

Static in C

Static is a keyword used in C programming language. It can be used with both variables and functions, i.e., we can declare a static variable and static function as well. An ordinary variable is limited to the scope in which it is defined, while the scope of the static variable is throughout the program.

Static keyword can be used in the following situations:

Static global variable

When a global variable is declared with a static keyword, then it is known as a static global variable. It is declared at the top of the program, and its visibility is throughout the program.

Static function

When a function is declared with a static keyword known as a static function. Its lifetime is throughout the program.

Static local variable

When a local variable is declared with a static keyword, then it is known as a static local variable. The memory of a static local variable is valid throughout the program, but the scope of visibility of a variable is the same as the automatic local variables. However, when the function modifies the static local variable during the first function call, then this modified value will be available for the next function call also.

Let's understand through an example.

```
#include <stdio.h>
int main()
{
    printf("%d",func());
    printf("\n%d",func());
    return 0;
}
int func()
{
    int count=0; // variable initialization
    count++; // incrementing counter variable

    return count;
}
```

In the above code, the func() function is called. In func(), count variable gets updated. As soon as the function completes its execution, the memory of the count variable will be removed. If we do not want to remove the count from memory, then we need to use the count variable as static. If we declare the variable as static, then the variable will not be removed from the memory even when the function completes its execution.

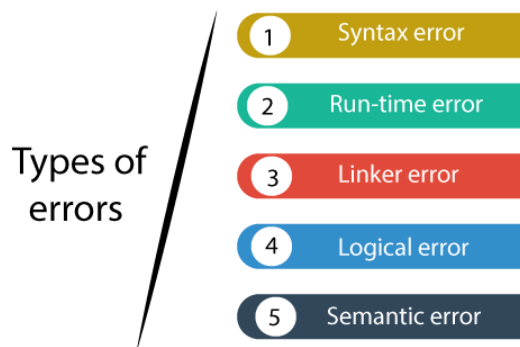
Programming Errors in C

Errors are the problems or the faults that occur in the program, which makes the behavior of the program abnormal, and experienced developers can also make these faults. Programming errors are also known as the bugs or faults, and the process of removing these bugs is known as debugging.

These errors are detected either during the time of compilation or execution. Thus, the errors must be removed from the program for the successful execution of the program.

There are mainly five types of errors exist in C programming:

- Syntax error
- Run-time error
- Linker error
- Logical error
- Semantic error



Syntax error

Syntax errors are also known as the compilation errors as they occurred at the compilation time, or we can say that the syntax errors are thrown by the compilers. These errors are mainly occurred due to the

mistakes while typing or do not follow the syntax of the specified programming language. These mistakes are generally made by beginners only because they are new to the language. These errors can be easily debugged or corrected.

For example:

If we want to declare the variable of type integer,

- `int a; // this is the correct form`
- `Int a; // this is an incorrect form.`

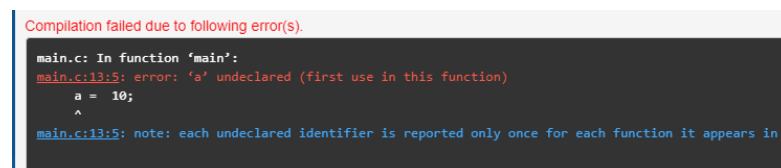
Commonly occurred syntax errors are:

- If we miss the parenthesis `{}` while writing the code.
- Displaying the value of a variable without its declaration.
- If we miss the semicolon `;` at the end of the statement.

Let's understand through an example.

```
#include <stdio.h>
int main()
{
    a = 10;
    printf("The value of a is : %d", a);
    return 0;
}
```

Output



```
Compilation failed due to following error(s).
main.c: In function 'main':
main.c:13:5: error: 'a' undeclared (first use in this function)
    a = 10;
    ^
main.c:13:5: note: each undeclared identifier is reported only once for each function it appears in
```

In the above output, we observe that the code throws the error that 'a' is undeclared. This error is nothing but the syntax error only.

There can be another possibility in which the syntax error can exist, i.e., if we make mistakes in the basic construct. Let's understand this scenario through an example.

