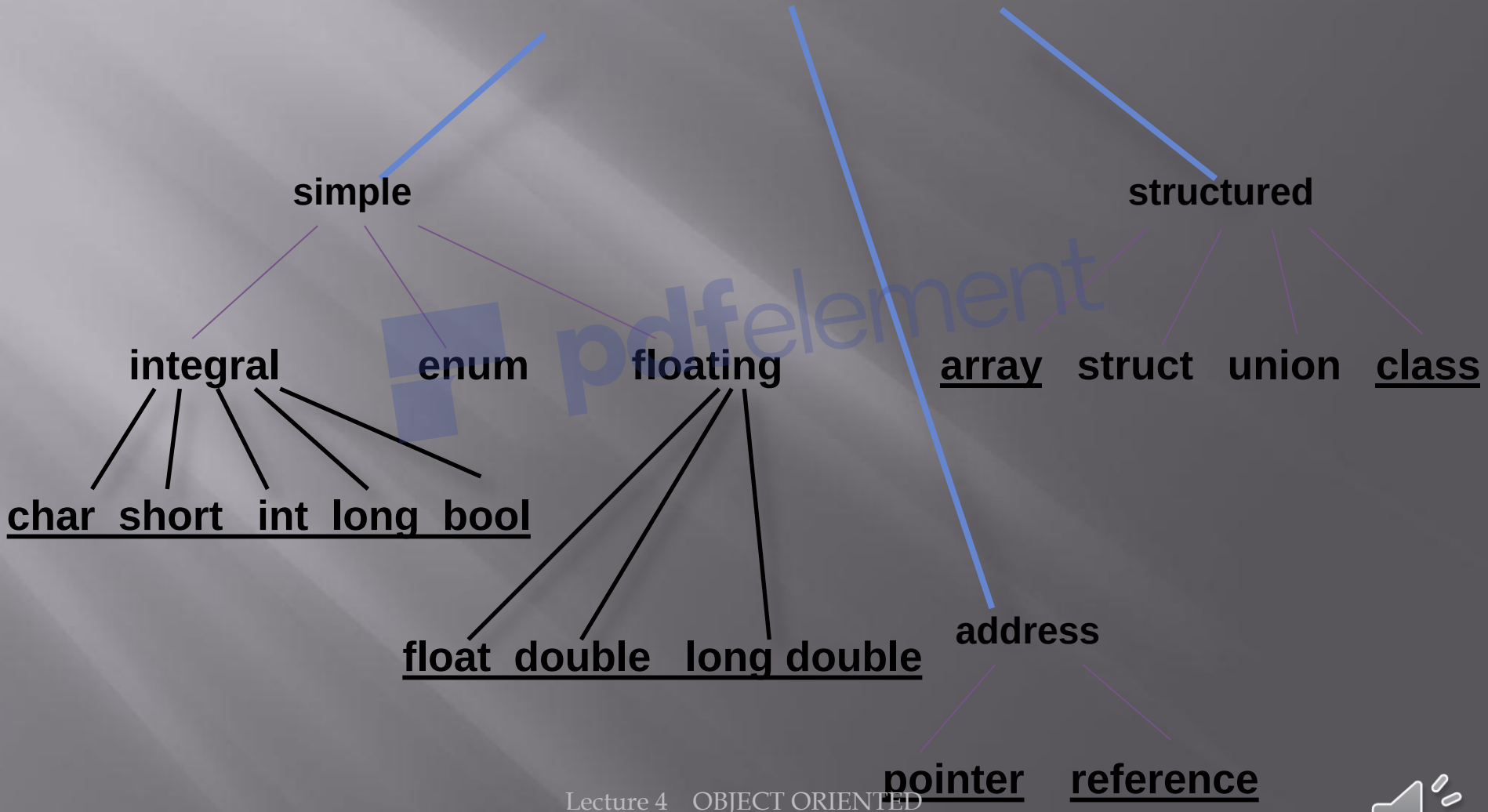
- The main purpose of C++ programming is to add object orientation to the C programming language.
- A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package.
- The data and functions within a class are called members of the class.

CLASS DEFINITIONS

- When you define a class, you define a **blueprint for a data type**. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will **consist of** and what **operations can be performed** on such an object.

- A class definition starts with the keyword **class** followed by the **class name;**

C++ Data Types



Why Member Function

- They model the behaviors of an object
- Objects can make their data invisible
- Object remains in consistent state

Object is an instantiation of a user defined type or a class

Declaring class variables

- Variables of classes (objects) are declared just like variables of structures and built-in data types

TypeName VaraibaleName;

int *var*;

Student st;

Accessing members

- Members of an object can be accessed using
  - dot operator (.) to access via the variable name(object)
  - arrow operator (->) to access via a pointer to an object
- Member variables and member functions are accessed in a similar fashion

CLASS ACCESS SPECIFIERS

- **Data hiding** is one of the important features of **Object Oriented Programming** which allows **preventing** the functions of a program to **access directly the internal representation of a class type**.

- A class can have **multiple public, protected, or private labeled sections**. Each section remains in **effect until either** another section label or the closing right brace of the class body is seen.

Access specifiers

- There are three access specifiers
  - 'public' is used to tell that member can be accessed whenever you have access to the object
  - 'private' is used to tell that member can only be accessed from a member function
  - 'protected' to be discussed when we cover inheritance

CLASS ACCESS MODIFIERS

```
1.  class Base {
2.    public:
3.    // public members go here
4.
5.    protected:
6.    // protected members go here
7.
8.    private:
9.    // private members go here
10. };
```

Default access specifiers

- When no access specifier is mentioned then by default the member is considered private member

Example

```
class Student
{
   char   name[10];
   int    No;
};
```

```
class Student
{
private:
   char   name[10];
   int    No;
};
```

Member Functions

- Member functions are the functions that operate on the data encapsulated in the class
- Public member functions are the interface to the class
- Define member function inside the class definition

OR

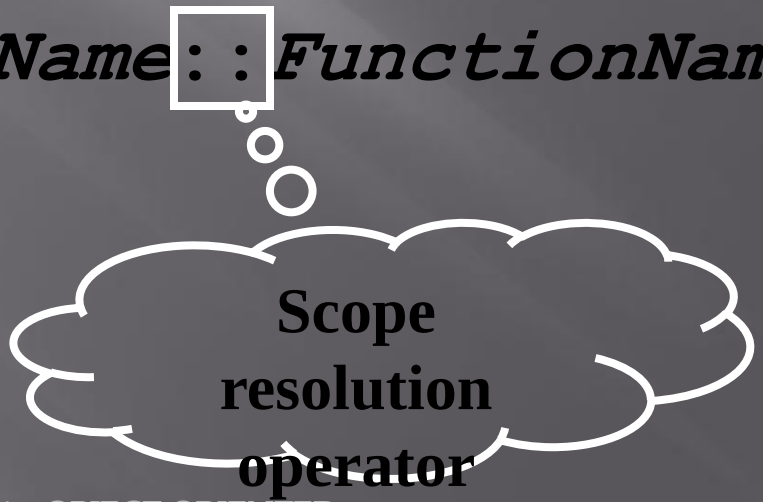
- Define member function outside the class definition
  - But they must be declared inside class definition

Function Inside Class Body

```
class ClassName {
  …
  public:
  ReturnType FunctionName() {
    …
  }
};
```


Function Outside Class Body

```
class ClassName{
   …
   public:
   ReturnType FunctionName();
};
ReturnType   ClassName::FunctionName()
{
   …
}
```



Define a Member Function


```
class Rectangle
{
    private:
       int width, length;
    public:
       void set (int w, int l);
       int area() {return width*length; }
};
```

class name

member function name

```
void Rectangle :: set (int w, int l)
{
    width = w;
    length = l;
}
```

inline

scope operator

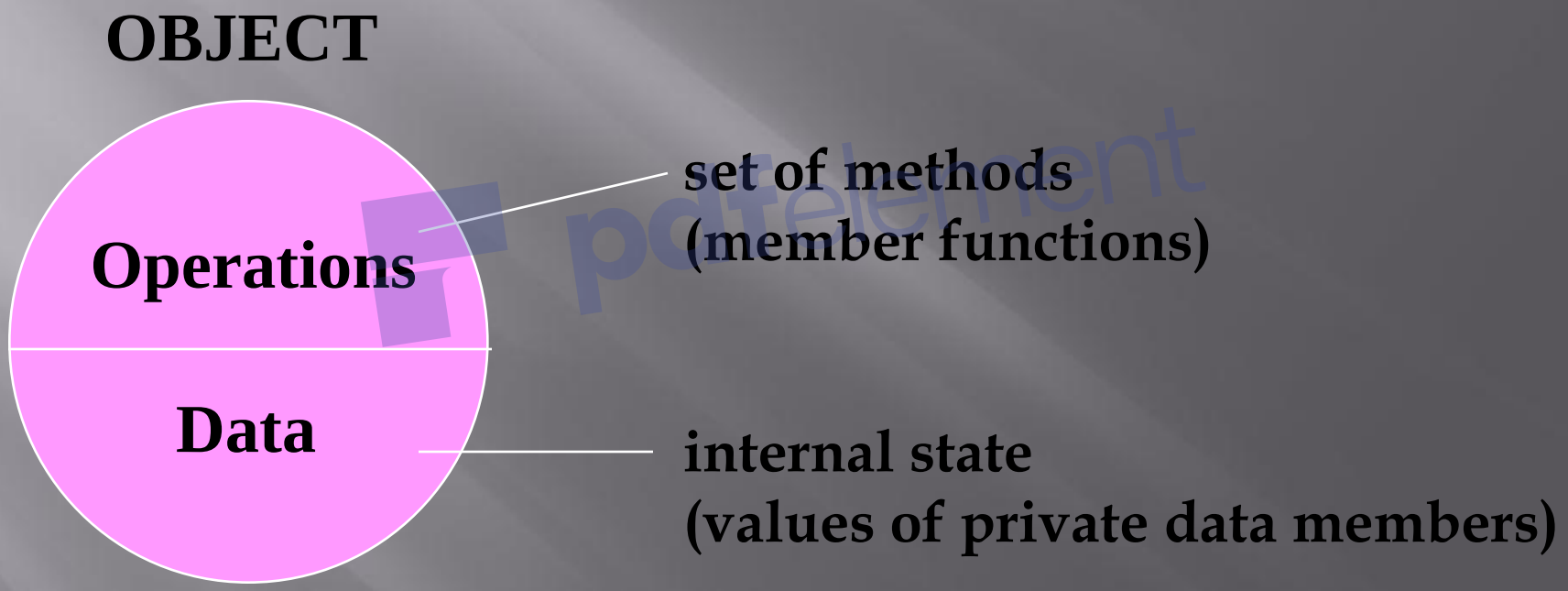
CLASS DEFINITIONS EXAMPLE

```
1. class Box
2. {
3.    public:
4.       double length;   // Length of a box
5.       double breadth;  // Breadth of a box
6.       double height;   // Height of a box
7. };
```


DEFINE C++ OBJECTS:

```
1. Box Box1;        // Declare Box1 of type Box
2. Box Box2;        // Declare Box2 of type Box
```

What is an object?



