Welcome

Object Oriented Programming (UTA018)

(Session: 2026-2027 ODD)

Instructor: Dr. Suresh Raikwar

Department of Computer Science and Engineering,

Thapar Institute of Engineering and Technology, Patiala, Punjab, India

Important Instruction(s):

- This material is intended for guidance and conceptual understanding. It does not guarantee that examination questions will be drawn directly from it.
- Merely studying this material is not sufficient for achieving good performance.
- Sessional Tests, MST, and EST will assess your understanding of Object-Oriented Programming concepts through **logical reasoning, analytical thinking, and problem-solving**.
- Students are strongly encouraged to attend regular classes and practice programming problems from multiple sources to develop a deeper conceptual understanding and improve problem-solving skills.

Week 4

Constructor

- A constructor is a special member function of a class whose task is to initialize the objects of its class.
- It has the same name as of that class.
- It is invoked on the creation of the object of the associated class.
- It can be defined either inside the class or outside the class.

Characteristics of Constructor

- It must be declared in the public section of the class.
- It does not have any return type (not even void) and therefore cannot return any value.
- It is automatically called when the object is created.
- It can also be called explicitly.
- It can take parameters.
- Constructors can have default arguments.
- Constructor can be overloaded.

Characteristics of Constructor (Cont..)

- It cannot be inherited (although a derived class constructor can call a base class constructor).
- Constructors cannot be virtual.
- We cannot refer to their addresses.
- Constructors make implicit call to the operators *new* and *delete* when memory allocation is required.

Note: When a constructor is declared in a class, initialization of class objects become mandatory.

Declaration of Constructor

- Declaration of constructor

```
//class with a constructor
class classname
{
  private:
    // variable and function declarations;
  public:
   // variable and function declarations;
      classname();      //constructor (having same name as the class)
};
```

Example of Constructor

```
#include<iostream>
using namespace std;
class example
{
  private:
    int a;
  public:
    example();     //constructor declared
    void display( );
};
   example:: example()  //constructor defined
      {
       a=5;
      }
   void example::display()
```

```
int main()
{
    example e1;                  //implicit call
    example e2=example();  //explicit call
    e1.display();
    e2.display();
    return 0;
}
```

When a class contains a constructor, it is guaranteed that an object of that class (when created) will be initialized automatically.

Not only creates the object **e1** of type **example** but also initializes its data member **a** to 5.

Types of Constructor

- Default Constructor
- Parameterized Constructor
- Copy Constructor (will be covered later)

Default Constructor

- A constructor that accepts no parameters is called default constructor.
- The default constructor for class **example** is **example::example( )**

Parameterized Constructors

- Sometimes, it may be necessary to initialize the various data elements of different objects with different values when they are created.
- This is achieved by passing arguments to the constructor function when the objects are created.
- The constructors that can take arguments are called parameterized constructors.

Example of Parameterized Constructor

```
class example{
private:
    int a;
public:
    example(int);     //Parameterized Constructor
    void display( );
};
example::example(int x){
    a = x;
}
void example::display(){
    cout<<a<<"\n";
}
```

When a constructor is parameterized, we must pass the initial values as arguments to the constructor function when an object is declared.

Two ways Calling:

1. **Explicit**

   example e1 = example(5);

2. **Implicit**

   example e1(5);

   //Shorthand method

Example 2

```
#include <iostream>
using namespace std;
class myclass {
    int a, b;
public:
    myclass(int i, int j){
    a=i;
    b=j;
}
void show() {
    cout << a << " " << b;
    }
};
```

```
int main(){
    myclass ob(3, 5);
    //You can also pass
arguments as
    // myclass ob=myclass
ob(3, 5);
    ob.show();
    return 0;
}
```

Constructors with Default Arguments

It is possible to define constructors with default arguments.

- Consider example (int a, int b= 0);
- The default value of the argument b is zero.

example e1(5);

assigns the value 5 to the variable a and 0 to b.

example e1(5, 3);

assigns the value 5 to the variable a and 3 to b.

Constructors with Default Arguments (Cont..)

- **example::example()** //Default Constructor
- **example::example(int a=0);** //Default Argument Constructor

- The default argument constructor can be called with either one argument or no arguments.

- When called with no arguments, it becomes a default constructor.

Constructors with One Parameter: A Special Case

```
#include <iostream>
using namespace std;
class X {
    int a;
public:
    X(int j) {
        a = j;
    }
    int geta() {
        return a;
    }
};
int main()
{
    X ob = 99; // passes 99 to j
    cout << ob.geta(); // outputs 99
    return 0;
}
```

Dynamic Constructor

- The constructor can also be used to allocate memory while creating objects.
- This enables the system to allocate the right amount of memory for each object when the objects are not of the same size.
- Allocation of memory to objects at time of their construction is known as dynamic construction of objects.
- The memory is allocated with the help of *new* operator.

