Module 2

Selection Constructs

The selection constructs are used to change the normal sequential flow of the program based on certain conditions. Selection statements are also known as, branching statements, or conditional statements, or decision-making statements. The C language supports the following selection constructs

- Simple if statement
- if else statement
- if else if ladder
- switch statement

1. Simple if statement

The **if** in C is the simplest decision-making statement. The if statement is a program control statement in C language that is used to execute a part of code based on some condition. It consists of the test condition and if block or body. This selection construct executes a sequence only if a certain condition is satisfied

Syntax is :

```
if (condition)
 {
          // block of code to be executed if the condition is true
}
```

Parentheses enclose the condition, which may be a relational expression or a logical expression. The sequence may be a single statement or a code block.

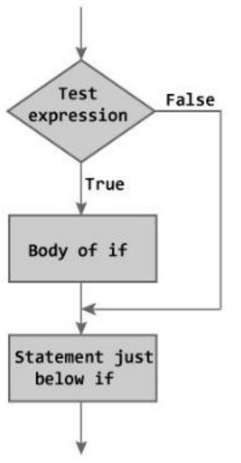


Figure: Flowchart of if Statement

Example 1:

```
if (x > y)
```

```
{
  printf("x is greater than y");
}
```

<u>Example 2: print even if the given number is even</u>

```
if (n % 2 == 0)

{

  printf("%d is Even", n);

 }
```

Advantages of if Statement

- It is the simplest decision-making statement.
- It is easy to use and understand.
- It can evaluate expressions of all types such as int, char, bool, etc.

Disadvantages of if Statement

- It contains only a single block. In case when there are multiply related if blocks, all the blocks will be tested even when the matching if block is found at the start
- When there are a large number of expressions, the code of the if block gets complex and unreadable.
- It is slower for a large number of conditions.

2. <u>if else statement</u>

It is an extension of the **if in C** that includes an **else** block along with the already existing if block. The else statement to specify a block of code to be executed if the condition is false. If the given condition is true, then the code inside the if block is executed, otherwise the code inside the else block is executed.

<u>Syntax of if-else</u>

```
if (condition) {
   // code executed when the condition is true
}
else {
   // code executed when the condition is false
}
```

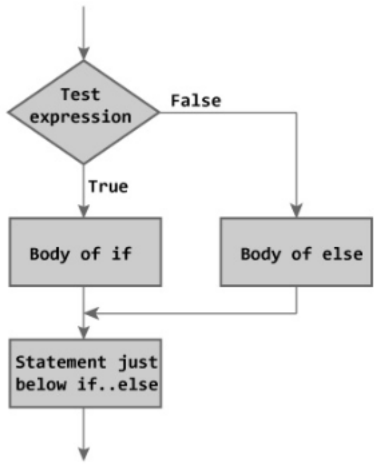


Figure: Flowchart of if...else Statement

Example: Check the given number is odd or even

```
if (n % 2 == 0)
{
            printf("%d is Even", n);
           }
else
{
            printf("%d is Odd", n);
           }
```

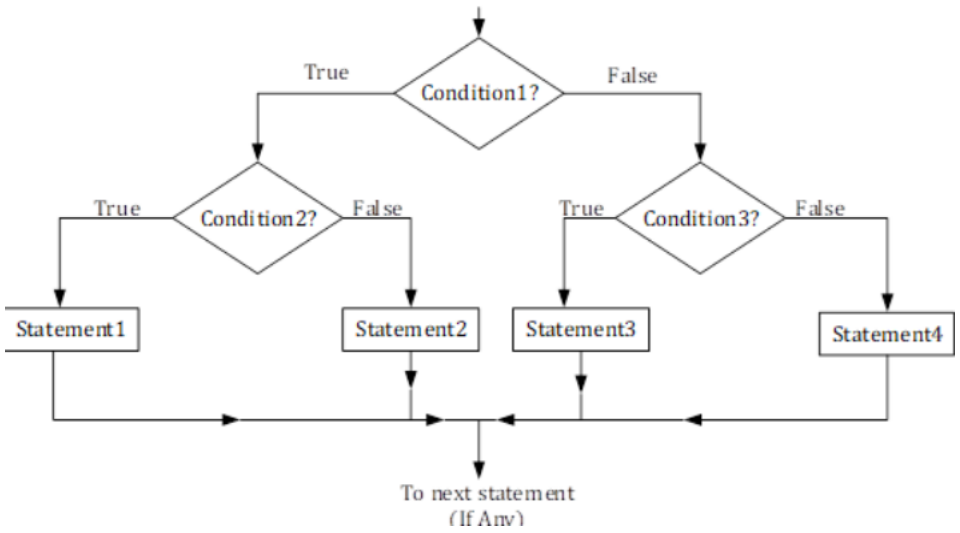
3. Nested if statement

We can have an ***"if-else"*** statement within another ***"if"*** or ***"else"*** block in C since it supports the nesting of ***"if-else"*** statements. It allows for adaptability when managing various environments.

