Objectives :

- Learn About Comparison Operators
- Learn About Logical Operator
- Learn Control Flow
- Learn About Loops

Comparison Operators :

C provides comparison operators to compare two values and determine their relationship. The result of a comparison is a **boolean** value, which can be either **true (1)** or **false (0)**.

- **> (Greater than):** Checks if the first operand is greater than the second operand.
  - Example: `5 > 4` (evaluates to `true`)
- **< (Less than):** Checks if the first operand is less than the second operand.
  - Example: `4 < 5` (evaluates to `true`)
- **== (Equal to):** Checks if the first operand is equal to the second operand.
  - Example: `4 == 4` (evaluates to `true`)
- **!= (Not equal to):** Checks if the first operand is not equal to the second operand.
  - Example: `1 != 0` (evaluates to `true`)
- **>= (Greater than or equal to):** Checks if the first operand is greater than or equal to the second operand.
  - Example: `2 >= 2` (evaluates to `true`)
- **<= (Less than or equal to):** Checks if the first operand is less than or equal to the second operand.
  - Example: `2 <= 3` (evaluates to `true`)

Example :

```
#include <stdio.h>

int main(){
    int A = 5;
    int B = 6;
    printf("is B Greater then A : %b\n", B > A);
    printf("is A Less then B : %b\n", A < B);
    printf("is A Equal B : %b\n", A == B);
    printf("is A Not Equal B : %b\n", A != B);
    return 0;
}
```

Logical Operator :

C provides logical operators to combine multiple conditions into more complex expressions. The result of a logical operation is also a boolean value (true or false).

- **|| (OR):**
  - Returns **true** if at least one of the conditions connected by `||` is true.
  - Returns **false** only if all the connected conditions are false.
- **&& (AND):**
  - Returns **true** only if all the conditions connected by `&&` are true.
  - Returns **false** if any of the connected conditions are false.
- **! (NOT):**
  - Reverses the logical state of the condition.
  - If the condition is true, `!` makes it false.
  - If the condition is false, `!` makes it true.

Example :

```
#include <stdio.h>

int main(){
    int A = 5;
    int B = 6;
    printf("is B Greater then A And B Less then 10 : %b\n", B > A && B < 10);
    printf("is A Greater then B Or A positive : %b\n", A < B || A > 0);
    printf("is A Equal B : %b\n", !(A != B));
    return 0;
}
```

Control Flow :

If Else statement :

Control flow refers to the order in which instructions in a program are executed. It allows us to make decisions and control the program's behavior based on specific conditions.
C provides the `if-else` statement to implement conditional execution. This allows the program to execute different blocks of code depending on whether a certain condition is true or false.

```
if (condition){
instructions
}
```

We place the condition within parentheses `()` and enclose the instructions to be executed when the condition is true within curly braces `{}`.
This will execute the instructions within the `if` block only if the condition is true. Otherwise, it will ignore them. We can add an `else` statement to provide an alternative set of instructions to be executed if the condition is false.

```
if (condition){
instructions to run if condition valide
}else{
instructions to run if condition not valide
}
```

The `else` statement does not require a condition because it executes only when the preceding `if` condition is **false**.
To check for multiple conditions and provide different execution paths, we can use the `else if` statement.

```
if (condition 1){
instructions to run if condition 1 is valide
}else if(condition 2){
instructions to run if condition 2 is valide
}else{
instructions to run if non of the conditions is valide
}
```

Example :

```
#include <stdio.h>

 int main(){
     int n;
     printf("Enter number :");
     scanf("%d", &n);
     if(n > 0){
             printf("Positive \n");
     }else if(n < 0){
             printf("Negative \n");
     }else{
             printf("Null \n");
     }
     return 0;
```

```
}
```

Switch Case statement :

Another way to control the flow of your program is by using the `switch-case` statement. The `switch-case` statement is used when you have a variable and you want to execute different blocks of code based on its value.

Important Note: The variable used in a `switch` statement must be `int`, `char`, or `enum`.

```
switch (variable){
case value1:
instruction that run if variable == value1
break;
case value2:
instruction that run if variable == value2
break;
case value3:
instruction that run if variable == value3
break;
case value4:
instruction that run if variable == value4
break;
default:
instruction that run if variable have other value then we provided
break;
}
```

We can see that to make a `switch-case` statement, we need a variable to test its value. Then, we provide some values to test in using `case` labels.
After each `case`, we see that we added the keyword `break`. This means that if the variable meets the value in a `case`, only the instructions inside that `case` will run. If we don't use the `break` keyword, the computer will run all the instructions that come inside the `case` labels after the valid one.
Finally, we can see we used the `default` label for the last case. The instructions inside it will run for any other values that don't match any of the `case` labels.

