# Objectives

○ Develop C++ programs using arrays, structures, and functions with inline and default arguments.
○ Understand and apply pass-by-reference and return-by-reference techniques.
○ Use dynamic memory allocation to manage data at runtime.
○ Solve basic computational problems through structured programming.

# Tools and Libraries Used

○ Programming Language: C++
○ IDE: Visual Studio Code
○ Libraries: #include<iostream>, #include<string>

# Theory

Arrays:
structure.
When you declare an array like
arrays in C++ are a fundamental way to store multiple elements of the same type in a fixed-size

```
1. int arr[5];
```

you are reserving space for five integers in contiguous memory. These can be initialized directly
using curly braces, for example: int arr[5] = {1, 2, 3, 4, 5}; Each element is accessed using an
index, starting from 0.
Here's a basic example of array usage:

```
1. #include <iostream>
2. using namespace std;
3.
5. 6. 7. 8. }
9. return 0;
10. }
4. int main() {
int numbers[3] = {10, 20, 30};
for (int i = 0; i < 3; ++i) {
cout << "Element " << i << ": " << numbers[i] << endl;
```

Structures:

structures (or struct) in C++ provide a way to group related variables of different types under a single name. They are useful when modeling real-world entities.
Syntax:

```
1. // Structure definition
2. struct StructName {
3. datatype member1;
4. datatype member2;
5. // ... other members
6. };
7.
8. // Declaring structure variables
9. StructName varName;
10.
11. // Initializing
12. StructName varName = {value1, value2};
```

```
// Accessing members
varName.member1;
```

For instance, if you are dealing with a student record, a structure can be used to combine the ID, name, and grade into one unit. Here's how a structure is defined and used:

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

struct Student {
int id;
string name;
float grade;
};

int main() {
Student s1 = {1, "Alice", 89.5};
cout << "ID: " << s1.id << ", Name: " << s1.name << ", Grade: " << s1.grade << endl;
return 0;
}
```

In this example, the Student structure contains an integer, a string, and a float. We initialize a student object and access its members using the dot operator.Functions: C++ supports both inline functions and functions with default arguments. An inline function is declared using the inline keyword. It's useful for small, frequently called functions, because the compiler attempts to expand the function body at the point of call, potentially reducing function-call overhead.

Syntax:

```
// Inline function declaration and definition
inline returnType functionName(parameterList) {
// function body
}
```

For example:

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

inline int square(int x) {
return x * x;
}

int main() {
cout << "Square of 5: " << square(5) << endl;
return 0;
}
```

This program outputs the square of 5 using an inline function. Inline functions are best used when the function body is short and simple. Default arguments, on the other hand, allow you to call functions without specifying all parameters. Parameters with default values must appear at the end of the parameter list.

Syntax:

```
// Function declaration or definition
returnType functionName(type param1, type param2 = defaultValue, ...);

// Call
functionName(value1); // uses default values
functionName(value1, value2); // overrides defaults
```

7.For instance:

```
1. #include <iostream>
2. using namespace std;
3.
5.
6. }
7.
8. int main() {
9. greet("Bob");
10. greet();
11. return 0;
12. }
13.
4. void greet(string name = "Guest") {
cout << "Hello, " << name << "!" << endl;
```

Here, the greet function has a default value of "Guest". If you call greet() without any argument, it will use the default value.

Parameter passing:
In terms of parameter passing, C++ allows functions to accept parameters by reference. This means the function can modify the original variable passed to it, since it works directly with the memory address of the variable. This is achieved using the & symbol in the function's parameter list.
Syntax:

```
1. // Function definition with reference parameter
2. void functionName(datatype ¶m) {
// param can be modified
3. 4. }
```

For example:

```
1. #include <iostream>
2. using namespace std;
3.
4. void increment(int &n) {
5. n++;
6. }
7.
8. int main() {
9. int a = 5;
10. increment(a);
11. cout << "After increment: " << a << endl;
12. return 0;
13. }
```

Here, the value of a is modified inside the increment function because it is passed by reference. In addition to passing by reference, C++ also allows functions to return a reference. This is particularly useful when you want a function to return a modifiable reference to an element in a data structure like an array.
Syntax:

```
1. // Function returning a reference
2. datatype& functionName(parameters) {
3. // must return a reference to a valid variable
4. }
```

For example:

```
1. #include <iostream>
2. using namespace std;
3.
4. int& getElement(int arr[], int index) {
```

```
5. return arr[index];
6. }
7.
8. int main() {
9. int nums[3] = {1, 2, 3};
10. getElement(nums, 1) = 100;
11. cout << "Updated array: ";
12. for (int i = 0; i < 3; i++) cout << nums[i] << " ";
13. return 0;
14. }
```

This program accesses and modifies an element in the array using a return-by-reference function. The result will show the updated second element as 100.

