Assignment 1

1. Write a program to declare a pointer, initialise it with the address of a variable, and print the value using both the pointer and the variable. Demonstrate pointer assignment using two integer variables

Code:-

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

int main() {
    // Declare an integer variable
    int a = 10;

    // Declare a pointer and initialize it with the address of the variable
    int *ptr = &a;

    // Print the value of the variable and its address using the pointer
    cout << "Value of a using variable: " << a << endl;
    cout << "Value of a using pointer: " << *ptr << endl;
    cout << "Address of a: " << &a << endl;
    cout << "Address stored in pointer: " << ptr << endl;

    // Demonstrate pointer assignment with two variables
    int b = 20;
    int *ptr2 = &b;

    cout << "\nBefore assignment:\n";
    cout << "Value of a: " << a << ", Address of a: " << &a << endl;
    cout << "Value of b: " << b << ", Address of b: " << &b << endl;

    // Assign pointer ptr2 to ptr
    ptr = ptr2;

    cout << "\nAfter assignment:\n";
    cout << "Pointer ptr now points to b:\n";
    cout << "Value of b using pointer ptr: " << *ptr << endl;
    cout << "Address stored in ptr: " << ptr << endl;

    return 0;
}
```

Output:-

```
Value of a using variable: 10
Value of a using pointer: 10
Address of a: 0xa4725ff6cc
Address stored in pointer: 0xa4725ff6cc

Before assignment:
Value of a: 10, Address of a: 0xa4725ff6cc
Value of b: 20, Address of b: 0xa4725ff6c8

After assignment:
Pointer ptr now points to b:
Value of b using pointer ptr: 20
Address stored in ptr: 0xa4725ff6c8
```

2. Write a program that explains the concept of a wild pointer and how it can lead to undefined behaviour. Show how initialising a pointer can resolve this issue.

Code:-

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


int main() {
   // Wild Pointer Example
   int *wildPtr; // Uninitialized pointer, this is a wild pointer


   // Accessing or dereferencing a wild pointer can lead to undefined behavior
   // Uncommenting the next line will cause undefined behavior
   // cout << "Value of wild pointer: " << *wildPtr << endl;


   // Solution: Initialize the pointer
   wildPtr = nullptr; // Initialize to nullptr to avoid undefined behavior


   // Now wildPtr is safely initialized
   if (wildPtr == nullptr) {
      cout << "Pointer is safely initialized to nullptr." << endl;
   }
```

```
    // Allocate memory and assign to the pointer
    wildPtr = new int(42); // Dynamically allocate memory and assign value 42

    cout << "Value of dynamically allocated memory: " << *wildPtr << endl;
    cout << "Address stored in wildPtr: " << wildPtr << endl;

    // Clean up dynamically allocated memory
    delete wildPtr;
    wildPtr = nullptr; // Reset pointer after deletion

    cout << "Pointer reset to nullptr after memory deallocation." << endl;

    return 0;
}
```

Output:-

```
Pointer is safely initialized to nullptr.
Value of dynamically allocated memory: 42
Address stored in wildPtr: 0x1ae68f679b0
Pointer reset to nullptr after memory deallocation.
```

3. Create a program to demonstrate the use of NULL and its importance in pointer initialisation. Write code to check for NULL before dereferencing a pointer.

Code:-

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

int main() {
    // Demonstration of NULL in pointer initialization
    int *ptr = NULL; // Initialize pointer to NULL

    // Check if the pointer is NULL before dereferencing
    if (ptr == NULL) {
        cout << "Pointer is initialized to NULL. It is not pointing to any valid memory." << endl;
    } else {
        cout << "Value pointed by ptr: " << *ptr << endl;
    }

    // Allocate memory and assign to the pointer
    ptr = new int(100); // Dynamically allocate memory and assign value 100

    if (ptr != NULL) {
        cout << "Pointer is now pointing to valid memory." << endl;
        cout << "Value pointed by ptr: " << *ptr << endl;
    }

    // Clean up dynamically allocated memory
    delete ptr;
    ptr = NULL; // Reset pointer to NULL after deletion


    if (ptr == NULL) {

        cout << "Pointer has been reset to NULL after memory deallocation." << endl;

    }


    return 0;

}
```

Output:-

```
Pointer is initialized to NULL. It is not pointing to any valid memory.

Pointer is now pointing to valid memory.