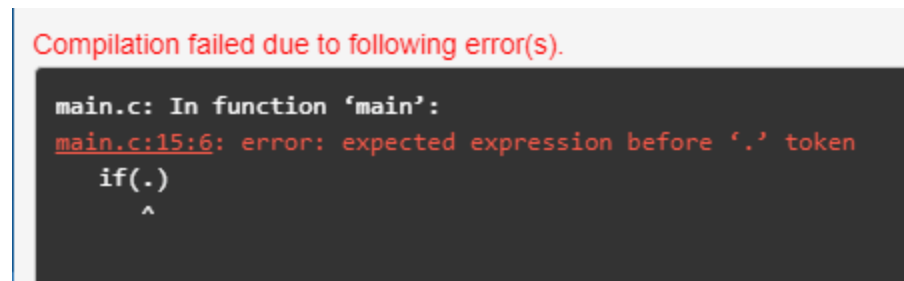
```
#include <stdio.h>
int main()
```

```
{
  int a=2;
  if(.) // syntax error
```

```
  printf("a is greater than 1");
  return 0;
}
```

In the above code, we put the (.) instead of condition in 'if', so this generates the syntax error as shown in the below screenshot.

Output



Run-time error

Sometimes the errors exist during the execution-time even after the successful compilation known as run-time errors. When the program is running, and it is not able to perform the operation is the main cause of the run-time error. The division by zero is the common example of the run-time error. These errors are very difficult to find, as the compiler does not point to these errors.

Let's understand through an example.

```
#include <stdio.h>
int main()
{
  int a=2;
  int b=2/0;
  printf("The value of b is : %d", b);
  return 0;
}
```

Output

```
main.c:14:12: warning: division by zero [-Wdiv-by-zero]
Floating point exception

...Program finished with exit code 136
Press ENTER to exit console.□
```

In the above output, we observe that the code shows the run-time error, i.e., division by zero.

Linker error

Linker errors are mainly generated when the executable file of the program is not created. This can be happened either due to the wrong function prototyping or usage of the wrong header file. For example, the main.c file contains the sub() function whose declaration and definition is done in some other file such as func.c. During the compilation, the compiler finds the sub() function in func.c file, so it generates two object files, i.e., main.o and func.o. At the execution time, if the definition of sub() function is not found in the func.o file, then the linker error will be thrown. The most common linker error that occurs is that we use Main() instead of main().

Let's understand through a simple example.

```
#include <stdio.h>
int main()
{
    int a=78;
    printf("The value of a is : %d", a);
    return 0;
}
```

Output

```
(.text+0x20): undefined reference to `main'
collect2: error: ld returned 1 exit status
```

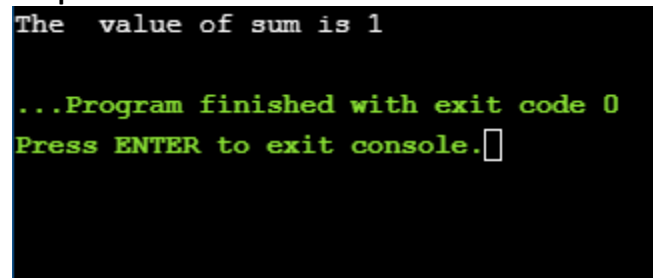
Logical error

The logical error is an error that leads to an undesired output. These errors produce the incorrect output, but they are error-free, known as logical errors. These types of mistakes are mainly done by beginners. The occurrence of these errors mainly depends upon the logical thinking of the developer. If the programmers sound logically good, then there will be fewer chances of these errors.

Let's understand through an example.

```
#include <stdio.h>
int main()
{
    int sum=0; // variable initialization
    int k=1;
    for(int i=1;i<=10;i++); // logical error, as we put the semicolon after loop
    {
        sum=sum+k;
        k++;
    }
    printf("The value of sum is %d", sum);
    return 0;
}
```

Output



```
The value of sum is 1

...Program finished with exit code 0
Press ENTER to exit console.█
```

In the above code, we are trying to print the sum of 10 digits, but we got the wrong output as we put the semicolon (;) after the for loop, so the inner statements of the for loop will not execute. This produces the wrong output.

Semantic error

Semantic errors are the errors that occurred when the statements are not understandable by the compiler.

The following can be the cases for the semantic error:

- Use of a un-initialized variable.

```
int i;
```

```
i=i+2;
```

- Type compatibility

```
int b = "ICS";
```

- Errors in expressions

```
int a, b, c;
```

```
a+b = c;
```

- Array index out of bound

```
int a[10];
```

```
a[10] = 34;
```

Let's understand through an example.

```
#include <stdio.h>
int main()
{
int a,b,c;
a=2;
b=3;
c=1;
a+b=c; // semantic error
return 0;
}
```

In the above code, we use the statement $a + b = c$, which is incorrect as we cannot use the two operands on the left-side.