ACCESSING THE DATA MEMBERS

```
1. #include <iostream>
2. using namespace std;
3. class Box
4. {
5.    public:
6.       double length;   // Length of a box
7.       double breadth;  // Breadth of a box
8.       double height;   // Height of a box
9. };
```

ACCESSING THE DATA MEMBERS CON…

```
1. int main( )
2. {
3.    Box Box1;        // Declare Box1 of type Box
4.    Box Box2;        // Declare Box2 of type Box
5.    double volume = 0.0;     // Store the volume of a box here


6.  // box 1 specification
7.  Box1.height = 5.0;
8.    Box1.length = 6.0;
9.    Box1.breadth = 7.0;
```

ACCESSING THE DATA MEMBERS CON…

```
1.  // box 2 specification
2.     Box2.height = 10.0;
3.     Box2.length = 12.0;
4.     Box2.breadth = 13.0;

5.     // volume of box 1
6.     volume = Box1.height * Box1.length * Box1.breadth;
7.     cout << "Volume of Box1 : " << volume <<endl;
8.     // volume of box 2
9.     volume = Box2.height * Box2.length * Box2.breadth;
10.    cout << "Volume of Box2 : " << volume <<endl;
11.    return 0;
12. }
```

```
Volume of Box1 : 210
Volume of Box2 : 1560
```


CLASS MEMBER FUNCTIONS

- A **member function** of a class is a function that has its definition or its prototype **within the class** definition like any other variable.

```
class Box
{
   public:
      double length;        // Length of a box
      double breadth;       // Breadth of a box
      double height;        // Height of a box
      double getVolume(void);// Returns box volume
};
```


CLASS MEMBER FUNCTIONS CON…

Member functions can be defined within the class definition or separately using scope resolution operator, ::.

Method 1

```
class Box
{
  public:
    double length;     // Length of a box
    double breadth;    // Breadth of a box
    double height;     // Height of a box

    double getVolume(void)
    {
      return length * breadth * height;
    }
};
```

Method 2

```
double Box::getVolume(void)
{
    return length * breadth * height;
}
```

```
Box myBox;          // Create an object
myBox.getVolume();
// Call member function for the object
```

Example

- Define a class of student that has a key number. This class should have a function that can be used to set the student number

Example

```
class Student
{
   int rollNo;
   char name[20];
   float CGPA;
   char address[20];
…
   void setName(char newName[20]);
   void setRollNo(int newRollNo);
…
};
```

Member variables

Member Functions

Example

```
class Student{
private:
   char name [10];
   int  No;
public:
   void setName(char x[10]);
   void setNo(int);
...
};
```

Cannot be accessed outside class

Can be accessed outside class

- **Notes:**
- Accessing a private member from outside member function is a syntax error
- You can use access specifiers as many times as you like
- Each access specifier must be followed by a ':'

Example

```
class Student
{
  char  name[10];
  int   No;
  void SetName(char x[10]);
};
Student a;
a.SetName("Ali");
```


Example

```
class Student
{
   char name[10];
   int  No;
public:
   void setName(char  x[10])
                {strcpy(name,x)}
};
Student aStudent;
aStudent.SetName("Ali");
```


Example

```
class Student{
   …
   int No;
public:
   void setNo(int aNo);
};
void Student::setNo(int aNo){
   …
   No = aNo;
}
```


THE PUBLIC MEMBERS EXAMPLE

- A **public** member is accessible from anywhere **outside the class** but **within a program**.

```
#include <iostream>
using namespace std;
class Line
{
   public:
      double length;
      void setLength( double len );
      double getLength( void );
};
```

THE PUBLIC MEMBERS EXAMPLE CON…

```
// Member functions definitions
double Line::getLength(void)
{
    return length ;
}

void Line::setLength( double len )
{
    length = len;
}
```

THE PUBLIC MEMBERS EXAMPLE CON…

```
// Main function for the program
int main( )
{   Line line;

   // set line length
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;

   // set line length without member function
   line.length = 10.0; // OK: because length is public
   cout << "Length of line : " << line.length <<endl;
   return 0; }
```

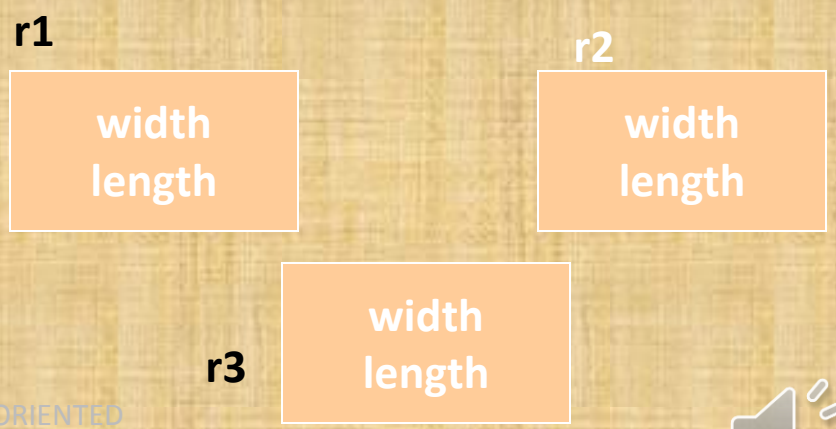
```
Length of line : 6
Length of line : 10
```

LECTURE 5

Data Members

```
class Rectangle
{
    private:
        int width;
        int length;
public:
        void set(int w, int l);
        int area();
    }
```

```
Rectangle  r1;
Rectangle  r2;
Rectangle  r3;
```



Member Function

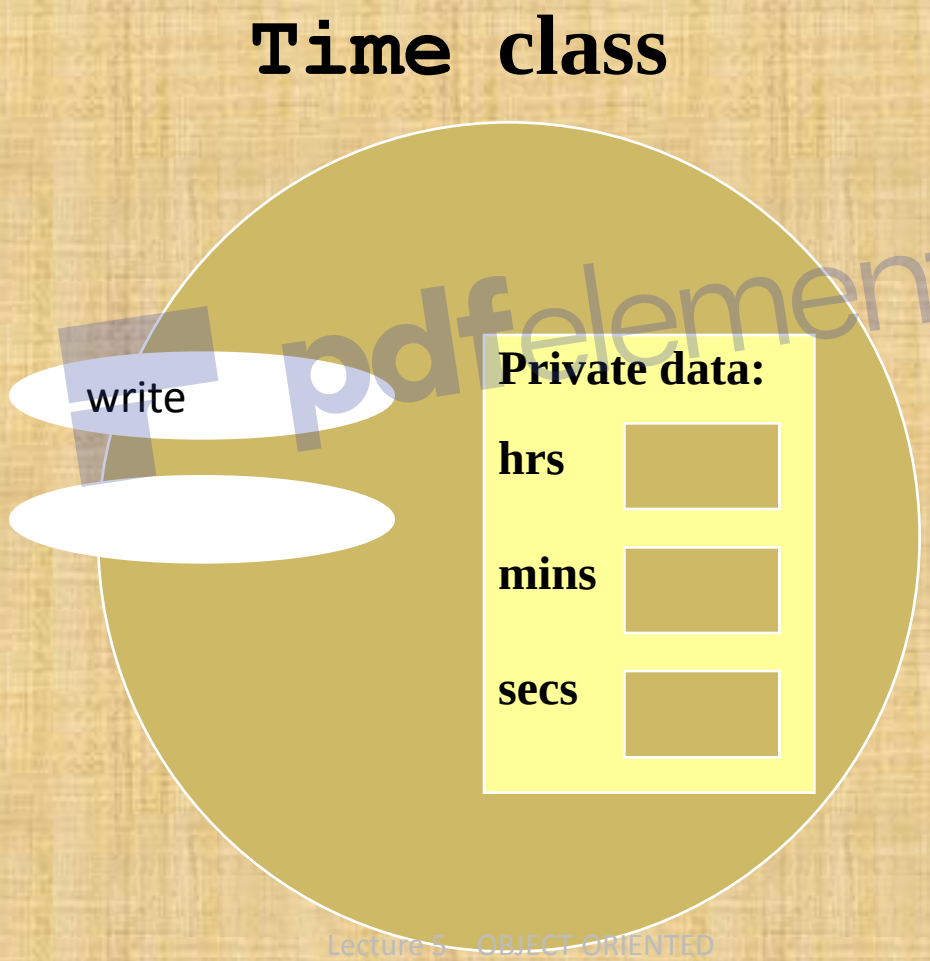
```
class  Time
{
  private :
     int     hrs, mins, secs ;

  public :
     void      Write ( )   ;
} ;
```

function declaration

function definition

```
void Time :: Write( )
{
    cout <<hrs << ":" << mins << ":" << secs << endl;
}
```

Class Interface Diagram



THE PRIVATE MEMBERS

- A **private** member variable or function cannot be accessed, or even viewed from outside the class. Only the class and friend functions can access private members.
- **By default all the members** of a class would be **private**, for example in the following class **width** is a **private** member.

```
class Box
{  double width;
   public:
      double length;
      void setWidth( double wid );
      double getWidth( void );
};
```

THE PRIVATE MEMBERS EXAMPLE

```
#include <iostream>
using namespace std;
class Box
{   public:
      double length;
      void setWidth( double wid );
      double getWidth( void );
   private:
      double width;
};
// Member functions definitions
double Box::getWidth(void)
{
    return width ;
}
```

THE PRIVATE MEMBERS EXAMPLE CON…

```
void Box::setWidth( double wid ) {
    width = wid;
}
int main( ) {
   Box box;
   // set box length without member function
   box.length = 10.0; // OK: because length is public
   cout << "Length of box : " << box.length <<endl;

   // set box width without member function
   // box.width = 10.0; // Error: because width is private
   box.setWidth(10.0);  // Use member function to set it.
   cout << "Width of box : " << box.getWidth() <<endl;
   return 0; }
```

```
Length of box : 10
Width of box : 10
```

Declaration of an Object

```
class Rectangle
{
    private:
      int width;
      int length;
    public:
     void set(int w, int l);
      int area(){return width*length ;}
};
void Rectangle::set(int w,int l)
{width=w;
Length=l;}
```

```
main()
{
    Rectangle r1;
    Rectangle r2;

   r1.set(5, 8);
    cout<<r1.area()<<endl;

    r2.set(8,10);
    cout<<r2.area()<<endl;
}
```

Another Example

```
#include <iostream.h>

class circle
{
    private:
       double radius;

    public:
       void store(double);
       double area(void);
       void display(void);

};
```

```
// member function definitions

void circle::store(double r)
{
    radius = r;
}
double circle::area(void)
{
    return 3.14*radius*radius;
}
void circle::display(void)
{
    cout << “r = “ << radius << endl;
}
```

```
int main(void) {
    circle c;   // an object of circle class
    c.store(5.0);
    cout << "The area of circle c is " << c.area() << endl;
    c.display();
}
```

Another Example

```
  #include <iostream.h>
class circle
  {
      private:
        double radius;
 public:
        void store(double);
        double area(void);
        void display(void);
        double get_r( );
};
```

```
// member function definitions

void circle::store(double r)
{  radius = r;}

double circle::area(void)
{ return 3.14*radius*radius;}

void circle::display(void)
{cout <<“r = “ << radius << endl;}
double circle:: get_r ( )
{ return radius ; }
```

```
int main(void) {
    circle c1, c2;   //  objects of circle class
    c1.store(5.0); c2.store(9.0);
//   if (c1.radius > c2.radius)    error radius is private
    if  (c1.get_r( ) > c2.get_r()) cout << c1._get_r();
    else if (c1.get_r ()< c2.get_r( ) ) cout<< c2.get_r( );
     else cout<<“equivalent”;}

}
```

Another Example

```
#include <iostream.h>
class circle
  {
      private:  double radius;
 public:
         void  read( );
         void store(double);
         double area(void);
         void display(void);
         double get_r( );
};
```

```
// member function definitions
void circle:: read( )
{ cin >> radius;}
void circle::store(double r)
{  radius = r;}

double circle::area(void)
{ return 3.14*radius*radius;}

void circle::display(void)
{cout <<“r = “ << radius << endl;}
double circle:: get_r ( )
{ return radius ; }
```

```
int main(void) {
    circle c1, c2;   // an object of circle class
    c1.store(5.0); c2.read();
    double r1=c1.get_r( );
    double r2=c2.get_r( );
    if (r1>r2) cout r1;
    else if ( r1< r2) cout <<r2;
    else cout <<“equivalent”;
}
```

Another Example

```
#include <iostream.h>

class circle
{
    private:
       double radius;
        void store(double);
    public:
       void call_store(double);
       double area(void);
       void display(void);

};
```

```
// member function definitions
void circle::store(double r)
{    radius = r;   }

double circle::area(void)
{  return 3.14*radius*radius; }

void circle::display(void)
{ cout << "r = " << radius << endl; }

void circle::call_store ( double r )
{  store (r ) ; }
```

```
int main(void) {
    circle c;   // an object of circle class
    c.store(5.0);//syntax error
    c.call_store ( 5.0 );
    cout << "The area of circle c is " << c.area() << endl;
    c.display();  }
```

Example : class Time Specification

```
class  time
{
  public :
      void              set ( int  hours , int  minutes , int  seconds ) ;
      void              increment ( ) ;
      void              write ( )   ;
 private :
     int           hrs ;
      int           min ;
      int           sec ;
} ;
void  time : : set( int hours,  int minutes,  int seconds)
{ hrs=hours;  min=minutes;  sec=seconds;   }
```

Home work

- 1. define function increment that adds 1 second to the assigned time data members.
- 2. define function write that prints data members of time object.
- Let your main function create two objects of type time, and make a call to each defined function in the class.