Example 1

```
#include<iostream>
using namespace std;
class test{
    int *ptr;
public:
    test();
    test(int);
    void display();
};
test::test(){
    ptr=new int;
    *ptr=100;
}
test::test(int t){
    ptr=new int;
    *ptr=t;
```

```
void test::display(){
   cout<<"The value of object's pointer
is:"<<*ptr<<endl;
}
int main(){
    test obj;
    test obj1(40);
    obj.display();
    obj1.display();
    return 0;
}
```

OUTPUT:
The value of object's pointer is:100
The value of object's pointer is:40

Example 2

```
class example{
    char  *name;
    int  length;
public:
    example();
    example(char *);
    void display();
};
example::example(){
    length = 0;
    name = new char[length+1];
}
example::example(char *e){
    length = strlen(e);
    name = new char[ length+1];
    strcpy( name,e);
}
```

```
void  example::display(){
 cout<<name<<endl;
}
int main(){
   char  *a= "Welcome to";
   example  e1(a), e2("C++"), e3("World");
   e1.display();
   e2.display();
   e3.display();
   return 0;
}
```

OUTPUT:
Welcome to
C++
World

Destructors

- A destructor is a special member function of class which is used to destroy the objects that have been created by a constructor.
- The destructor is called automatically by the compiler when the object goes out of scope.
- Like a constructor, the destructor is a member function whose name is the same as class name but is preceded by a tilde ~ sign.

```
class A
{
  public:
    ~A();    //Destructor declaration
 };
```

Destructors (Cont..)

- Destructor never takes any argument.
- Destructors does not return any value.
- Destructor cannot be overloaded.
- Whenever *new* is used to allocate memory in the constructors, *delete* should be used to free that memory.

Note: Objects are destroyed in reverse order of their creation.

Example

```
#include<iostream>
using namespace std;
class ABC{
    int a;
public:
    ABC(int);  //Constructor
    void display();
    ~ABC();   //Destructor
  };
ABC::ABC(int x){
    a=x;
}
void ABC::display(){
    cout<<"a="<<a;
}
```

```
ABC::~ABC(){ //Definition of Destructor
    cout<<"\nObject is destroyed";
}
int main(){
    ABC obj1(10);
    obj1.display();
    return 0;
}
```

OUTPUT:
a=10
Object is destroyed

Destructor is automatically called by the compiler once the object goes out of scope.

When Constructors and Destructors Are Executed

- Global objects have their constructors execute before main( ) begins execution.
- Global constructors are executed in order of their declaration, within the same file.
- You cannot know the order of execution of global constructors spread among several files.
- Global destructors execute in reverse order after main( ) has terminated.

Example

```
#include<iostream>
using namespace std;
class myclass {
public:
    int who;
    myclass(int id);
    ~myclass();
} glob_ob1(1), glob_ob2(2);
myclass::myclass(int id){
    cout << "Initializing " << id << "\n";
    who = id;
}
myclass::~myclass(){
    cout << "Destructing " << who << "\n";
}
```

```
int main(){
    myclass local_ob1(3);
    cout << "This will not be first line
displayed.\n";
    myclass local_ob2(4);
    return 0;
}
```

Output

Initializing 1
Initializing 2
Initializing 3
This will not be first line displayed. Initializing 4
Destructing 4
Destructing 3
Destructing 2
Destructing 1

From Problem Statement to C++

A step-by-step OOP design using two related classes

Employee and Department

Problem statement

An institute needs a small system to maintain employees and their departments.

- Each Employee has an employee ID, name and salary.
- Each Department has a department ID and department name.
- An employee works in one department. A department can have many employees.

Task: Design Object Oriented Solution

\ Identify the classes

Look for important nouns in the problem statement.

Employee

A person who works in the institute.

Department

An organizational unit in the institute.

Nouns are the candidate classes

Step-2: Identify attributes

Ask: What information must each object remember?

Employee
− employeeId : int − name : string − salary : double + setDepartment(Department&) + display()

Department
− deptId : int − deptName : string + addEmployee(Employee&) + display()

Attribute = data belonging to an object

Representation Rule(s)

- Access specifier identifier : data_type [For attributes/Data members]
- Access specifier identifier(argument_list:data_type) : return_type [For member function]

+ indicates Public, - indicates Private and ~ indicates Protected

Step-3: Identify the relationship

Use the verb in the problem statement.

As per the problem statement,

- every Employee works in exactly one Department.
- One Department may have many Employees.

Therefore: Employee to Department is many-to-one (N:1).

