**UNIT-2**

**EXPERIMENT 1**

**AIM:**

To write a C++ program to demonstrate the concepts of inline functions and function overloading.

**DESCRPTION:**

1. **Inline Function:**
   * An inline function is a function where the compiler places a copy of the code of that function at each point the function is called.
   * This is used to reduce function call overhead, especially for small, frequently called functions.
   * Declared using the inline keyword.
2. **Function Overloading:**
   * Function overloading allows multiple functions to have the same name with different parameter types or numbers.
   * The compiler differentiates them by their signature (number/type/order of parameters).
   * It supports polymorphism at compile-time (static polymorphism).

**STEPS:**

1. Start your C++ program and include the necessary headers.
2. Define one or more inline functions using the inline keyword.
3. Define multiple functions with the same name but different parameter types or counts to demonstrate function overloading.
4. In the main() function:
   * Call the inline function.
   * Call the overloaded functions with different arguments.
5. Compile and run the program.
6. Observe the results and verify the behavior of inline and overloaded functions.

**PROGRAM:**

// demonstration INLINE FUNCTIONS

#include<iostream>

using namespace std;

inline int square(int x)

{

return x \* x;

}

int main() {

cout << "Square of 5 is: " << square(5) << endl;

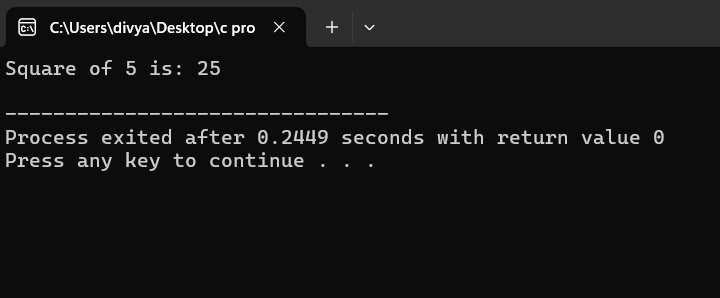
// here, instead of calling the square() function,

// the compiler replaces it with 5 \* 5 at compile time.

return 0;

}

**OUTPUT:**



// Demonstration of OVERLOADING MEMBER FUNCTION

#include<iostream>

using namespace std;

class DemoOverloading

{

public:

void add()

{

cout<<"\nI am in add () - No Arg ";

}

void add(int a, int b)

{

cout<<"\nI am in add (int, int ), The sum = "<<a+b;

}

void add(float a, float b)

{

cout<<"\nI am in add (float, float ) - The sum = "<<a+b;

}

void add(int a, float b)

{

cout<<"\nI am in add(int, float) - The sum="<<a+b;

}

};

int main()

{

DemoOverloading ob;

ob.add();

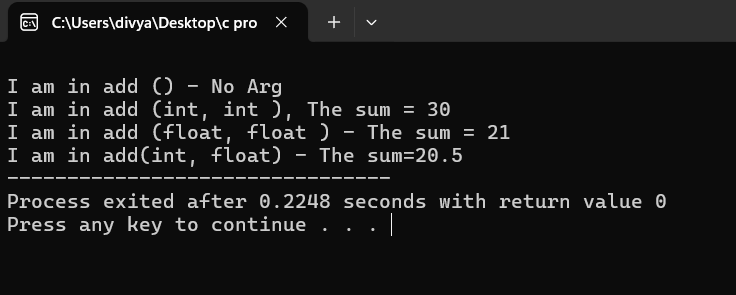
ob.add(10,20);

ob.add(10.5f,10.5f);

ob.add(10,10.5f);

}

**OUTPUT:**



**RESULT:**

The C++ program was successfully compiled and executed.  
It demonstrated the use of inline functions and function overloading.

* The inline function square() correctly returns the square of the number.
* The overloaded add() functions are called according to the types and number of arguments passed, demonstrating function overloading.

**EXPERIMENT 2**

**AIM:**

To write a C++ program to illustrate the concept of a friend function.

**DESCRIPTION:**

In C++, a friend function is a function that is not a member of a class but is given access to theprivate and protected members of the class. It is declared using the keyword friend inside the class.

A friend function can be:

* A normal function
* A member of another class
* A global function

Friend functions are useful when two or more classes need to share some private data.

