## C If else statement

**Syntax of if else statement:**  
If condition returns true then the statements inside the body of “if” are executed and the statements inside body of “else” are skipped.  
If condition returns false then the statements inside the body of “if” are skipped and the statements in “else” are executed.

if(condition) {

// Statements inside body of if

}

else {

//Statements inside body of else

}

**Flow diagram of if else statement**

**Example of if else statement**

In this program user is asked to enter the age and based on the input, the if..else statement checks whether the entered age is greater than or equal to 18. If this condition meet then display message “You are eligible for voting”, however if the condition doesn’t meet then display a different message “You are not eligible for voting”.

#include <stdio.h>

int main()

{

int age;

printf("Enter your age:");

scanf("%d",&age);

if(age >=18)

{

/\* This statement will only execute if the

\* above condition (age>=18) returns true

\*/

printf("You are eligible for voting");

}

else

{

/\* This statement will only execute if the

\* condition specified in the "if" returns false.

\*/

printf("You are not eligible for voting");

}

return 0;

}

Output:

Enter your age:14

You are not eligible for voting

**Note:** If there is **only one statement** is present in the “if” or “else” body then you do not need to use the braces (parenthesis). For example the above program can be rewritten like this:

#include <stdio.h>

int main()

{

int age;

printf("Enter your age:");

scanf("%d",&age);

if(age >=18)

printf("You are eligible for voting");

else

printf("You are not eligible for voting");

return 0;

}

**C Nested If..else statement**

When an if else statement is present inside the body of another “if” or “else” then this is called nested if else.  
**Syntax of Nested if else statement:**

if(condition) {

//Nested if else inside the body of "if"

if(condition2) {

//Statements inside the body of nested "if"

}

else {

//Statements inside the body of nested "else"

}

}

else {

//Statements inside the body of "else"

}

**Example of nested if..else**

#include <stdio.h>

int main()

{

int var1, var2;

printf("Input the value of var1:");

scanf("%d", &var1);

printf("Input the value of var2:");

scanf("%d",&var2);

if (var1 != var2)

{

printf("var1 is not equal to var2\n");

//Nested if else

if (var1 > var2)

{

printf("var1 is greater than var2\n");

}

else

{

printf("var2 is greater than var1\n");

}

}

else

{

printf("var1 is equal to var2\n");

}

return 0;

}

Output:

Input the value of var1:12

Input the value of var2:21

var1 is not equal to var2

var2 is greater than var1

**C – else..if statement**

The else..if statement is useful when you need to check multiple conditions within the program, nesting of if-else blocks can be avoided using else..if statement.

**Syntax of else..if statement:**

if (condition1)

{

//These statements would execute if the condition1 is true

}

else if(condition2)

{

//These statements would execute if the condition2 is true

}

else if (condition3)

{

//These statements would execute if the condition3 is true

}

.

.

else

{

//These statements would execute if all the conditions return false.

}

**Example of else..if statement**

Lets take the same example that we have seen above while discussing nested if..else. We will rewrite the same program using else..if statements.

#include <stdio.h>

int main()

{

int var1, var2;

printf("Input the value of var1:");

scanf("%d", &var1);

printf("Input the value of var2:");

scanf("%d",&var2);

if (var1 !=var2)

{

printf("var1 is not equal to var2\n");

}

else if (var1 > var2)

{

printf("var1 is greater than var2\n");

}

else if (var2 > var1)

{

printf("var2 is greater than var1\n");

}

else

{

printf("var1 is equal to var2\n");

}

return 0;

}

Output:

Input the value of var1:12

Input the value of var2:21

var1 is not equal to var2

As you can see that only the statements inside the body of “if” are executed. This is because in this statement as soon as a condition is satisfied, the statements inside that block are executed and rest of the blocks are ignored.

Important Points:  
1. else and else..if are optional statements, a program having only “if” statement would run fine.  
2. else and else..if cannot be used without the “if”.  
3. There can be any number of else..if statement in a if else..if block.  
4. If none of the conditions are met then the statements in else block gets executed.  
5. Just like relational operators, we can also use logical operators such as AND (&&), OR(||) and NOT(!).

