**Object Oriented Programming Lab 04**

|  |  |
| --- | --- |
| **Course**: Object Oriented Programming (CL1004) | **Semester**: Spring 2025 |
| **Instructor**: Muhammad Monis |  |
| Note:   * Maintain discipline during the lab. * Listen and follow the instructions as they are given. * Just raise hand if you have any problem. * Completing all tasks of each lab is compulsory. * Get your lab checked at the end of the session. |  |

**Class**:

A class is a user-defined data type. It 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.

**Types of Classes:**

* Concrete Classes
* Generalized classes
* Specialized classes

**Concrete Classes:**

A concrete class is a class that has an implementation for all of its methods. They cannot have any unimplemented methods.

**Example:**

class Concrete {

private:

    string info;

public:

    Concrete(string s) : info(s) { }

    void printContent() {

        cout << "Concrete Object Information\n" << info << endl;

    }};

int main()

{

string s;

s = "Object Creation";

    Concrete c(s);

    c.printContent();

}

**Generalized Classes:**

A class which tells the main features but not the specific details. The classes situated at the top of the inheritance hierarchy can be said as General.

**Example**:

 "Car" can be considered generalized class.

#include <iostream>

using namespace std;

class Car {

    public:

        int price;

        int year;

        string make;

        string model;

        Car(int p, int y, string m, string mo) {

            price = p;

            year = y;

            make = m;

            model = mo;

        }

        void displayInformation() {

            cout << "Price: " << price << endl;

            cout << "Year: " << year << endl;

            cout << "Make: " << make << endl;

            cout << "Model: " << model << endl;

        }

};

**Specialized Classes:**

 A class which is very particular and states the specific details. The classes situated at the bottom of the inheritance hierarchy can be said as Specific.

**Example**:

In the code provided, the class "ToyotaCars" is an example of a specialized class. This class represents a specific type of car, namely Toyota cars, and provides specific information about them, such as the model, year, and color. The class is defined with private variables to store this information, and public methods to display it.

#include<iostream>

using namespace std;

class ToyotaCars {

    private:

        string model;

        int year;

        string color;

    public:

        ToyotaCars(string model, int year, string color)

        {

            this->model = model;

            this->year = year;

            this->color = color;

        }

        void displayInfo()

        {

            cout<<"Model: "<<model<<endl;

            cout<<"Year: "<<year<<endl;

            cout<<"Color: "<<color<<endl;

        }

};

**Constructor**

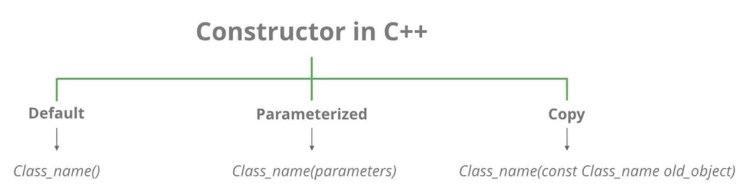
Constructor is the special type of member function in C++ classes. It is automatically invoked when an object is being created. It is special because its name is same as the class name. A constructor can be used for the following functions:

* **To initialize data member of class:** In the constructor member function (which the programmer will declare), we can initialize the default vales to the data members and they can be used further for processing.
* **To allocate memory for data member:** Constructor is also used to declare run time memory (dynamic memory for the data members).

Constructor has the following properties:

* Constructor has the same name as the class name.
* The name is case sensitive.
* Constructor does not have return type.
* We can overload constructor; it means we can create more than one constructor of class (we will discuss this in later labs).
* It must be public type (declared inside the public access modifier in the class).

