**THE LOOP CONTROL STRUCTURE**

**Loops:**

when a block of code needs to be executed several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. Given below is the general form of a loop statement in most of the programming languages −



**There are mainly two types of loops:**

1. **Entry Controlled loops**: In this type of loop, the test condition is tested before entering the loop body. **For Loop** and **While Loop** is entry-controlled loops.
2. **Exit Controlled Loops**: In this type of loop the test condition is tested or evaluated at the end of the loop body. Therefore, the loop body will execute at least once, irrespective of whether the test condition is true or false. the do-while**loop** is exit controlled loop.



C programming language provides the following types of loops to handle looping requirements.

|  |  |
| --- | --- |
| **S.No** |  **Loop type and description** |
| 1 | **while loop** – First checks the condition, then executes the body. |
| 2 | **for loop**– firstly initializes, then, condition check, execute body, update. |
| 3 | **do-while** **loop**– firstly, execute the body then condition check |

**While loop**

* A **while** loop in C programming repeatedly executes a target statement as long as a given condition is true.
* A while loop is the most straightforward looping structure.
* It is an entry-controlled loop.
* In while loop, a condition is evaluated before processing a body of the loop.
* If a condition is true then and only then the body of a loop is executed. After the body of a loop is executed then control again goes back at the beginning, and the condition is checked if it is true, the same process is executed until the condition becomes false.
* Once the condition becomes false, the control goes out of the loop.
* After exiting the loop, the control goes to the statements which are immediately after the loop.
* The body of a loop can contain more than one statement. If it contains only one statement, then the curly braces are not compulsory. It is a good practice though to use the curly braces even we have a single statement in the body.
* In while loop, if the condition is not true, then the body of a loop will not be executed, not even once.

**Syntax**

The syntax of a **while** loop in C programming language is −

 **while(condition) {**

 **statement(s);**

 **}**

**Flow Diagram**



Here, the key point to note is that a while loop might not execute at all. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

**Example**

#include <stdio.h>

int main () {

 /\* local variable definition \*/

 int a = 10;

 /\* while loop execution \*/

 while( a < 20 ) {

 printf("value of a: %d\n", a);

 a++;

 }

 return 0;

}

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

**value of a: 10**

**value of a: 11**

**value of a: 12**

**value of a: 13**

**value of a: 14**

**value of a: 15**

**value of a: 16**

**value of a: 17**

**value of a: 18**

**value of a: 19**

***do-while* Loop**

* The do..while loop is similar to the while loop with one important difference. The body of do...while loop is executed at least once. Only then, the test expression is evaluated.
* Using the do-while loop, we can repeat the execution of several parts of the statements.

**The syntax of the do...while loop is:**

**do {**

 **// the body of the loop**

**}**

**while (testExpression);**

### Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop executes once before the condition is tested. If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop executes again. This process repeats until the given condition becomes false.

### **How do...while loop works?**

* The body of do...while loop is executed once. Only then, the testExpression is evaluated.
* If testExpression is **true**, the body of the loop is executed again and testExpression is evaluated once more.
* This process goes on until testExpression becomes **false**.
* If testExpression is **false**, the loop ends.

## Flow Diagram

## do...while loop in C

## Example:

#include <stdio.h>

int main () {

 /\* local variable definition \*/

 int a = 10;

 /\* do loop execution \*/

 do {

 printf("value of a: %d\n", a);

 a = a + 1;

 }while( a < 20 );

 return 0;

}

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

**value of a: 10**

**value of a: 11**

**value of a: 12**

**value of a: 13**

**value of a: 14**

**value of a: 15**

**value of a: 16**

**value of a: 17**

**value of a: 18**

**value of a: 19**

## For loop

* A for loop is a more efficient loop structure in ‘C’ programming.
* A for loop is a repetition control structure that allows us to write a loop that is executed a specific number of times. The loop enables us to perform n number of steps together in one line.

