C++

# Assignment No :- 17

Q.1] What is mean by function overloading.

→ Function overloading is compile time polymorphism
- Function overloading is a programming concept where functions can have the same name but differ in the number or type of parameters.
- It allows a single function name to handle different types of input, making the code more readable and reusable.
- All overloaded functions have the same name
- But different parameter like -
The numbers of parameters.
The types of parameters.
The order of parameters.

Q.2] Why function overloading is considered as compile time polymorphism.

→ Function overloading considered as compile-time polymorphism because the function to be executed is determined at compile time.
- Based on the function's name, the number of arguments, and their types.
- This decision-making happens before the program runs, distinguishing it from runtime polymorphism.
where decisions are made during execution.

Q.3] What do you mean by ~~m~~ e name mangling/ naming decoration.

→ Name Mangling is the process by which a compiler generates unique names for functions, variables or objects in order to avoid naming

conflicts in programs,
Particularly when features like function overloading or namespace are used.
Name Mangling Needed because
In C++ which support function overloading, multiple function can have the same name but differ in parameter types or numbers.
The compiler needs a way to distinguish between these functions internally, so it mangles the names by appending extra information to the function name.
e.g. int Addition (int no1, int no2)
Addition@ii = Addition @ 2ii
initials of datatypes of every parameters

Q4) Why return value is not considered as function overloading criteria?
→ The return value is not considered as a valid criterion for function overloading because the compiler cannot determine which function to call based on the return type alone.
Function calls are resolved at compile time, and the compiler uses the function name and the parameters to distinguish between overloaded functions not the return type.
Reasons:
1) No return type in function call.
2) Ambiguity at the call site
3) Function Resolution is Based on.

Q5] Why & what is the use of function overloading

→ Function overloading allows multiple functions with the same name to coexist in a program, distinguished by their parameter. It enhances the readability, flexibility and reusability of code.

Uses

1] Improves Code Readability-
Using the same name for similar tasks makes code easier to understand. Instead of creating different function names.

2] Simplifies Function Calls.
Developers can call the same function name without worrying about details like parameters types or numbers.

3] Supports Polymorphism
Function overloading is a form of compile time polymorphisms, where the function to be executed is determined at compile time.

4] Makes Code Reusable.
By overloading a function, developers can reuse the same logic for different types of data without duplicating code.

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

→ The function overloading is powerful, but there are specific scenarios where it cannot be performed due to language constraints or ambiguities.

Below are key scenarios where function overloading fails:

1) Functions Differ only by Return Type
Overloading cannot be done if the functions differ only in their return type because the compiler resolves function calls based on their name and parameters, not their return type.

2) Functions Differ only by Default Parameters
Overloading cannot occur if the difference between two functions lies solely in their default parameters.

3) Ambiuity in Type Conversion
Overloading fails when type conversion creates ambiguity in function calls.

4) Overloading with Ellipsis (...)
Overloading cannot occur when one function has an ellipsis as a parameter because it accepts any argument type, causing ~~ambiitue~~ ambiguity.

Q7) What are the scenarios in which we can overload the function.

→ Function overloading can be performed in the following scenarios:

1) Different number of parameters.
2) Different types of parameters.
3) Different order of parameters.
4) Using const and non-const parameters.
5) Differentiating between references and non-referenc
6) Differentiate between pointers of different type
7) Using templates to handle multiple return types.
8) Differentiating const and non-const member functions.

Q8] Predict the output 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;
}
```

→ Output

first definition second definition.

Q9] Predict the output

```
#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;
}
```

→ Output - Third definition first definition second definition

Q 10] Draw object layout & class diagram of below code snippets and explain its internal. Explain the type of inheritance in the below code.

```
#include<iostream>
using namespace std;
class Base
{ public:
    int i, j;
    static int k;

    Base()
    { i = 10;
      j = 20;
    }
    void fun()
    { cout<<"Base fun"; }
};
int Base :: k = 12;
```

```
class Derived : public Base
{ public:
    int x, y;

    Derived ()
    { x = 1000;
      y = 2000;
    }
    void gun()
    { cout << " Derived Gun";
    }
};
int main()
{ Base bobj();
  Derived dobj();
  cout << sizeof (bobj);
  cout << size of (dobj);

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

  bobj.fun();
  dobj.fun();
  dobj.gun();
return 0;
}
```

- this is single level Inheritance where the Derived class inherits from the Base class.