Output

Compilation failed due to following error(s).

```
main.c: In function 'main':
main.c:17:6: error: lvalue required as left operand of assignment
  a+b=c;
    ^
```

Compile time vs Runtime

Compile-time and Runtime are the two programming terms used in the software development. Compile-time is the time at which the source code is converted into an executable code while the run time is the time at which the executable code is started running. Both the compile-time and runtime refer to different types of error.

Compile-time errors

Compile-time errors are the errors that occurred when we write the wrong syntax. If we write the wrong syntax or semantics of any programming language, then the compile-time errors will be thrown by the compiler. The compiler will not allow to run the program until all the errors are removed from the program. When all the errors are removed from the program, then the compiler will generate the executable file.

The compile-time errors can be:

- Syntax errors
- Semantic errors

Syntax errors

When the programmer does not follow the syntax of any programming language, then the compiler will throw the syntax error.

For example,

```
int a, b:
```

The above declaration generates the compile-time error as in C, every statement ends with the semicolon, but we put a colon (:) at the end of the statement.

Semantic errors

The semantic errors exist when the statements are not meaningful to the compiler.

For example,

```
a + b=c;
```


The above statement throws compile-time errors. In the above statement, we are assigning the value of 'c' to the summation of 'a' and 'b' which is not possible in C programming language as it can contain only one variable on the left of the assignment operator while right of the assignment operator can contain more than one variable.

The above statement can be re-written as:

```
c=a+b;
```

Runtime errors

The runtime errors are the errors that occur during the execution and after compilation. The examples of runtime errors are division by zero, etc. These errors are not easy to detect as the compiler does not point to these errors.

Let's look at the differences between compile-time and runtime:

Compile-time	Runtime
The compile-time errors are the errors which are produced at the compile-time, and they are detected by the compiler.	The runtime errors are the errors which are not generated by the compiler and produce an unpredictable result at the execution time.
In this case, the compiler prevents the code from execution if it detects an error in the program.	In this case, the compiler does not detect the error, so it cannot prevent the code from the execution.
It contains the syntax and semantic errors such as missing semicolon at the end of the statement.	It contains the errors such as division by zero, determining the square root of a negative number.

Example of Compile-time error

```
#include <stdio.h>
```

```
int main()
{
    int a=20;
    printf("The value of a is : %d",a):
    return 0;
}
```

In the above code, we have tried to print the value of 'a', but it throws an error. We put the colon at the end of the statement instead of a semicolon, so this code generates a compile-time error.

Output

Compilation failed due to following error(s).

```
main.c: In function 'main':
main.c:14:39: error: expected ';' before ':' token
    printf("The value of a is : %d",a):
                                   ^
```

Example of runtime error

```
#include <stdio.h>
int main()
{
    int a=20;
    int b=a/0; // division by zero
    printf("The value of b is : %d",b):
    return 0;
}
```

In the above code, we try to divide the value of 'b' by zero, and this throws a runtime error.

Output

Compilation failed due to following error(s).

```
main.c: In function 'main':
main.c:14:12: warning: division by zero [-Wdiv-by-zero]
    int b=a/0;
           ^
main.c:15:39: error: expected ';' before ':' token
    printf("The value of b is : %d",b):
                                   ^
```

Conditional Operator in C

The conditional operator is also known as a ternary operator. The conditional statements are the decision-making statements which depends upon the output of the expression. It is represented by two symbols, i.e., '?' and ':'.

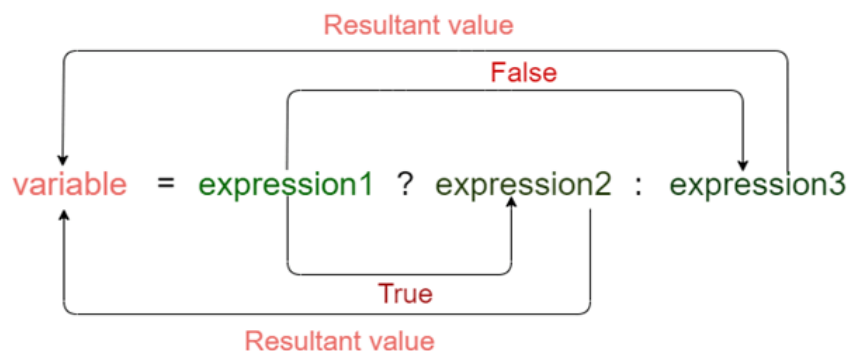
As conditional operator works on three operands, so it is also known as the ternary operator.