Example :

```
#include <stdio.h>

 int main(){
```

```
    int n;
    printf("Enter number :");
    scanf("%d", &n);
    switch(n){
            case 1:
            printf("One \n");
            break;
            case 2:
            printf("Two \n");
            break;
            case 3:
            printf("Three \n");
            break;
            case 4:
            printf("Four \n");
                break;
            case 5:
            printf("Five \n");
            break;
            default:
            printf("Other numbers \n");
            break;

    }
}
```

Loops :

Sometimes we need to use and repeat some instructions more than once. We could repeatedly rewrite the lines of code, but this quickly becomes cumbersome and error-prone. Imagine trying to repeat a block of code 100 times! This would result in a very large and unmanageable file for a relatively simple task.
To address this, we can use **loops**.
Loops are a powerful programming construct that allow us to repeatedly execute a block of code as long as a specific condition is met. This significantly improves code readability, maintainability, and reduces the risk of errors.
There are three primary types of loops in C:

- `for` **loop:** Used when the number of iterations is known in advance.
- `while` **loop:** Used when the number of iterations is not known in advance, and the loop continues as long as a specific condition remains true.
- `do-while` **loop:** Similar to the `while` loop, but it guarantees that the code within the loop will execute at least once, as the condition is checked after the first iteration.

For Loop :

For loop is used when we know how many times we need to repeat the instructions or when we have a range of values over which we want to run the instructions.

```
for (initialization;condition; increment/decrement){
instructions we want to repeat
}
```

- **Initialization:** This part is executed only once at the beginning of the loop. It typically initializes a counter variable.
  - Example: `int i = 0;`
- **Condition:** This condition is checked before each iteration of the loop. If the condition is true, the code within the loop body is executed. If the condition is false, the loop terminates.
  - Example: `i < 10;`
- **Increment/Decrement:** This part is executed at the end of each iteration. It usually modifies the counter variable to control the loop's progress.
  - Example: `i++;`

Example :

```
#include <stdio.h>

int main(){
     for (int i = 0; i < 5; i++){
              printf("Hello \n");
     }
 }
```

While Loop :

While loop is used when we don't know how many times our instruction should run, but we know a condition that our program should repeat and run while it is true.

```
while (condition){
instructions we want to repeat
}
```

Inside the loop, our instructions should influence and change the variable used in the condition; otherwise, we risk having a condition that is always true, leading to an infinite loop. If the condition is false from the start, the instructions inside the loop will never run.

Example :

```
#include <stdio.h>

int main(){
        int i = 0;
    while (i < 5){
             printf("Hello \n");
             i++;
     }
 }
```

Do While Loop :

The last type of loop provided by C is the `do-while` loop. While similar to the `while` loop, it is used when we don't know the exact number of times the instructions need to run but have a condition to evaluate. The key difference between the `do-while` and `while` loop lies in their execution order: the `do-while` loop executes the code block first and then checks the condition. As a result, even if the condition is false at the beginning, the code block will still execute at least once.

```
do{
instructions we want to repeat
}while (condition);
```

As before, the instruction inside the `do-while` loop should change the value of the variable in the condition; otherwise, we risk creating an infinite loop.

Example :

```
#include <stdio.h>

int main(){
        int i = 0;
    do{
             printf("Hello \n");
             i++;
     }while (i < 5);
 }
```

Break and Continue :

We can enhance the control of our loops by using the keywords `break` and `continue`:

- **`break`**: The `break` statement is used to terminate and exit a loop immediately when a specific condition is met.
- **`continue`**: The `continue` statement is used to skip the current iteration of the loop and proceed with the next one, effectively ignoring the instructions for that iteration when a specific condition is true.

Example :

```
#include <stdio.h>

int main(){
        int i = 0;
    while (i < 50){
            i++;
            if(i % 2 == 0){
                    continue;
            }
            else if(i == 25){
                    break;
            }
             printf("%d \n", i);
     }
     return 0;
 }
```

Tasks :

Task 1 :

Write a C program that reads three numbers, **a**, **b**, and **c**, from the user, and then solves the quadratic equation:

$$a^2 + bx + c = 0$$

The program should handle the following cases:

1. When the discriminant $b^2 - 4ac$ is positive, there are two real and distinct roots.
2. When the discriminant is zero, there is one real root .
3. When the discriminant is negative, the equation has no real solutions.

Task 2:

Write a program that reads a number **n** from the user and displays all the prime numbers from 0 to **n**.

Hint :

A prime number is a number that can only be divided by 1 and itself.