Control Structures in C++

- Control structures decide the **flow of execution** of a program.
- They help in decision making, looping, and jumping in C++.

C++ has mainly **3 categories** of control structures:

1. **Decision Making (Selection)** → `if`, `if-else`, `nested if`, `if-else ladder`, `switch`
2. **Looping (Iteration)** → `for`, `while`, `do-while`
3. **Jump Statements** → `break`, `continue`, `goto`

1. Decision Making Statements

(a) `if` Statement

Executes a block of code only if condition is true.

Example:

```
int age = 18;
if (age >= 18) {
    cout << "Eligible to vote.";
}
```

(b) `if-else` Statement

Provides two paths: one if condition is true, another if false.

Example:

```
int marks = 45;
if (marks >= 40) {
    cout << "Pass";
} else {
    cout << "Fail";
}
```

(c) `if-else if` Ladder

Used to test multiple conditions.

Example:

```
int marks = 85;
if (marks >= 90) cout << "Grade A";
else if (marks >= 75) cout << "Grade B";
else if (marks >= 50) cout << "Grade C";
else cout << "Fail";
```

(d) Nested `if`

`if` statement inside another `if`.

Example:

```
int age = 20;
if (age >= 18) {
    if (age < 60) {
        cout << "Adult";
    }
}
```

(e) `switch` Statement

Best alternative when there are multiple choices.

```
switch (expression) {
    case value1:
        // code
        break;
    case value2:
        // code
        break;
    default:
        // code if no case matches
}
```

Example:

```
int choice = 2;
switch (choice) {
    case 1: cout << "Start"; break;
    case 2: cout << "Stop"; break;
    default: cout << "Invalid Choice";
}
```

2. Looping Statements

(a) `for` Loop

Used when the number of iterations is known.

```
for (initialization; condition; update) {
    // code
}
```

Example:

```
for (int i = 1; i <= 5; i++) {
    cout << i << " ";
}
```

Output:

```
1 2 3 4 5
```

(b) `while` Loop

Used when the number of iterations is not fixed.

```
while (condition) {
    // code
}
```

Example:

```
int i = 1;
while (i <= 5) {
    cout << i << " ";
    i++;
}
```

(c) `do-while` Loop

Executes code **at least once**, even if condition is false.

```
do {
    // code
} while (condition);
```

Example:

```
int i = 1;
do {
    cout << i << " ";
    i++;
} while (i <= 5);
```

3. Jump Statements

(a) `break`

Exits the loop or switch immediately.

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) break;
    cout << i << " ";
}
```

Output: `1 2`

(b) `continue`

Skips the current iteration and jumps to the next iteration.

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) continue;
    cout << i << " ";
}
```

Output: `1 2 4 5`

(c) `goto`

Transfers control to a labeled statement. *(Not recommended in practice)*

```
int i = 1;
label:
cout << i << " ";
i++;
if (i <= 5) goto label;
```

Output: `1 2 3 4 5`

C++ Pointers Notes

- A **pointer** is a variable that stores the **memory address** of another variable.
- Instead of storing a value directly, it "points" to where the value is located in memory.

Example:

```
int *p;   // p is a pointer to an int
char *c;  // c is a pointer to a char
```

Assigning Address to Pointer

We use the **address-of operator &** to store a variable's address inside a pointer.

```
int a = 10;
int *p = &a;   // p stores the address of a
```

Accessing Value Using Pointer

We use the **dereference operator** * to access the value stored at the address.

```
int a = 10;
int *p = &a;

cout << "Address stored in p: " << p << endl;
cout << "Value at p: " << *p << endl; // prints 10
```

Null Pointer

A pointer that does not point to any valid memory location is called a **null pointer**.

```
int *p = nullptr;   // C++11 way
```

Pointer to Pointer

A pointer can also store the address of another pointer.

```
int a = 5;
int *p = &a;       // pointer to int
int **q = &p;      // pointer to pointer

cout << **q;       // prints 5
```

Pointer Arithmetic

Pointers can be incremented or decremented.
When incremented, they move to the **next memory location** of their type.

```
int arr[3] = {10, 20, 30};
int *p = arr;

cout << *p << endl;    // 10
p++;
cout << *p << endl;    // 20
```

Arrays and Arrays with Pointers in C++

Array is a collection of elements of the **same data type** stored in **contiguous memory location.**

Example:

```
int arr[5] = {10, 20, 30, 40, 50};

cout << arr[0];   // 10
cout << arr[2];   // 30
```

- Array index starts from **0**.
- Memory is allocated **continuously**.

Relationship Between Array and Pointer

In C++, the **array name acts like a pointer** to the first element of the array.

```
int arr[3] = {10, 20, 30};

cout << arr;     // prints address of arr[0]
cout << *arr;    // prints value of arr[0] → 10
```

- `arr` → address of first element.
- `*arr` → value of first element.

Accessing Array Elements Using Pointer

You can use pointer arithmetic (p+1, p+2) to access array elements.

```
int arr[3] = {10, 20, 30};
int *p = arr;   // p points to arr[0]

cout << *p;       // 10
cout << *(p+1);   // 20
cout << *(p+2);   // 30
```

Difference Between Array and Pointer

Array	Pointer
Array is a fixed-size collection of elements.	Pointer is a variable that stores an address.

Array	Pointer
Size must be defined at compile-time (unless using `new`).	Size can be changed by pointing to different memory.
`arr` always points to the same memory block (cannot be reassigned).	Pointer can point to different locations.
Example: `int arr[5];`	Example: `int *p;`

Pointer to an Array

We can create a pointer that points to an **entire array** (not just the first element).

```
int arr[5] = {1, 2, 3, 4, 5};
int (*p)[5] = &arr;   // pointer to whole array

cout << (*p)[2];   // 3
```

Array of Pointers

Instead of one array, you can store multiple pointers in an array.

```
const char *names[3] = {"Alice", "Bob", "Charlie"};

cout << names[0];   // Alice
cout << names[1];   // Bob
```

Dynamic Array with Pointer

We can create arrays dynamically using pointers.

```
int *arr = new int[5];   // dynamic array of 5 integers
arr[0] = 10;
arr[1] = 20;

cout << arr[1];    // 20

delete[] arr;      // free memory
```