Object Initialization


```
#include <iostream.h>

class circle
{
    public:
       double radius;
};
```

1. By Assignment

- Only work for public data members
- No control over the operations on data members


```
int main()
{
   circle c1;                    // Declare an instance of the class circle
   c1.radius = 5;                // Initialize by assignment

}
```


Object Initialization


```
#include <iostream.h>

class circle
{
  private:
      double radius;

   public:
      void set (double r)
          {radius = r;}
      double get_r ()
          {return radius;}

};
```

2. By Public Member Functions


```
int main(void) {
    circle c;              // an object of circle class
    c.set(5.0);            // initialize an object with a public member function
    cout << "The radius of circle c is " << c.get_r() << endl;
      // access a private data member with an accessor
}
```


LECTURE 6

Object Initialization

1. By Assignment

```
#include <iostream.h>

class circle
{
    public:
       double radius;
};
```

- Only work for public data members
- No control over the operations on data members


```
int main()
{
   circle c1;                  // Declare an instance of the class circle
   c1.radius = 5;              // Initialize by assignment

}
```


Object Initialization


```
#include <iostream.h>

class circle
{
  private:
      double radius;

   public:
      void set (double r)
          {radius = r;}
      double get_r ()
          {return radius;}

};
```

2. By Public Member Functions


```
int main(void) {
    circle c;              // an object of circle class
    c.set(5.0);            // initialize an object with a public member function
    cout << "The radius of circle c is " << c.get_r() << endl;
       // access a private data member with an accessor
}
```

CONSTRUCTOR

Function Overloading

- Is the process of using the same name for two or more functions
- Requires each redefinition of a function to use a different function signature that is:
  - different types of parameters,
  - or sequence of parameters,
  - or number of parameters
- Is used so that a programmer does not have to remember multiple function names

Constructor

- Constructor is a special function having same name as the class name
- Constructor does not have return type
- Constructors are commonly public members
- Constructor is used to initialize the objects of a class
- Constructor is used to ensure that object is in well defined state at the time of creation
- Constructor is automatically called when the object is created
- Constructor are not usually called explicitly

Example

```
class Student{
   int number;
public:
   Student(){
      number = 0;
      …
   }
};
int main()
{
   Student aStudent;
   /*constructor is implicitly called at this
   point*/
}
```


Object Initialization


```
class Rectangle
{
    private:
      int width;
      int length;
    public:
      Rectangle()
    {width=0;length=0;}
      Rectangle(int w, int l)
       { width=w; length=l ; }
      void set(int w, int l);
      int area();
}
main( ){ Rectangle r1 , r2(3,5);}
```

3. By Constructor

- Default constructor
- Copy constructor
- Constructor with parameters

They are publicly accessible

Have the same name as the class

There is no return type

Are used to initialize class data members

They have different signatures

Default Constructor

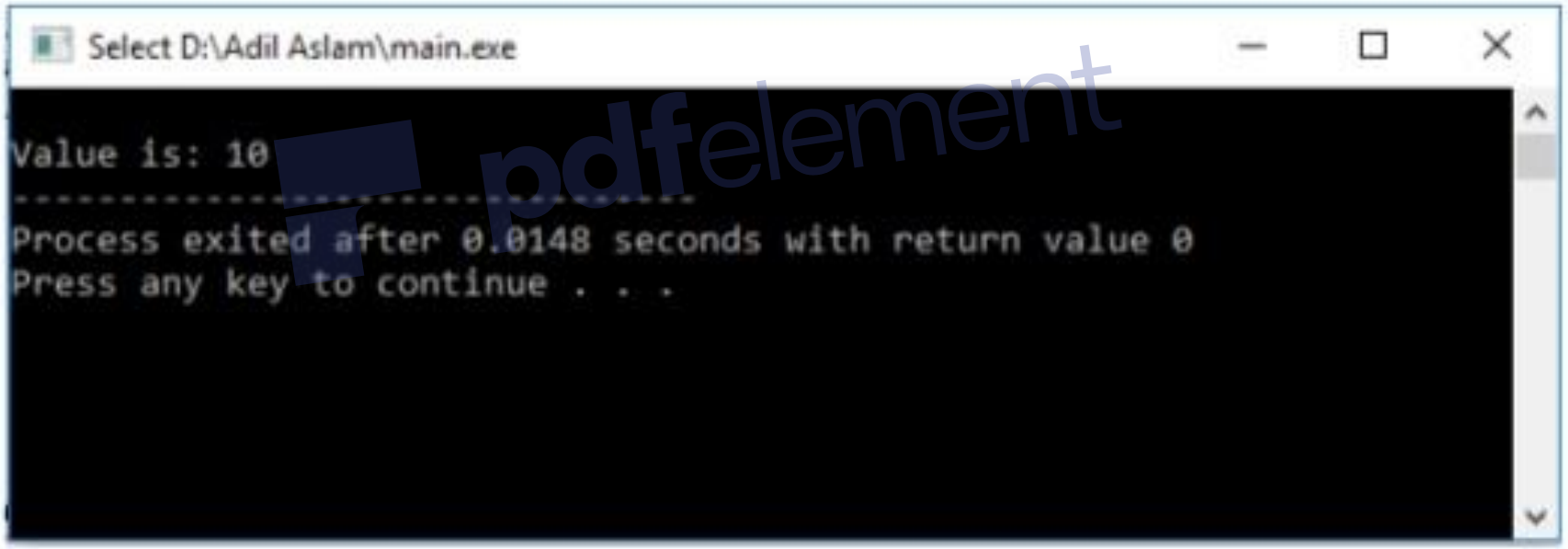
- Constructor without any argument is called default constructor
- If we do not define a default constructor the compiler will generate a default constructor
- This compiler generated default constructor initialize the data members to their default values

Default Constructor Example

```
class Cube
 {
        public:
        int side;
        public:
        Cube()
         {
                side=10;
         }
};
int main()
 {
        Cube c;
        cout <<"\nValue is: "<<c.side;
 }
```

Default Constructor

Output of the Previous Program is :



C++ Parameterize Constructor

- A constructor that receives arguments/parameters, is called parameterized constructor.
- Construct with parameter is called parameterize constructor.
- All constructor executes at the time of object creation.
- When we will create parameterized constructor then we have to pass argument value at the time of object creation of that class.

Another Example of Default Constructor-1

```
class Student
{
        public:
        int Roll;
        string Name;
        float Marks;
        public:
        Student()
        {
                Roll = 1;
                Name="Adil";
                Marks = 85;
        }
```

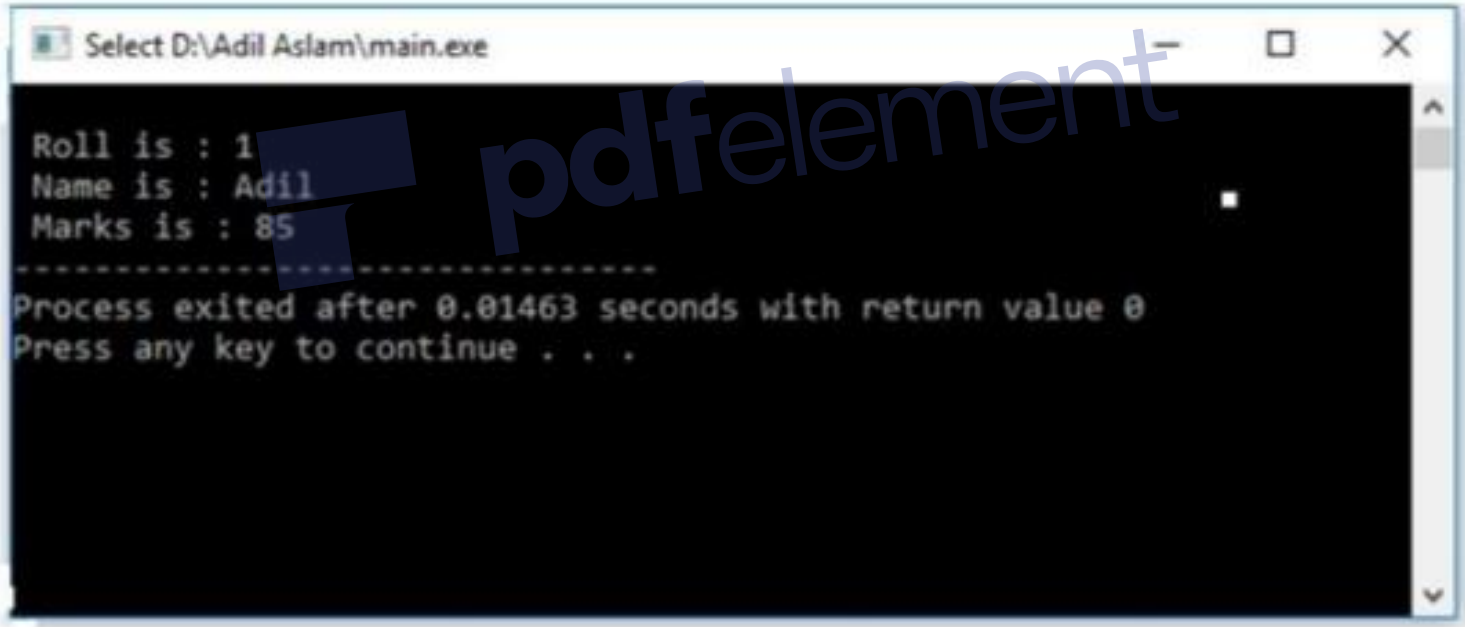
Default Constructor

Another Example of Default Constructor-2

```
void Display()
  {
        cout<<"\n Roll is : "<<Roll;
        cout<<"\n Name is : "<<Name;
        cout<<"\n Marks is : "<<Marks;
  }
};
int main()
 {

   Student S;        //Creating Object
   S.Display();      //Displaying Student Details
   return 0;

}
```

Object Oriented Programming in C++

Output of the Previous Program is :



C++ Parameterize Constructor

Syntax:

```
class Student
{
    Student(int a , int b)  //parameterized constructor
    {
       //Statements
    }
};
```