**Types of Constructor**

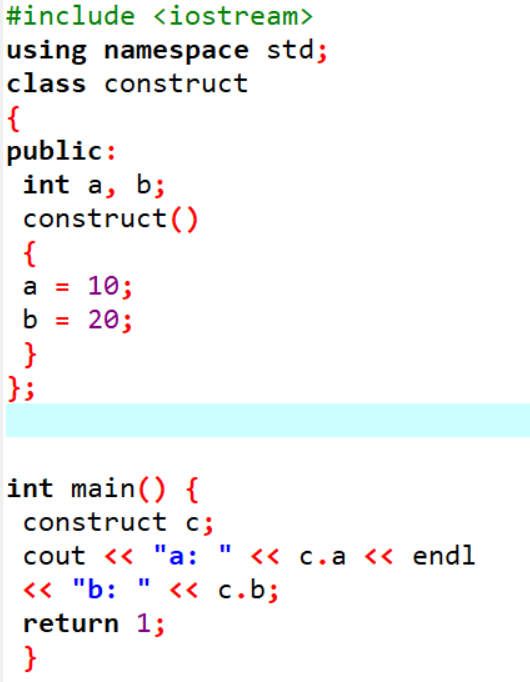


Let us understand the types of constructors in C++ by taking a real-world example. Suppose you went to a shop to buy a marker. When you want to buy a marker, what are the options. The first one you go to a shop and say give me a marker. So just saying give me a marker mean that you did not set which brand name and which color, you didn’t mention anything just say you want a marker. So, when we said just I want a marker so whatever the frequently sold marker is there in the market or in his shop he will simply hand over that. And this is what a default constructor is!

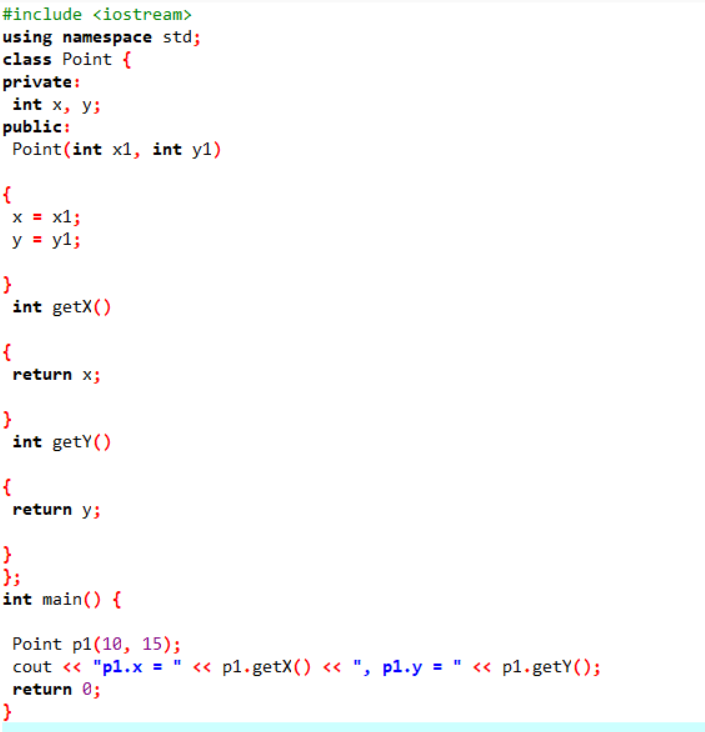
The second method is you go to a shop and say I want a marker a red in color and XYZ brand. So, you are mentioning this and he will give you that marker. So, in this case you have given the parameters and this is what a parameterized constructor is! Then the third one you go to a shop and say I want a marker like this (a physical marker on your hand). So, the shopkeeper will see that marker. Okay, and he will give a new marker for you. So copy of that marker. And that’s what a copy constructor is!

* **Default Constructors:** Default constructor is the constructor, which does not take any argument. It has no parameters.
* **Null constructors:** Null constructors in C++ are a special type of constructor that does nothing. The compiler knows that there is no code to execute, so it will not generate any executable code for the constructor.
* **Parameterized Constructors:** It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor’s body, use the parameters to initialize the object.
* **Copy Constructor:** A copy constructor is a member function, which initializes an object using another object of the same class. The copy constructor in C++ is used to copy data of one object to another.