1. */\**
2. *\* C program to find the biggest of three numbers*
3. *\*/*
4. #include <stdio.h>
6. void main()
7. {
8. int num1, num2, num3;
10. printf("Enter the values of num1, num2 and num3**\n**");
11. scanf("%d %d %d", &num1, &num2, &num3);
12. printf("num1 = %d**\t**num2 = %d**\t**num3 = %d**\n**", num1, num2, num3);
13. if (num1 > num2)
14. {
15. if (num1 > num3)
16. {
17. printf("num1 is the greatest among three **\n**");
18. }
19. else
20. {
21. printf("num3 is the greatest among three **\n**");
22. }
23. }
24. else if (num2 > num3)
25. printf("num2 is the greatest among three **\n**");
26. else
27. printf("num3 is the greatest among three **\n**");
28. }

Case:1

Enter the values of num1, num2 and num3

6 8 10

num1 = 6 num2 = 8 num3 = 10

num3 is the greatest among three

Case:2

Enter the values of num1, num2 and num3

10 87 99

num1 = 10 num2 = 87 num3 = 99

num3 is the greatest among three

**switch** statement :

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each **switch case**.

Syntax

The syntax for a **switch** statement in C programming language is as follows −

switch(expression) {

case constant-expression :

statement(s);

break; /\* optional \*/

case constant-expression :

statement(s);

break; /\* optional \*/

/\* you can have any number of case statements \*/

default : /\* Optional \*/

statement(s);

}

The following rules apply to a **switch** statement −

* The **expression** used in a **switch** statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.
* You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
* The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
* When the variable being switched on is equal to a case, the statements following that case will execute until a **break** statement is reached.
* When a **break** statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
* Not every case needs to contain a **break**. If no **break** appears, the flow of control will *fall through* to subsequent cases until a break is reached.
* A **switch** statement can have an optional **default** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No **break** is needed in the default case.

Flow Diagram



Example

#include <stdio.h>

int main () {

/\* local variable definition \*/

char grade = 'B';

switch(grade) {

case 'A' :

printf("Excellent!\n" );

break;

case 'B' :

case 'C' :

printf("Well done\n" );

break;

case 'D' :

printf("You passed\n" );

break;

case 'F' :

printf("Better try again\n" );

break;

default :

printf("Invalid grade\n" );

}

printf("Your grade is %c\n", grade );

return 0;

}

When the above code is compiled and executed, it produces the following result −

Well done

Your grade is B

**Loops**

A loop is used for executing a block of statements repeatedly until a given condition returns false.

## C For loop

This is one of the most frequently used loop C programming.  
**Syntax of for loop:**

for (initialization; condition test; increment or decrement)

{

//Statements to be executed repeatedly

}

### Flow Diagram of For loop

**Step 1:** First initialization happens and the counter variable gets initialized.  
**Step 2:** In the second step the condition is checked, where the counter variable is tested for the given condition, if the condition returns true then the C statements inside the body of for loop gets executed, if the condition returns false then the for loop gets terminated and the control comes out of the loop.  
**Step 3:** After successful execution of statements inside the body of loop, the counter variable is incremented or decremented, depending on the operation (++ or –).

### Example of For loop

#include <stdio.h>

int main()

{

int i;

for (i=1; i<=3; i++)

{

printf("%d\n", i);

}

return 0;

}

Output:

1

2

3

### Various forms of for loop in C

I am using variable num as the counter in all the following examples –  
1) Here instead of num++, I’m using num=num+1 which is same as num++.

for (num=10; num<20; num=num+1)

2) Initialization part can be skipped from loop as shown below, the counter variable is declared before the loop.

int num=10;

for (;num<20;num++)

**Note:**Even though we can skip initialization part but semicolon (;) before condition is must, without which you will get compilation error.  
3) Like initialization, you can also skip the increment part as we did below. In this case semicolon (;) is must after condition logic. In this case the increment or decrement part is done inside the loop.

for (num=10; num<20; )

{

//Statements

num++;

}

4) This is also possible. The counter variable is initialized before the loop and incremented inside the loop.

int num=10;

for (;num<20;)

{

//Statements

num++;

}

5) As mentioned above, the counter variable can be decremented as well. In the below example the variable gets decremented each time the loop runs until the condition num>10 returns false.

for(num=20; num>10; num--)

