ASSIGNMENT 17

Q1] What is meant by function overloading

Refer Assignment 13 Q10

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

Function overloading allows multiple functions having same name but different parameter lists.

It is considered as compile time polymorphism because the decision about which overloaded function to call is made by the compiler at compile time.

The compiler determines the appropriate function to call during the compilation process by analyzing the arguments provided in the function call.

Ex
```
#include <iostream>
using namespace std;
void display(int a){
  cout << "Integer: " << a << endl;
}
void display(double a){
  cout << "Double: " << a << endl;
}
int main(){
  display(5);
  display(5.5);
  return 0;
}
```

Q3] What do you mean by Name Mangling / Naming Decoration

Refer Assignment 14 Q5

Q4] Why return value is not considered as function overloading criteria ?

The return value is not considered as function overloading criteria because function calls are resolved at compile time based on their name & parameter list

1> If the compiler relied on the return type to differentiate between overloaded functions, it would create ambiguity

Ex
```
int func();
float func();
func(); // Compiler Ambiguity which function to call
```

2> The return value is used after the function call completes. However, function overloading is resolved before execution.

Q5] Why & What is the use of function overloading ?

Why =>

1> Code Readability & Reusibility:

function overloading allows multiple functions with the same name to co-exist differentiated by their parameter list.

2> It enables the same function name to handle different types without requiring multiple distinct function names.

3> Polymorphism in Compile Time:

It demonstrates compile time polymorphism, where the same function name performs different operations depending on the arguments.

Uses ⇒

1> Handling Different Data Types:
A function can perform the same logical operation on various data types such as int, float, double etc.

2> Varying no. of Arguments:
Functions can be overloaded to accept different nos. of parameters

3> Improving Code Reusability:
Overloading eliminates the need to create multiple function names for similar operations, simplifying code maintainence

Q6] What are the scenarios in which we cannot perform function overloading?

1> When functions differ only by return type, As discussed return type is not part of signature used for overloading

2> Ambigous Overloads: overloading with default arguments can create ambigouity

3> Functions differing obj by 'const':
Non-constant & constant variables of a function cannot be overloaded unless used in classes with 'this' pointer.

Q7] What are the scenarios in which we can overload the function?

1> Different Parameter Types

```
void print (int a);
void print (double d);
```

2> Different no. of Parameter

```
void print (int a);
void print (int a, b);
```

3> Different Parameter Order:

```
Void print (int a, double b);
void print (double a, int b);
```

4> With ref or Pointer Parameters:

```
void print (int &a);
void print (int *a);
```

• Rules:

1> Names of all functions should be same
2> Data Types of parameters should be different
3> No. of Parameters should be different
4> We cannot overload function by changing its return value.

We can overload function by Changing:-

1> No. of Arguments
2> Sequence of Arguments
3> Data Type of Arguments

Q8] Predict o/p of below program

```
#include <iostream>
using namespace std;
class Demo{
public:
void fun(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;
}
```

o/p :- First definition Second definition

Q9] Predict o/p

```
#include<iostream>
using namespace std;
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 = 10;
    float f = 12.3;
    Demo obj();
    obj.fun(no);
    obj.fun(&no);
    obj.fun(&f);
    return 0;
}
```

O/P : Third definitionfirst definitionSecond definition

Q10] Draw object layout & class diagram of below code & explains its internal. Explain type of inheritance in below code.

→ Classes & members :

1> Class Base :

Non static data members : i, j,

Static data member : k

Constructor : Initializes i=10, j = 20

Member function : void fun(), prints "Base fun".

2> Class Derived :

Inherits Base publicly (public Base) means Base's public & protected members are accessible as public & protected in Derived).

Adds new data members : x, y

Constructor : Initializes x=100, y=200

member function : void gon()

Types of Inheritance :

Single public Inheritance : The derived class inherits all public and protected members of the Base class.

Object Layout :

Memory layout for Base Object :

Member	Type	Value (after construction)
i	int	10
j	int	20
k	static int	12 (shared by all objects)

Memory Layout for Derived Object :

Member	Type	Value (After Construction)
Base :: i	int	10
Base :: j	int	20
Derived :: x	int	100
Derived :: y	int	200
Base :: k	static int	12

Internal Working:

1> Static Member k:

k is a class level variable & shared across all objects of Base & Derived.

2> Single Inheritance:

The Derived class can access the public members of Base, including fun().

The Derived class adds its own members (x, y) & method go

3> Object Instantiation:

When a derived object is created:

The Base Constructor is called first to initialize i & j.

Then the Derived Constructor initializes x & y.