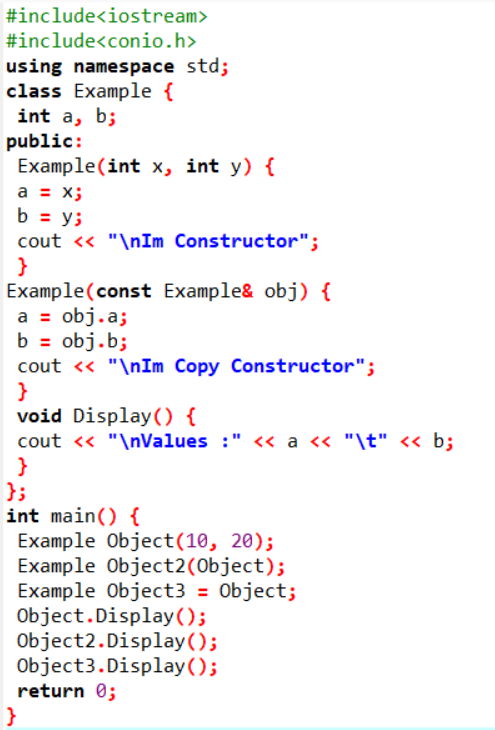
**Example: Default Constructor**



**Example: Parameterized Constructor**



**Example: Copy Constructor**



Constructor Overloading:

In C++, We can have more than one constructor in a class with same name, as long as each has a different list of arguments. This concept is known as Constructor Overloading. Overloaded constructors have the same name (name of the class) but the different number of arguments. Depending upon the number and type of arguments passed, the corresponding constructor is called

#include <iostream>

using namespace std;

class Student

{

    private:

        string name;

        int age;

        string address;

        string department;

    public:

        // Default constructor

        Student()

        {

            name = "";

            age = 0;

            address = "";

            department = "";

        }

        // Overloaded constructor with name and age parameters

        Student(string studentName, int studentAge)

        {

            name = studentName;

            age = studentAge;

            address = "";

            department = "";

        }

        // Overloaded constructor with name, age, and address parameters

        Student(string studentName, int studentAge, string studentAddress)

        {

            name = studentName;

            age = studentAge;

            address = studentAddress;

            department = "";

        }

        // Overloaded constructor with all parameters

        Student(string studentName, int studentAge, string studentAddress, string studentDepartment)

        {

            name = studentName;

            age = studentAge;

            address = studentAddress;

            department = studentDepartment;

        }

        // Accessor functions to access private data members

        string getName()

        {

            return name;

        }

        int getAge()

        {

            return age;

        }

        string getAddress()

        {

            return address;

        }

        string getDepartment()

        {

            return department;

        }

};

//Driver code

int main()

{

    // Creating objects using different constructors

    Student student1;

    Student student2("Ali Hasan", 20);

    Student student3("Junaid Khan", 22, "Gulshan, Khi");

    Student student4("Ayesha Usman", 23, "Saddar, Khi", "Computer Science");

    // Printing details of each student

    cout << "Student 1 Details:" << endl;

    cout << "Name: " << student1.getName() << endl;

    cout << "Age: " << student1.getAge() << endl;

    cout << "Address: " << student1.getAddress() << endl;

    cout << "Department: " << student1.getDepartment() << endl;

    cout << endl;

    cout << "Student 2 Details:" << endl;

    cout << "Name: " << student2.getName() << endl;

    cout << "Age: " << student2.getAge() << endl;

    cout << "Address: " << student2.getAddress() << endl;

    cout << "Department: " << student2.getDepartment() << endl;

    cout << endl;

    cout << "Student 3 Details:" << endl;

    cout << "Name: " << student3.getName() << endl;

    cout << "Age: " << student3.getAge() << endl;

    cout << "Address: " << student3.getAddress() << endl;

    cout << "Department: " << student3.getDepartment() << endl;

    cout << endl;

    cout << "Student 4 Details:" << endl;

    cout << "Name: " << student4.getName() << endl;

    cout << "Age: " << student4.getAge() << endl;

    cout << "Address: " << student4.getAddress() << endl;

    cout << "Department: " << student4.getDepartment() << endl;

    cout << endl;

