PAGE No.
DATE / /

Assignment ~~-16~~ -17

Q.] What is meant by Function overloading?

Ans · ~~In object orientation~~ Class

1] In object orientation, polymorphism means ~~the name~~ single name & multiple behaviours.

2] Polymorphism is of two types
   1] Compile - Time Polymorphism
   2] Run - Time Polymorphism

3] To achieve the ~~term~~ compile time polymorphism we use the term overloading
   there are two types of overloadding:
   1] Function overloading
   2] Operator overloading

4] According to creator of C++, polymorphism is considered as syntatic sugar which just adds a flavour in our application.

5] In case of of function overloading we can define multiple functions in a class with the same name & with different prototype

Example:

```
Class Demo
{
    Public:
Address   int addition(int a, int b)   addition@2ii
1000      {
              int ans = 0;
              ans = a+b;
```

```
        return ans;
    }
//address int addition(int a, int b, int c)  Addition@2000
    {
        int ans = 0;
        ans = a + b + c;
        return ans;
    }
//address float addition(float a, float b)  Addition@3000
    {
        float fans = 0.0f;
        fans = a+b;
        return fans;
    }
};

int main()
{
    Demo obj;
    int i=20, j=30, k=40
    int ret = 0;

    ret = obj.addition(i,j)      //CALL 1000
    cout<<"Addition is : "<<ret<<endl;

    ret = obj.addition(i,j,k);   //CALL 2000
    cout<<"Addition is : "<<ret<<endl;

    float p=90.0, q=80.0, fret = 0.0f;
    fret = obj.addition(p,q)     //CALL 3000
    cout<<"Addition is : "<<fret<<endl;
    return 0;
}
```

PAGE NO
DATE / /

Q2] Why function overloading is considered as compile time polymorphism?

Ans

1] When we compile the code, compiler changes the name of every function

2] Which function should be called gets decided at the time of compilation, due to which it is called as early-binding / compile time polymorphism.

Q3] What is the use of function overloading?

Ans Due to concept of function overloading as a user of class, there is no need to remember names of different functions.

Q4] What are the scenarios in which we can overload the function?

Ans

1] Name of all functions should be same

2] Data types used as parameters to function must be different.

3] No. of parameters should be different

Q5] What is mean by name mangling / naming decoration?

Ans 1] In function overloading as a user of class, there is no need to remember different names of functions

2] When we compile the code, compiler changes the name of every function with the mangled-name (edited-name)

3] When we overload the function all the names of function are same but due to the concept of name-mangling, the compiler will edit, the name of every function with a new name.

For eg:

mangled name

int Addition (int no1, int no2) Addition@2ii

float Addition (float fno) Addition@1f

4] According to above point we conclude that there is no such concept of function overloading after code gets compiled.

Q 6] Why return value is not considered as function overloading criteria? (Doubt)

Ans

1] In Function overloading, we can define multiple functions in a class with same name & with different prototype

2] During Compilation the function prototype is checked & converted into the mangled name.

~~3] So Function can After Compilation there is no such concept of function overloading~~

3] Mangled names are different. for every fun

4] ~~Hence~~ Therefore after compilation there is no su concept of overloading of function & the desire function gets called.

Q7] What are the scenarios in which we cannot perform function overloading?

Ans

1] We cannot overload the function by changing its return value.

3] We cannot overload the function by changing the access specifier.

Q8] Predict the output of below program.

Ans

```
Class Demo
{
    Public:
    void fun (~~int~~ int i)
    {
        cout << " First Definition";
    }
    void fun ( int i, int j)
    {
        cout << "Second Definition";
    }
};
int main()
{
    Demo obj ();

    Obj. fun (10);
    Obj. fun (10,20);
    return 0;
}
```

OUTPUT

cmd.

First Definition
Second Definition

PAGE NO.
DATE / /

Q9] Predict the o/p of below

```
Class Demo
{
    public:
    void fun(int *p)
    {
        cout<<"First Definition";
    }
    void fun(float *p)
    {
        cout<<"Second Definition";
    }
    void fun(int no)
    {
        cout<<"Third Definition";
    }
};
int main()
{
    int no = 11;
    float f = 3.14;

    Demo obj();
    obj.fun(no);
    obj.fun(&no);
    obj.fun(&f);

    return 0;
}
```

cmd

Compilation Error

PAGE No
DATE / /

Q.10] Draw object layout & Class Diagram of below code snippets & explain its internal working in detail. Explain the type of inheritance in below code snippet.