**EXPLAINATION:**

->The class Number has a private member value.

->The function add() is declared as a friend inside the class.

->Although add() is not a member of the class, it can access value directly.

->Two Number objects are created in main(), and the add() function accesses their private data to compute the sum.

**PROGRAM:**

// Adding Two Classes using FRIEND FUNCTIONS

#include<iostream>

using namespace std;

class Demo

{ private:

int x, y;

void fun1()

{

x = 10;

y = 20;

cout << "I am inside member function (fun1()) in the class" << endl;

}

friend void add(Demo d); // Declare friend function with parameter

};

void add(Demo d) { // Friend function

// Cannot call d.fun1() because it is private

// So we need to set values directly if possible, or use constructor

d.x = 10; // Accessing private data

d.y = 20;

cout << "Sum = " << d.x + d.y << endl;

}

int main() {

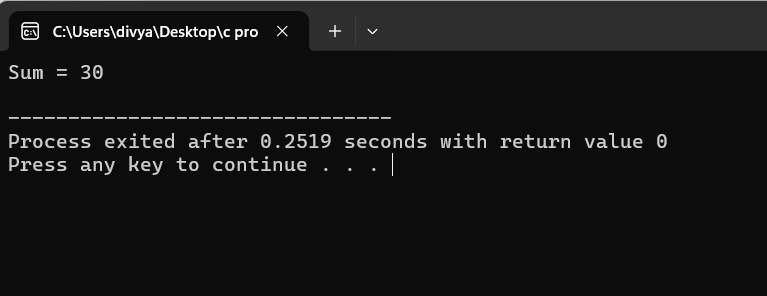
Demo ob;

add(ob); // Pass object to friend function

return 0;

}

**OUTPUT:**



**RESULT:**

The program was successfully compiled and executed. It demonstrates the use of a friend function to access and manipulate the private data of a class.

**EXPERIMENT 3**

**AIM:**

To write a C++ program to demonstrate the use of constructors and destructors.

**DESCRIPTION:**

In C++, constructors and destructors are special member functions of a class.

* A constructor is automatically called when an object is created. It is used to initialize object data members. It has the same name as the class and no return type.
* A destructor is automatically called when an object is destroyed. It is used to perform cleanup operations like releasing memory or closing files. It has the same name as the class, prefixed with a tilde ~, and no return type or parameters.

Constructors and destructors are useful for automatic resource management and initialization/cleanup tasks.

**EXPLAINATION**:

* The constructor Demo(int n) is called automatically when an object is created (e.g., Demo A(10);).
* It initializes the private data member number and prints a message.
* The destructor ~Demo() is called automatically when the object goes out of scope (at the end of main() in this case), printing a message.
* This shows how constructors and destructors work behind the scenes to manage object lifecycle.

**PROGRAM:**

// demo constructor and destructor

#include <iostream>

using namespace std;

class Demo

{

public:

Demo() {

cout << "Constructor called: Object is created." << endl;

}

// Destructor

~Demo() {

cout << "Destructor called: Object is destroyed." << endl;

}

// Member function

void display() {

cout << "Inside display function." << endl;

}

};

int main() {

{

Demo obj; // Constructor will be called

obj.display();

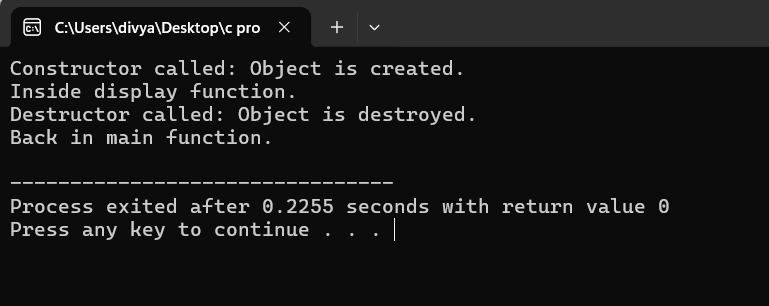
} // Destructor will be called automatically when obj goes out of scope

cout << "Back in main function." << endl;

return 0;

}

**OUTPUT:**



**RESULT:**

The program was executed successfully.  
It demonstrated the use of constructor for initializing objects and destructor for cleaning up when the object goes out of scope.