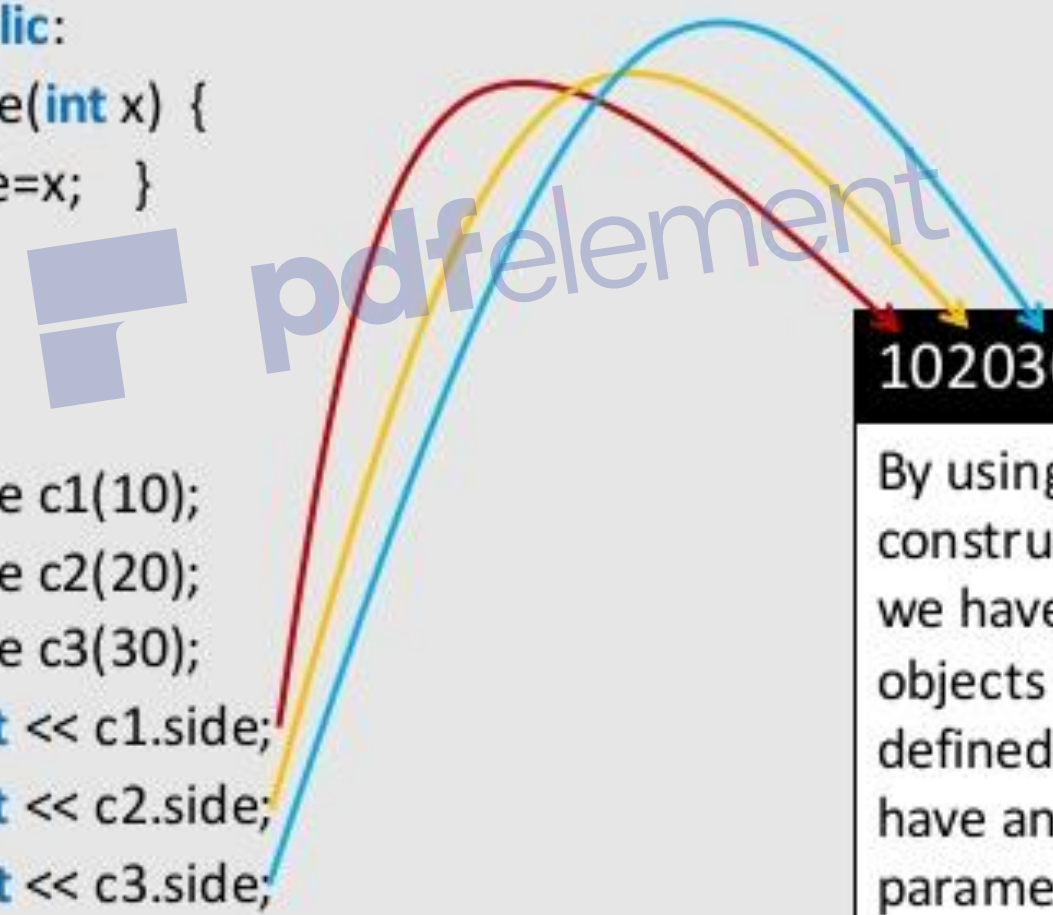
Parameterize Constructor Example

```
class Cube  {
        public:
        int side;
        public:
        Cube(int x)  {
         side=x;   }
 };
int main()
{
        Cube c1(10);
        Cube c2(20);
        Cube c3(30);
        cout << c1.side;
        cout << c2.side;
        cout << c3.side;
}
```

Parameterized Constructor

Parameterize Constructor Example

```
class Cube  {
        public:
        int side;
        public:
        Cube(int x)  {
         side=x;   }
 };
int main()
 {
        Cube c1(10);
        Cube c2(20);
        Cube c3(30);
        cout << c1.side;
        cout << c2.side;
        cout << c3.side;
 }
```



102030

By using parameterized constructor in this case, we have initialized 3 objects with user defined values. We can have any number of parameters in a constructor.

Example of Parameterized Constructor-1

```
class Student
{
        public:
        int Roll;
        string Name;
        float Marks;
        public:
        Student(int r, string nm, float m)
        {
                Roll = r;
                Name=nm;
                Marks = m;
        }
```

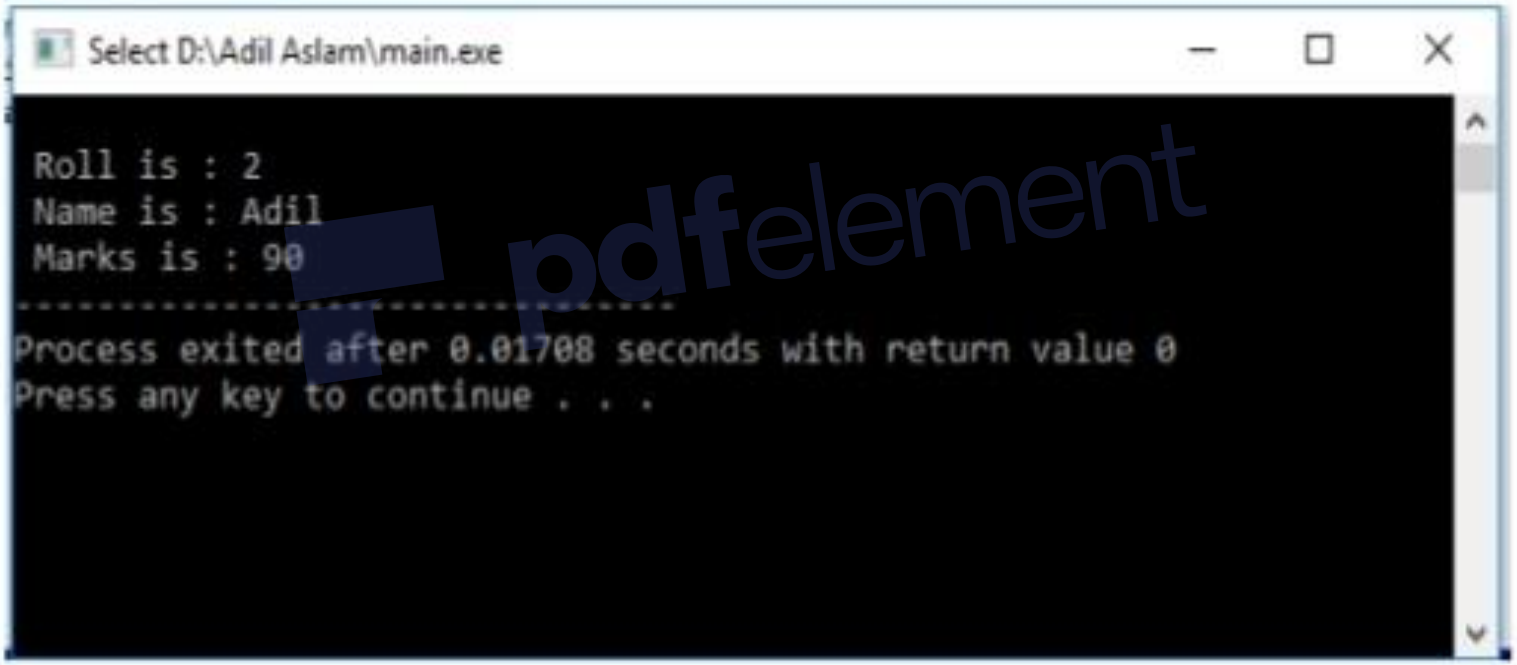
Parameterized Constructor with Three Parameters

Example of Parameterized Constructor-2

```
void Display() {
                cout<<"\n Roll is : "<<Roll;
                cout<<"\n Name is : "<<Name;
                cout<<"\n Marks is : "<<Marks;
        }
};   //end of class
int main()
{
                Student S(2,"Adil",90);
                S.Display();     //Displaying Student Details
                return 0;

}
```

In parameterize constructor, we have to pass values to the constructor through object.

Output of the Previous Program is :



Constructor Overloading In C++

- Constructor can be overloaded in a similar way as function overloading.
- Overloaded constructors have the same name (name of the class) but different number of arguments.
- Depending upon the number and type of arguments passed, specific constructor is called.
- Since, there are multiple constructors present, argument to the constructor should also be passed while creating an object.

Constructor Overloading

- Constructors can have parameters
- These parameters are used to initialize the data members with user supplied data

Object Initialization

```
class Rectangle
{
    private:
       int width;
       int length;
    public:
       Rectangle(int w, int l)
            {width =w; length=l;}
       void set(int w, int l);
       int area();
}
```

If any constructor with any number of parameters is declared, no default constructor will exist, unless you define it.

```
Rectangle r4;      // error
```

3. Initialize with constructor

```
Rectangle r5(60,80);
```

Object Initialization

Write your own constructors

```
class Rectangle
{
    private:
       int width;
       int length;
    public:
       Rectangle();
       Rectangle(int w, int l);
       void set(int w, int l);
       int area();
}
```

```
Rectangle :: Rectangle()
{
    width = 20;
    length = 50;
};
```

```
Rectangle  r7;
```

r7

width = 20
length = 50

Object Initialization


```
class Account
{
     private:
       char name[10];
       double balance;
        int id;
     public:
       Account();

       Account( char person[10]);
}
```

With constructors, we have more control over the data members


```
Account :: Account()
{
     name = NULL; balance = 0.0;
     id = 0;
};
```

```
Account :: Account( char person[10])
{

     strcpy (name, person);
     balance = 0.0;
     id = 0;
};
```

So far, …

- An object can be initialized by a class constructor
  - default constructor
  - copy constructor
  - constructor with parameters
- Resources are allocated when an object is initialized
- Resources should be revoked when an object is about to end its lifetime

CONSTRUCTOR OVERLOADING

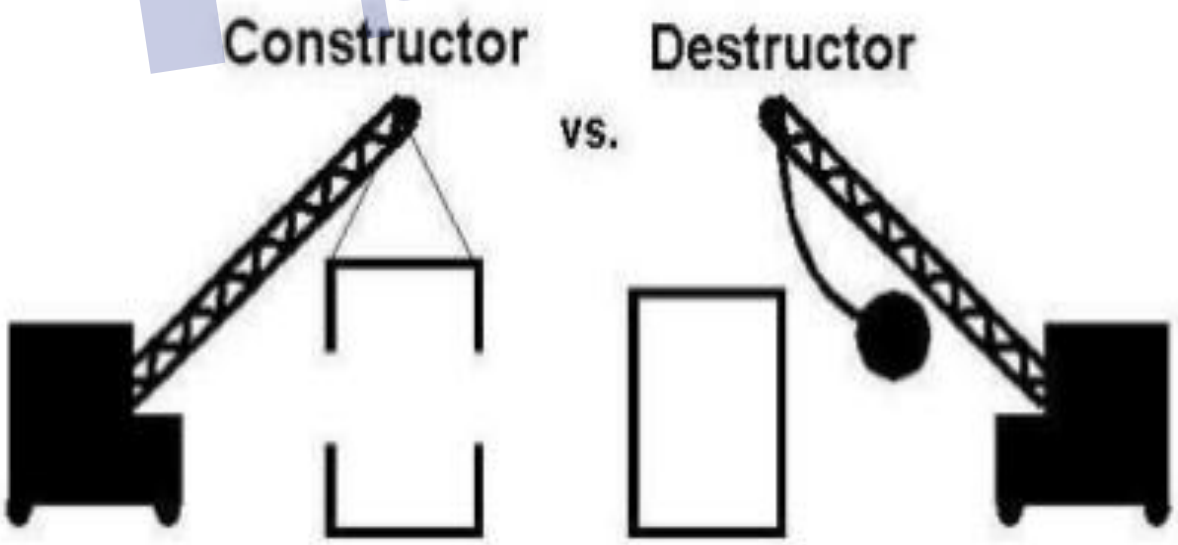
C++ allows more than one constructors to be defined in a single class.
Defining more than one constructors in a single class provided they all differ in either type or number of argument is called constructor overloading.

```
class student
{
 private:
      int rollno;
     char name[20];
     float marks;
public:
    student( )      // Default Constructor
     { rollno=0; marks=0.0; }
    student(int roll, char nam[20],
float mar) // Parametrized Constructor
     { rollno=roll;
     strcpy(name, nam);
     marks =mar;
     }
```

```
student(int roll, char nam[20])  //
Parametrized Constructor
     { rollno=roll;
     strcpy(name, nam);
     marks =100;
     }

student( student & s)  // Copy
Constructor
     { rollno= s.rollno;
     strcpy(name, s.name);
     marks =s.marks;
     }
}
```

- In the class student, there are 4 constructors:
  1 default constructor,
  2 parametrized constructors
  and 1 copy constructor.
- Defining more than one constructor in a single class is called constructor overloading.
- We can define as many parametrized constructors as we want in a single class provided they all differ in either number or type of arguments.