Syntax:

```
if (condition1)

{

  // Code to execute if condition1 is true

   if (condition2)

          {

           // Code to execute if condition2 is true

           }

  else

          {
            // Code to execute if condition2 is false
            }
}
else
{
  // Code to execute if condition1 is false
}
```

Flow chart of nested if else is given below



Example : Program to check the givrn number is positive or negative and if positive, check even or odd

```
#include <stdio.h>
int main()
{
   int num = 10;
   if (num > 0)
   {
      printf("Number is positive.\n");
      if (num % 2 == 0)
           {
           printf("Number is even.\n");
            }
       else
```

```
            {
            printf("Number is odd.\n");
            }


    }
else
{
      printf("Number is non-positive.\n");
   }
   return 0;
}
```

4. <u>if else if ladder</u>

if else if ladder in C programming is used to test a series of conditions sequentially. Furthermore, if a condition is tested only when all previous if conditions in the if-else ladder are false. If any of the conditional expressions are evaluated to be true, the appropriate code block will be executed, and the entire if-else ladder will be terminated.

<u>Syntax:</u>

```
// any if-else ladder starts with an if statement only
if(condition)
{
  // code executed when the condition is true
}
else if(condition)
 {
 // this else if will be executed when condition in if is false and
 // the condition of this else if is true
}
.... // once if-else ladder can have multiple else if
else {
 // at the end we put else
}
```

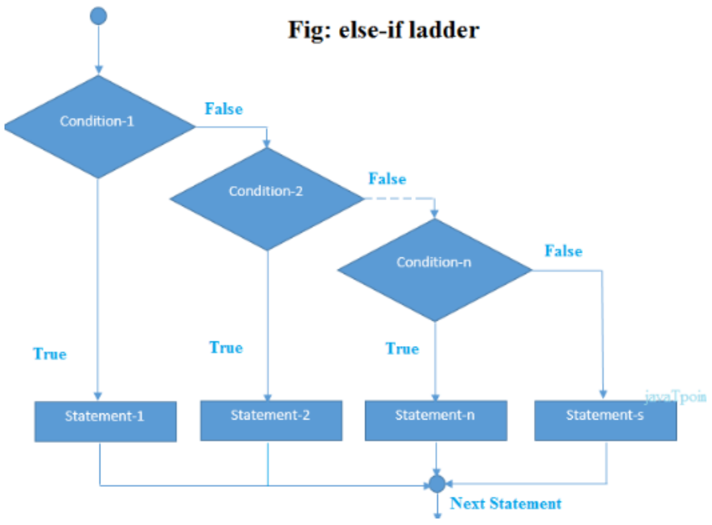


Fig: else-if ladder

<u>Example 1: Check whether the given number is positive, negative or zero</u>

```
if (n > 0) {
    printf("Positive");
  }

  else if (n < 0) {
    printf("Negative");
  }



  else {
    printf("Zero");
  }
```

The condition in a selection construct may be a ***compound*** condition. A compound condition takes the form of a logical expression

<u>Example 2</u>: Print grade equivalent to the marks obtained

```
if (marks <= 100 && marks >= 90)
    printf("A+ Grade");
  else if (marks < 90 && marks >= 80)
    printf("A Grade");
  else if (marks < 80 && marks >= 70)
    printf("B Grade");
```

```
else if (marks < 70 && marks >= 60)
    printf("C Grade");
else if (marks < 60 && marks >= 50)
    printf("D Grade");
else
    printf("F Failed");
return 0;
```

5. Switch Statement

Instead of writing **many** if..else statements, we can use the switch statement. The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression. Switch case statement evaluates a given expression and based on the evaluated value(matching a certain condition) and executes the statements associated with it. The switch statement selects one of many code blocks to be executed. The switch statement consists of conditional-based cases and a default case.

