MidTerm Solution			
Subject	**Object-Oriented Programming language using C++**	**Code**	**CIC-211**
Community Name	**SnapED codeCampus**		

Ans-1(a): When the inline function is called, the whole code of the inline function gets inserted or substituted at the point of the inline function call. This substitution is performed by the C++ compiler at compile time. An inline function may increase efficiency if it is small.

Syntax:

```
inline return-type function-name(parameters)
{
 // function code
}
```

Example:

```
#include <iostream>
using namespace std;
inline int cube(int s) { return s * s * s; }
int main()
{
   cout << "The cube of 3 is: " << cube(3) << "\n";
   return 0;
}
```

Output:

```
The cube of 3 is: 27
```

Ans-1(b): Array of Objects: An array of objects is a collection of objects of the same type stored in contiguous memory locations.

```
#include <iostream>

using namespace

#include <string>


class Person {

public:

    std::Parneeg name;

    int age;


    // Constructor

    Person(std::string n, int a) : name(n), age(a) {}

};


int main() {

    // Creating an array of objects

    const int arraySize = 3;

    Person people[arraySize] = {{"Alice", 25}, {"Bob", 30}, {"Charlie", 22}};


    // Accessing objects in the array

    std::cout << "People in the array:\n";

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

        std::cout << "Name: " << people[i].name << ", Age: " << people[i].age << "\n";

    } return 0;

}
```

```
#include <iostream>
using namespace std;
#include <string>

class Car {
public:
    string brand;
    int year;
    Car(std::string b, int y) : brand(b), year(y) {}
};

int main() {

    Car* myCar = new Car("Toyota", 2022);



     cout << "My car details: " << myCar->brand << " " << myCar->year << "\n";


    delete myCar;


    return 0;
}
```

Ans-1(b)Pointers to Objects:

Pointers to objects hold the memory address of an object, allowing for dynamic memory allocation and indirect access to object members.

Example:

```
#include <iostream>

using namespace std;

#include <string>

class Car {

public:

    std::string brand;

    int year;

    Car(std::string b, int y) : brand(b), year(y) {}

};

int main() {

    Car* myCar = new Car("Toyota", 2022);

    std::cout << "My car details: " << myCar->brand << " " << myCar->year << "\n";

    delete myCar;

    return 0;

}
```

Ans-1(c) this Pointer:

In C++, a "this" pointer is a special pointer that is automatically created within member functions of a class. It points to the object for which the member function is called. The "this" pointer is used to access the members of the calling object.

Ans-1(d):Data Abstraction:

Definition: Hides complex implementation details, exposing only essential functionalities.

Importance:

-Simplifies complexity for users.

Promotes modularity in design.

Encapsulation:

Definition: Bundles data and methods into a single unit (class), controlling access to data.

Importance:

Protects data integrity.

Organizes code, improving readability and maintainability.

Real-time Example:

Car: Abstraction allows users to interact with simplified driving functionalities. Encapsulation in a car class bundles related data and methods, protecting internal components.

Ans-2(a):

```
#include <iostream>

using namespace std;

class Singleton {

private:

  Singleton() {

    cout << "Object created" << endl;

  }


public:

  static Singleton& getInstance() {

    static Singleton instance;

    return instance;

  }
```

```
    void doSomething() {
        cout << "Doing something..." << endl;
    }
};

int main() {
    // Create the first object
    Singleton& obj1 = Singleton::getInstance();
    obj1.doSomething();

    // Attempt to create a second object (this should terminate the program)
    // Uncommenting the next line would result in a compilation error.
    // Singleton& obj2 = Singleton::getInstance();

    return 0;
}
```

Ans-2(b):

The operators that cannot be overloaded are:

1. `.` (Member Access Operator)

2. `.*` (Member Access through Pointer to Member)

3. `::` (Scope Resolution Operator)

4. `?:` (Ternary Conditional Operator)

These operators cannot be overloaded because they are either involved in low-level operations related to memory management or are fundamental to the language's syntax and semantics.

```
#include <iostream>
using namespace std;
class Complex {
private:
    double real;
    double imag;
public:
    Complex(double r = 0, double i = 0) : real(r), imag(i) {}
    friend ostream& operator<<(ostream& os, const Complex& c);
    friend istream& operator>>(istream& is, Complex& c);
};
ostream& operator<<(ostream& os, const Complex& c) {
    os << "(" << c.real << " + " << c.imag << "i)";
    return os;
}
istream& operator>>(istream& is, Complex& c) {
    cout << "Enter real part: ";
    is >> c.real;
    cout << "Enter imaginary part: ";
    is >> c.imag;
    return is;
}
int main() {
    Complex c1, c2;
    cin >> c1 >> c2;
    cout << "Complex Number 1: " << c1 << endl;
    cout << "Complex Number 2: " << c2 << endl;
    return 0;
}
```

