

Date:

Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

Program No.: 5(A)

Develop Create a C++ program that demonstrates the use of constructors and destructors in a class.

Aim: To develop a C++ program that demonstrates the use of constructors and destructors in a class.

Description:

A **constructor** is a special member function of a class that is **automatically called** when an object is created.

- It is mainly used to **initialize data members** of a class.
- It has the **same name as the class** and **no return type**.

A **destructor** is a special member function that is **automatically called** when an object goes out of scope or is explicitly deleted.

Syntax:

```
class ClassName {  
public:  
    // Constructor  
    ClassName() {  
        // Initialization code  
    } // Destructor  
    ClassName() {  
        // Cleanup code };
```



A D I T Y A
UNIVERSITY

Program code:

```
#include <iostream>  
using namespace std;  
class Demo {  
private:  
    int value;  
public:  
    Demo(int v)  
{
```

Date:

Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

```
value = v;

}

Demo()

{

    cout << "Destructor called, value = " << value << endl;

}

void display()

{

    cout << "Value is: " << value << endl;

}

};

int main() {

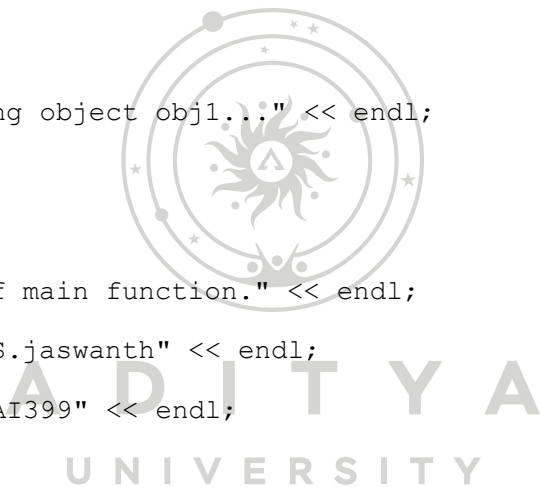
    cout << "Creating object obj1..." << endl;
    Demo obj1(20);
    obj1.display();

    cout << "End of main function." << endl;

    cout << "Name:S.jaswanth" << endl;
    cout << "24B11AI399" << endl;

    return 0;

}
```



Date:

Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

Output:

```
C:\Users\Jaswa\OneDrive\Doc × + v
Creating object obj1...
Value is: 20
End of main function.
Name:S.jaswanth
24B11AI399
```

CO Mapped: CO2

POs Mapped: PO1, PO2, PO3, PO4, PO5, PO9, PO11

PSOs Mapped: PSO1



A D I T Y A
U N I V E R S I T Y

Date:

Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

Program No.: 5(B)

Develop a C++ program that illustrates constructor overloading.

Aim: To develop a C++ program that illustrates constructor overloading

Description:

- **Constructor Overloading** in C++ means having **multiple constructors** in the same class, but with **different parameter lists** (number or type of parameters).
- This allows creating objects in **different ways** depending on the arguments passed.
- The compiler decides which constructor to call based on the arguments (compile-time polymorphism).

Syntax:

```
class ClassName {  
public:  
    // Default constructor  
    ClassName() {  
        // Code to initialize  
    } // Parameterized constructor (1 argument)  
  
    ClassName(int x) {  
        // Initialize with x  
    } // Parameterized constructor (2 arguments)  
  
    ClassName(int x, int y) {  
        // Initialize with x and y  
    }  
};
```

Program:

```
#include<iostream>  
  
using namespace std;  
  
class Aravind  
{  
public:  
    void display()  
{
```

Date:

Roll No.:

2	4	B	1	1	A	I	3	9	9
---	---	---	---	---	---	---	---	---	---

```
        cout<<"I am display function without any parameters\n";
    }
void display(string a)
{
    cout<<"I am display funtion with paramter string:"<<a<<endl;
}
void display(int a)
{
    cout<<"I am display function with paramter int:"<<a<<endl;
}
int display(int a,int b)
{
    return (a+b);
}

};

int main(){
    Aravind v;
    cout<<"the sum of two numbers is:"<<v.display(16,19) <<endl;

v.display();
v.display(10);
v.display("s.jaswanth");
cout<<"24B11AI399" ;
return 0;
}
```



A D I T Y A
U N I V E R S I T Y

Date:

Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

Output:

```
C:\Users\Jaswa\OneDrive\Doc  X + v
the sum of two numbers is:35
I am display function without any parameters
I am display funtion with paramter int:10
I am display funtion with paramter string:s.jaswanth
24B11AI399
-----
```