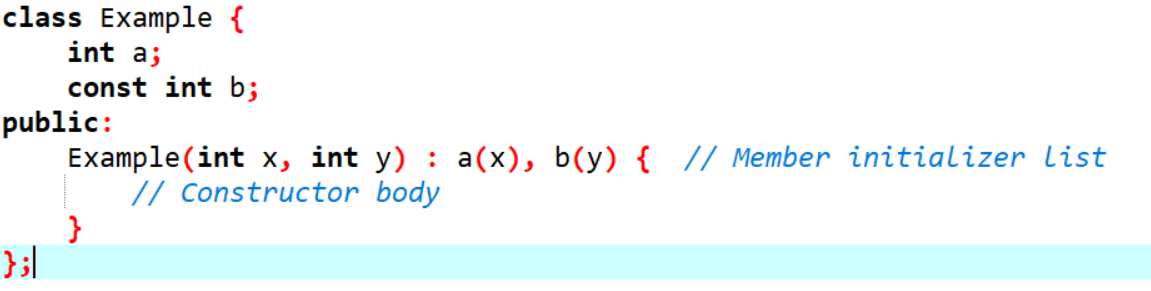
}

**Member Initializer List**

In C++, a member initializer list is used to initialize class members directly when an object is created. It is written after the constructor’s parameter list and before the constructor’s body. This approach is especially useful for constant variables, reference variables, and base class initialization. Instead of assigning values inside the constructor body, we can directly initialize the variables in the initializer list. It is written after the constructor’s parameter list and before the constructor body.

This method is very useful for initializing constant variables, reference variables, and base class members because they must be set at the time of object creation. It also helps in making the code faster and more efficient by avoiding extra assignments inside the constructor.

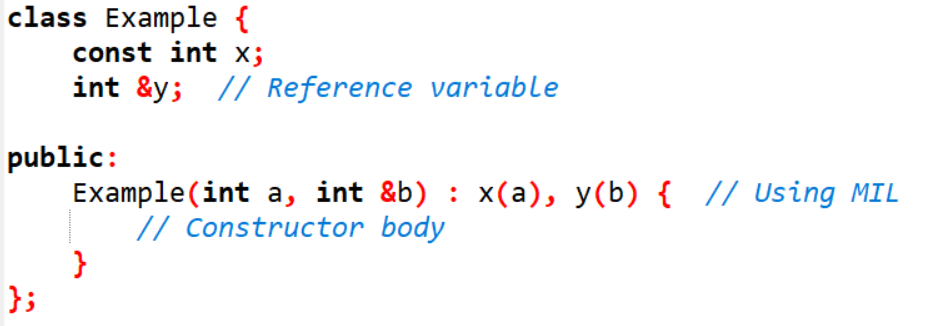
Consider the example below:



Here, **a** and **b** are initialized using the member initializer list before entering the constructor body.

**Advantages**

* Improves performance by avoiding extra assignments.
* Required for const and reference members since they must be initialized at declaration.

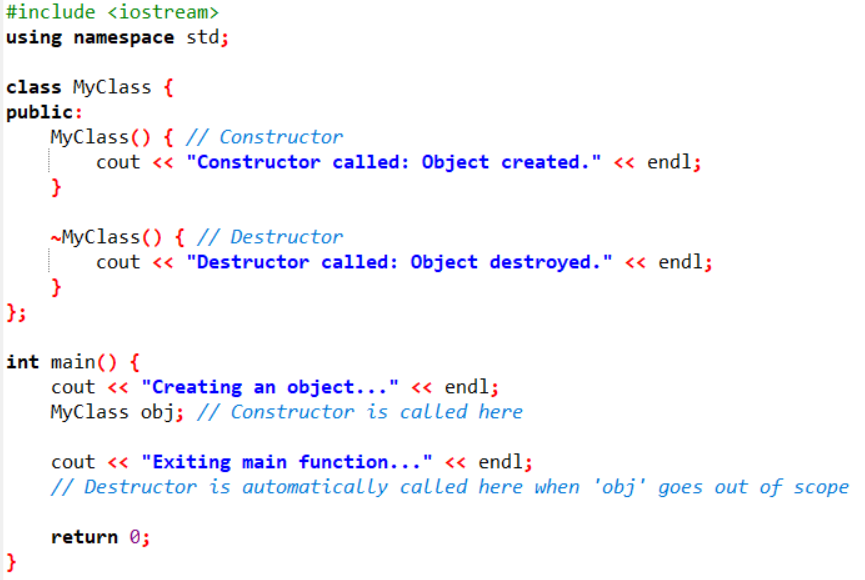


* Ensures base class constructors are called properly in inheritance (will be discussed later in future labs).