Destructor in C++



C++ Destructors

- Destructor is a special class function which destroys the object as soon as the scope of object ends. The destructor is called automatically by the compiler when the object goes out of scope.

- The syntax for destructor is same as that for the constructor, the class name is used for the name of destructor, with a tilde ~ sign as prefix to it.


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

Rules of Destructors

- destructors cannot be overloaded
- destructors take no arguments
- they don't return a value

- Constructor is invoked automatically when the object created.
- Destructor is invoked when the object goes out of scope. In other words, Destructor is invoked, when compiler comes out form the function where an object is created.

Destructor

- **When is destructor called?**
- A destructor function is called automatically when the object goes out of scope:
- (1) the function ends
- (2) the program ends
- (3) a block containing local variables ends
- (4) a delete operator is called

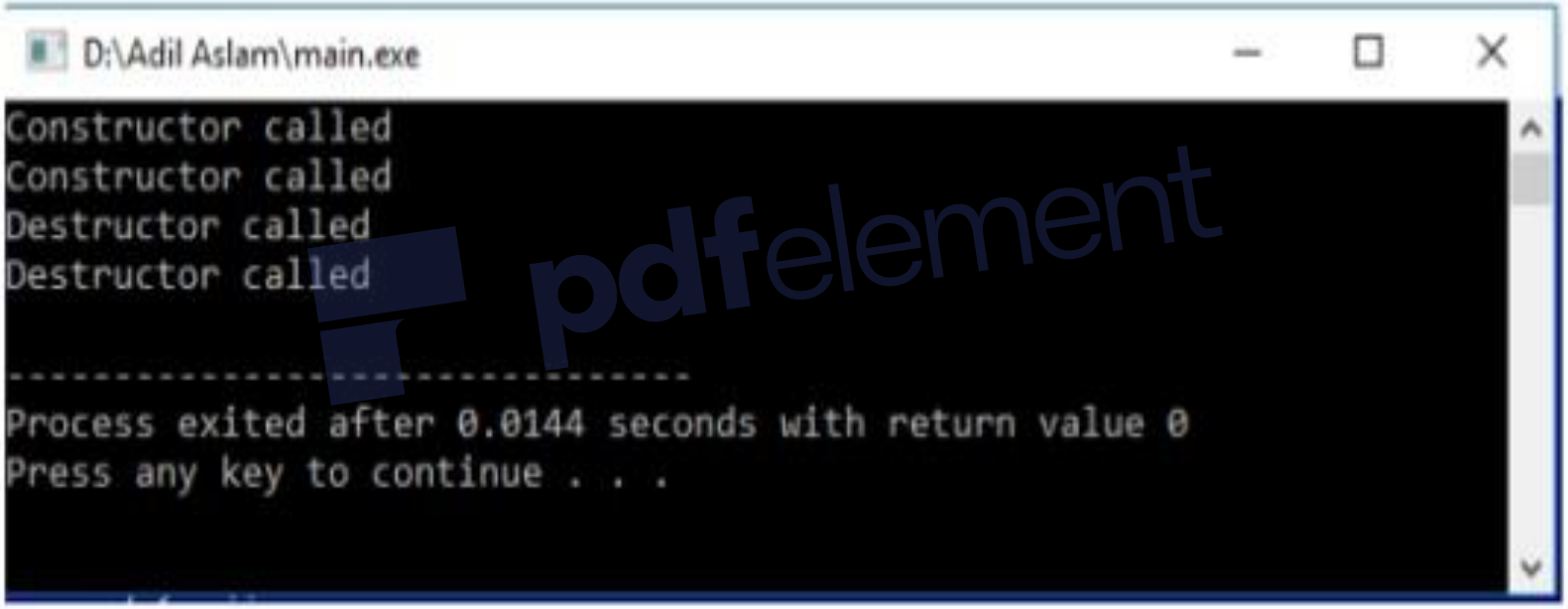
Destructor

- **Can there be more than one destructor in a class?**
  No, there can only one destructor in a class with class name preceded by ~, no parameters and no return type.
- **When do we need to write a user-defined destructor?**
  If we do not write our own destructor in class, compiler creates a default destructor for us. The default destructor works fine unless we have dynamically allocated memory or pointer in class. When a class contains a pointer to memory allocated in class, we should write a destructor to release memory before the class instance is destroyed. This must be done to avoid memory leak.

Example of Destructor

```
class A {
        public:
         A() {
                cout << "Constructor called"<<endl;
              }
        ~A() {
                cout << "Destructor called"<<endl;
              }
};
int main() {
        A obj1;  // Constructor Called
        int x=1;
        if(x) {
                A obj2;  // Constructor Called
  } // Destructor Called for obj2
} //  Destructor called for obj1
```

Output of the Previous Program is :



Cleanup of An Object


```
class Account
{
    private:
      char name[10];
      double balance;
      unsigned int id;  //unique
    public:
      Account();
      Account( char person[10]);
      ~Account();
}
```

Destructor

```
Account :: ~Account()
{
    cout<<"end of object"<<id;
}
```


- Its name is the class name preceded by a ~ (tilde)
- It has no argument
- It is used to release dynamically allocated memory and to perform other "cleanup" activities
- It is executed automatically when the object goes out of scope

Sequence of Calls

- Constructors and destructors are called automatically
- Constructors are called in the sequence in which object is declared
- Destructors are called in reverse order

EXAMPLE

```
1.  class Cube { public: int side;
2.  Cube() { side = 10; }
3.  };
4.  int main() { Cube c;
5.  cout << c.side; }
```

Default constructor

6. OUTPUT 10
7. ==========

```
8.  class Cube { public: int side;
9.  Cube(int x) { side=x; }
```

Parameterized

```
10. };
11. int main()
12. { Cube c1(10); Cube c2(20); Cube c3(30);
13. cout << c1.side; cout << c2.side; cout << c3.side; }
```

constructor

14. Output:
15. 10 20 30

OVERLODING CONSTRUCTORS

- class Student
- { public: int rollno; string name;
- // first constructor
- Student(int x) { rollno = x; name = "None"; }
- // second constructor
- Student(int x, string str) { rollno = x; name = str; }
- };
- void main() {
- Student c ; // syntax error
- // student A initialized with roll no 10 and name None Student A(10);
- // student B initialized with roll no 11 and name John Student B(11, "John"); }

```
class Student
{ public: int rollno; string name;
            Student(int x) { rollno = x; name = "None"; }
            Student(int x, string str) { rollno = x; name = str; }
            ~Student ( ) { cout<<“OBJECT DESTRUCTED WITH NAME=“
                                <<name<<“ROLLN
};
void main() {
Student A(10); Student B(11, “HASAN");
cout<<“OBJECT A AND B ARE INITIALI
if (5<10)  { Student  C( 12,”AHMED”);
              cout<<“OBJECT C IS INITIAL
  cout<<“PROGRAM CONTINUED”;
   }
```

OUTPUT

OBJECT A AND B ARE INITIALIZED

OBJECT C IS INITIALIZED

OBJECT DESTRUCTED WITH NAME=AHMED AND ROLLNO=12

PROGRAM CONTINUED

OBJECT DESTRUCTED WITH NAME =HASAN AND ROLLNO=11

OBJECT DESTRUCTED WITH NAME=None AND ROLLNO=10

LECTURE 7

Example : class Time Specification


```
class  time
{
 public :
      void          set ( int  hours , int  minutes , int  seconds ) ;
      void          increment ( ) ;
      void          write ( )  ;
 private :
     int          hrs ;
      int          min ;
      int          sec ;
} ;
```

Write three different constructors in class time.

Example : class Time Specification


```
class  time
{ private :
      int           hrs ;
       int           min ;
       int           sec ;
Public:
  time ( ) { hrs=0; min=0 ; sec=0;}
  time ( int x ) { hrs=x; min = x ; sec=x; }
  time ( int x , int y ,int z ) { hrs= x; min= y ; sec = z ; }
};
main( ) { time a ,  b (3) ,  c( 10,30,5);
            }
```


Constructor Overloading In C++

- Constructor can be overloaded in a similar way as function overloading.
- Overloaded constructors have the same name (name of the class) but different number of arguments.
- Depending upon the number and type of arguments passed, specific constructor is called.
- Since, there are multiple constructors present, argument to the constructor should also be passed while creating an object.

Constructor Overloading

- Constructors can have parameters
- These parameters are used to initialize the data members with user supplied data

Object Initialization

```
class Rectangle
{
    private:
      int width;
      int length;
    public:
      Rectangle(int w, int l)
          {width =w; length=l;}
      void set(int w, int l);
      int area();
}
```

If any constructor with any number of parameters is declared, no default constructor will exist, unless you define it.

```
Rectangle r4;      // error
```

3. Initialize with constructor

```
Rectangle r5(60,80);
r5.set(10,20);
```

Object Initialization

Write your own constructors

```
class Rectangle
{
    private:
       int width;
       int length;
    public:
       Rectangle();
       Rectangle(int w, int l);
       void set(int w, int l);
       int area();
}
```

```
Rectangle :: Rectangle()
{
     width = 20;
     length = 50;
};
```

```
Rectangle  r7;
```

r7

width = 20
length = 50

Object Initialization


```
class Account
{
     private:
       char name[10];
       double balance;
        int id;
     public:
       Account();

       Account( char person[10]);
}
```

With constructors, we have more control over the data members


```
Account :: Account()
{
     name = NULL; balance = 0.0;
     id = 0;
};
```

```
Account :: Account( char person[10])
{

     strcpy (name, person);
     balance = 0.0;
     id = 0;
};
```

So far, …

- An object can be initialized by a class constructor
  - default constructor
  - copy constructor
  - constructor with parameters
- Resources are allocated when an object is initialized
- Resources should be revoked when an object is about to end its lifetime