Syntax

```
switch(expression)
{
        case value1: statement_1;
                break;
        case value2: statement_2;
                break;
        .
        .
        .
        case value_n: statement_n;
                break;
        default: default_statement;
}
```

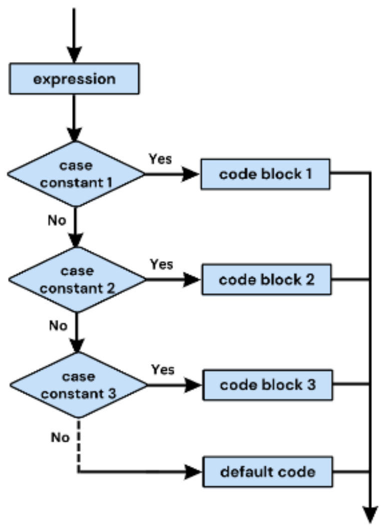
- The switch expression is evaluated once
- The value of the expression is compared with the values of each case
- If there is a match, the associated block of code is executed
- The break statement breaks out of the switch block and stops the execution
- The default statement is optional, and specifies some code to run if there is no case match

Rules of the switch case statement

1. In a switch statement, the “case value” must be of “char” and “int” type.
2. There can be one or N number of cases.
3. The values in the case must be unique.

4. Each statement of the case can have a break statement. It is optional.
5. The default Statement is also optional.

Flow chart of switch statement is given below



Example 1: Using the weekday number calculate the weekday name

```
int day = 4;

switch (day)
{
  case 1:
        printf("Monday");
        break;
  case 2:
        printf("Tuesday");
        break;
  case 3:
        printf("Wednesday");
        break;
  case 4:
        printf("Thursday");
        break;
  case 5:
        printf("Friday");
        break;
  case 6:
        printf("Saturday");
        break;
  case 7:
        printf("Sunday");
        break;
}
```

Example 2 : Program to create a simple calculator

```
#include <stdio.h>

int main()
{
   char operation;
   double n1, n2;

   printf("Enter an operator (+, -, *, /): ");
   scanf("%c", &operation);
   printf("Enter two operands: ");
   scanf("%lf %lf",&n1, &n2);

   switch(operation)
   {
      case '+':
         printf("%.1lf + %.1lf = %.1lf",n1, n2, n1+n2);
         break;

      case '-':
         printf("%.1lf - %.1lf = %.1lf",n1, n2, n1-n2);
         break;

      case '*':
         printf("%.1lf * %.1lf = %.1lf",n1, n2, n1*n2);
         break;

      case '/':
         printf("%.1lf / %.1lf = %.1lf",n1, n2, n1/n2);
         break;

      // operator doesn't match any case constant +, -, *, /
      default:
         printf("Error! operator is not correct");
   }

   return 0;
}
```

Branching statements

Branching statements are used in C programs to alter the flow of control. Branching statements are mainly categorized into two

1. **Conditional Branching Statements**
2. **Unconditional Branching Statements**

Conditional Branching Statements

In C, ***conditional branching statements*** are used to execute particular code blocks based on a condition (as needed). These branching instructions in C allow programmers to run the code only when specific conditions are met. The various categories of conditional branching statements in C are as follows:

- **if Statement**
- **if-else Statement**
- **nested if-else Statement**
- **if-else-if ladder**
- **switch Statement**

Unconditional Branching Statements

In C programming, **unconditional branching statements** are used to alter the normal flow of program execution without any condition. These statements allow the program to jump to a specific point in the code, and also known as unconditional jump statements. The various categories of unconditional branching statements in C are as follows:

- **goto Statement**
- **break Statement**
- **continue Statement**

1. goto Statement

The **C goto statement** is a jump statement which is sometimes also referred to as an **unconditional jump** statement. The goto statement can be used to jump from anywhere to anywhere within a function. The ***"goto"*** statement executes the code in the block following a jump to the ***"label"-labeled statement***. Any legal identifier can be used as the ***"label"*** after a colon ***(:)***. The 'label:' can appear before or after the 'goto label;' statement.

```
Syntax1      |   Syntax2
----------------------------
goto label;  |    label:
.            |    .
.            |    .
.            |    .
label:       |    goto label;
```