**Destructor**

A destructor is also a special member function as a constructor. Destructor destroys the class objects created by the constructor. Destructor has the same name as their class name preceded by a tilde (~) symbol. It is not possible to define more than one destructor. The destructor is only one way to destroy the object created by the constructor. Hence destructor can-not be overloaded. Destructor neither requires any argument nor returns any value. It is automatically called when the object goes out of scope. Destructors release memory space occupied by the objects created by the constructor. In destructor, objects are destroyed in the reverse of object creation.

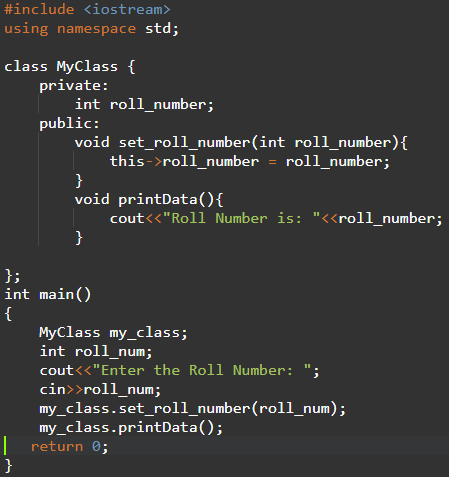
**Example: Destructor**



**This Keyword**

This keyword is a pointer to the current object. It is used to refer to the member variables and member functions of the current object. It can be particularly useful when there is a naming conflict between a member variable and a local variable or parameter with the same name in a member function. By using this, you can disambiguate between the two and access the member variable specifically.

Consider the below example:



In this example, the MyClass class has a private member variable roll\_number. The set\_ roll\_number member function takes an integer parameter roll\_number and sets the value of the member variable to it using the this keyword. The printData member function simply prints the value of the member variable to the console, again using the this keyword to refer to the member variable.

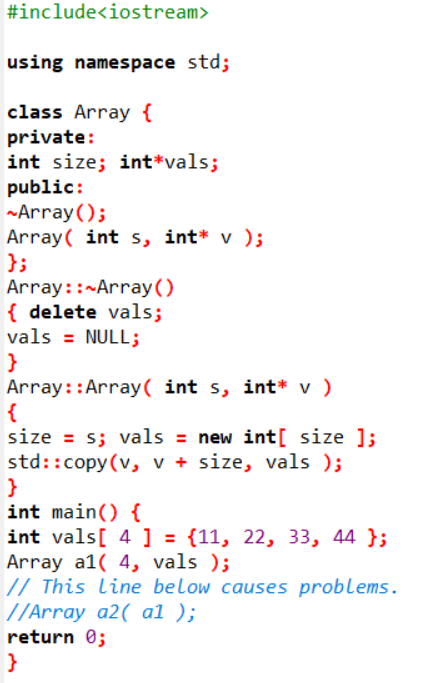
**Let`s Explore:**

1. What will happen if we will not use “this pointer” in above example?
2. Which value will be assigned to member variable roll\_number? And why?
3. Which value will be assigned to local variable roll\_number? And why?

**Rule of Three**

If you need to explicitly declare either the destructor, copy constructor or copy assignment operator yourself, you probably need to explicitly declare all three of them. The default constructors and assignment operators do shallow copy and we create our own constructor and assignment operators when we need to perform a deep copy (For example when a class contains pointers pointing to dynamically allocated resources).

First, what does a destructor do? It contains code that runs whenever an object is destroyed. Only affecting the contents of the object would be useless. An object in the process of being destroyed cannot have any changes made to it. Therefore, the destructor affects the program’s state as a whole. Now, suppose our class does not have a copy constructor. Copying an object will copy all of its data members to the target object. In this case when the object is destroyed the destructor runs twice. Also, the destructor has the same information for each object being destroyed. In the absence of an appropriately defined copy constructor, the destructor is executed twice when it should only execute once. This duplicate execution is a source for trouble.



**The Rule of Five in C++**

The **Rule of Five** in C++ is an extension of the **Rule of Three**, introduced in C++11 to cover resource management for classes that utilize dynamic memory, file handles, sockets, or other system resources. It recognizes that if a class needs a custom destructor, copy, or assignment operations, it might also need custom move semantics to handle resources efficiently.

The **five special member functions** are:

1. **Destructor**  
   Cleans up resources (e.g., dynamic memory) when an object goes out of scope.