CONSTRUCTOR OVERLOADING

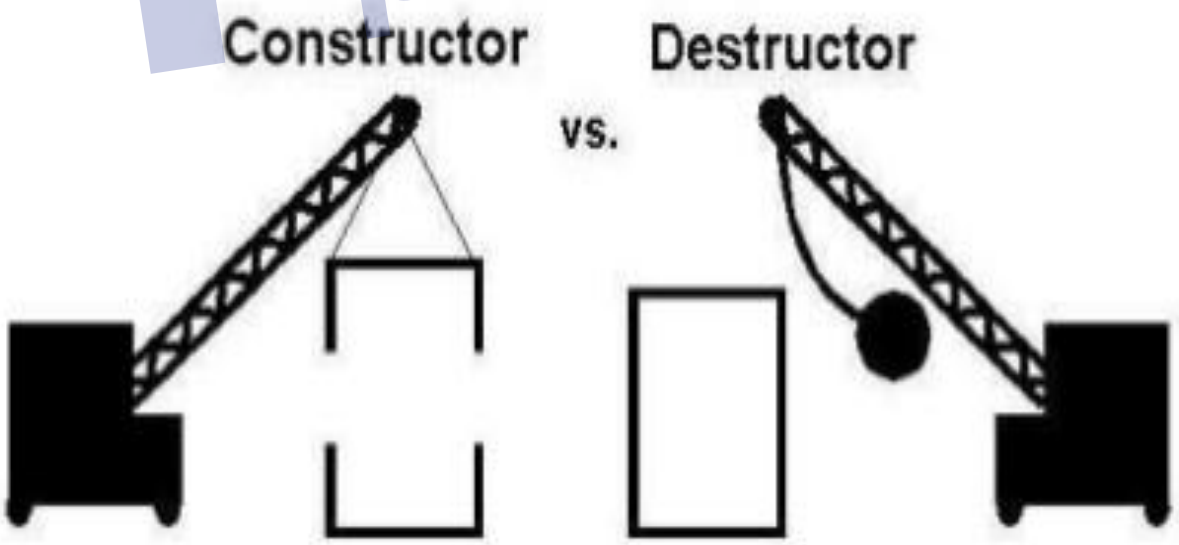
C++ allows more than one constructors to be defined in a single class.
Defining more than one constructors in a single class provided they all differ in either type or number of argument is called constructor overloading.

```
class student
{
 private:
      int rollno;
     char name[20];
     float marks;
public:
    student( )      // Default Constructor
     { rollno=0; marks=0.0; }
    student(int roll, char nam[20],
float mar) // Parametrized Constructor
     { rollno=roll;
     strcpy(name, nam);
     marks =mar;
     }
```

```
student(int roll, char nam[20])  //
Parametrized Constructor
     { rollno=roll;
     strcpy(name, nam);
     marks =100;
     }

student( student & s)  // Copy
Constructor
     { rollno= s.rollno;
     strcpy(name, s.name);
     marks =s.marks;
     }
}
```

- In the class student, there are 4 constructors:
  1 default constructor,
  2 parametrized constructors
  and 1 copy constructor.
- Defining more than one constructor in a single class is called constructor overloading.
- We can define as many parametrized constructors as we want in a single class provided they all differ in either number or type of arguments.

Destructor in C++



C++ Destructors

- Destructor is a special class function which destroys the object as soon as the scope of object ends. The destructor is called automatically by the compiler when the object goes out of scope.

- The syntax for destructor is same as that for the constructor, the class name is used for the name of destructor, with a tilde ~ sign as prefix to it.


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

Rules of Destructors

- destructors cannot be overloaded
- destructors take no arguments
- they don't return a value

- Constructor is invoked automatically when the object created.
- Destructor is invoked when the object goes out of scope. In other words, Destructor is invoked, when compiler comes out form the function where an object is created.

```
Main()
{int x ;
x=3;
cout <<x;
if (x==3){ int y;
y=4;
cout<<y;}
cout <<x<<y;}
```

Destructor

- **When is destructor called?**
- A destructor function is called automatically when the object goes out of scope:
- (1) the function ends
- (2) the program ends
- (3) a block containing local variables ends
- (4) a delete operator is called

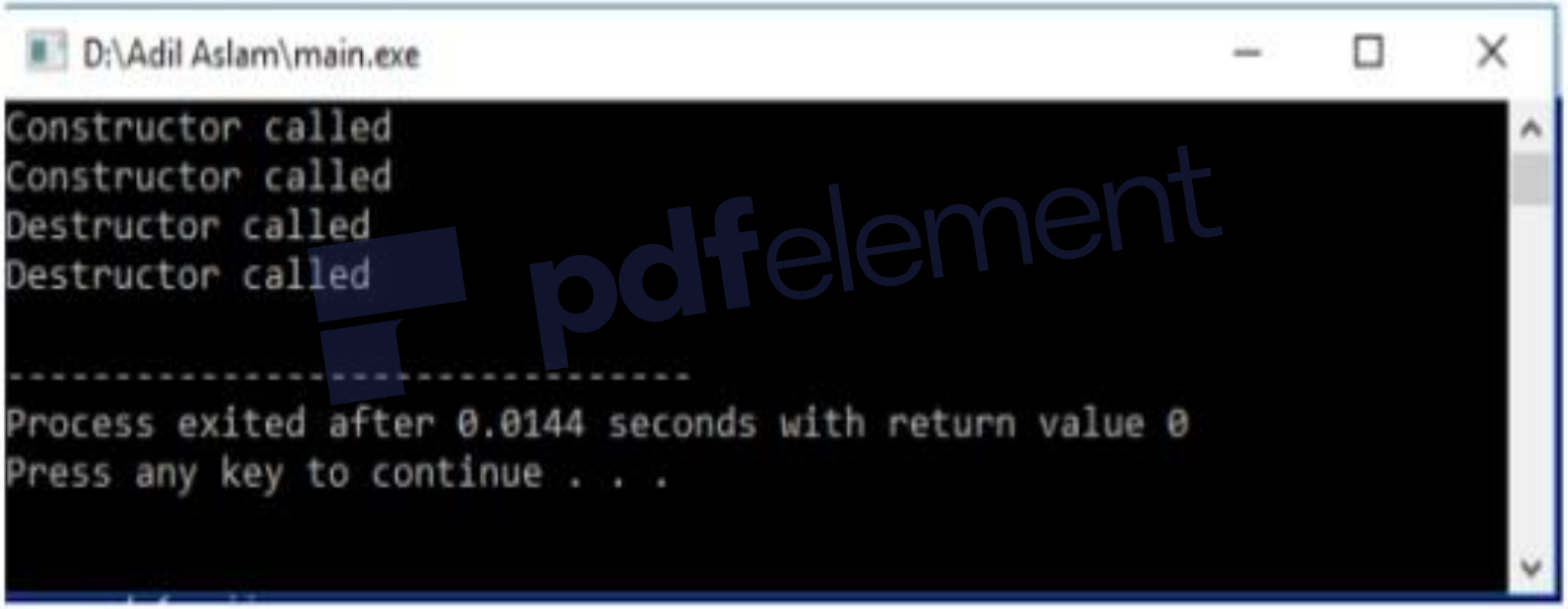
Destructor

- **Can there be more than one destructor in a class?**
  No, there can only one destructor in a class with class name preceded by ~, no parameters and no return type.
- **When do we need to write a user-defined destructor?**
  If we do not write our own destructor in class, compiler creates a default destructor for us. The default destructor works fine unless we have dynamically allocated memory or pointer in class. When a class contains a pointer to memory allocated in class, we should write a destructor to release memory before the class instance is destroyed. This must be done to avoid memory leak.

Example of Destructor

```
class A {
        public:
         A() {
                cout << "Constructor called"<<endl;
              }
        ~A() {
                cout << "Destructor called"<<endl;
              }
};
int main() {
        A obj1;  // Constructor Called
        int x=1;
        if(x) {
                A obj2;  // Constructor Called
  } // Destructor Called for obj2
} //  Destructor called for obj1
```

Output of the Previous Program is :



Cleanup of An Object


```
class Account
{
    private:
      char name[10];
      double balance;
      unsigned int id;  //unique
    public:
      Account();
      Account( char person[10]);
      ~Account();
}
```

Destructor

```
Account :: ~Account()
{
    cout<<"end of object"<<id;
}
```


- Its name is the class name preceded by a ~ (tilde)
- It has no argument
- It is used to release dynamically allocated memory and to perform other "cleanup" activities
- It is executed automatically when the object goes out of scope

Sequence of Calls

- Constructors and destructors are called automatically
- Constructors are called in the sequence in which object is declared
- Destructors are called in reverse order

EXAMPLE


```
class Cube { public: int side;
 Cube() { side = 10; }
};
 int main() { Cube c;
 cout << c.side; }
```

Default constructor

OUTPUT 10

==========

```
class Cube { public: int side;
 Cube(int x) { side=x; }
};
 int main()
 { Cube c1(10); Cube c2(20); Cube c3(30);
 cout << c1.side; cout << c2.side; cout << c3.side; }
```

Parameterized constructor

Output:

10 20 30

OVERLODING CONSTRUCTORS

```
class Student
{ public: int rollno; string name;
  // first constructor
 Student(int x) { rollno = x; name = "None"; }
 // second constructor
Student(int x, string str) { rollno = x; name = str; }
};
void main() {
Student c ; // syntax error
// student A initialized with roll no 10 and name None
 Student A(10);
// student B initialized with roll no 11 and name John
Student B(11, "John"); }
```

```
class Student
{ public: int rollno; string name;
            Student(int x) { rollno = x; name = "None"; }
            Student(int x, string str) { rollno = x; name = str; }
            ~Student ( ) { cout<<“OBJECT DESTRUCTED WITH NAME=“
                                <<name<<“ROLLN
};
void main() {
Student A(10); Student B(11, “HASAN");
cout<<“OBJECT A AND B ARE INITIALI
if (5<10)  { Student  C( 12,”AHMED”);
              cout<<“OBJECT C IS INITIAL
  cout<<“PROGRAM CONTINUED”;
   }
```

OUTPUT

OBJECT A AND B ARE INITIALIZED

OBJECT C IS INITIALIZED

OBJECT DESTRUCTED WITH NAME=AHMED AND ROLLNO=12

PROGRAM CONTINUED

OBJECT DESTRUCTED WITH NAME =HASAN AND ROLLNO=11

OBJECT DESTRUCTED WITH NAME=None AND ROLLNO=10

Example : class Time Specification


```
class  time
{ private :
      int          hrs ;
       int          min ;
       int          sec ;
Public:
  time ( ) { hrs=0; min=0 ; sec=0;}
  time ( int x ) { hrs=x; min = x ; sec=x; }
  time ( int x , int y ,int z ) { hrs= x; min= y ; sec = z ; }
~time ( ) {
cout <<"Destruct object:"<<hrs<<"hours"<<min<<"minute"<<sec<<"seconds\n";
}
};
main( ) { time a ,  b (3) ,  c( 10,30,5);
          }
```

Output

Destruct object 10 hours 30 minute 5 seconds
Destruct object 3 hours 3 minute 3 seconds
Destruct object 0 hours 0 minute 0 seconds

LECTURE 8

Sequence of Calls

- Constructors and destructors are called automatically
- Constructors are called in the sequence in which object is declared
- Destructors are called in reverse order

EXAMPLE


```
class Cube { public: int side;
 Cube() { side = 10; }
};
 int main() { Cube c;
 cout << c.side; }
```

Default constructor

OUTPUT 10

==========

```
class Cube { public: int side;
 Cube(int x) { side=x; }
};
 int main()
 { Cube c1(10); Cube c2(20); Cube c3(30);
 cout << c1.side; cout << c2.side; cout << c3.side; }
```

Parameterized constructor

Output:

10 20 30

OVERLODING CONSTRUCTORS

```
class Student
{ public: int rollno; string name;
  // first constructor
 Student(int x) { rollno = x; name = "None"; }
 // second constructor
Student(int x, string str) { rollno = x; name = str; }
};
void main() {
Student c ; // syntax error
// student A initialized with roll no 10 and name None
 Student A(10);
// student B initialized with roll no 11 and name John
Student B(11, "John"); }
```

```
class Student
{ public: int rollno; string name;
            Student(int x) { rollno = x; name = "None"; }
            Student(int x, string str) { rollno = x; name = str; }
            ~Student ( ) { cout<<"OBJECT DESTRUCTED WITH NAME="
                                <<name<<"ROLLN
};
void main() {
Student A(10); Student B(11, "HASAN");
cout<<"OBJECT A AND B ARE INITIALI
if (5<10)  { Student  C( 12,"AHMED");
              cout<<"OBJECT C IS INITIAL
  cout<<"PROGRAM CONTINUED";
   }
```

