

Aim:

To understand the concept of **Function Overloading** with the help of an example to compute the area of rectangle, triangle and circle

Theory:

Function overloading (also method overloading) is a programming concept that allows programmers to define two or more functions with the same name and in the same scope.

Each function has a unique signature (or header), which is derived from:

- function/procedure name
- number of arguments
- arguments' type
- arguments' order

Since functions' names are in this case the same, we must preserve uniqueness of signatures, by changing something from the parameter list (last three alienees). If the functions' signatures are sufficiently different, the compiler can distinguish which function was intended to be used at each occurrence. This process of searching for the appropriate function is called function resolution and can be quite an intensive one, especially if there are a lot of equally named functions. Programming languages supporting implicit type conventions usually use promotion of arguments (i.e. type casting of integer to floating-point) when there is no exact function match. The demotion of arguments is rarely used. When two or more functions match the criteria in function resolution process, an ambiguity error is reported by compiler. Adding more information for the compiler by editing the source code (using for example type casting), can address such doubts.

Syntax :

```
int test() { }  
int test(int a) { }  
float test(double a) { }  
int test(int a, double b) { }
```

Code :

```
#include <iostream>  
#include <math.h>  
using namespace::std ;  
  
float area( int a , int b , int c ) {  
    float s = (a+b+c)/2.0 ;  
    return sqrt(s*(s-a)*(s-b)*(s-c)) ;  
}  
  
float area(int l , int b){
```

```

        return l * b ;
    }

    float area( int radius) {
        return 3.14 * radius * radius ;
    }

    int main () {

        cout << area(5,12,13) << endl ;
        cout << area(1,2) << endl ;
        cout << area(1) << endl ;

    }

```

Output :

```

~/College/OOPS ➤ master ➤ ./function-overloading
Find the Area of
1. Triangle
2. Rectangle
3. Circle
Enter -1 to Quit
1
Enter Sides
1 2 2
Area of Traingeles is 0.968246 sq units
Find the Area of
1. Triangle
2. Rectangle
3. Circle
Enter -1 to Quit
2
Enter Sides
2 3
Area of Rectangle is 6 sq units
Find the Area of
1. Triangle
2. Rectangle
3. Circle
Enter -1 to Quit
3
Enter Sides
2
Area of Circle is 12.56 sq units
Find the Area of
1. Triangle
2. Rectangle
3. Circle
Enter -1 to Quit
-1

```

Discussion :

The above program illustrates the concept of function overloading where the function `area` is overloaded to find the area of a triangle, rectangle or circle depending upon the number of arguments given to it.

The area of triangle is calculated using Heron's Formula. Each `area` function has a unique signature which helps the compiler to resolve which function to call.

Learing Outcomes :

- Function Overloading enables consistency in the naming of methods / functions which logically perform very similar tasks, and differ slightly in by accepting different parameters. This allows the same method name to be reused across multiple implementations.
- Consistency in notation, which is good both for reading and for writing code

Aim:

To understand the concept of **Static Members** with the help of an example.

Theory:

A static member is shared by all objects of the class. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member. All static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it can be initialized outside the class as done in the following example by redeclaring the static variable, using the scope resolution operator `::` to identify which class it belongs to.

By declaring a function member as static, you make it independent of any particular object of the class. A static member function can be called even if no objects of the class exist and the static functions are accessed using only the class name and the scope resolution operator `::`.

A static member function can only access static data member, other static member functions and any other functions from outside the class.

Syntax :

```
class A {
    static int a ;
    public :
        static void test(){ }
};

// Using Static Members

A::a ;
A::test() ;
```

Code :

```
#include <iostream>

using namespace std;

class Test {
    int code ;
    static int count ;

    public :
        void setcode(){
            code = ++count ;
        }
        void showcode(){
            cout << "Obj No : " << code << endl;
        }
}
```

```

        static void showcount(){
            cout << "Object Count : " << count << endl ;
        }
};

int Test::count ;

int main (){
    Test T1, T2 ;
    T1.setcode() ;
    T2.setcode() ;

    Test::showcount() ;

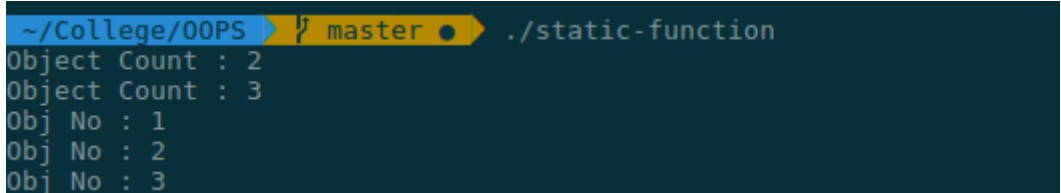
    Test T3 ;
    T3.setcode() ;

    Test::showcount() ;
    T1.showcode();
    T2.showcode();
    T3.showcode();

    return 0;
}

```

Output :



```

~/College/OOPS } master ● ./static-function
Object Count : 2
Object Count : 3
Obj No : 1
Obj No : 2
Obj No : 3

```

Discussion :

The above program illustrates the concept of static members. The static member `count` of class `Test` is declared as static and is used to store the number of objects created. A static member function `showcount()` is used to display that count and is called using the class name.