### Nested For Loop in C

Nesting of loop is also possible. Lets take an example to understand this:

#include <stdio.h>

int main()

{

for (int i=0; i<2; i++)

{

for (int j=0; j<4; j++)

{

printf("%d, %d\n",i ,j);

}

}

return 0;

}

Output:

0, 0

0, 1

0, 2

0, 3

1, 0

1, 1

1, 2

1, 3

In the above example we have a for loop inside another for loop, this is called nesting of loops

## Multiple initialization inside for Loop in C

We can have multiple initialization in the for loop as shown below.

for (i=1,j=1;i<10 && j<10; i++, j++)

**What’s the difference between above for loop and a simple for loop?**  
1. It is initializing two variables. Note: both are separated by comma (,).  
2. It has two test conditions joined together using AND (&&) logical operator. Note: You cannot use multiple test conditions separated by comma, you must use logical operator such as && or || to join conditions.  
3. It has two variables in increment part. **Note:** Should be separated by comma.

### Example of for loop with multiple test conditions

#include <stdio.h>

int main()

{

int i,j;

for (i=1,j=1 ; i<3 || j<5; i++,j++)

{

printf("%d, %d\n",i ,j);

}

return 0;

}

## while loop

The syntax of a while loop is:

while (testExpression)

{

//codes

}

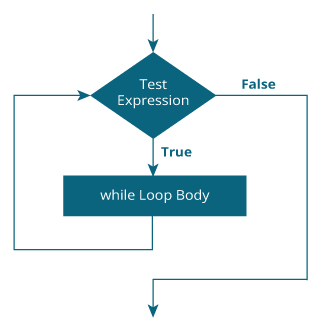
### How while loop works?

The while loop evaluates the test expression.

If the test expression is true (nonzero), codes inside the body of while loop is executed. The test expression is evaluated again. The process goes on until the test expression is false.

When the test expression is false, the while loop is terminated.

### Flowchart of while loop



### Example 1: while loop

// Program to find factorial of a number

// For a positive integer n, factorial = 1\*2\*3...n

#include <stdio.h>

int main()

{

int number;

long long factorial;

printf("Enter an integer: ");

scanf("%d",&number);

factorial = 1;

// loop terminates when number is less than or equal to 0

while (number > 0)

{

factorial \*= number; // factorial = factorial\*number;

--number;

}

printf("Factorial= %lld", factorial);

return 0;

}

**Output**

Enter an integer: 5

Factorial = 120

To learn more on test expression (when test expression is evaluated to nonzero (true) and 0 (false)).

## do… while loop

**Syntax:**

|  |  |
| --- | --- |
| 1  2  3  4  5 | do{     // body of do while loop     statement 1;     statement 2;  }while(condition); |

In do while loop first the statements in the body are executed then the condition is checked. If the condition is true then once again statements in the body are executed. This process keeps repeating until the condition becomes false. As usual, if the body of do while loop contains only one statement, then braces ({}) can be omitted. Notice that unlike the [while loop](https://overiq.com/c-programming-101/the-while-loop-in-c/), in do while a semicolon(;) is placed after the condition.

The do while loop differs significantly from the [while loop](https://overiq.com/c-programming-101/the-do-while-loop-in-c/) because in do while loop statements in the body are executed at least once even if the condition is false. In the case of while loop the condition is checked first and if it true only then the statements in the body of the loop are executed.

The following program print numbers between 1 and 100 which are multiple of 3 using the do while loop:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | #include<stdio.h> // include the stdio.h    int main()  {      int i = 1; // declare and initialize i to 1        do      {          // check whether i is multiple of 3 not or not          if(i % 3 == 0)          {              printf("%d ", i);   // print the value of i          }          i++; // increment i by 1      }while(i < 100);  // stop the loop when i becomes greater than 100        // signal to operating system everything works fine      return 0;  } |

**Expected Output:**

|  |  |
| --- | --- |
| 1  2 | 3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48 51 54 57 60 63 66 69 72 7  5 78 81 84 87 90 93 96 99 |

How it works:

In line 5, we have declared and initialized variable i. Then, the control comes inside the body of the do while loop. Inside the body of the loop the if condition (i%3==0) is tested, if it is true, then the statement inside the if block  is executed. The statement i++ increments the value of i by 1. At last, the do while condition (i<100) is checked. If it is true then statements inside the body of the loop are executed once again. This process keeps repeating as long as the value of i is less than 100.

