

## Unit-2

### Chapter-1

#### Decision Making and Control Statements

Control statements play a crucial role in C programming, enabling programmers to direct the flow of execution within a program as per their specific requirements. They provide the means to make decisions, iterate over code blocks, and control the program's behavior based on certain predefined conditions.

Control Statement	Type	Example
If Statement	Selection	<code>if (condition) { // code block }</code>
Else Statement	Selection	<code>if (condition) { // code block } else { // code block }</code>
If-Else-If Statement	Selection	<code>if (condition) { // code block } else if (condition) { // code block }</code>
Switch Statement	Selection	<code>switch (expression) { case value: // code block break; default: // code block }</code>
While Loop	Iterative	<code>while (condition) { // code block }</code>
Do-While Loop	Iterative	<code>do { // code block } while (condition);</code>
For Loop	Iterative	<code>for (initialization; condition; increment) { // code block }</code>
Break Statement	Jump	<code>break;</code>
Continue Statement	Jump	<code>continue;</code>
Goto Statement	Jump	<code>goto label;</code>
Return	Return	Used by function for returning values

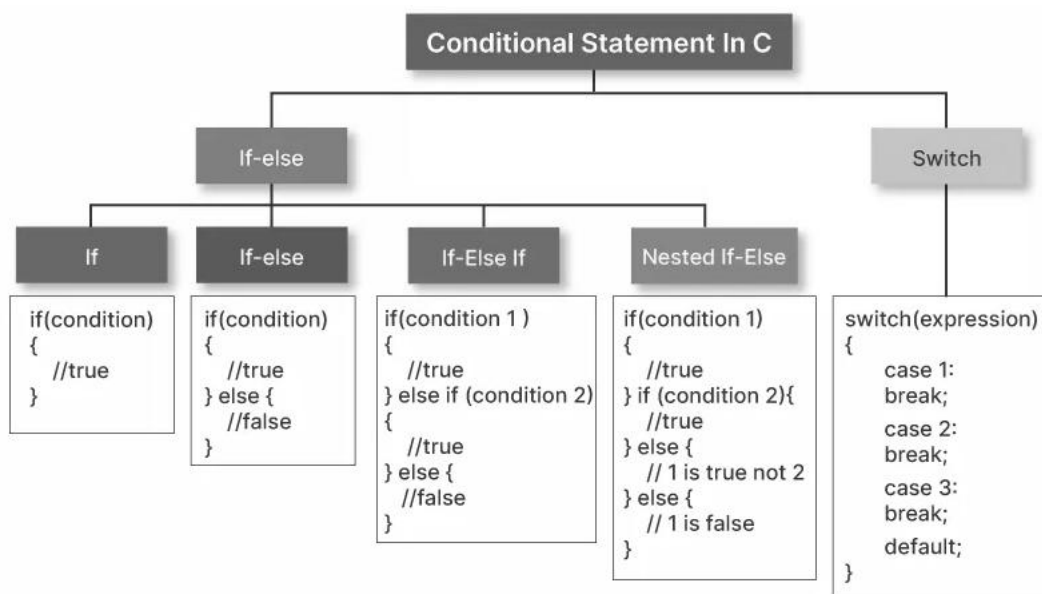
## 2.1 Decision Making Statements / Conditional Statements:

C program executes program sequentially. Sometimes, a program requires checking of certain conditions in program execution. C provides various key condition statements to check condition and execute statements according conditional criteria.

These statements are called as 'Decision Making Statements' or 'Conditional Statements.'

Followings are the different conditional statements used in C.

- 1.If Statement
- 2.If-Else Statement
- 3.Nested If-Else Statement
- 4.Switch Case



**If Statement:** This is a conditional statement used in C to check condition or to control the flow of execution of statements. This is also called as 'decision making statement or control statement.' The execution of a whole program is done in one direction only.

**Syntax:**

```
if(condition)
```

```
{
    Statements;
}
```

In above syntax, the condition is checked first. If it is true, then the program control flow goes inside the braces and executes the block of statements associated with it. If it returns false, then program skips the braces. If there are more than 1 (one) statements in if statement then use { } braces else it is not necessary to use.

**Program :**

```
#include <stdio.h>

#include <conio.h>

void main()
{
    int a;
    a=5;
    clrscr();
    if(a>4)
        printf("\nValue of A is greater than 4 !");
    if(a==4)
        printf("\n\n Value of A is 4 !");
    getch();
}
```

**Output :**

Value of A is greater than 4 !\_

**a. If-Else Statement:**

This is also one of the most useful conditional statement used in C to check conditions.