Ans-3(a):

```
#include <iostream>
using namespace std;
#include <vector>
class Stack {
private:
    vector<int> stackData;
public:
    Stack() {}
    Stack(const vector<int>& initialValues) : stackData(initialValues) {}
    void Push(int value) {
        stackData.push_back(value);
        cout << "Pushed: " << value << endl;
    }
    void Pop() {
        if (!stackData.empty()) {
            cout << "Popped: " << stackData.back() <<endl;
            stackData.pop_back();
        } else {
            cout << "Stack is empty. Cannot pop." <<endl;
        }
    }
    friend void displayStackContent(const Stack& stack);
};
void displayStackContent(const Stack& stack) {
    cout << "Stack Content: ";
    for (const auto& element : stack.stackData) {
        cout << element << " ";
    }
    cout << endl;
}
int main() {
    Stack myStack({1, 2, 3});
    displayStackContent(myStack);
    myStack.Push(4);
    myStack.Push(5);
    displayStackContent(myStack);
    myStack.Pop();
    myStack.Pop();
    displayStackContent(myStack);
    return 0;
}
```

Ans-3(b):

Compile Time Polymorphism	**Runtime Polymorphism**
In Compile time Polymorphism, the call is resolved by the compiler.	In Run time Polymorphism, the call is not resolved by the compiler.
It is also known as Static binding, Early binding and overloading as well.	It is also known as Dynamic binding, Late binding and overriding as well.
Method overloading is the compile-time polymorphism where more than one methods share the same name with different parameters or signature and different return type.	Method overriding is the runtime polymorphism having the same method with same parameters or signature but associated with compared, different classes.
It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers.
It provides fast execution because the method that needs to be executed is known early at the compile time.	It provides slow execution as compare to early binding because the method that needs to be executed is known at the runtime.

Ans-4(a):

1. Default Constructor:

Definition: A default constructor is a constructor that is automatically called when an object of a class is created. It has no parameters or has all parameters with default values.

Purpose: Initializes the object's attributes to default values.

Example:

```
#include <iostream>

using namespace std;

class MyClass {

public:

    // Default Constructor

    MyClass() {
        cout << "Default Constructor called" << endl;
    }
};

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

2. Parameterized Constructor:

Definition: A parameterized constructor is a constructor with parameters that allows you to initialize the object's attributes with specific values during object creation.

Purpose: Customized initialization based on provided values.

Example:

```
#include <iostream>
using namespace std;
class Point {
private:
    int x;
    int y;
public:
    // Parameterized Constructor
    Point(int xCoord, int yCoord) : x(xCoord), y(yCoord) {
        cout << "Parameterized Constructor called" << endl;
    }
    void display() {
        cout << "Point: (" << x << ", " << y << ")" << endl;
    }
};

int main() {
    Point p1(3, 5);
    p1.display();
    return 0;
}
```

3. Copy Constructor:

Definition: A copy constructor is a special constructor that creates an object by copying the attributes of an existing object of the same class.

Purpose: Used when an object is passed by value, returned by value, or explicitly copied.

Example:

```
#include <iostream>
using namespace std;
class Person {
private:
    string name;
    int age;

public:
    // Parameterized Constructor
    Person(const string& n, int a) : name(n), age(a) {}


    // Copy Constructor

    Person(const Person& other) : name(other.name), age(other.age) {

        cout << "Copy Constructor called" << endl;

    }


    void display() {

       cout << "Person: " << name << ", Age: " << age << endl;

    }

};


int main() {

    Person person1("John", 30);

    Person person2 = person1;

    person1.display();

    person2.display();

    return 0;

}
```

Ans-4(b):

Dynamic Memory Allocation:

Dynamic memory allocation refers to the process of allocating memory during the program's execution, rather than at compile time. In C++, the `new` and `delete` operators are used for dynamic memory allocation and deallocation, respectively.

C++ Program Illustrating `new` and `delete` Operators:

```
#include <iostream>
using namespace std;
int main() {
    int* dynamicInt = new int;
    *dynamicInt = 42;
    cout << "Dynamically allocated integer: " << *dynamicInt << endl;
    int* dynamicArray = new int[5];
    for (int i = 0; i < 5; ++i) {
        dynamicArray[i] = i * 10;
    }
    cout << "Dynamically allocated array: ";
    for (int i = 0; i < 5; ++i) {
        cout << dynamicArray[i] << " ";
    }
    cout << endl;
    delete dynamicInt;
    delete[] dynamicArray;
    return 0;
}
```

1. Type Safety:

- `new` and `delete` operators in C++ are type-safe, meaning they automatically calculate the size of the object or array based on the type, eliminating the need for manual size calculations.
- `malloc()` returns a `void*` and requires an explicit cast, leading to potential type-related errors.

2. Constructor Invocation:

- `new` invokes the constructor of the object being created, ensuring proper initialization.

- `malloc()` only allocates memory and does not invoke constructors, which can lead to uninitialized data.

3. Operator Overloading:

- `new` and `delete` can be overloaded in C++ for user-defined types, providing more flexibility.

- `malloc()` is a function and cannot be overloaded, limiting its customization.

4. Easier Syntax:

- `new` and `delete` have a more straightforward syntax, especially when dealing with objects and arrays.

- `malloc()` requires manual size calculations and casting, leading to less readable code.