Value pointed by ptr: 100

Pointer has been reset to NULL after memory deallocation.
```

4. Write code to show the behaviour of pointers with const qualifier in various scenarios:
   i. Pointer to a const value.
   ii. const pointer to a value.
   iii. const pointer to a const value.

Code:-

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

int main() {
    // i. Pointer to a const value
    const int constValue = 10;
    const int *ptrToConst = &constValue; // Pointer to a const value

    cout << "Pointer to a const value:" << endl;
    cout << "Value pointed by ptrToConst: " << *ptrToConst << endl;

    // *ptrToConst = 20; // Error: cannot modify the value through ptrToConst
    // ii. const pointer to a value
    int value = 30;
    int *const constPtr = &value; // const pointer to a value

    cout << "\nconst pointer to a value:" << endl;
    cout << "Value pointed by constPtr: " << *constPtr << endl;

    *constPtr = 40; // Allowed: can modify the value through constPtr
    cout << "Value after modification: " << *constPtr << endl;

    // constPtr = &constValue; // Error: cannot change the address stored in constPtr

    // iii. const pointer to a const value
    const int anotherConstValue = 50;
    const int *const constPtrToConst = &anotherConstValue; // const pointer to a const value

    cout << "\nconst pointer to a const value:" << endl;
    cout << "Value pointed by constPtrToConst: " << *constPtrToConst << endl;

    // *constPtrToConst = 60; // Error: cannot modify the value through constPtrToConst
    // constPtrToConst = &value; // Error: cannot change the address stored in constPtrToConst

    return 0;
}
```

Output:-

```
 Pointer to a const value:
 Value pointed by ptrToConst: 10

 const pointer to a value:
 Value pointed by constPtr: 30
 Value after modification: 40

 const pointer to a const value:
 Value pointed by constPtrToConst: 50
```

5. Write a program demonstrating the difference between const int *ptr, int *const ptr, and const int *const ptr.

   Code:-

```
#include <iostream>

int main() {
    int a = 10, b = 20;

    // 1. const int *ptr: Pointer to a constant integer
    const int *ptr1 = &a; // Pointer can point to different integers, but the value at the pointed location cannot be modified
    std::cout << "Value pointed by ptr1: " << *ptr1 << std::endl;
    // *ptr1 = 15; // Error: cannot modify the value through ptr1
    ptr1 = &b; // Valid: ptr1 can point to another integer
    std::cout << "After changing ptr1 to point to b, value: " << *ptr1 << std::endl;

    // 2. int *const ptr: Constant pointer to an integer
    int *const ptr2 = &a; // Pointer must always point to the same integer, but the value at the pointed location can be modified
    std::cout << "Value pointed by ptr2: " << *ptr2 << std::endl;
    *ptr2 = 15; // Valid: can modify the value at the location
    std::cout << "After modifying value via ptr2: " << *ptr2 << std::endl;
    // ptr2 = &b; // Error: cannot change the address stored in ptr2

    // 3. const int *const ptr: Constant pointer to a constant integer
    const int *const ptr3 = &a; // Pointer cannot change its address, and the value at the pointed location cannot be modified
    std::cout << "Value pointed by ptr3: " << *ptr3 << std::endl;
    // *ptr3 = 25; // Error: cannot modify the value through ptr3
    // ptr3 = &b; // Error: cannot change the address stored in ptr3

    return 0;
     }
```

   Output:-

```
Value pointed by ptr1: 10
After changing ptr1 to point to b, value: 20
Value pointed by ptr2: 10
After modifying value via ptr2: 15
Value pointed by ptr3: 15
```

6. Create a program that demonstrates how type-casting a const pointer can lead to unexpected behaviour.

Code:-

```
 #include <iostream>

 int main() {
     int a = 10;
     const int *ptr = &a; // Pointer to a constant integer

     std::cout << "Initial value pointed by ptr: " << *ptr << std::endl;

     // Type-casting const pointer to non-const pointer
     int *modifiablePtr = const_cast<int *>(ptr);

     // Modifying the value through the non-const pointer
     *modifiablePtr = 20; // Undefined behavior: modifying a const object

     std::cout << "Value after modification through modifiablePtr: " << *ptr << std::endl;
     std::cout << "Value of a directly: " << a << std::endl;

     // Demonstrating unexpected behavior
     const int b = 30;
     const int *ptrToConstB = &b;
     int *invalidModifiablePtr = const_cast<int *>(ptrToConstB);

     *invalidModifiablePtr = 40; // Dangerous: Undefined behavior

     std::cout << "Value of b after invalid modification: " << b << std::endl;
     std::cout << "Value pointed by ptrToConstB: " << *ptrToConstB << std::endl;

     return 0;
}
```

Output:-

```
 Initial value pointed by ptr: 10
 Value after modification through modifiablePtr: 20
 Value of a directly: 20
 Value of b after invalid modification: 30
 Value pointed by ptrToConstB: 40
```