OUTPUT

OBJECT A AND B ARE INITIALIZED

OBJECT C IS INITIALIZED

OBJECT DESTRUCTED WITH NAME=AHMED AND ROLLNO=12

PROGRAM CONTINUED

OBJECT DESTRUCTED WITH NAME =HASAN AND ROLLNO=11

OBJECT DESTRUCTED WITH NAME=None AND ROLLNO=10

Example : class Time Specification


```
class  time
{ private :
      int          hrs ;
       int          min ;
       int          sec ;
Public:
  time ( ) { hrs=0; min=0 ; sec=0;}
  time ( int x ) { hrs=x; min = x ; sec=x; }
  time ( int x , int y ,int z ) { hrs= x; min= y ; sec = z ; }
~time ( ) {
cout <<"Destruct object:"<<hrs<<"hours"<<min<<"minute"<<sec<<"seconds\n";
}
};
main( ) { time a ,  b (3) ,  c( 10,30,5);
          }
```

Output

Destruct object 10 hours 30 minute 5 seconds
Destruct object 3 hours 3 minute 3 seconds
Destruct object 0 hours 0 minute 0 seconds

Example

```
#include <iostream>
 #include <string>
using namespace std;
 class Student { string name;
                 int marks;
public: void getName() { cin>> name; }
void getMarks() { cin >> marks; }
void displayInfo() { cout << "Name : " << name << endl;
                     cout << "Marks : " << marks << endl; }
};
 int main() { Student st[5];
 for( int i=0; i<5; i++ ) { cout << "Student " << i + 1 << endl;
                            cout << "Enter name" << endl;
                            st[i].getName();
                            cout << "Enter marks" << endl;
                            st[i].getMarks(); }
          for( int i=0; i<5; i++ )
            { cout << "Student " << i + 1 << endl;
              st[i].displayInfo(); }
 return 0; }
```

Passing and Returning Objects in C++

and Returning Objects in C++

In C++ we can pass class's objects as arguments and also return them from a function the same way we pass and return other variables. No special keyword or header file is required to do so.

Passing an Object as argument

- To pass an object as an argument we write the object name as the argument while calling the function the same way we do it for other variables.
  Syntax:
- function_name(object_name);

- // C++ program to calculate the average marks of two students
- #include <iostream>using namespace std;

```
class Student {    public:
                   double marks;
                   Student(double m) {  marks = m; }};
 void calculateAverage(Student s1, Student s2)
{          double average = (s1.marks + s2.marks) / 2;
           cout << "Average Marks = " << average <<
endl; }
  main() {    Student  student1(88.0),  student2(56.0);
    calculateAverage(student1, student2); }
```

- **Output**
- Average Marks = 72

Lecture 9

Array of Objects in C++

- Like array of other user-defined data types, an array of type class can also be created.
- The array of type class contains the objects of the class as its individual elements.Thus, an array of a class type is also known as an array of objects.
- An array of objects is declared in the same way as an array of any built-in data type.
- The syntax for declaring an array of objects is

```
class_name    array_name [size] ;
```


Example

```
#include <iostream>
 #include <string>
using namespace std;
 class Student { string name;
                 int marks;
public: void getName() { cin>> name; }
void getMarks() { cin >> marks; }
void displayInfo() { cout << "Name : " << name << endl;
                     cout << "Marks : " << marks << endl; }
};
 int main() { Student st[5];
 for( int i=0; i<5; i++ ) { cout << "Student " << i + 1 << endl;
                            cout << "Enter name" << endl;
                            st[i].getName();
                            cout << "Enter marks" << endl;
                            st[i].getMarks(); }
          for( int i=0; i<5; i++ )
           { cout << "Student " << i + 1 << endl;
             st[i].displayInfo(); }
 return 0; }
```

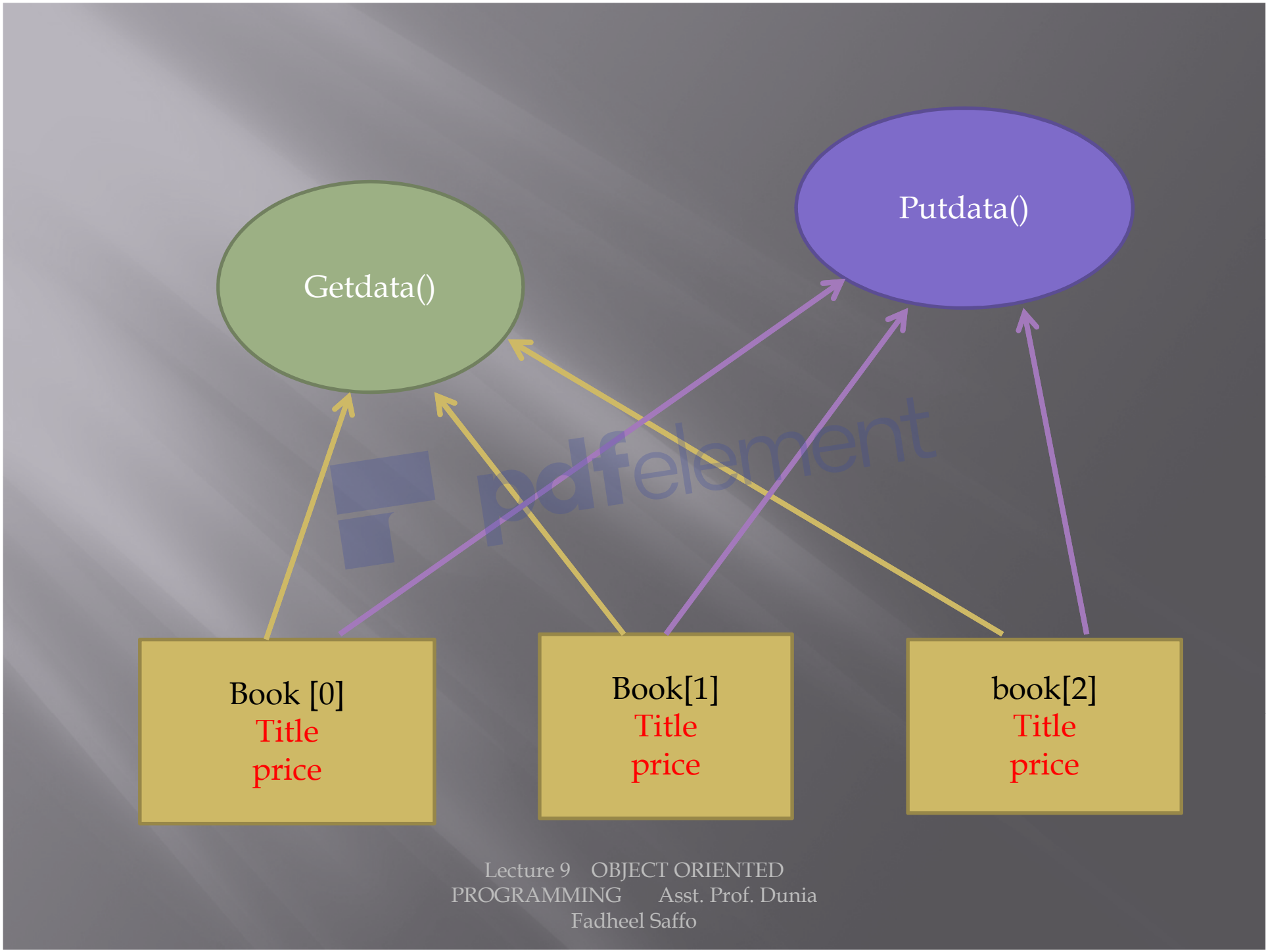
Array of Objects

- suppose we have 50 students in a class and we have to input the name and marks of all the 50 students. Then creating 50 different objects and then inputting the name and marks of all those 50 students is not a good option. In that case, we will create an **array of objects** as we do in case of other data-types.
- Let's see an example of taking the input of name and marks of 5 students by creating an array of the objects of students.

- Example:
- class book { string title;
- int price;
- public: void getdata( );
- void putdata()
- };

When an array of objects is declared:

- The memory is allocated in the same way as to multidimensional arrays.
- For example, for the array book, a separate copy of title and price is created for each member book[0], book[l] and book[2].
- For instance, the memory space is allocated to the array of objects book of the class books


Getdata()
Putdata()
Book [0]
Title
price
Book[1]
Title
price
book[2]
Title
price

The rest of the program is as follows:

```
void books :: getdata ()
{
cout<<"Title:";cin>>title;
cout<<"Price:";cin>>price;
 }
void books :: putdata ()
{
cout<<"Title:"<<title<< "\n";
cout<<"Price:"<<price<< "\n";
}
```

```
int main ()
{const int size=3 ;
books book[size] ;
for(int i=0;i<size;i++)
{
cout<<"Enter details of
book "<<(i+1)<<"\n";
book[i].getdata();
}
for(int i=0;i<size;i++)
{
cout<<"\nBook "<<(i+l)<<"\n";
book[i].putdata() ;
}
return 0;
}
```


The output of the program is

```
Enter details of book
1
Title: c++
Price: 325
Enter details of book
2
Title: DBMS
Price: 455
Enter details of book
3
Title: Java
Price: 255
```

→

```
Book 1
Title: c++
Price: 325
Book 2
Title: DBMS
Price: 455
Book 3
Title: Java
Price: 255
```

Passing an Object as argument

- To pass an object as an argument we write the object name as the argument while calling the function the same way we do it for other variables.
  Syntax:
- function_name(object_name);

PASSING OBJECTS TO NON-MEMBER FUNCTION

- // C++ program to calculate the average marks of two students
- #include <iostream>using namespace std;

```
class Student {    public:
                        double marks;
                        Student(double m) {  marks = m; }};
 void calculateAverage(Student s1, Student s2)
{          double average = (s1.marks + s2.marks) / 2;
           cout << "Average Marks = " << average << endl; }
  main() {    Student  student1(88.0),  student2(56.0);
    calculateAverage(student1, student2); }
```

- **Output**
- Average Marks = 72

Here, we have passed two Student objects *s1* and *s2* as arguments to the calculateAverage() function.

```
#include<iostream>

class Student {...};

void calculateAverage(Student s1, Student s2) {
    // code
}

int main() {
    ... ...
    calculateAverage(student1, student2);
    ... ...
}
```

Passing object to a member function

- Example :

```
class Student {    public:
                        double marks;
                        Student(double m) {  marks =
m; }
 void calculateAverage(Student s1)
{          double average = (marks + s1.marks) / 2;
          cout << "Average Marks = " << average
<<endl; }
};
  main() {    Student student1(88.0),
student2(56.0);
```

Passing object to a member function of different class

- Example :

```
class Student {    public:   int mark1, mark2;
   Student(int m1, int m2) {  mark1 =
m1;mark2=m2 }};
class average{ public : av;
 void calculateAverage(Student s1)
{                  av = (s1.mark1 + s1.mark2) / 2; } };
 main() {    Student  student1(88,77); average a;
    a.calculateAverage(student1);
     cout <<"average of student is "<<a.av; }
```

PASSING OBJECTS TO NON-MEMBER FUNCTION

```
// C++ program to calculate the average marks of two students
 #include <iostream>using namespace std;
   class Student {    int  marks;
          public:   Student(int m) {  marks = m; }
                     int ret ( ){ return marks;}
};
 void calculateAverage(Student s1, Student s2) {
//   error double average = (s1.marks + s2.marks) / 2;
    double   average=(s1.ret( )+s2.ret( ))/2;
          cout << "Average Marks = " << average << endl; }
  main() {   Student  student1(88.0),  student2(56.0);
    calculateAverage(student1, student2); }
```

- **Output**
- Average Marks = 72

Lecture 10

Passing an Object as argument

- To pass an object as an argument we write the object name as the argument while calling the function the same way we do it for other variables.
  Syntax:
- function_name(object_name);