CO Mapped: CO2

POs Mapped: PO1, PO2, PO3, PO4, PO5, PO9, PO11

PSOs Mapped: PSO1



Date:

Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

Program No.: 5(C)

Develop a C++ program that illustrates the use of a copy constructor

Aim: To develop a C++ program that illustrates the use of a copy constructor

Description:

- A **copy constructor** is a special constructor in C++ that initializes an object **by copying data from another object** of the same class.
- It is invoked when:
 1. A new object is created from an existing object (ClassName obj2 = obj1;).
 2. An object is passed **by value** to a function.
 3. An object is returned **by value** from a function.
- If you don't define one, the compiler provides a **default copy constructor** that performs a **shallow copy** (bitwise copy).
- You can define your own **user-defined copy constructor** to perform a **deep copy** when needed.

Syntax:

```
class ClassName {  
public:  
    // Copy constructor  
    ClassName(const ClassName &obj) {  
        // Copy data from obj    }  
};
```

Program:


```
#include <iostream>  
using namespace std;  
class Student {  
private: string name;  
        int age;  
public: Student(string n, int a) {  
        name = n;  
        age = a;  
}  
        Student(const Student &s) {  
        name = s.name;  
        age = s.age;  
}
```

Date:

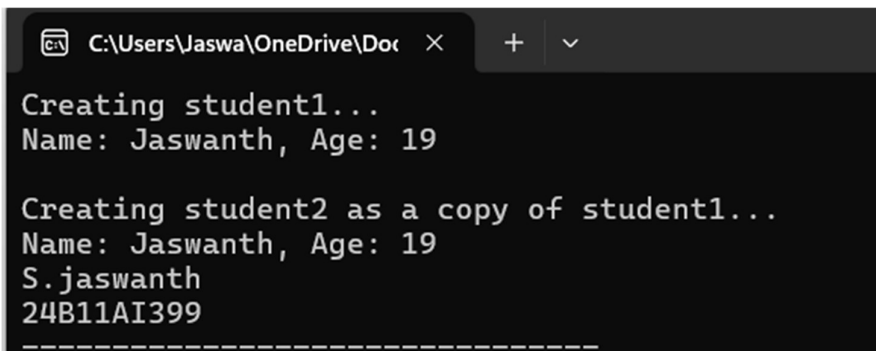
Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

```
void display() {  
    cout << "Name: " << name << ", Age: " << age << endl;  
}  
  
};  
  
int main() {  
    cout << "Creating student1..." << endl;  
    Student student1("Jaswanth", 19);  
    student1.display();  
  
    cout << "\nCreating student2 as a copy of student1..." << endl;  
    Student student2 = student1;  
    student2.display();  
    cout<<"S.jaswanth"<<endl;  
    cout<<"24B11AI399";  
  
    return 0;  
}
```



Output:



```
C:\Users\Jaswa\OneDrive\Doc >  
Creating student1...  
Name: Jaswanth, Age: 19  
  
Creating student2 as a copy of student1...  
Name: Jaswanth, Age: 19  
S.jaswanth  
24B11AI399  
-----
```

CO Mapped: CO2

POs Mapped: PO1, PO2, PO3, PO4, PO5, PO9, PO11

PSOs Mapped: PSO1

Date:

Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

Program No.: 6(A)

Develop a C++ program that demonstrates how to overload both unary and binary operators using member functions.

Aim: Develop a C++ program that demonstrates how to overload both unary and binary operators using member functions.

Description:

In C++, **operator overloading** allows you to redefine the behavior of operators (+, -, ++, etc.) for **user-defined types (classes/objects)**.

- **Unary operator overloading:** Deals with operators that act on a **single operand** (e.g., ++, --, -, !).
- **Binary operator overloading:** Deals with operators that act on **two operands** (e.g., +, -, *, /).
- Overloading is done using a **member function** or a **friend function**.
- Syntax inside a **member function**:

Syntax:

```
// Unary operator overloading
ClassName operator-();

// Binary operator overloading
ClassName operator+(const ClassName &obj);

// Display function
void display();};
```

Program:

```
#include<iostream>
#include<sstream>
using namespace std;
class Complex
{
private:
int real;
int img;
public:
Complex(int r,int i) :real(r),img(i){}
friend Complex operator-(Complex& obj);
friend Complex operator+(Complex& obj1, Complex& obj2);
```

Date:

Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

```
string display()
{
    stringstream s;
    if (img<0)
    {
        s<<real<<" "<<img<<"i\n";
        return s.str();
    }
    else{s<<real<<"+"<<img<<"i\n";
        return s.str();
    }
}

Complex operator-(Complex& obj)
{
    return Complex(-obj.real,-obj.img);
}

Complex operator+(Complex& obj1,Complex& obj2)
{
    return Complex(obj1.real+obj2.real,obj1.img+obj2.img);
}

int main()
{
    Complex c1(11,20);
    cout<<"The Complex c1:"<<c1.display();
    Complex c2(7,45);
    cout<<"The Complex c2:"<<c2.display();
    Complex c3=-c2;
    cout<<"The Complex c3:"<<c3.display();
    Complex c4=c1+c2;
    cout<<"The Complex c4:"<<c4.display();
    return 0;
}
```

Date:

Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

}

Output:

```
Output
The Complex c1:11+20i
The Complex c2:7+45i
The Complex c3:-7-45i
The Complex c4:18+65i

=== Code Execution Successful ===
```

CO Mapped: CO2

POs Mapped: PO1, PO2, PO3, PO4, PO5, PO9, PO11

PSOs Mapped: PSO1



A D I T Y A
U N I V E R S I T Y

Date:

Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

Program No.: 6(B)

Develop a C++ program to demonstrate operator overloading for unary and binary operators using friend functions

Aim: Develop a C++ program that demonstrates operator overloading for unary and binary operators using friend functions

Description:

In C++, **operator overloading** allows you to redefine the behavior of operators (+, -, ++, etc.) for **user-defined types (classes/objects)**.

- **Unary operator overloading:** Deals with operators that act on a **single operand** (e.g., ++, --, -, !).
- **Binary operator overloading:** Deals with operators that act on **two operands** (e.g., +, -, *, /).
- Overloading is done using a **member function** or a **friend function**.
- Syntax inside a **member function**:

Syntax:

```
class ClassName {  
private:  
    // data members  
public:  
    // Constructor  
    ClassName();  
    // Unary operator (friend function)  
    friend ClassName operator-(ClassName obj);  
    // Binary operator (friend function)  
    friend ClassName operator+(ClassName obj1, ClassName obj2);  
};
```

Program:

```
#include<iostream>  
  
#include<sstream>  
  
using namespace std;  
  
class Complex{  
private:  
    int real;
```

Date:

Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

```
int img;

public:

Complex(int r,int i) :real(r),img(i){}

friend Complex operator-(Complex& obj);

friend Complex operator+(Complex& obj1, Complex& obj2);

string display(){

ostringstream s;

if (img<0){

s<<real<<" "<<img<<"i\n";

return s.str();

}

else{s<<real<<"+"<<img<<"i\n";

return s.str();

}}};

Complex operator-(Complex& obj)

{

return Complex(-obj.real,-obj.img);

}

Complex operator+(Complex& obj1,Complex& obj2)

{

return Complex(obj1.real+obj2.real,obj1.img+obj2.img);

}

int main(){

Complex c1(7,45);

cout<<"The Complex c1:"<<c1.display();

Complex c2(18,17);

cout<<"The Complex c2:"<<c2.display();

Complex c3=-c2;

cout<<"The Complex c3:"<<c3.display();
```



A D I T Y A
U N I V E R S I T Y

Date:

Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

```
Complex c4=c1+c2;  
cout<<"The Complex c4:"<<c4.display();  
return 0;  
}
```

Output:

```
Output  
The Complex c1:5+35i  
The Complex c2:18+17i  
The Complex c3:-18-17i  
The Complex c4:23+52i  
  
=== Code Execution Successful ===
```

CO Mapped: CO2

POs Mapped: PO1, PO2, PO3, PO4, PO5, PO9, PO11

PSOs Mapped: PSO1

A D I T Y A
U N I V E R S I T Y

Date:

Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

Program No.: 7(A)

Develop C++ programs to demonstrate different forms of inheritance

Aim: Develop C++ programs to demonstrate different forms of inheritance

Description:

1. Single Inheritance

- In single inheritance, a derived class inherits from only one base class.
- It is the simplest form of inheritance and allows the derived class to reuse the properties and methods of the base class.

2. Multiple Inheritance

- In multiple inheritance, a derived class inherits from two or more base classes.
- This allows the derived class to combine functionalities from multiple base classes.

3. Multi-level Inheritance

- Multi-level inheritance forms a chain of classes, where a class is derived from another derived class.
- The properties are inherited from the topmost base class down the chain.

4. Hierarchical Inheritance

- In hierarchical inheritance, multiple derived classes inherit from a single base class.
- Each derived class can have its own additional properties and methods while sharing the base class features.