Step-4: Identify the number of instances

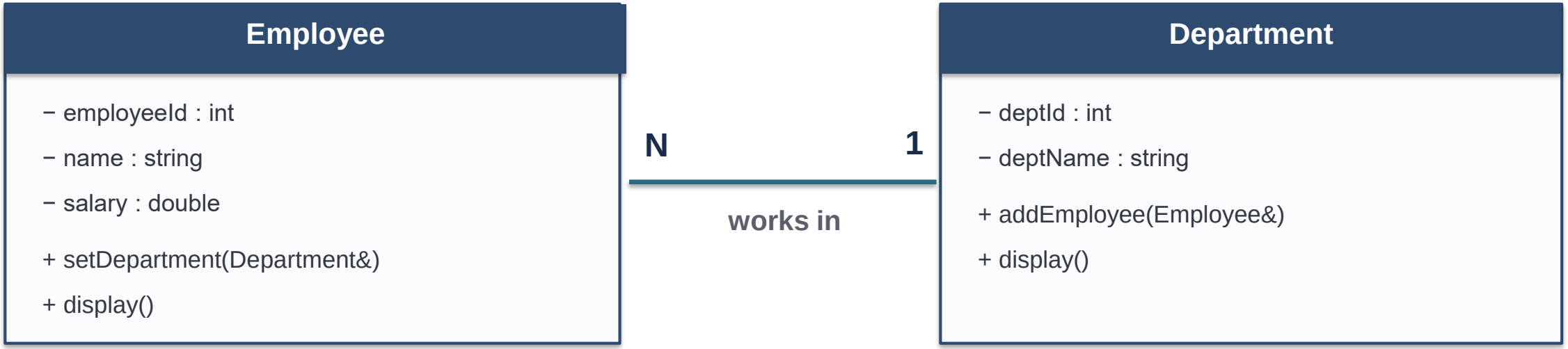
Class = blueprint; object/instance = actual entity.

Department	Department object	Department object
− deptId − deptName	− deptId = D01 − deptName = CSE	− deptId = D02 − deptName = ECE
class	2 Department instances	

Suppose **D01 has 3 employees**, then it indicates that 03 Employee objects are related to the same Department object.

Step-5: Class diagram

Translate the analysis into a visual design.



Step-6: Convert the diagram to code

Create the classes and attributes first.


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

class Department {
    int deptId;
    string deptName;
};

class Employee {
    int employeeId;
    string name;
    double salary;
};

int main() {
    Department d1;
    Employee e1;
}
```

Step-7: Add constructors and arguments

Arguments initialize object attributes.

```
class Department {
    int deptId;
    string deptName;
public:
    Department(int id, string name) {
        deptId = id;
        deptName = name;
    }
};

class Employee {
    int employeeId;
    string name;
    double salary;
public:
    Employee(int id, string n, double s) {
        employeeId = id;
        name = n;
        salary = s;
    }
};
```

Argument → parameter → attribute

Step-8: Implement the relationship with a reference/pointer

The Employee stores a reference to its Department.

```
class Department {
    int deptId;
    string deptName;
public:
    Department(int id, string name)
        : deptId(id), deptName(name) {}
};
```

```
class Employee {
    int employeeId;
    string name;
    double salary;
    Department& department;     // relationship
public:
    Employee(int id, string n, double s,
Department& d)
        : employeeId(id), name(n), salary(s),
department(d) {}

    void display() {
        cout << name << " works in "
             << department.getName();
    }
};
```

Step-9: Add the missing member function

The reference lets Employee access its related Department object.

```
class Department {
    int deptId;
    string deptName;
public:
    Department(int id, string name)
        : deptId(id), deptName(name) {}

    string getName() const {
        return deptName;
    }
};
```

department is a reference to an existing Department object.

`department.getName()` calls a member function of that related object.

No new Department object is created inside Employee.

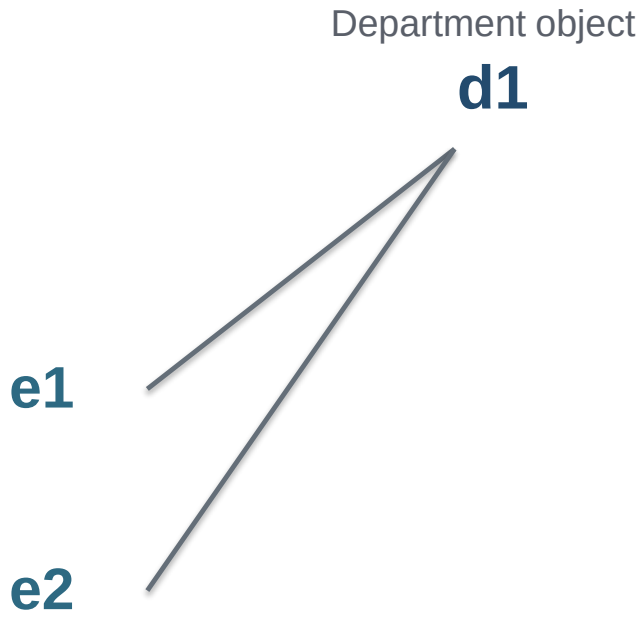
Step-10: Create instances and connect them

Objects make the class diagram concrete.

```
int main() {
    Department d1(101, "Computer Science");

    Employee e1(1, "Aman", 55000, d1);
    Employee e2(2, "Riya", 62000, d1);

    e1.display();
    e2.display();
}
```



Both Employee objects refer to the same d1 object.


दृश्य एवं संकेत सूचना प्रसंसकरण समूह
Visual Signal and Information Processing Group
TIET


ti
THAPAR INSTITUTE
OF ENGINEERING & TECHNOLOGY
(Deemed to be University)

Thank You!