**Syntax:**

```
if(condition)
{
    Statements;
}
else
{
    Statements;
}
```

In above syntax, the condition is checked first. If it is true, then the program control flow goes inside the braces and executes the block of statements associated with it. If it returns false, then it executes the else part of a program.

**Program :**

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int no;
    clrscr();
    printf("\n Enter Number :");
    scanf("%d",&no);
    if(no%2==0)
        printf("\n\n Number is even !");
    else
        printf("\n\n Number is odd !");
    getch();
}
```

**Output :**

```
Enter Number : 11
Number is odd !_
```

**b. Nested If-Else Statement :**

It is a conditional statement which is used when we want to check more than 1 conditions at a time in a same program. The conditions are executed from top to bottom checking each condition whether it meets the conditional criteria or not. If it found the condition is true then it executes the block of associated statements of true part else it goes to next condition to execute.

**Syntax:**

```
if(condition)
{
    if(condition)
    {
        statements;
    }
    else
    {
        statements;
    }
}
else
```

```

{
    statements;
}

```

In above syntax, the condition is checked first. If it is true, then the program control flow goes inside the braces and again checks the next condition. If it is true then it executes the block of statements associated with it else executes else part.

#### **Program :**

```

#include <stdio.h>
#include <conio.h>
void main()
{
    int no;
    clrscr();
    printf("\n Enter Number :");
    scanf("%d",&no);
    if(no>0)
    {
        printf("\n\n Number is greater than 0 !");
    }
    else
    {
        if(no==0)
        {
            printf("\n\n It is 0 !");
        }
        else
        {
            printf("Number is less than 0 !");
        }
    }
    getch();
}

```

#### **Output :**

```

Enter Number : 0
It is 0 !_

```

#### **c. Switch case Statement :**

This is a multiple or multiway branching decision making statement.

When we use nested if-else statement to check more than 1 conditions then the complexity of a program increases in case of a lot of conditions. Thus, the program is difficult to read and maintain. So to overcome this problem, C provides 'switch case'.

Switch case checks the value of a expression against a case values, if condition matches the case values then the control is transferred to that point.

#### **Syntax:**

```

switch(expression)

```

```

{
    case expr1:
        statements;
        break;
    case expr2:
        statements;
        break;

    case exprn:
        statements;
        break;
    default:
        statements;
}

```

In above syntax, switch, case, break are keywords.

expr1, expr2 are known as 'case labels.'

Statements inside case expression need not to be closed in braces.

Break statement causes an exit from switch statement.

Default case is optional case. When neither any match found, it executes.

#### **Program :**

```

#include <stdio.h>
#include <conio.h>
void main()
{
    int no;
    clrscr();
    printf("\n Enter any number from 1 to 3 :");
    scanf("%d",&no);
    switch(no)
    {
        case 1:
            printf("\n\n It is 1 !");
            break;
        case 2:
            printf("\n\n It is 2 !");
            break;
        case 3:
            printf("\n\n It is 3 !");
            break;
    }
}

```

```
        default:
            printf("\n\n Invalid number !");
    }
    getch();
}
```

**Output 1 :**

Enter any number from 1 to 3 : 3

It is 3 !\_

**Output 2 :**

Enter any number from 1 to 3 : 5

Invalid number !\_

**\* Rules for declaring switch case :**

- The case label should be integer or character constant.
- Each compound statement of a switch case should contain break statement to exit from case.
- Case labels must end with (:) colon.

**\* Advantages of switch case :**

- Easy to use.
- Easy to find out errors.
- Debugging is made easy in switch case.
- Complexity of a program is minimized.

## 2.2. Loops

Loops are used to repeat a block of code. Being able to have your program repeatedly execute a block of code is one of the most basic but useful tasks in programming -- many programs or websites that produce extremely complex output (such as a message board) are really only executing a single task many times. (They may be executing a small number of tasks, but in principle, to produce a list of messages only requires repeating the operation of reading in some data and displaying it.) Now, think about what this means: a loop lets you write a very simple statement to produce a significantly greater result simply by repetition.

- A loop is a part of code of a program which is executed repeatedly.
- A loop is used using condition. The repetition is done until condition becomes condition true.
- A loop declaration and execution can be done in following ways.
- Check condition to start a loop
- Initialize loop with declaring a variable.
- Executing statements inside loop.
- Increment or decrement of value of a variable.

### Types of looping statements :

Basically, the types of looping statements depends on the condition checking mode. Condition checking can be made in two ways as : Before loop and after loop. So, there are 2(two) types of looping statements.

Entry controlled loop & Exit controlled loop.

### 1. Entry controlled loop :

In such type of loop, the test condition is checked first before the loop is executed.

Some common examples of this looping statements are : **while loop & for loop**

### 2. Exit controlled loop :

In such type of loop, the loop is executed first. Then condition is checked after block of statements are executed. The loop executed atleast one time compulsarily.

Some common example of this looping statement is : **do-while loop**

### While loop :

This is an entry controlled looping statement. It is used to repeat a block of statements until condition becomes true.

#### Syntax:

```
while(condition)
{
    statements;
    increment/decrement;
}
```



In above syntax, the condition is checked first. If it is true, then the program control flow goes inside the loop and executes the block of statements associated with it. At the end of loop increment or decrement is done to change in variable value. This process continues until test condition satisfies.

**Program :**

```
#include <stdio.h>

#include <conio.h>

void main()
{
    int a;
    clrscr();
    a=1;
    while(a<=5)
    {
        printf("\n TechnoExam");
        a+=1    // i.e. a = a + 1
    }
    getch();
}
```

Output :

```
TechnoExam
TechnoExam
TechnoExam
TechnoExam
TechnoExam_
```

**For loop :**

This is an entry controlled looping statement.

In this loop structure, more than one variable can be initialized. One of the most important feature of this loop is that the three actions can be taken at a time like variable initialization, condition checking and increment/decrement. The for loop can be more concise and flexible than that of while and do-while loops.

**Syntax:**

```
for(initialisation; test-condition; incre/decre)
{
    statements;
}
```

In above syntax, the given three expressions are separated by ';' (Semicolon)

**Features :**

- More concise
- Easy to use
- Highly flexible
- More than one variable can be initialized.
- More than one increments can be applied.
- More than two conditions can be used.

**Program :**

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int a;
    clrscr();
    for(i=0; i<5; i++)
    {
        printf("\n\t TechnoExam"); // 5 times
    }
    getch();
}
```

**Output :**

```
TechnoExam
TechnoExam
TechnoExam
TechnoExam
TechnoExam
```

**Do-While loop :**

This is an exit controlled looping statement.

Sometimes, there is need to execute a block of statements first then to check condition. At that time such type of a loop is used. In this, block of statements are executed first and then condition is checked.

**Syntax:**

```
do
{
```

```

statements;

(increment/decrement);

}while(condition);

```

In above syntax, the first the block of statements are executed. At the end of loop, while statement is executed. If the resultant condition is true then program control goes to evaluate the body of a loop once again. This process continues till condition becomes true. When it becomes false, then the loop terminates.

**Note: The while statement should be terminated with ; (semicolon).**

**Program :**

```

#include <stdio.h>

#include <conio.h>

void main()
{
    int a;
    clrscr();
    a=1;
    do
    {
        printf("\n\t TechnoExam"); // 5 times
        a+=1;    // i.e. a = a + 1
    }while(a<=5);
    a=6;
    do
    {
        printf("\n\n\t Technowell"); // 1 time
        a+=1;    // i.e. a = a + 1
    }while(a<=5);
    getch();
}

```

**Output :**

```

TechnoExam
TechnoExam
TechnoExam
TechnoExam
TechnoExam
Technowell_

```

## 2.3. Break Statement :

Sometimes, it is necessary to exit immediately from a loop as soon as the condition is satisfied.

When break statement is used inside a loop, then it can cause to terminate from a loop. The statements after break statement are skipped.

### Syntax :

```
break;

while (condition)
{
    _ _ _ _
    break ;
    _ _ _ _
} ←
```

### Program :

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int i;
    clrscr();
    for(i=1; ; i++)
    {
        if(i>5)
            break;
        printf("%d",i); // 5 times only
    }
    getch();
}
```

### Output :

12345

## 2.4 Continue Statement:


Sometimes, it is required to skip a part of a body of loop under specific conditions. So, C supports 'continue' statement to overcome this anomaly.

The working structure of 'continue' is similar as that of that break statement but difference is that it cannot terminate the loop. It causes the loop to be continued with next iteration after skipping statements in between. Continue statement simply skips statements and continues next iteration.

### Syntax :

continue;

```
while (condition)
{
    _ _ _ _
    continue;
    _ _ _ _
}
```



### Program :

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int i;
    clrscr();
    for(i=1; i<=10; i++)
    {
        if(i==6)
            continue;
        printf("\n\t %d",i); // 6 is omitted
    }
    getch();
}
```

### Output :

```
1
2
3
4
5
7
8
9
10
```

## 2.5 Goto Statement :

It is a well known as 'jumping statement.' It is primarily used to transfer the control of execution to any place in a program. It is useful to provide branching within a loop.