Learning Outcomes :

- It eliminates the need for an object as the members can be accessed using class.
- Static Data Members can be Encapsulated. A static member can be a private member, but a global variable cannot.
- It acts as a way to interact between different objects of the same class.

Aim:

To understand the concept of **Classes** with the help of an example.

Theory:

The building block of C++ that leads to Object Oriented programming is a Class. It is a user defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

A class is defined in C++ using keyword `class` followed by the name of class. The body of class is defined inside the curly brackets and terminated by a semicolon at the end.

Syntax :

```
class class_name {
    access_specifier_1:
        member1;
    access_specifier_2:
        member2;
    ...
} ;

// Initializing

class_name object_name ;
```

Code :

```
#include <iostream>

using namespace::std;

class Customer {
    char name[50] ;
    double account_number ;
    char account_type ;
    float balance ;

    public :

    void input() {
        cin.clear();
        cout << "Enter Name : " ;
        cin.getline(name , 50 , '\n' ) ;
        cout << "Enter Account Number : " ;
        cin >> account_number ;
```

```

        cout << "Enter Account Type : " ;
        cin >> account_type ;
        cout << "Enter Balance : Rs. " ;
        cin >> balance ;
    }

    void display() {

        cout << "Name : " << name << endl ;
        cout << "Account Number : " << account_number << endl ;
        if (account_type == 'S' ) {
            cout << "Type : Saving" << endl ;
        } else {
            cout << "Type : Current" << endl ;
        }
        cout << "Balance : Rs. " << balance << endl ;
    }

    void deposit(){
        int amount ;
        cout << "Enter Amount to Deposit : Rs. " ;
        cin >> amount ;
        balance += amount ;
        cout << "New Balance : Rs. " << balance << endl ;
    }

    void withdraw() {
        int amount ;
        cout << "Enter Amount to Withdraw : Rs. " ;
        cin >> amount ;
        if ( balance - amount < 1000) {
            cout << "Insufficient Balance " << endl ;
            return ;
        }
        balance -= amount ;
        cout << "New Balance : Rs. " << balance << endl ;
    }
};

int main () {

    Customer customers[10] ;
    int choice;
    char cont ;

    for( int i = 0 ; i < 10 ; i++ ) {

        cout << "#####\n" ;
        cout << "##### Customer " << i + 1 << " #####\n" ;
        cout << "#####\n" ;

        customers[i].input();
        while(1) {
            cout << "1. Withdraw" << endl ;
            cout << "2. Deposit" << endl ;
            cout << "3. Display" << endl ;

```

```

        cout << "Enter Choice : ";
        cin >> choice ;
        if (choice == 1 ) {
            customers[i].withdraw() ;
        } else if (choice == 2 ) {
            customers[i].deposit() ;
        } else if (choice == 3 ) {
            customers[i].display() ;
        } else {
            cout << "Invalid Choice" << endl ;
        }
        cout << "Wanna Continue (y or n) : " ;
        cin >> cont ;
        cin.clear();
        if (cont == 'n' || cont == 'N') {
            break;
        }
    }

    return 0 ;
}

```

Output :

```

~/College/00PS ➤ master ➤ ./bank
#####
##### Customer 1 #####
#####
Enter Name : Dhruv
Enter Account Number : 123
Enter Account Type : S
Enter Balance : Rs. 1200
1. Withdraw
2. Deposit
3. Display
Enter Choice : 1
Enter Amount to Withdraw : Rs. 150
New Balance : Rs. 1050
Wanna Continue (y or n) : y
1. Withdraw
2. Deposit
3. Display
Enter Choice : 2
Enter Amount to Deposit : Rs. 300
New Balance : Rs. 1350
Wanna Continue (y or n) : y
1. Withdraw
2. Deposit
3. Display
Enter Choice : 3
Name : Dhruv
Account Number : 123
Type : Saving
Balance : Rs. 1350
Wanna Continue (y or n) : n
#####
##### Customer 2 #####
#####

```


Discussion :

The program stimulates a simple queue of 10 people in bank, where 1 person can withdraw , deposit cash in account or display the details of its account. The process is stimulated using the concept of classes where each Customer is represented by an Object. All the related data to customer and functions like display are encapsulated within the `Customer` class.

Learning Outcomes :

- Classes are a way to organize your code into generic, reusable pieces. At their best they are generic blueprints for things that will be used over and over again with little modification.
- Classes use Data Hiding and Encapsulation and knowledge of its implementation is not necessary for its use.
- Classes can restrict access to data, removing bug-opportunities that direct access could give.