5. Hybrid Inheritance

- Hybrid inheritance is a combination of two or more types of inheritance.
- It may combine single, multiple, multi-level, or hierarchical inheritance in a single program.

Syntax:

1. Single Inheritance:

```
class Base {  
    // base class members  
};  
  
class Derived : public Base {  
    // derived class members  
};
```

2. Multiple Inheritance

```
class Base1 {  
    // members of Base1
```

Date:

Roll No.:

2	4	B	1	1	A	I	3	9	9
---	---	---	---	---	---	---	---	---	---

```
};

class Base2 {
    // members of Base2
};

class Derived : public Base1, public Base2 {
    // derived class members
};
```

3. Multi-level Inheritance

```
class Base {
    // base class members
};

class Derived1 : public Base {
    // derived from Base
};

class Derived2 : public Derived1 {
    // derived from Derived1
};
```

4. Hierarchical Inheritance

```
class Base {
    // base class members
};

class Derived1 : public Base {
    // derived from Base
};

class Derived2 : public Base {
    // derived from Base
};
```

5. Hybrid Inheritance

```
class Base {
    // base class members
};

class Derived1 : public Base {
    // derived from Base
};

class Derived2 : public Base {
    // derived from Base
};
```



ADITYA
UNIVERSITY

Date:

Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

```
class Derived3 : public Derived1, public Derived2 {  
    // derived from multiple classes  
};
```

Program:

```
#include <iostream>  
  
using namespace std;  
  
class SingleBase {  
public:  
  
    void showSingleBase() {  
  
        cout << "SingleBase class function" << endl;  
  
    }  
};  
  
class SingleDerived : public SingleBase {  
public:  
  
    void showSingleDerived() {  
  
        cout << "SingleDerived class function" << endl;  
  
    }  
};  
  
class MultiBase1 {  
public:  
  
    void showMultiBase1() {  
  
        cout << "MultiBase1 class function" << endl;  
  
    }  
};  
  
class MultiBase2 {  
public:  
  
    void showMultiBase2() {  
  
        cout << "MultiBase2 class function" << endl;  
  
    }  
};  
  
class MultiDerived : public MultiBase1, public MultiBase2 {  
public:  
  
    void showMultiDerived() {
```

Date:

Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

```
        cout << "MultiDerived class function" << endl;

    });

class LevelBase {
public:

    void showLevelBase() {

        cout << "LevelBase class function" << endl;

    });

class LevelDerived1 : public LevelBase {
public:

    void showLevelDerived1() {

        cout << "LevelDerived1 class function" << endl;

    });

class LevelDerived2 : public LevelDerived1 {
public:

    void showLevelDerived2() {

        cout << "LevelDerived2 class function" << endl;

    });

class HierBase {
public:

    void showHierBase() {

        cout << "HierBase class function" << endl;

    });

class HierDerived1 : public HierBase {
public:

    void showHierDerived1() {

        cout << "HierDerived1 class function" << endl;

    });

class HierDerived2 : public HierBase {
public:
```

Date:

Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

```
void showHierDerived2() {
    cout << "HierDerived2 class function" << endl;
};

class HybridBase {
public:
    void showHybridBase() {
        cout << "HybridBase class function" << endl;
    };
};

class HybridDerived1 : public HybridBase {
public:
    void showHybridDerived1() {
        cout << "HybridDerived1 class function" << endl;
    };
};

class HybridDerived2 : public HybridBase {
public:
    void showHybridDerived2() {
        cout << "HybridDerived2 class function" << endl;
    };
};

class Hybrid : public HybridDerived1, public HybridDerived2 {
public:
    void showHybrid() {
        cout << "Hybrid class function" << endl;
    };
};

int main() {
    cout << "--- Single Inheritance ---" << endl;
    SingleDerived sObj;
    sObj.showSingleBase();
    sObj.showSingleDerived();
    cout << "\n--- Multiple Inheritance ---" << endl;
```

Date:

Roll No.:

2	4	B	1	1	A	I	3	9	9
---	---	---	---	---	---	---	---	---	---

```
MultiDerived mObj;

mObj.showMultiBase1();

mObj.showMultiBase2();

mObj.showMultiDerived();

cout << "\n--- Multi-level Inheritance ---" << endl;

LevelDerived2 lObj;

lObj.showLevelBase();

lObj.showLevelDerived1();

lObj.showLevelDerived2();

cout << "\n--- Hierarchical Inheritance ---" << endl;

HierDerived1 hObj1;

HierDerived2 hObj2;

hObj1.showHierBase();

hObj1.showHierDerived1();

hObj2.showHierBase();

hObj2.showHierDerived2();

cout << "\n--- Hybrid Inheritance ---" << endl;

Hybrid hyObj;

hyObj.HybridDerived1::showHybridBase();

hyObj.HybridDerived2::showHybridBase();

hyObj.showHybridDerived1();

hyObj.showHybridDerived2();

hyObj.showHybrid();

cout<<"24B11AI399";

return 0;

}
```