Example 1:

```
#include <stdio.h>

    int main()

    {
      int num = 1;
      if (num == 1)
    {
      goto label;
    }
    printf("This statement is skipped.\n");
    label:
    printf("The value of num is 1.\n");
    return 0;
    }
```

In this example, the ***'goto' statement*** switches control to the ***labelled statement 'label'***, skipping the succeeding ***printf statement***, when the condition ***num ==1*** is ***true***. Only the line ***"The value of num is 1"*** is printed consequently

<u>Example 2:</u>C program to check if a number is even or not using goto statement

```
#include <stdio.h>
void checkEvenOrNot(int num)
{
    if (num % 2 == 0)
            // jump to even
            goto even;
    else
            // jump to odd
            goto odd;

even:
    printf("%d is even", num);
    // return if even
    return;
odd:
    printf("%d is odd", num);
}
int main()
```

```
{
    int num = 26;
    checkEvenOrNot(num);
    return 0;
}
```

2. break Statement

This statement is used to exit from loops or switch statements prematurely. The **break in C** is a loop control statement that breaks out of the loop when encountered. It can be used inside loops or switch statements to bring the control out of the block. The break statement can only break out of a single loop at a time.

Syntax :

```
break;
```

The ***"break"*** statement is frequently employed in switch statements as well as looping constructions like ***"for", "while"***, and ***"do-while"***. It enables you to skip the statement that follows the ***loop*** or ***switch*** and end the execution of the closest enclosing loop or switch statement.

Example1:

```
#include <stdio.h>
int main()
{
  int i;
  for (i = 1; i<= 5; i++)
   {
          if (i == 3)
          {
          break;
          }
          printf("%d ", i);
   }
  return 0;
}
```

The ***'for' loop*** iterates from ***1*** to ***5***. However, the ***'break' statement*** is encountered when ***i*** equals ***3***, ending the loop early. Only the numerals ***1*** and ***2*** are printed as a result.

Example 2: C program to illustrate use of break statement in Nested loops

```
#include <stdio.h>

int main()

{

   for (int i = 1; i <= 6; ++i)

   {

      for (int j = 1; j <= i; ++j)

      {

                  if (i <= 4)

                      {

                          printf("%d ", j);

                          }

                  else {

                          // if i > 4 then this innermost loop will break

                          break;

                          }

      }

      printf("\n");

   }

   return 0;

}
```

Output will be:

```
1

1 2

1 2 3

1 2 3 4
```

break statement only breaks out of one loop at a time. So if in nested loop, we have used break in inner loop, the control will come to outer loop instead of breaking out of all the loops at once. We will have to use multiple break statements if we want to break out of all the loops.

3. continue Statement

In the C programming language, the *continue statement* is used to go to the next iteration of a loop while skipping the current iteration. Unlike break, it cannot be used with a C switch case. The **C continue statement** resets program control to the **beginning** of the loop when encountered. As a result, the current iteration of the loop gets skipped and the control moves on to the next iteration. Statements after the continue statement in the loop are not executed.

Syntax:

Continue ;

Example:

```
#include <stdio.h>
int main()
{
    int i;
    for (i = 0; i< 10; i++)
    {
        if (i % 2 == 0)
            {
                continue; // skip even numbers
            }

            printf("%d ", i);
     }
     return 0;
    }
```

Advantages of Branching Statements:

- **Better Decision Making:** Branching statements give programmers the ability to decide what to do next in their code depending on certain circumstances.
- **Readability of the code:** ***Branching statements*** make the logic of the code clearer and simpler to understand.
- **Code effectiveness:** Branching statements optimize program execution by only running the necessary code blocks in accordance with the predetermined circumstances. It can make programs run more quickly and effectively.
- **Flexibility:** Programming ***flexibility*** is provided through branching statements, which enables several code paths to be performed in response to diverse circumstances.
- **Code Reusability:** Branching statements make it easier to reuse code by enabling the execution of code blocks in response to various situations.

Disadvantages of Branching Statements:

- **Code Complexity:** When branching statements are overused or deeply nested, the code may become unclearer and difficult to comprehend
- **Readability Issues:** The code may become more difficult to read and understand if there are too many nested levels, complex conditional expressions, or duplicative sections of code.
- **Potential for Logical Mistakes:** If the conditions are not adequately established and validated, branching statements run the danger of logical mistakes.
- **Code Maintenance:** The intricacy of branching statements makes it more difficult to maintain the code.