**Syntax:**

**for (initialization; condition; increment)**

**{**

 **statements;**

**}**

Here is the flow of control in a 'for' loop −

* The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
* Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and the flow of control jumps to the next statement just after the 'for' loop.
* After the body of the 'for' loop executes, the flow of control jumps back up to the **increment** statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
* The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the 'for' loop terminates.

**Flow Diagram**



**Example**

#include <iostream>

using namespace std;

int main()

{

    for (int i = 1; i <= 5; i++) {

        cout << "Hello World\n";

    }

    return 0;

}

**Output**

**Hello World**

**Hello World**

**Hello World**

**Hello World**

**Hello World**

**ODD Loop:**

Sometimes a user may not know about how many times a loop is to be executed. If we want to execute a loop for unknown number of times, then the concept of odd loops should be implemented. This can be done using for-loop, while-loop or do-while-loops.

### **Example**

Following is the C program for the odd loop −

#include<stdio.h>

int main(){

   int number;

   number=1;

   while(number==1) // odd loop don’t know how many times loop executes{

      printf("enter a number:\n");

      scanf("%d",&number);

      if((number%2)==0)

         printf("number is even\n");

      else

         printf("number is odd\n");

         printf("do you want to test any number\n");

         printf("if yes then press '1'\n");// if press 1 loop executes again

         printf("else press '0'\n");//if press 0 exist from loop

         scanf("%d",&number);

   }

   return 0;

}

### **Output**

When the above program is executed, it produces the following output −

**enter a number:**

3

number is odd

do you want to test any number

if yes then press '1'

else press '0'

1

**enter a number:**

4

number is even

do you want to test any number

if yes then press '1'

else press '0'

1

**enter a number:**

9

number is odd

do you want to test any number

if yes then press '1'

else press '0'

0

**The *break* Statement**

* The break is a keyword in C which is used to bring the program control out of the loop.
* The break statement is used inside loops or switch statement.
* The break statement breaks the loop one by one, i.e., in the case of nested loops, it breaks the inner loop first and then proceeds to outer loops.

The break statement in C can be used in the following two scenarios:

* **With switch case**
* **With loop**

If you are using nested loops, the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

## How break statement works?

## Working of break statement

## Syntax

The syntax for a **break** statement in C is as follows −

**break;**

**Flow Diagram**



**Example:**

#include<stdio.h>

#include<stdlib.h>

**void** main ()

{

    **int** i;

    **for**(i = 0; i<10; i++)

    {

        printf("%d ",i);

        **if**(i == 5)

        **break**;

    }

     printf("came outside of loop i = %d",i);

}

**Output**

**0 1 2 3 4 5 came outside of loop i = 5**

**The continue Statement**

* The continue statement skips the current iteration of the loop and continues with the next iteration.
* The continue statement is almost always used with the if...else statement.

**How continue statement works?**



## Syntax

The syntax for a **continue**statement in C is as follows −

 **continue;**

**Flow Diagram**



**Example**

#include <stdio.h>

int main () {

 /\* local variable definition \*/

 int a = 10;

 /\* do loop execution \*/

 do {

 if( a == 15) {

 /\* skip the iteration \*/

 a = a + 1;

 continue;

 }

 printf("value of a: %d\n", a);

 a++;

 } while( a < 20 );

 return 0;

}

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

**value of a: 10**

**value of a: 11**

**value of a: 12**

**value of a: 13**

**value of a: 14**

**value of a: 16**

**value of a: 17**

**value of a: 18**

**value of a: 19**

**The Case Control Structure:**

C provides a special control statement that allows us to handle such cases effectively; rather than using a series of **if** statements. This control instruction is in fact the topic of this chapter. Towards the end of the chapter we would also study a keyword called **goto**, and understand why we should avoid its usage in C programming.

**Decisions Using *switch:***

* The switch statement in C is an alternate to if-else-if ladder statement which allows us to execute multiple operations for the different possible values of a single variable called switch variable.
* Basically, it is used to perform different actions based on different conditions(cases).
* 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.

**Syntax:**

**switch**(expression){

**case** value1:

 //code to be executed;

 **break**;  //optional

**case** value2:

 //code to be executed;

 **break**;  //optional

......