When the loops are deeply nested at that if an error occurs then it is difficult to get exited from such loops. Simple break statement cannot work here properly. In this situations, goto statement is used.

### Syntax :

```
goto [expr];
```

```

while (condition)
{
    for( ; ; )
    {
        _ _ _ _
        goto err;
        _ _ _ _
    }
    err: ←
}

```

### Program :

```

#include <stdio.h>
#include <conio.h>
void main()
{
    int i=1, j;
    clrscr();
    while(i<=3)
    {
        for(j=1; j<=3; j++)
        {
            printf(" * ");
            if(j==2)
                goto stop;
        }
        i = i + 1;
    }
    stop:
        printf("\n\n Exited !");
        getch();
}

```

### Output :

```

* *
Exited

```

## return

The return in C returns the flow of the execution to the function from where it is called. This statement does not mandatorily need any conditional statements. As soon as the statement is executed, the flow of the program stops immediately and returns the control from where it was called. The return statement may or may not return anything for a void function, but for a non-void function, a return value must be returned.

Flowchart of return

Flow

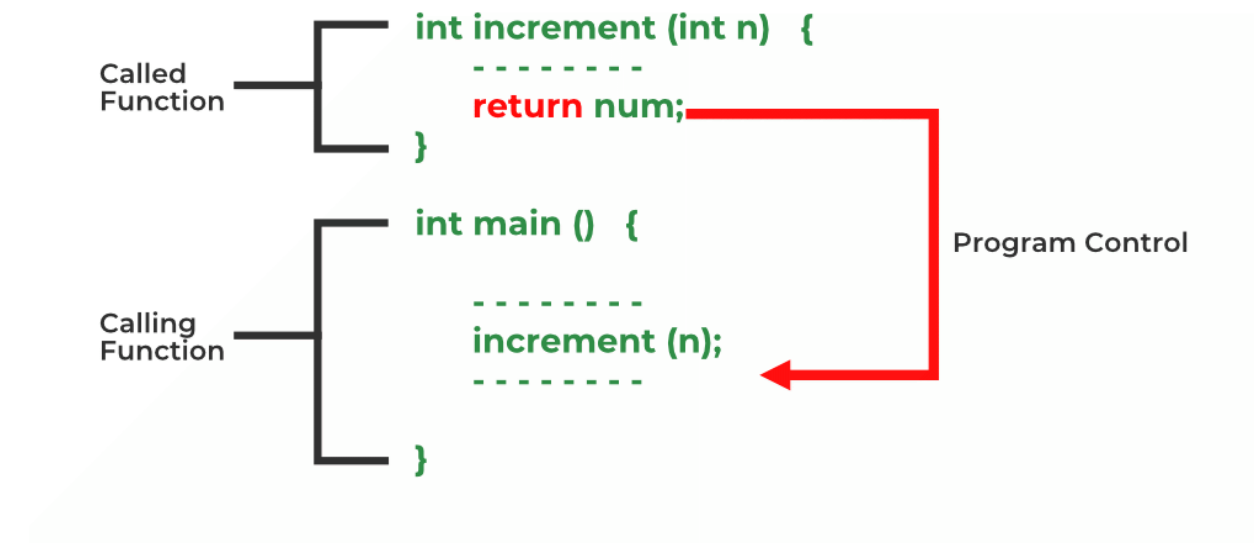


Diagram of return

Syntax of return

**return** [*expression*];

## Storage Class:

'Storage' refers to the scope of a variable and memory allocated by compiler to store that variable. Scope of a variable is the boundary within which a variable can be used. Storage class defines the the scope and lifetime of a variable.

From the point view of C compiler, a variable name identifies physical location from a computer where variable is stored. There are two memory locations in a computer system where variables are stored as : Memory and CPU Registers.