## Where should I use do while loop?

Most of the time you will use while loop instead of do while. However, there are some scenarios where do while loop suits best. Consider the following problem.

Let’s say you want to create a program to find the factorial of a number. As you probably know that factorial is only valid for 0 and positive numbers. Here is one way you can approach this problem.

Let’s say the user entered a negative number, so instead of displaying an error message and quitting the program, a better approach would be to ask the user again to enter a number. You have to keep asking until the user enters a positive number or 0. Once a positive number or 0 is entered, calculate factorial and display the result.

Let’s see how we can implement it using while and do while loop.

## Using while loop

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | #include<stdio.h> // include the stdio.h    int main()  {      int num;      char num\_ok = 0;        // keep asking for numbers until num\_ok == 0      while(num\_ok==0)      {          printf("Enter a number: ");          scanf("%d", &num);            // if num >= 0 set num\_ok = 1 and stop asking for input          if(num>=0)          {              num\_ok = 1;          }      }       // calculate factorial  } |

## Using do while loop

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | #include<stdio.h> // include the stdio.h    int main()  {      int num;        do      {          printf("Enter a number: ");          scanf("%d", &num);      }while(num<0); // keep asking for numbers until num < 0       // calculate factorial  } |

Notice that the solution using while loop is more involved, to achieve the same thing we have to create an extra variable num\_ok, and an additional if statement. On the other hand, the do while loop achieves the same thing without any trickery and it’s more elegant and concise.

Before we leave do while loop, let’s take one more example.

The Following program calculates Simple interest:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Program to calculate the Simple interest  \*  \* SI = (Principal \* Rate \* Time) / 100  \*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/    #include<stdio.h> // include stdio.h    int main()  {      float p, r, t;      char ch = 'y';        do      {          printf("Enter principal: ");          scanf("%f", &p);            printf("Enter rate: ");          scanf("%f", &r);            printf("Enter t: ");          scanf("%f", &t);            printf("SI = %.2f", (p \*r \* t)/100 );            printf("\n\nCalculate SI one more time ? ('y' for Yes, 'n' for no ) : ");          scanf(" %c", &ch);    // notice the preceding white space before %c      }while(ch == 'y');        // keep asking for P, R and T til the input is 'y'        // signal to operating system everything works fine      return 0;  } |

**Expected Output:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | Enter principal: 15000  Enter rate: 4.5  Enter t: 3  SI = 2025.00    Calculate SI one more time ? ('y' for Yes, 'n' for no ) : y  Enter principal: 20000  Enter rate: 5.4  Enter t: 4  SI = 4320.00    Calculate SI one more time ? ('y' for Yes, 'n' for no ) : n |

Operator Precedence

It usually means, if an expression has multiple operators in it, which operator get the precedence over others. To understand what's meant by operator precedence, let's take an arithmetic expression as an example.  
  
Consider, int a = 2 + 3 \* 4. Value of 'a' would be 14, not 20. This is because operator '\*' is executed before '+', like the [BODMAS](http://www.mathsisfun.com/operation-order-bodmas.html) rule. So we can say that operator '\*' has precedence over operator '+' and the expression would be grouped as int a = (2 + (3 \* 4)).

Operator Associativity

Associativity specifies how the operators are grouped for evaluation. For example, consider the expression 2 + 3 + 4. Note: In this case (+ operator), the result of different grouping doesn't matter. But that might not be the case with other operators.   
  
can you tell how the expression will be grouped? ((2 + 3) + 4) or (2 + (3 + 4)). This is determined by the associativity of operators. From the above example we can understand that, associativity comes into picture only when expression has operators of same precedence.

Different Types of Operator Associativity

There are two types of associativity possible known as 'Right to Left' associativity and 'Left to Right' associativity. L->R (left to right) associativity means operator on the left hand side will get priority over right hand operator and R->L (right to left) associativity means right most operators will get more priority. To understand the associativity in detail, lets consider two example expressions.