Output:

Date:

Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

```
C:\Users\Jaswa\OneDrive\Doc  X + v
--- Single Inheritance ---
SingleBase class function
SingleDerived class function

--- Multiple Inheritance ---
MultiBase1 class function
MultiBase2 class function
MultiDerived class function

--- Multi-level Inheritance ---
LevelBase class function
LevelDerived1 class function
LevelDerived2 class function

--- Hierarchical Inheritance ---
HierBase class function
HierDerived1 class function
HierBase class function
HierDerived2 class function

--- Hybrid Inheritance ---
HybridBase class function
HybridBase class function
HybridDerived1 class function
HybridDerived2 class function
Hybrid class function
24B11AI399
-----
```

CO Mapped: CO3

POs Mapped: PO1, PO2, PO3, PO4, PO5, PO11

PSOs Mapped: PSO1

Date:

Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

Program No.: 7(B)

Develop a C++ program that illustrates the order of execution for constructors and destructors in the context of inheritance

Aim: Develop a C++ program that illustrates the order of execution for constructors and destructors in the context of inheritance

Description:

In C++ inheritance, the **order of execution of constructors and destructors** is important:

1. Constructor Execution Order:

- o When creating an object of a derived class, **base class constructors** are called first, then the **derived class constructors**.
- o This ensures the base part of the object is initialized before the derived part.

2. Destructor Execution Order:

- o When an object is destroyed, the **derived class destructor** is called first, then the **base class destructor**.
- o This ensures resources in the derived class are released before the base class.

This applies to **single, multi-level, or multiple inheritance**, but the principle is **base first (construction), derived first (destruction)**.

Syntax:

```
class Base {  
public:  
    Base() { // base constructor }  
    ~Base() { // base destructor }  
};  
class Derived : public Base {  
public:  
    Derived() { // derived constructor }  
    ~Derived() { // derived destructor }  
};
```

Program:

```
#include <iostream>  
  
using namespace std;  
  
class Base  
{  
public:  
    Base()  
};
```

Date:

Roll No.:

2	4	B	1	1	A	I	3	9	9
---	---	---	---	---	---	---	---	---	---

```
{
cout << "Base class constructor called" << endl;
}};

class Derived : public Base
{
public:
    Derived()
    {
        cout << "Derived class constructor called" << endl;
    }
};

class MultiDerived : public Derived
{
public:
    MultiDerived()
    {
        cout << "MultiDerived class constructor called" << endl;
    }
};

int main()
{
    MultiDerived obj;

    cout << "\n\nName:S.jaswanth\n24B11AI399" ;

    return 0;
}
```



A D I T Y A
U N I V E R S I T Y

Date:

Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

Output:

```
C:\Users\Jaswa\OneDrive\Doc × + v
Base class constructor called
Derived class constructor called
MultiDerived class constructor called

Name:S.jaswanth
24B11AI399
-----
```

CO Mapped: CO3

POs Mapped: PO1, PO2, PO3, PO4, PO5, PO11

PSOs Mapped: PSO1



A D I T Y A
U N I V E R S I T Y

Date:

Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

Program:8a

Develop a C++ program that demonstrates how to use pointers to access and manipulate objects of a class.

Aim:

To demonstrate the use of pointers to access and manipulate objects of a class.

Description:

The program creates an object dynamically using a pointer to a class Person. It uses the pointer to call member functions and modify data members. It also highlights the importance of freeing allocated memory to prevent memory leaks.

Syntax:

```
// Syntax for creating and using a pointer to a class object
ClassName * ptr = new ClassName(constructor_arguments) ;
// dynamically create object
ptr->memberFunction();
// access members using pointer and -> operator
ptr->setData(value);
// modify object data using pointer
delete ptr;
// free dynamically allocated memory
```

Program code:

```
#include <iostream>
using namespace std;
class Person {
    string name;
    int age;
public:
    Person(string n, int a) : name(n), age(a) {
        cout << "Person constructor called for " << name << endl;
    }
    void display() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
    void setAge(int a) {
```

Date:

Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

```
        age = a;
    }
    ~Person() {
        cout << "Person destructor called for " << name << endl;
    }
};

int main() {

    // Creating object dynamically using pointer

    Person* p = new Person("Nani", 30);

    // Accessing members via pointer

p->display();
    // Modifying object through pointer
    p->setAge(31);
    p->display();
    // Deleting dynamically allocated object
    delete p;

    cout << "Name:S.jaswanth" << endl;
    cout << "Roll NO: 24B11AI399" << endl;
    return 0;

}
```

Date:

Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

output:

```
C:\Users\Jaswa\OneDrive\Doc X + v
Person constructor called for Nani
Name: Nani, Age: 30
Name: Nani, Age: 31
Person destructor called for Nani
Name:S.jaswanth
Roll NO: 24B11AI399
```

CO Mapped: CO1

POs Mapped: PO1, PO2, PO3, PO4, PO5, PO9, PO11

PSOs Mapped: PSO1



A D I T Y A
U N I V E R S I T Y

Date:

Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

Program:8b

Develop a C++ program to demonstrate the concept of virtual base classes in the context of multiple inheritance, which resolves ambiguity in the inheritance hierarchy

AIM :

To demonstrate the concept of virtual base classes in multiple inheritance and how they resolve the diamond problem in C++.

Description:

This program demonstrates the diamond problem that arises in multiple inheritance when two classes inherit from the same base class. It shows how declaring the base class as a virtual base class ensures only one copy of the base class is shared among derived classes. The program also illustrates the order in which constructors and destructors are called in this inheritance hierarchy.

Syntax:

```
// Syntax for virtual base class inheritance
class Base {
    // Base class members
};
class Derived1 : virtual public Base {
    // Derived class 1 members
};
class Derived2 : virtual public Base {
    // Derived class 2 members
};
class FinalDerived : public Derived1, public Derived2 {
    // Final derived class members
};
int main() {
    FinalDerived obj;
    // Creating object invokes constructors in order

    return 0;
}
```

Program:

```
#include <iostream>
```

Date:

Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

```
using namespace std;

class A {
public:
    A() { cout << "Constructor A called" << endl; }
    virtual ~A() { cout << "Destructor A called" << endl; }
};

class B : virtual public A {
public:
    B() { cout << "Constructor B called" << endl; }
    ~B() { cout << "Destructor B called" << endl; }
};

class C : virtual public A {
public:
    C() { cout << "Constructor C called" << endl; }
    ~C() { cout << "Destructor C called" << endl; }
};

class D : public B, public C {
public:
    D() { cout << "Constructor D called" << endl; }
    ~D() { cout << "Destructor D called" << endl; }
};

int main() {
    D obj;
    return 0;
}
```

Date:

Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

Output:

```
Output
Constructor A called
Constructor B called
Constructor C called
Constructor D called
Destructor D called
Destructor C called
Destructor B called
Destructor A called

=== Code Execution Successful ===
```

CO Mapped: CO2

POs Mapped: PO1, PO2, PO3, PO4, PO6, PO9, PO11

PSOs Mapped: PSO1



A D I T Y A
U N I V E R S I T Y

Date:

Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

Program No: 11(a)

Aim:

Develop a C++ program that demonstrates exception handling using try, throw, and catch blocks.

Description:

This program illustrates how exceptions are handled in C++ using try, throw, and catch.

When an exceptional condition occurs, the program throws an exception, which is then caught and handled to prevent abnormal termination.

Syntax:

```
try {  
    // code that may throw  
}  
  
catch (exceptionType e) {  
    // handle exception  
}
```

Program:

```
#include <iostream>  
  
using namespace std;  
  
int divide(int a, int b) {  
    if (b == 0) {  
        throw "Division by zero error!";  
    }  
    return a / b;  
}  
  
int main() {  
    int x = 10, y = 0;  
    try {  
        int result = divide(x, y);  
    }
```



A D I T Y A
U N I V E R S I T Y

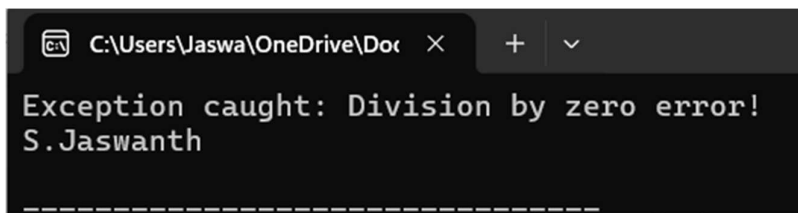
Date:

Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

```
cout << "Result: " << result << endl;
} catch (const char* msg) {
    cout << "Exception caught: " << msg << endl;
}
cout << "S.Jaswanth" << endl;
return 0;
}
```

Sample output:



CO Mapped: CO4

POs Mapped: PO1 , PO2 , PO3 , PO4 , PO5 , PO9 , PO11

PSOs Mapped: PSO1

A D I T Y A
U N I V E R S I T Y

Date:

Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

Program No: 11(b)

Aim:

Develop a C++ program to illustrate the use of multiple catch statements, where different types of exceptions are caught and handled differently.