7. Write a short program in both C and C++ that declares a structure, initializes it, and prints its members.

   Code:-

```
#include <iostream>
#include <cstdio> // For C-style I/O

// C++ Program
void cpp_structure_demo() {
   struct Person {
      std::string name;
      int age;
   } person = {"Harshal Bodhe", 30};

   std::cout << "C++ Structure Example:" << std::endl;
   std::cout << "Name: " << person.name << std::endl;
   std::cout << "Age: " << person.age << std::endl;
}

// C Program
void c_structure_demo() {
   struct Person {
      char name[50];
      int age;
   } person = {"Harshal bodhe", 25};

   printf("C Structure Example:\n");
   printf("Name: %s\n", person.name);
   printf("Age: %d\n", person.age);
}

int main() {
   cpp_structure_demo();
   c_structure_demo();
   return 0;
}
```

Output:-

```
C++ Structure Example:
Name: Harshal Bodhe
Age: 30
C Structure Example:
Name: Harshal bodhe
Age: 25
```

8. Create a struct in C++ and add member functions to initialize data members and display their values.
   Code:-

```
#include <iostream>
#include <cstdio> // For C-style I/O

// C++ Program with Member Functions
void cpp_structure_demo() {
   struct Person {
      std::string name;
      int age;

      // Member function to initialize data members
      void initialize(const std::string& personName, int personAge) {
         name = personName;
         age = personAge;
      }

      // Member function to display data members
      void display() const {
         std::cout << "Name: " << name << std::endl;
         std::cout << "Age: " << age << std::endl;
      }
   };

   Person person;
   person.initialize("Harshal Bodhe", 30);
   std::cout << "C++ Structure Example with Member Functions:" << std::endl;
   person.display();
}

// C Program
void c_structure_demo() {
   struct Person {
      char name[50];
      int age;
   } person = {"Harshal Bodhe", 25};

   printf("C Structure Example:\n");
   printf("Name: %s\n", person.name);
   printf("Age: %d\n", person.age);
}

int main() {
   cpp_structure_demo();
```

```
    c_structure_demo();
    return 0;
}
```

Output:-

```
C++ Structure Example with Member Functions:
Name: Harshal Bodhe
Age: 30
C Structure Example:
Name: Harshal Bodhe
Age: 25
```

9. Write a program to declare an array of structures to store information about 5 students (e.g., Name, Age, Marks). Allow the user to input details and print the list.

Code:-

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

// Define the structure for student information
struct Student {
    string name;
    int age;
    float marks;
};

int main() {
    // Declare an array of 5 students
    Student students[5];

    // Input student details
    for(int i = 0; i < 5; i++) {
        cout << "Enter details for student " << i + 1 << ":" << endl;

        cout << "Name: ";
        cin.ignore();  // To clear the input buffer before reading a string
        getline(cin, students[i].name);

        cout << "Age: ";
        cin >> students[i].age;

        cout << "Marks: ";
        cin >> students[i].marks;
        cout << endl;  // Add a newline for better formatting
    }

    // Print the details of all students
    cout << "\nStudent List:\n";
    for(int i = 0; i < 5; i++) {
        cout << "\nStudent " << i + 1 << " details:\n";
        cout << "Name: " << students[i].name << endl;
        cout << "Age: " << students[i].age << endl;
        cout << "Marks: " << students[i].marks << endl;
    }

    return 0;
}
```

Output:-

```
Enter details for student 1:
Name: Harshal
Age: 22
Marks: 85

Enter details for student 2:
Name: Raj
Age: 23
Marks: 75

Enter details for student 3:
Name: Kunal
Age: 25
Marks: 65

Enter details for student 4:
Name: Tanmay
Age: 22
Marks: 35

Enter details for student 5:
Name: Sakshi
Age: 23
Marks: 85


Student List:

Student 1 details:
Name: arshal
Age: 22
Marks: 85
```

```
Student 2 details:
Name: Raj
Age: 23
Marks: 75

Student 3 details:
Name: Kunal
Age: 25
Marks: 65

Student 4 details:
Name: Tanmay
Age: 22
Marks: 35

Student 5 details:
Name: Sakshi
Age: 23
Marks: 85
```

10. Write a C program that uses typedef to define a struct for a 2D point (x, y) and performs operations like distance calculation between two points.

    Code:-

```
#include <stdio.h>
#include <math.h>

// Define a typedef for the struct Point
typedef struct {
    float x;
    float y;
} Point;

// Function to calculate the distance between two points
float calculateDistance(Point p1, Point p2) {
    return sqrt((p2.x - p1.x) * (p2.x - p1.x) + (p2.y - p1.y) * (p2.y - p1.y));
}

int main() {
    Point point1, point2;

    // Input details for the first point
    printf("Enter the coordinates of point 1 (x y): ");
    scanf("%f %f", &point1.x, &point1.y);
    // Input details for the second point
    printf("Enter the coordinates of point 2 (x y): ");
    scanf("%f %f", &point2.x, &point2.y);