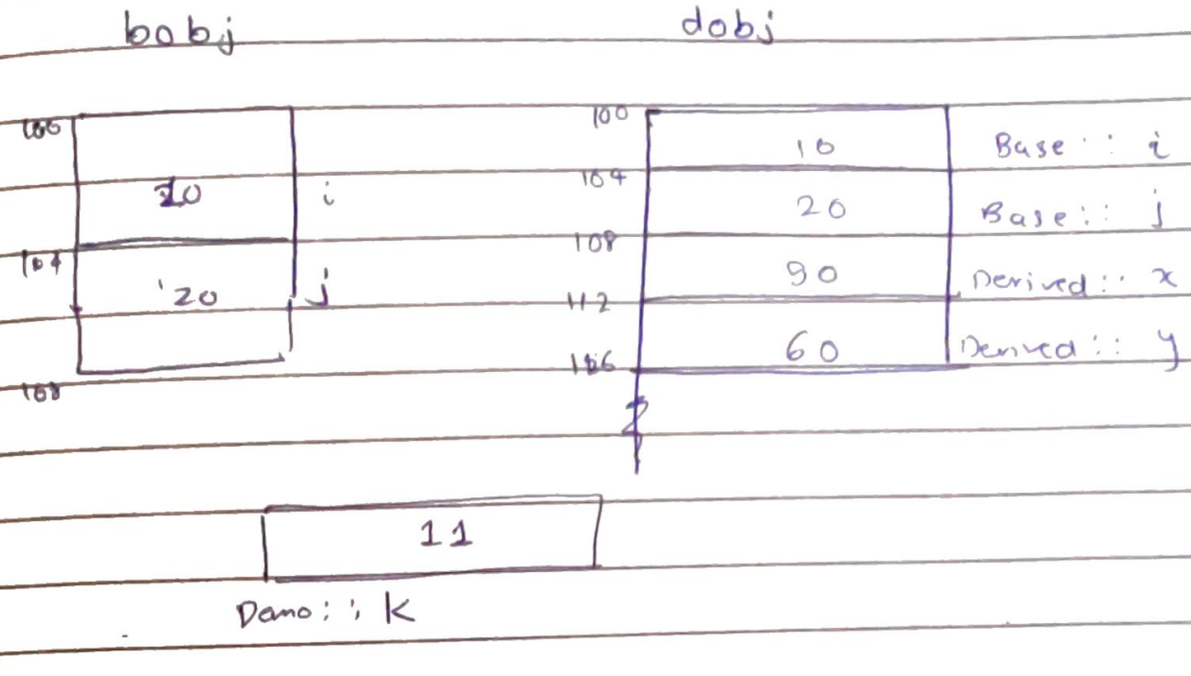
Ans

```
class Base
{
    public:
        int i, j;
        Static int k;

    Base()
    {
        i = 10;
        j = 20;
    }
    void fun()
    {
        cout << "Base Fun";
};

int Base :: k = 11;

class Derived : public Base
{
    public:
        int x, y;
    Derived()
    {
        x = 50;
        y = 60;
    }
```

```
    void gun()
    {
        cout<<" Derived Gun";
    }
};

int main()
{
    Base bobj;
    Derived dobj;

cout<< sizeof(bobj);      // 8 bytes
cout << sizeof(dobj);     // 16 bytes

cout << bobj.i;
cout << bobj.i;
cout << dobj.i;
cout << dobj.j;
cout << bobj.k;
cout<< bobj.x;

bobj.fun();
dobj.fun();
dobj.gun();

    return 0;
}
```

PAGE No
DATE / /

Object Layout



Class Diagram

```
Class Base                                  <-- Class Definition
{
    public:                                 <-- Access specifier
        int i,j;                            <-- Non-static characteristics of class
        static int k;                       <-- static characteristics
    Base ()                                 <-- constructor
    {
        i = 10;                             <-- Initialisation of
        j = 20;                                 characteristic
    }
    void fun ()                             <-- Behaviour of class
    {
        cout << "Base Fun";
    }
};
```

PAGE No
DATE / /

```
int Base::k=11;
```
← Initialization of static characteristic

```
class Derived: public Base
{
    public:
        int x,y;
    Derived()
    {
        x=50;
        y=60;
    }
    void gun()
    {
        cout<<"Derived Gun";
    }
};
```

← ~~Multi~~-Single Inheritance
← Access specifier
← characteristics of class
← Constructor
← Initialization of characteristics
← Behaviour of class

1] This is a Single ~~type~~level Inheritance in which the Derived class inherits characteristics & behaviours of the Base class.

2] Total memory allocated for the derived class object 16 bytes. ~~for~~

3] A~~s~~ static ~~var~~ characteristic 'k' is also initialised

4] Constructor & Destructor calling sequence:
When an object is created, Constructor of Base ~~cat~~ class gets called first & the Constructor of Derived class gets called. The Constructor calling sequence is top to bottom. the Destructor calling sequence is bottop to top.