Dynamic Memory Allocation:
dynamic memory allocation in C++ is used to allocate memory at runtime, which is especially useful when the size of data structures cannot be determined at compile time. You use the newkeyword to allocate memory and delete to free it.
Syntax:

```
1. // Single variable
2. datatype* ptr = new datatype;
4. // Array
5. datatype* arr = new datatype[size];
7. // Access
8. *ptr = value;
9. arr[index] = value;
11. // Deallocation
12. delete ptr; 13. delete[] arr; // for single variable
// for arrays
```

For example:

```
1. #include <iostream>
2. using namespace std;
3.
4. int main() {
5. int *ptr = new int;
6. *ptr = 25;
7. cout << "Value: " << *ptr << endl;
8. delete ptr;
9. return 0;
10. }
11.
```

This code dynamically allocates memory for a single integer and stores the value 25 in it. Similarly, you can create dynamic arrays:

```
1. #include <iostream>
2. using namespace std;
3.
4. int main() {
5. int size = 4;
6. int *arr = new int[size];
7. for (int i = 0; i < size; i++) {
8. arr[i] = i + 1;
9. }
10. 11. for (int i = 0; i < size; i++) {
cout << arr[i] << " ";
12. }
13. delete[] arr;
14. return 0;
15. }
16.
```

**Q1:** Write a program to enter the angles of the triangle and check whether the triangle is cute, right angled or obtuse.

Code:

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

int main(){
    int a,b,c;
    cout <<"enter the largest angle:"<<endl;
    cin>>a;
    cout <<"enter the second angle:"<<endl;
    cin>>b;
    cout<<"enter the third angle:"<<endl;
    cin>>c;
    if (a+b+c==180){


        if (a>b+c){
            cout<<"triangle is acute"<<endl;
        }
        else if (a<b+c){
            cout<<"triangle is obtuse"<<endl;
        }
        else if(a==b+c){
            cout<<"triangle is right angled"<<endl;
        }
    }
    else{
        cout<<"it is not a triangle"<<endl;
    }
    return 0;


}
```

Output:

```
enter the largest angle:
90
enter the second angle:
30
enter the third angle:
60
triangle is right angled
Program ended with exit code: 0
```

**Q2:** Write a C++ program to determine the nature of the roots of a quadratic equation of the form: Ax^2 + Bx + C = 0

Code:

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

int main()
{
    int A,B,C,D;
    cout <<"enter the values of A:"<<endl;
    cin >>A;
    cout<<"enter value of B:"<<endl;
    cin >>B;
    cout<<"enter value of c:"<<endl;
    cin >>C;
    D=B*B-4*A*C;
    if (D > 0) {
            cout << "The equation has two real and distinct roots." << endl;
        }
        else if (D == 0) {
            cout << "The equation has one real and repeated root." << endl;
        }
        else {
            cout << "The equation has two complex (imaginary) roots." << endl;

    }
    return 0;


}
```

Output:

```
enter the values of A:
4
enter value of B:
7
enter value of c:
9
The equation has two complex (imaginary) roots.
Program ended with exit code: 0
```

**Q no 3:** Check password strength based on length and character rules.

Code:

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

 int main()
{
     char pass[100];
     bool U=0,L=0,S=0,N=0;
     cout<<"Enter the password for checking : ";
     cin>>pass;
     int n=strlen(pass);  Implicit conversion loses integer precision: 'size_t' (aka 'unsigned long') to 'int'
     if(n<8){
         cout<<"Error password length.";
     }
     else{
         for(int i=0;i<n;i++){
             if(pass[i]>='A' && pass[i]<='Z'){
                 U = 1;}
             if(pass[i]>='a' && pass[i]<='z'){
                 L = 1;}
             if(pass[i]>='0' && pass[i]<='9'){
                 N = 1;}
             if(pass[i]>=33 && pass[i]<=47){
                 S = 1;}
         }
         if(U ==1 && L ==1 && N ==1 && S ==1){
             cout<<"The password is accepted."<<endl;
         }
         else{
             cout<<"Password error...Please make sure that uppercase, lowercase, number,
                 and special symbols are included."<<endl;
         }}
     return 0;
 }
```

Output:

```
Enter the password for checking : ADityq@#7365465
The password is accepted.
Program ended with exit code: 0
```

## Conclusion:

In this lab, we delved into fundamental C++ programming concepts by solving practical problems that enhanced our comprehension of control structures, functions, and string manipulation. We acquired valuable experience in crafting efficient and structured code. This foundational work not only sharpened our problem-solving abilities but also served as a prelude to more advanced topics, including object-oriented programming, file handling, and data structures in C++.