Thus, the concept of constructors and destructors was successfully implemented and verified.

**EXPERIMENT 4**

**AIM:**

To write a C++ program to demonstrate constructor overloading, where multiple constructors with different parameters initialize objects in different ways.

**DESCRIPTION:**

Constructor overloading means having more than one constructor in a class with different parameter lists. This allows creating objects in different ways by calling the appropriate constructor based on the arguments passed.

It improves flexibility in object initialization and supports different ways of setting up class members.

**PROGRAM:**

// Constructor overloading

#include<iostream>

using namespace std;

class Demo

{

public:

Demo()

{

cout<<"I am in Default Constructor";

}

Demo(int x)

{

cout<<"\nI am in One arg constructor"<<x;

}

Demo(char str[12])

{

cout<<"\nI am in one arg string constructor"<<str;

}

~Demo()

{

cout<<"\nI am in DESTRUCTOR";

}

};

int main()

{

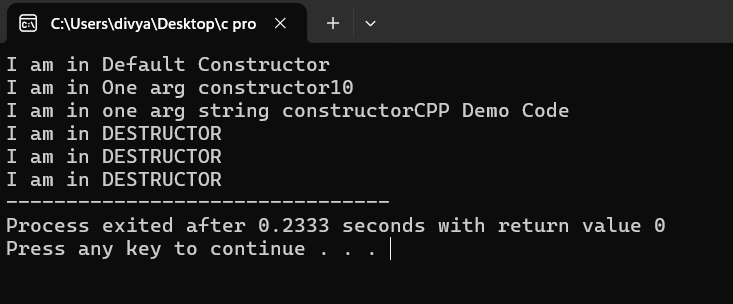
Demo ob;

Demo ob1(10);

Demo ob2("CPP Demo Code");

}

**OUTPUT:**



**RESULT:**

The program was executed successfully.  
It demonstrated constructor overloading by creating objects using different constructors based on the number of parameters passed.  
Thus, the concept of constructor overloading was successfully implemented and verified.

**EXPERIMRNT 5**

**AIM:**

To write a C++ program to illustrate the use of a copy constructor.

**DESCRIPTION:**

A copy constructor in C++ is a special constructor used to create a new object as a copy of an existing object.  
It is automatically called when:

* An object is passed by value.
* An object is returned from a function.
* An object is explicitly copied during initialization.

Syntax:

ClassName(const ClassName &obj) {

// body

}

If a copy constructor is not defined by the programmer, the compiler provides a default copy constructor that performs a shallow copy.

**PROGRAM:**

// Demonstration of Copy Constructor

#include<iostream>

using namespace std;

class Student

{ int id;

public: Student(int i)

{ id = i;

cout << "Constructor called\n";

}

Student(Student &s) // Copy constructor

{ id = s.id;

cout << "Copy constructor called\n";

}

void display()

{ cout << "ID: " << id << endl;

}

};

int main() {

Student s1(500); // Constructor called here

Student s2 = s1; // Copy constructor called here

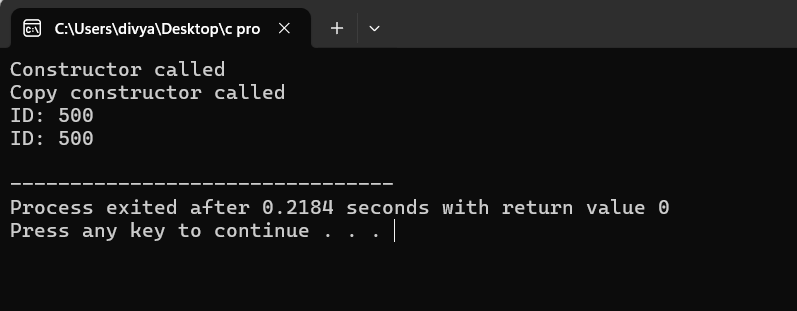
s1.display();

s2.display();

return 0;

}

**OUTPUT:**



**RESULT:**

The program was successfully compiled and executed.  
It demonstrated the use of a copy constructor by creating a duplicate object from an existing one.  
The output confirmed that the copy constructor was invoked automatically when a new object was initialized using an existing object.  
Thus, the concept of copy constructor was successfully implemented and verified.