## Functions of storage class :

- (a) Where the variable would be stored.
- (b) What will be the initial value of the variable, if initial value is not specifically assigned.(i.e. the default initial value).
- (c) What is the scope of the variable; i.e. in which functions the value of the variable would be available.
- (d) What is the life of the variable; i.e. how long would the variable exist

## Types of Storage Classes:

Storage classes are categorised in 4 (four) types as,

1. Automatic Storage Class
2. Register Storage Class
3. Static Storage Class
4. External Storage Class

### 1. Automatic Storage Class :

- Keyword : auto
- Storage Location : Main memory
- Initial Value : Garbage Value
- Life : Control remains in a block where it is defined.
- Scope : Local to the block in which variable is declared.

#### Syntax :

```
auto [data_type] [variable_name];
```

**Example :** auto int a;

#### Program :

```
#include <stdio.h>
#include <conio.h>
void main()
{
    auto int i=10;
    {
        auto int i=20;
        printf("\n\t %d",i);
    }
}
```



```

    }
    printf("\n\n\t %d",i);
    getch();
}

```

**Output :**

```

20
10_

```

## 2. Register Storage Class :

- Keyword : register
- Storage Location : CPU Register
- Initial Value : Garbage
- Life : Local to the block in which variable is declared.
- Scope : Local to the block.

**Syntax :**

```
register [data_type] [variable_name];
```

**Example :** register int a;

When the calculations are done in CPU, then the value of variables are transferred from main memory to CPU. Calculations are done and the final result is sent back to main memory. This leads to slowing down of processes.

Register variables occur in CPU and value of that register variable is stored in a register within that CPU. Thus, it increases the resultant speed of operations. There is no waste of time, getting variables from memory and sending it to back again.

It is not applicable for arrays, structures or pointers. It cannot be used with static or external storage class.

Unary and address of (&) cannot be used with these variables as explicitly or implicitly.

**Program :**

```

#include <stdio.h>
#include <conio.h>
void main()
{
    register int i=10;
    clrscr();
    {
        register int i=20;
        printf("\n\t %d",i);
    }
    printf("\n\n\t %d",i);
    getch();
}

```

**Output :**

```

20
10_

```

### 3. Static Storage Class :

- Keyword : static
- Storage Location : Main memory
- Initial Value : Zero and can be initialize once only.
- Life : depends on function calls and the whole application or program.
- Scope : Local to the block.

#### Syntax :

```
static [data_type] [variable_name];
```

#### Example :

```
static int a;
```

There are two types of static variables as :

a) Local Static Variable

b) Global Static Variable

Static storage class can be used only if we want the value of a variable to persist between different function calls.

<pre>main( ) {     increment( ) ;     increment( ) ;     increment( ) ; }  increment( ) {     auto int i = 1 ;     printf ( "%d\n", i ) ;     i = i + 1 ; }</pre>	<pre>main( ) {     increment( ) ;     increment( ) ;     increment( ) ; }  increment( ) {     static int i = 1 ;     printf ( "%d\n", i ) ;     i = i + 1 ; }</pre>
The output of the above programs would be:	
1	1
1	2
1	3

### 4. External Storage Class :

- Keyword : extern
- Storage Location : Main memory

- Initial Value : Zero
- Life : Until the program ends.
- Scope : Global to the program.

**Syntax :**

```
extern [data_type] [variable_name];
```

**Example :**

```
extern int a;
```

The variable access time is very fast as compared to other storage classes. But few registers are available for user programs.

The variables of this class can be referred to as 'global or external variables.' They are declared outside the functions and can be invoked at anywhere in a program.

```
int i;
main()
{
    printf ( "\ni = %d", i );

    increment( );
    increment( );
    decrement( );
    decrement( );
}

increment( )
{
    i = i + 1 ;
    printf ( "\non incrementing i = %d", i );
}

decrement( )
{
    i = i - 1 ;
    printf ( "\non decrementing i = %d", i );
}
```

The output would be:

```
i = 0
on incrementing i = 1
on incrementing i = 2
on decrementing i = 1
on decrementing i = 0
```

## Summary: Storage Classes

Properties Storage Class	Storage	Default Initial Value	Scope	Life
<b>Automatic</b>	Memory	Garbage Value	Local to the block in which the variable is defined	Till the control remains within the block in which the variable is defined
<b>Register</b>	CPU Registers	Garbage Value	Local to the block in which the variable is defined	Till the control remains within the block in which the variable is defined
<b>Static</b>	Memory	Zero	Local to the block in which the variable is defined	Value of the variable continues to exist between different function calls
<b>External</b>	Memory	Zero	Global	Till the program's execution doesn't come to an end