Description:

This program demonstrates multiple catch blocks to handle various exception types. Each catch block can handle a different exception, allowing fine-grained control over error handling.

Syntax:

```
try {  
    // code  
}  
catch(Type1 e) { }  
catch(Type2 e) { }
```

program:

```
#include <iostream>  
using namespace std;  
  
int main() {  
    try {  
        int choice;  
        cout << "Enter 1 for int exception, 2 for double exception: ";  
        cin >> choice;  
  
        if (choice == 1) throw 100;  
        else if (choice == 2) throw 3.14;  
        else throw "Unknown exception";  
    }  
    catch (int i) {
```



A D I T Y A
U N I V E R S I T Y

Date:

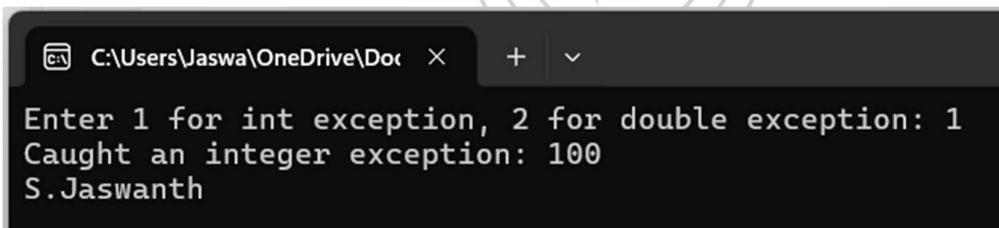
Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

```
    cout << "Caught an integer exception: " << i << endl;
}
catch (double d) {
    cout << "Caught a double exception: " << d << endl;
}
catch (const char* msg) {
    cout << "Caught a string exception: " << msg << endl;
}

cout << "S.Jaswanth" << endl;
return 0;
}
```

Sample output:

A screenshot of a terminal window with a dark background. The title bar shows the file path 'C:\Users\Jaswa\OneDrive\Doc' and standard window controls. The terminal text shows the program's execution: a prompt 'Enter 1 for int exception, 2 for double exception:' followed by the user input '1', the output 'Caught an integer exception: 100', and the name 'S.Jaswanth'.

CO Mapped: CO4

POs Mapped: PO1 , PO2 , PO3 , PO4 , PO5 , PO9 , PO11

PSOs Mapped: PSO1

Date:

Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

Program No: 12(a)

Aim:

Develop a C++ program to implement List and Vector containers and perform basic operations such as insertion, deletion, traversal

Description:

This program demonstrates the use of STL containers list and vector, performing operations like inserting elements, deleting elements, and traversing through the container.

Program:

```
#include <iostream>
```

```
#include <vector>
```

```
#include <list>
```

```
using namespace std;
```

```
int main() {
```

```
    // Vector operations
```

```
    vector<int> v = {10, 20, 30};
```

```
    v.push_back(40);
```

```
    cout << "Vector elements: ";
```

```
    for (int i : v) cout << i << " ";
```

```
    cout << endl;
```

```
    v.pop_back();
```

```
    cout << "Vector after pop_back: ";
```

```
    for (int i : v) cout << i << " ";
```

```
    cout << endl;
```



A D I T Y A
U N I V E R S I T Y

Date:

Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

```
// List operations
list<int> l = {1, 2, 3};
l.push_back(4);
l.push_front(0);
cout << "List elements: ";
for (int i : l) cout << i << " ";
cout << endl;

l.pop_back();
l.pop_front();
cout << "List after deletion: ";
for (int i : l) cout << i << " ";
cout << endl;

cout << "S.Jaswanth" << endl;
return 0;
}
```



A D I T Y A
UNIVERSITY

Sample Output:

```
C:\Users\Jaswa\OneDrive\Doc × + v
Vector elements: 10 20 30 40
Vector after pop_back: 10 20 30
List elements: 0 1 2 3 4
List after deletion: 1 2 3
S.Jaswanth
```

CO Mapped: CO4

POs Mapped: PO1 , PO2 , PO3 , PO4 , PO5 , PO9 , PO11

PSOs Mapped: PSO1

Date:

Roll No.:

2	4	B	1	1	A	I	3	9	9
---	---	---	---	---	---	---	---	---	---

Program No: 12(b)

Aim:

Implement Deque in C++ and demonstrate basic operations.