**default**:

 code to be executed **if** all cases are not matched;

}

### **Rules for switch statement in C language**

1) The *switch expression* must be of an integer or character type.

2) The *case value* must be an integer or character constant.

3) The case value can be used only inside the switch statement.

4) The break statement in switch case is not must. It is optional. If there is no break statement found in the case, all the cases will be executed present after the matched case. It is known as fall through the state of C switch statement.

## 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**

**Switch Versus if-else Ladder:**

**if-else Ladder:**

The if else ladder statement in C programming language is used to test set of conditions in sequence. An if condition is tested only when all previous if conditions in if-else ladder is false. If any of the conditional expression evaluates to true, then it will execute the corresponding code block and exits whole if-else ladder.

**Syntax:**

if (condition)

 statement 1;

else if (condition)

 statement 2;

.

.

else

 statement;

**Flow Chart:**



|  |  |  |
| --- | --- | --- |
|  | **If-else** |  **switch** |
| **Definition** | Depending on the condition in the 'if' statement, 'if' and 'else' blocks are executed. | The user will decide which statement is to be executed. |
| **Expression** | It contains either logical or equality expression. | It contains a single expression which can be either a character or integer variable. |
| **Evaluation** | It evaluates all types of data, such as integer, floating-point, character or Boolean. | It evaluates either an integer, or character. |
| **Sequence of execution** | First, the condition is checked. If the condition is true then 'if' block is executed otherwise 'else' block | It executes one case after another till the break keyword is not found, or the default statement is executed. |
| **Default execution** | If the condition is not true, then by default, else block will be executed. | If the value does not match with any case, then by default, default statement is executed. |
| **Editing** | Editing is not easy in the 'if-else' statement. | Cases in a switch statement are easy to maintain and modify. Therefore, we can say that the removal or editing of any case will not interrupt the execution of other cases. |
| **Speed** | If there are multiple choices implemented through 'if-else', then the speed of the execution will be slow. | If we have multiple choices then the switch statement is the best option as the speed of the execution will be much higher than 'if-else'. |

**The goto Keyword**

* The goto statement is a jump statement which is sometimes also referred to as unconditional jump statement.
* The goto statement can be used to jump from anywhere to anywhere within a function.

NOTE − Use of goto statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a goto can be rewritten to avoid them.

**Syntax**:

Syntax1 | Syntax2

----------------------------

goto label; | label:

. | .

. | .

. | .

label: | goto label;

In the above syntax, the first line tells the compiler to go to or jump to the statement marked as a label. Here label is a user-defined identifier which indicates the target statement. The statement immediately followed after ‘label:’ is the destination statement. The ‘label:’ can also appear before the ‘goto label;’ statement in the above syntax.

## Flow Diagram



## Example 1:

#include <stdio.h>

int main () {

 /\* local variable definition \*/

 int a = 10;

 /\* do loop execution \*/

 LOOP:do {

 if( a == 15) {

 /\* skip the iteration \*/

 a = a + 1;

 goto LOOP;

 }

 printf("value of a: %d\n", a);

 a++;

 }while( a < 20 );

 return 0;

}

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

**value of a: 10**

**value of a: 11**

**value of a: 12**

**value of a: 13**

**value of a: 14**

**value of a: 16**

**value of a: 17**

**value of a: 18**

**value of a: 19**

**Example 2**

#include <stdio.h>

**int** main()

{

  **int** num,i=1;

  printf("Enter the number whose table you want to print?");

  scanf("%d",&num);

  table:

  printf("%d x %d = %d\n",num,i,num\*i);

  i++;

  **if**(i<=10)

  **goto** table;

}

**Output:**

**Enter the number whose table you want to print?10**

**10 x 1 = 10**

**10 x 2 = 20**

**10 x 3 = 30**

**10 x 4 = 40**

**10 x 5 = 50**

**10 x 6 = 60**

**10 x 7 = 70**

**10 x 8 = 80**

**10 x 9 = 90**

**10 x 10 = 100**