1. **a = b = c** : '=' has right to left associativity. So the expression will be grouped as (a = (b = c))
2. **a + b + c** : '+' has left to right associativity. So the expression will be grouped as ((a + b) + c)

Precedence and Associativity Table

Table below shows the precedence and associativity of the operators available in C language. The operators in top rows have more precedence compared to the ones in bottom rows. Operators in same cell have same precedence.

|  |  |  |
| --- | --- | --- |
| **Operator(s)** | **Description** | **Associativity** |
| ++ -- | Post increment/post decrement operator | Left to right associativity |
| ++ -- | Pre-increment/pre-decrement operator | Right to left associativity |
| + - | Unary plus/minus operator | Right to left associativity |
| ! ~ | Logical NOT/bitwise NOT operator | Right to left associativity |
| \* | "Value at address" operator | Right to left associativity |
| & | "Address of" operator | Right to left associativity |
| sizeof | "Size of" operator | Right to left associativity |
| \* / % | Multiplication/division/modulus operator | Left to right associativity |
| + - | Addition/subtraction operator | Left to right associativity |
| << >> | Bitwise left shift/bitwise right shift operator | Left to right associativity |
| < <= | "Less than"/"Less than or equal to" relational operator | Left to right associativity |
| > >= | "Greater than"/"Greater than or equal to" relational operator | Left to right associativity |
| == != | "Equal to"/"Not equal to" relational operator | Left to right associativity |
| & | Bitwise AND operator | Left to right associativity |
| ^ | Bitwise XOR operator | Left to right associativity |
| | | Bitwise OR operator | Left to right associativity |
| && | Logical AND operator | Left to right associativity |
| || | Logical OR operator | Left to right associativity |
| ?: | Ternary conditional operator | Right to left associativity |
| = | Assignment operator | Right to left associativity |
| += -= | Assignment with addition/subtraction | Right to left associativity |
| \*= /= %/ | Assignment with multiplication/division/modulus | Right to left associativity |
| <<= >>= | Assignment with bitwise left shift/bitwise right shift | Right to left associativity |
| &= ^= |= | Assignment with bitwise AND/bitwise XOR/bitwise OR | Right to left associativity |

precedence - Associativity Examples

1. #include <stdio.h>
3. main()
4. {
5. float a = 10;
6. float b = 20;
7. float c = 30;
8. float d = 40;
9. float e;
11. e = a + b \* c / d;
12. printf("Value of a + b \* c / d is : %f\n", e );
14. e = (a + b) \* c / d;
15. printf("Value of (a + b) \* c / d is : %f\n" , e );
17. e = a + (b \* c) / d;
18. printf("Value of a + (b \* c) / d is : %f\n", e );
20. e = a + b \* (c / d);
21. printf("Value of a + b \* (c / d) is : %f\n" , e );
23. return 0;
24. }
26. Output:
28. Value of a + b \* c / d is : 25.000000
29. Value of (a + b) \* c / d is : 22.500000
30. Value of a + (b \* c) / d is : 25.000000
31. Value of a + b \* (c / d) is : 25.000000

An applicant must meet two conditions:

* 4 or more years of college, AND
* 2 years experience programming in Java OR a grade point average greater than 3.5.

### Answer:

if ( college >= 4 && **(** experience >= 2 || gpa > 3.5 **)** )

System.out.println("Interview applicant");

else

System.out.println("Send resume to circular file.");

# Precedence of Logical Operators

You have seen that when expressions mix && with || that evaluation must be done in the correct order. Parentheses can be used to group operands with their correct operator, just like in arithmetic. Also like arithmetic operators, logical operators have precedence that determines how things are grouped in the absence of parentheses.

In an expression, the operator with the highest precedence is grouped with its operand(s) first, then the next highest operator will be grouped with its operands, and so on. If there are several logical operators of the same precedence, they will be examined left to right.

It is common for programmers to use parentheses to group operands together for readability even when operator precedence alone would work.

|  |  |
| --- | --- |
| **Operator** | **precedence** |
| ! | High |
| && | Medium |
| || | Low |

|  |  |  |
| --- | --- | --- |
| A || B && C | means | A || (B && C) |
| A && B || C && D | means | (A && B) || (C && D) |
| A && B && C || D | means | ((A && B) && C) || D |
| !A && B || C | means | ((!A) && B) || C |