    // Calculate the distance between the two points
    float distance = calculateDistance(point1, point2);

    // Output the result
    printf("The distance between the points (%.2f, %.2f) and (%.2f, %.2f) is: %.2f\n",
           point1.x, point1.y, point2.x, point2.y, distance);

    return 0;
     }
```

Output:-

```
1 2
4 6
Enter the coordinates of point 1 (x y): Enter the coordinates of point 2 (x y): The distance
between the points (1.00, 2.00) and (4.00, 6.00) is: 5.00
```

11. Create a C++ program that declares a class with public, private, and protected access specifiers. Demonstrate how access specifiers control access to members.

    Code:-

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

// Define a class with public, private, and protected members
class Demo {
private:
    int privateValue;  // Private member

protected:
    int protectedValue;  // Protected member

public:
    int publicValue;  // Public member

    // Constructor to initialize values
    Demo(int priv, int prot, int pub) {
        privateValue = priv;
        protectedValue = prot;
        publicValue = pub;
    }

    // Public method to access private and protected members
    void displayValues() {
        cout << "Private Value (accessible within class only): " << privateValue << endl;
        cout << "Protected Value (accessible within class and derived classes): " <<
     protectedValue << endl;
        cout << "Public Value (accessible everywhere): " << publicValue << endl;
    }

    // Public method to access private members
    int getPrivateValue() {
        return privateValue;
    }

    // Public method to access protected members
    int getProtectedValue() {
        return protectedValue;
    }
};

// Derived class to demonstrate access to protected members
class DerivedDemo : public Demo {
public:
    DerivedDemo(int priv, int prot, int pub) : Demo(priv, prot, pub) {}

    void showProtectedValue() {
        cout << "Accessing protected value in derived class: " << protectedValue << endl;
    }
};

int main() {
    // Create an object of the Demo class
    Demo obj(10, 20, 30);

    // Access public member directly
    cout << "Public Value from main: " << obj.publicValue << endl;

    // Access methods to display private and protected members
    obj.displayValues();

    // Access private and protected members via getter methods
    cout << "Private Value from getter method: " << obj.getPrivateValue() << endl;
    cout << "Protected Value from getter method: " << obj.getProtectedValue() << endl;

    // Create an object of the DerivedDemo class (which inherits Demo)
    DerivedDemo derivedObj(100, 200, 300);
    derivedObj.showProtectedValue();  // Access protected member from derived class
    // Access public member from derived class
    cout << "Public Value from derived class: " << derivedObj.publicValue << endl;

    // Trying to access private member from main (this will result in an error)
    // cout << "Private Value from main: " << obj.privateValue << endl;  // Error: 'privateValue' is private

    return 0;
     }
```

Output:-

```
Public Value from main: 30
Private Value (accessible within class only): 10
Protected Value (accessible within class and derived classes): 20
Public Value (accessible everywhere): 30
Private Value from getter method: 10
Protected Value from getter method: 20
Accessing protected value in derived class: 200
Public Value from derived class: 300
```

12. Write a program to create a class called Employee with the data members name, id, and salary. Implement member functions to initialize and display data. Create multiple objects to show how the class works.

Code:-

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

// Define the Employee class
class Employee {
private:
    string name;
    int id;
    float salary;

public:
    // Member function to initialize data members
    void initializeData(string empName, int empId, float empSalary) {
        name = empName;
        id = empId;
        salary = empSalary;
    }

    // Member function to display employee data
    void displayData() {
        cout << "Employee ID: " << id << endl;
        cout << "Employee Name: " << name << endl;
        cout << "Employee Salary: $" << salary << endl;
    }
};

int main() {
    // Create multiple Employee objects
    Employee emp1, emp2, emp3;

    // Initialize data for the first employee
    emp1.initializeData("Harshal", 101, 50000.50);

    // Initialize data for the second employee
    emp2.initializeData("Mona", 102, 55000.75);

    // Initialize data for the third employee
    emp3.initializeData("Sakshi", 103, 60000.25);

    // Display data for all employees
    cout << "\nEmployee 1 Details:\n";
    emp1.displayData();

    cout << "\nEmployee 2 Details:\n";
    emp2.displayData();

    cout << "\nEmployee 3 Details:\n";
    emp3.displayData();

    return 0;
}
```

Output:-

```
Employee 1 Details:
Employee ID: 101
Employee Name: Harshal
Employee Salary: $50000.5

Employee 2 Details:
Employee ID: 102
Employee Name: Mona
Employee Salary: $55000.8

Employee 3 Details:
Employee ID: 103
Employee Name: Sakshi
Employee Salary: $60000.2
```