**~MyClass() {**

**delete ptr;**

**}**

1. **Copy Constructor**  
   Initializes a new object as a copy of an existing object.

**MyClass(const MyClass& other) {**

**ptr = new int(\*other.ptr);**

**}**

1. **Copy Assignment Operator**  
   Handles the assignment of one object to another.

**MyClass& operator=(const MyClass& other) {**

**if (this != &other) {**

**delete ptr;**

**ptr = new int(\*other.ptr);**

**}**

**return \*this;**

**}**

1. **Move Constructor**  
   Transfers ownership of resources from a temporary (rvalue) to a new object, avoiding deep copies.

**MyClass(MyClass&& other) noexcept {**

**ptr = other.ptr;**

**other.ptr = nullptr;**

**}**

1. **Move Assignment Operator**Transfers ownership of resources during assignment, again avoiding deep copies.

**MyClass& operator=(MyClass&& other) noexcept {**

**if (this != &other) {**

**delete ptr;**

**ptr = other.ptr;**

**other.ptr = nullptr;**

**}**

**return \*this;**

**}**

**Why is the Rule of Five Important?**

Without implementing these functions, your class may experience issues like **double deletion**, **memory leaks**, and **inefficient copies**. Properly defining move semantics enhances performance by reducing expensive deep copies in situations where temporary objects are used.

Example:

#include <iostream>

class MyClass {

private:

int\* ptr;

public:

// Constructor

MyClass(int value) : ptr(new int(value)) {}

// Destructor

~MyClass() { delete ptr; }

// Copy Constructor

MyClass(const MyClass& other) : ptr(new int(\*other.ptr)) {}

// Copy Assignment Operator

MyClass& operator=(const MyClass& other) {

if (this != &other) {

delete ptr;

ptr = new int(\*other.ptr);

}

return \*this;

}

// Move Constructor

MyClass(MyClass&& other) noexcept : ptr(other.ptr) {

other.ptr = nullptr;

}

// Move Assignment Operator

MyClass& operator=(MyClass&& other) noexcept {

if (this != &other) {

delete ptr;

ptr = other.ptr;

other.ptr = nullptr;

}

return \*this;

}

void show() const {

if (ptr)

std::cout << "Value: " << \*ptr << std::endl;

else

std::cout << "Pointer is null" << std::endl;

}

};

int main() {

MyClass obj1(10);

MyClass obj2 = obj1; // Copy constructor

MyClass obj3 = std::move(obj1); // Move constructor

obj2.show(); // Output: Value: 10

obj1.show(); // Output: Pointer is null

obj3.show(); // Output: Value: 10

return 0;

}

Exercise:

CW Task Must complete:

**Task - 01: Design a Matrix Class with Multiple Constructors**  
Create a class Matrix that represents a 2D matrix of **double** values. It must handle **row-major** dynamic allocation internally.

**Details**:

* + **Constructors**:
    1. **Default constructor**: Initializes a 0x0 matrix (rows = 0, cols = 0) or a nullptr-managed array.
    2. **Parameterized constructor (rows, cols)**: Allocates a 2D matrix of size rows x cols. Initialize elements to 0.
    3. **Copy constructor**: Deep copy.
    4. **Move constructor**: Transfers ownership of the matrix data.
    5. **Destructor**: Deallocates the dynamic memory.
  + **Member functions**:
    1. getRows() and getCols() to return the dimensions.
    2. at(int r, int c): returns a reference to the element at row r, column c.
    3. fill(double value): fills the entire matrix with the provided value.
    4. transpose(): returns a **new** Matrix object that is the transpose of the current matrix. Make sure to print it as well.

**Task - 02: Implement a Polynomial Class**

1. **Overview**:  
   A Polynomial can be represented by its coefficients. For example,

a0​+a1​x+a2​x2+…+an​xn

Create a class Polynomial that stores these coefficients dynamically.