The behavior of the conditional operator is similar to the 'if-else' statement as 'if-else' statement is also a decision-making statement.

Syntax of a conditional operator

expression1? expression2: expression3;

The pictorial representation of the above syntax is shown below:



Meaning of the above syntax.

- In the above syntax, the expression1 is a Boolean condition that can be either true or false value.
- If the expression1 results into a true value, then the expression2 will execute.
- The expression2 is said to be true only when it returns a non-zero value.

- If the expression1 returns false value then the expression3 will execute.
- The expression3 is said to be false only when it returns zero value.

Let's understand the ternary or conditional operator through an example.

```
#include <stdio.h>
int main()
{
    int age; // variable declaration
    printf("Enter your age");
    scanf("%d",&age); // taking user input for age variable
    (age>=18)? (printf("eligible for voting")) : (printf("not eligible for voting")); // conditional operator
    return 0;
}
```

In the above code, we are taking input as the 'age' of the user. After taking input, we have applied the condition by using a conditional operator. In this condition, we are checking the age of the user. If the age of the user is greater than or equal to 18, then the statement1 will execute, i.e., (printf("eligible for voting")) otherwise, statement2 will execute, i.e., (printf("not eligible for voting")).

Let's observe the output of the above program.

If we provide the age of user below 18, then the output would be:

```
Enter your age 12
not eligible for voting

...Program finished with exit code 0
Press ENTER to exit console.
```

If we provide the age of user above 18, then the output would be:

```
Enter your age 24
eligible for voting

...Program finished with exit code 0
Press ENTER to exit console.
```

As we can observe from the above two outputs that if the condition is true, then the statement1 is executed; otherwise, statement2 will be executed.

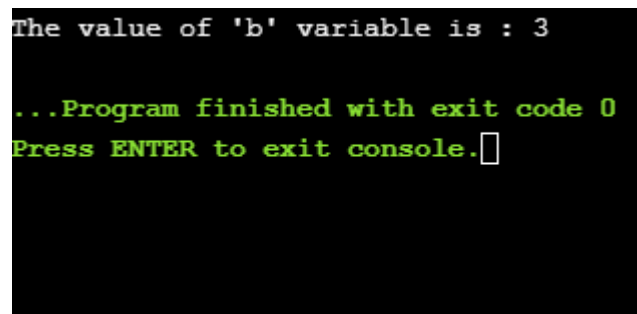
Till now, we have observed that how conditional operator checks the condition and based on condition, it executes the statements. Now, we will see how a conditional operator is used to assign the value to a variable.

Let's understand this scenario through an example.

```
#include <stdio.h>
int main()
{
    int a=5,b; // variable declaration
    b=((a==5)?(3):(2)); // conditional operator
    printf("The value of 'b' variable is : %d",b);
    return 0;
}
```

In the above code, we have declared two variables, i.e., 'a' and 'b', and assign 5 value to the 'a' variable. After the declaration, we are assigning value to the 'b' variable by using the conditional operator. If the value of 'a' is equal to 5 then 'b' is assigned with a 3 value otherwise 2.

Output



```
The value of 'b' variable is : 3
...Program finished with exit code 0
Press ENTER to exit console.█
```

The above output shows that the value of 'b' variable is 3 because the value of 'a' variable is equal to 5.

- As we know that the behavior of conditional operator and 'if-else' is similar but they have some differences. Let's look at their differences.
- A conditional operator is a single programming statement, while the 'if-else' statement is a programming block in which statements come under the parenthesis.

- A conditional operator can also be used for assigning a value to the variable, whereas the 'if-else' statement cannot be used for the assignment purpose.
- It is not useful for executing the statements when the statements are multiple, whereas the 'if-else' statement proves more suitable when executing multiple statements.
- The nested ternary operator is more complex and cannot be easily debugged, while the nested 'if-else' statement is easy to read and maintain.

Bitwise Operator in C

The bitwise operators are the operators used to perform the operations on the data at the bit-level.

When we perform the bitwise operations, then it is also known as bit-level programming. It consists of two digits, either 0 or 1. It is mainly used in numerical computations to make the calculations faster.

We have different types of bitwise operators in the C programming language. The following is the list of the bitwise operators:

Operator	Meaning of operator
&	Bitwise AND operator
	Bitwise OR operator
^	Bitwise exclusive OR operator
~	One's complement operator (unary operator)
<<	Left shift operator
>>	Right shift operator

Let's look at the truth table of the bitwise operators.