Program:

```
#include <iostream>
```

```
#include <deque>
```

```
using namespace std;
```

```
int main() {
```

```
    deque<int> dq;
```

```
    dq.push_back(10);
```

```
    dq.push_front(5);
```

```
    dq.push_back(20);
```

```
    cout << "Deque elements: ";
```

```
    for (int i : dq) cout << i << " ";
```

```
    cout << endl;
```

```
    dq.pop_front();
```

```
    dq.pop_back();
```

```
    cout << "Deque after deletions: ";
```

```
    for (int i : dq) cout << i << " ";
```

```
    cout << endl;
```

```
    cout << "S.Jaswanth" << endl;
```

```
    return 0;}
```



A D I T Y A
U N I V E R S I T Y

Date:

Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

Sample Output:

```
C:\Users\Jaswa\OneDrive\Doc × + v
Deque elements: 5 10 20
Deque after deletions: 10
S.Jaswanth
```

CO Mapped: CO4

POs Mapped: PO1 , PO2 , PO3 , PO4 , PO5 , PO9 , PO11

PSOs Mapped: PSO1



A D I T Y A
U N I V E R S I T Y

Date:

Roll No.:

2	4	B	1	1	A	I	3	9	9
---	---	---	---	---	---	---	---	---	---

Program No: 12(c)

Aim:

Implement Map and demonstrate operations such as insertion, deletion, access, and searching.

Description:

This program demonstrates the use of the map container in C++ STL.

It shows how to:

- Insert key–value pairs using assignment and the insert() method.
 - Delete entries from the map using erase().
 - Access values through their keys using the subscript operator [].
 - Search for a specific key using find().
- The map stores unique keys in sorted order, and each key is automatically associated with its corresponding value.

Program:

```
#include <iostream>

#include <map>

using namespace std;
```

```
int main() {
    map<int, string> m;

    m[1] = "Nithin";
    m[2] = "Sarvan";
    m.insert({3, "Akhil"});

    cout << "Map elements:" << endl;
    for (auto &p : m) {
        cout << p.first << " -> " << p.second << endl;
    }

    cout << "Access key 2: " << m[2] << endl;
```



A D I T Y A
U N I V E R S I T Y

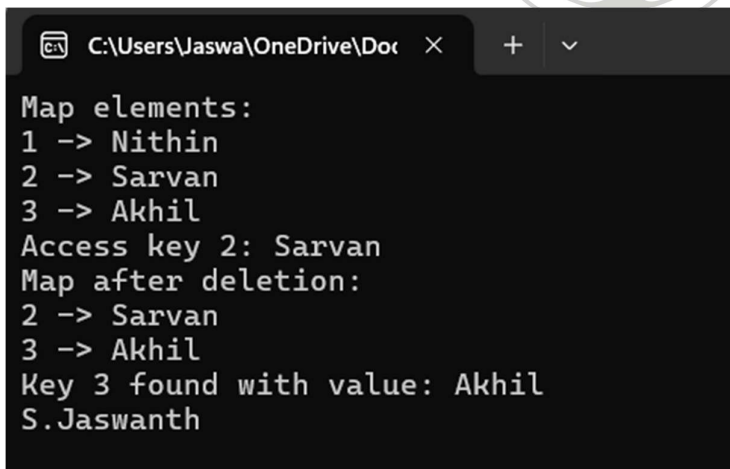
Date:

Roll No.:

2	4	B	1	1	A	1	3	9	9
---	---	---	---	---	---	---	---	---	---

```
m.erase(1);  
  
cout << "Map after deletion:" << endl;  
  
for (auto &p : m) {  
    cout << p.first << " -> " << p.second << endl;  
}  
  
int key = 3;  
  
if (m.find(key) != m.end())  
    cout << "Key " << key << " found with value: " << m[key] << endl;  
  
cout << "S.Jaswanth" << endl;  
  
return 0;  
}
```

Sample Output:



```
Map elements:  
1 -> Nithin  
2 -> Sarvan  
3 -> Akhil  
Access key 2: Sarvan  
Map after deletion:  
2 -> Sarvan  
3 -> Akhil  
Key 3 found with value: Akhil  
S.Jaswanth
```

CO Mapped: CO4

POs Mapped: PO1 , PO2 , PO3 , PO4 , PO5 , PO9 , PO11

PSOs Mapped: PSO1