1. **Details**:
   * **Data members**:
     + A dynamically allocated array of double (for coefficients).
     + An int that represents the highest power (degree) of the polynomial.
   * **Constructors**:
     + **Default constructor**: Creates a polynomial of degree 0 with a coefficient array of size 1 (initialized to 0).
     + **Parameterized constructor**: Takes an int degree and an array of double for coefficients. Copies them dynamically.
     + **Copy constructor**: Deep copy of the coefficients.
     + **Move constructor**: Transfers ownership of the coefficient array.
     + **Destructor**: Cleans up the allocated memory.
   * **Member functions**:
     + int getDegree() const: returns the polynomial’s degree.
     + double evaluate(double x) const: computes the polynomial value at x.
     + Polynomial add(const Polynomial& other) const: returns a new Polynomial that is the sum of the current polynomial and other.
     + Polynomial multiply(const Polynomial& other) const: returns a new Polynomial that is the product of the two polynomials.

**Task - 03: Implement a DynamicArray Class with the “Rule of Five”**

1. **Overview**:  
   Create a class called DynamicArray that manages a **dynamic** array of integers.
   * The class should allocate and deallocate its own memory.
   * It must correctly handle copy and move operations.
2. **Details**:
   * Provide these **constructors and operators**:
     1. **Default constructor**: Initializes an empty array or an array of size zero.
     2. **Parameterized constructor**: Accept an integer n that indicates the array size, allocate an array of that size, and default-initialize all elements to 0.
     3. **Copy constructor**: Performs a deep copy of the array.
     4. **Move constructor**: Efficiently transfers ownership of the dynamic array from an rvalue object (leaving the source in a valid but empty state).
     5. **Copy assignment operator (operator=)**: Deep copies the array from one object to another, properly handling self-assignment.
     6. **Move assignment operator (operator=)**: Transfers ownership from an rvalue object, deallocating any previously held memory.
     7. **Destructor**: Deallocates the dynamic array.
   * Provide additional **member functions**:
     1. size(): returns the current size of the array.
     2. at(int index): returns a reference to the element at index (and possibly a const version for read-only).
     3. resize(int newSize): resizes the dynamic array to a new size, copying old data if newSize > 0.

H.W

Task - 01:

Your task is to create a Circle constructor that creates a circle with a radius provided by an argument. The circles constructed must have two getters getArea() (PIr^2) and getPerimeter() (2PI\*r) which give both respective areas and perimeter (circumference).

Task - 02:

Create an Account class that a bank might use to represent customers bank accounts. Include a data member to represent the account balance. Provide three member functions. Member function credit should add an amount to the current balance. Member function debits should withdraw money from the Account. Member function get Balance should return the current balance.

Task - 03:

Create A class called Invoice that a hardware store might use to represent an invoice for an item sold at the store. An Invoice should include four pieces of information as instance variables - a part number (type String), a part description (type String), a quantity of the item being purchased (type int) and a price per item (double). Your class should have a constructor that initialize the four instance variables. In addition, provide a method named getInvoiceAmount that calculates the invoice amount (i.e., multiples the quantity by the price per item), then returns the amount as a double value. If the quantity is not positive, it should be set to 0. If the price per item is not positive, it should be set to 0.0.

Task - 04:

A book shop maintains the inventory of books that are being sold at the shop. The list includes details such as author, title, price, publisher and stock position. Whenever a customer wants a book, the sales person inputs the title and author and the system searches the list and displays whether it is available or not. If it is not, an appropriate message is displayed. If it is, then the system displays the book details and requests for the number of copies required. If the requested copies are available, the total cost of the requested copies is displayed; otherwise “Required copies not in stock” is displayed. Design a system using a class called books with suitable member functions and constructors.

Task - 05:

Write a class called CoffeeShop, which has three instance variables:

**Name:** a string (basically, of the shop)  
**Menu:** an array of items (of type MenuItem), with each item containing the item (name of the item), type (whether a food or a drink) and price.  
**Orders:** an empty array and seven methods:  
**addOrder:** adds the name of the item to the end of the orders array if it exists on the menu. Otherwise, return “This item is currently unavailable”  
**fulfillOrder:** if the orders array is not empty, return “The {item} is ready” . If the orders array is empty, return “All orders have been fulfilled”  
**listOrders**: returns the list of orders taken, otherwise, an empty array.  
**dueAmount**: returns the total amount due for the orders taken.  
**cheapestItem**: returns the name of the cheapest item on the menu.  
**drinksOnly**: returns only the item names of type drink from the menu.  
foodOnly: returns only the item names of type food from the menu.