PASSING OBJECTS TO NON-MEMBER FUNCTION

- // C++ program to calculate the average marks of two students
- #include <iostream>using namespace std;

```
class Student {    public:
                        double marks;
                        Student(double m) {  marks = m; }};
 void calculateAverage(Student s1, Student s2)
{          double average = (s1.marks + s2.marks) / 2;
           cout << "Average Marks = " << average << endl; }
  main() {    Student  student1(88.0),  student2(56.0);
    calculateAverage(student1, student2); }
```

- **Output**
- Average Marks = 72

Here, we have passed two Student objects *s1* and *s2* as arguments to the calculateAverage() function.

```
#include<iostream>

class Student {...};

void calculateAverage(Student s1, Student s2) {
    // code
}

int main() {
    ... ...
    calculateAverage(student1, student2);
    ... ...
}
```

Passing object to a member function

- Example :

```
class Student {    public:
                        double marks;
                        Student(double m) {  marks = m; }
 void calculateAverage(Student s1)
{          double average = (marks + s1.marks) / 2;
          cout << "Average Marks = " << average
<<endl; }
};
  main() {    Student  student1(88.0),  student2(56.0);
     student1.calculateAverage(student2); }
```

Passing object to a member function of different class

- Example :

```
class Student {    public:   int mark1, mark2;
   Student(int m1, int m2) {  mark1 =
m1;mark2=m2 }};
class average{ public : av;
 void calculateAverage(Student s1)
{                    av = (s1.mark1 + s1.mark2) / 2; } };
 main() {    Student  student1(88,77); average a;
    a.calculateAverage(student1);
     cout <<“average of student is “<<a.av; }
```

PASSING OBJECTS TO NON-MEMBER FUNCTION

```
// C++ program to calculate the average marks of two students
 #include <iostream>using namespace std;
   class Student {    int  marks;
          public:   Student(int m) {  marks = m; }
                     int ret ( ){ return marks;}
};
 void calculateAverage(Student s1, Student s2) {
//   error double average = (s1.marks + s2.marks) / 2;
    double   average=(s1.ret( )+s2.ret( ))/2;
          cout << "Average Marks = " << average << endl; }
  main() {   Student  student1(88.0),  student2(56.0);
    calculateAverage(student1, student2); }
```

- **Output**
- Average Marks = 72

Passing object to a member function

- Example :

```
class Student {  double marks;
        public:  Student(double m) {  marks = m; }
 void calculateAverage(Student s1)
{          double average = (marks + s1.marks) / 2;
         cout << "Average Marks = " << average
<<endl; }
};
  main() {    Student  student1(88.0),  student2(56.0);
     student1.calculateAverage(student2); }
```

Passing object to a member function of different class

- Example :

```
class Student {     int mark1, mark2;
    Student(int m1, int m2) {  mark1 = m1;mark2=m2 }
     int ret_m1 ( ) { return m1;}
     int ret_m2( ) { return m2; }
};
class average{ public :  float av;
 void calculateAverage(Student s1)
{  av = (s1.ret_m1( ) + s1.ret_m2) / 2; }     };
 main() {    Student  student1(88,77);    average a;
     a.calculateAverage(student1);
      cout <<“average of student is “<<a.av; }
```

- // C++ program to calculate the average marks of two students
- #include <iostream>using namespace std;

```
class Student {    int  marks;
        public:   Student(int m) {  marks = m; }
                    int ret ( ){ return marks;}
            // define calculate average as a member function
                    void new-calculate ( student s1)
                    { cout<<(marks+s1.marks)/2 ;}
};
 void calculateAverage(Student s1, Student s2) {
double   average=(s1.ret( )+s2.ret( ))/2;
          cout << "Average Marks = " << average << endl; }
  main() {    Student  student1(88.0),  student2(56.0);
     calculateAverage(student1, student2);
      student1.new-calculate(student2)}
```

- **Output**
- Average Marks = 72

Passing object to a member function

- Example :

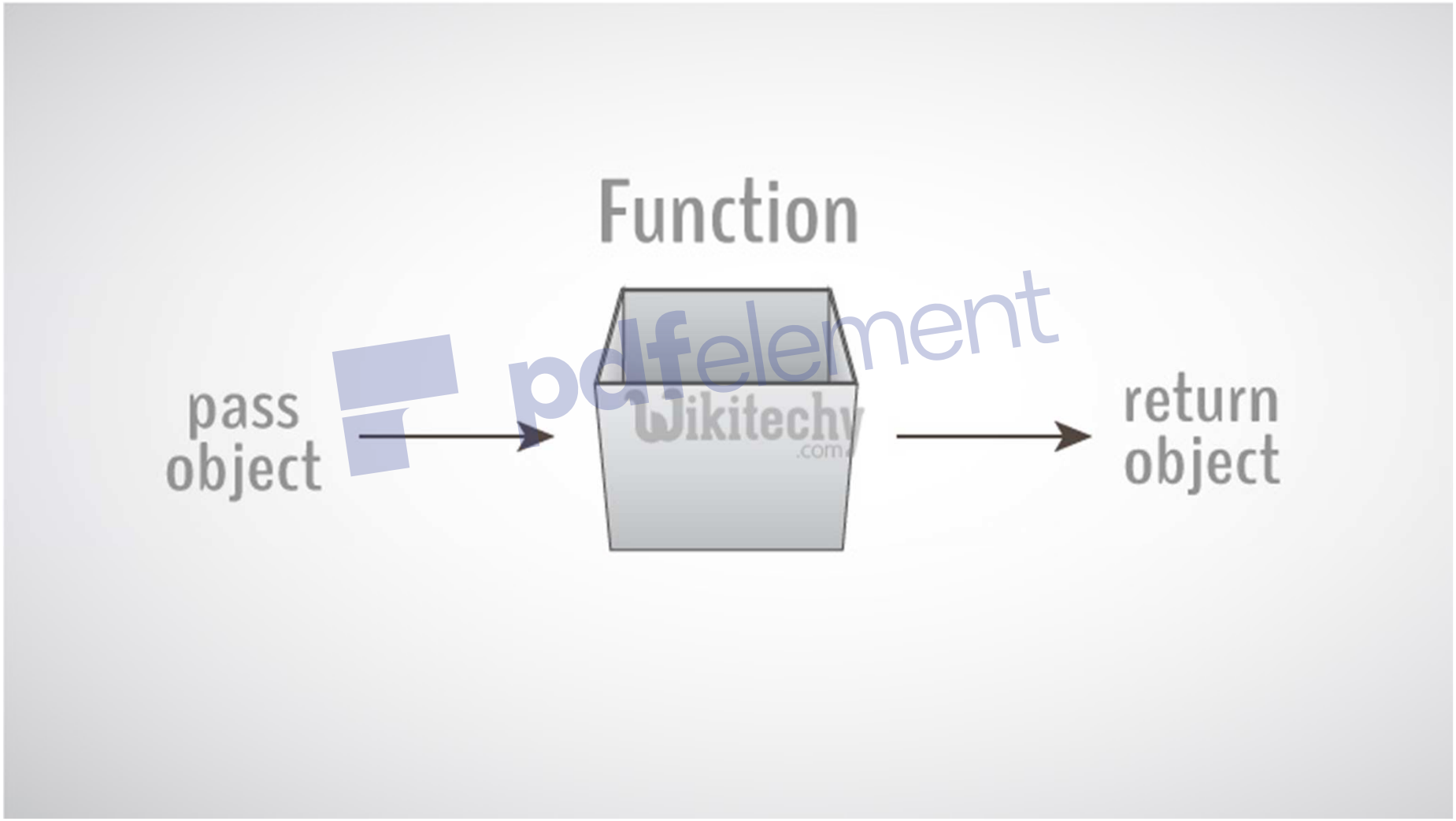
```
class Student {  double marks;
        public:  Student(double m) {  marks = m; }
                 int ret_m(){ return marks;}
 void calculateAverage(Student s1)        {        double
average = (marks + s1.marks) / 2;
        cout << "Average Marks = " << average <<endl; }
};
//define  calculateaverage as a non member function
void newcalculate( Student s1 , Student s2){
cout<< (s1.ret_m( )+s2.ret_m( ))/2;}
  main() {    Student  student1(88.0),  student2(56.0);
    student1.calculateAverage(student2);
    newcalculate(student1,student2); }
```

Passing object to a member function of different class

- Example :

```
class Student {     int mark1, mark2;
   Student(int m1, int m2) {  mark1 = m1;mark2=m2 }
    int ret_m1 ( ) { return m1;}
    int ret_m2( ) { return m2; }
//rewrite function calculateAverage as a member in class student
void  new_calculate ( average   a ){ float f = (m1+m2)/2;
                                    // a.av=f ;error av is private to class average
                                  a.set( f);}
  };
class average{ public :  float av;
 void calculateAverage(Student s1)
{  av = (s1.ret_m1( ) + s1.ret_m2) / 2; }
void  set ( float x ) {  av = x ; }
  };
 main() {    Student  student1(88,77);    average a;
    a.calculateAverage(student1);
     cout <<"average of student is "<<a.av;
     student1.new_calculate( a );
      }
```

Returning object from function



In C++ programming, object can be returned from a function

```
class className {
    ... .. ...

    public:
    className functionName(className agr1)
    {
        className obj;
        ... .. ...
        return obj;
    }

    ... .. ..

};

int main() {

    className o1, o2, o3;
    o3 = o1.functionName (o2);
}
```

Notes

- An object is an instance of a class. Memory is only allocated when an object is created and not when a class is defined.
- An object can be returned by a function using the return keyword.

Returning object from a Function

```
#include <iostream>
 using namespace std;
class Point {    private:    int x;    int y;
 public:    Point(int x1 = 0, int y1 = 0)
                    { x = x1; y   y1;   }
            Point addPoint(Point p) {   Point  temp;
            temp.x = x + p.x;       temp.y = y + p.y;
            return temp;    }
  void display()
{ cout<<"x = "<< x <<"\n";      cout<<"y = "<< y <<"\n";    } };
int main() {    Point p1(5,3);  Point p2(12,6);    Point p3;
 cout<<"Point 1\n";    p1.display();
 cout<<"Point 2\n";    p2.display();
 p3 = p1.addPoint(p2);
cout<<"The sum of the two points is:\n";    p3.display();
  return 0; }
```

Example

```
// C++ program to show passing
// of objects to a function

#include <bits/stdc++.h>
using namespace std;

class Example {
public:
    int a;
 // This function will take object as
arguments and return object
  Example add(Example Ea, Example Eb)
    {  Example Ec;
        Ec.a = 50;
        Ec.a = Ec.a + Ea.a + Eb.a;
  // returning the object
        return Ec;
    }
};
```

```
int main()
{ Example E1, E2, E3;
// Values are initialized for both
objects
E1.a = 50; E2.a = 100; E3.a = 0;
cout << "Initial Values \n";
cout <<"Value of object 1:" << E1.a
<< ", \n object 2: " << E2.a<<
"\nobject 3: " << E3.a<< "\n";
 // Passing object as an argument
    to function add()
  E3 = E3.add(E1, E2);
// Changed values after passing
object as an argument
cout << "New values \n";
cout <<"Value of object 1:" << E1.a
<< ", \nobject 2: " << E2.a<<
", \nobject 3: " << E3.a
return 0;}
```

OUTPUT

Initial values

Value of object 1 : 50

object 2 : 100

object 3 : 0

New values

Value of object 1 : 50

Object 2 : 100

Object 3 : 200

LAB. WORK 23-2-2021

Consider the following classes:

Class square { int side ; };

1. Define a default constructor that reads the data member value.
2. Your main function should declare an array of 3 squares objects, and print side of object with largest area.
3. Write a non-member function that returns average of two squares sides . (Note : the function should receives the two squares objects as arguments).
4. Change the function in step 3 to be a member function.