X	Y	X&Y	X Y	X^Y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	1

Bitwise AND operator

Bitwise AND operator is denoted by the single ampersand sign (&). Two integer operands are written on both sides of the (&) operator. If the corresponding bits of both the operands are 1, then the output of the bitwise AND operation is 1; otherwise, the output would be 0.

For example,

1. We have two variables a and b.
 - a = 6;
 - b = 4;
2. The binary representation of the above two variables are given below:
 - a = 0110
 - b = 0100
3. When we apply the bitwise AND operation in the above two variables, i.e., a&b, the output would be:
 - Result = 0100

As we can observe from the above result that bits of both the variables are compared one by one. If the bit of both the variables is 1 then the output would be 1, otherwise 0.

Let's understand the bitwise AND operator through the program.

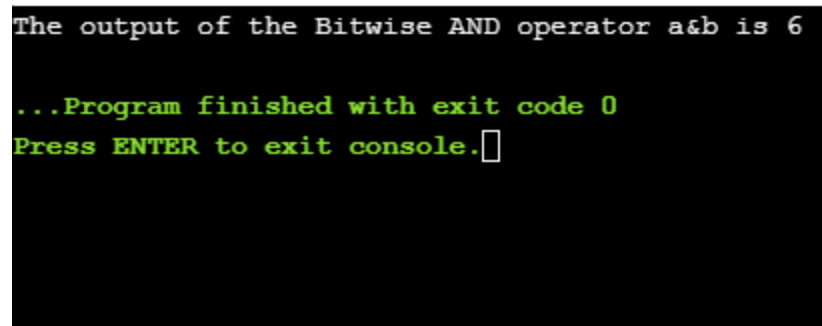
```
#include <stdio.h>
```

```
int main()
{
    int a=6, b=14; // variable declarations
    printf("The output of the Bitwise AND operator a&b is %d",a&b);
    return 0;
}
```

In the above code, we have created two variables, i.e., 'a' and 'b'. The values of 'a' and 'b' are 6 and 14 respectively. The binary value of 'a' and 'b' are 0110 and 1110, respectively. When we apply the AND operator between these two variables,

a AND b = 0110 && 1110 = 0110

Output



```
The output of the Bitwise AND operator a&b is 6
...Program finished with exit code 0
Press ENTER to exit console.█
```

Bitwise OR operator

The bitwise OR operator is represented by a single vertical sign (|). Two integer operands are written on both sides of the (|) symbol. If the bit value of any of the operand is 1, then the output would be 1, otherwise 0.

For example,

1. We consider two variables,
 - a = 23;
 - b = 10;
2. The binary representation of the above two variables would be:
 - a = 0001 0111
 - b = 0000 1010

3. When we apply the bitwise OR operator in the above two variables, i.e., $a|b$, then the output would be:

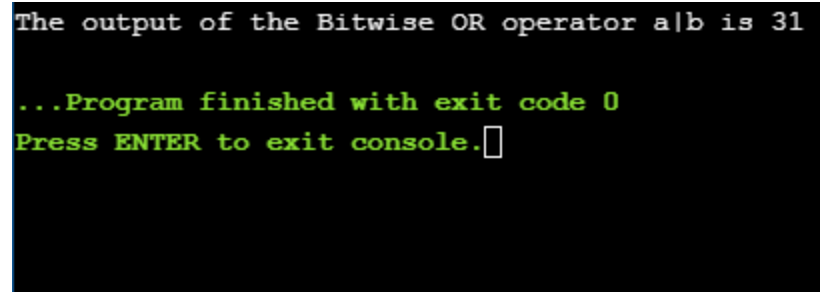
- Result = 0001 1111

As we can observe from the above result that the bits of both the operands are compared one by one; if the value of either bit is 1, then the output would be 1 otherwise 0.

Let's understand the bitwise OR operator through a program.

```
#include <stdio.h>
int main()
{
    int a=23,b=10; // variable declarations
    printf("The output of the Bitwise OR operator a|b is %d",a|b);
    return 0;
}
```

Output



```
The output of the Bitwise OR operator a|b is 31
...Program finished with exit code 0
Press ENTER to exit console.█
```

Bitwise exclusive OR operator

Bitwise exclusive OR operator is denoted by (^) symbol. Two operands are written on both sides of the exclusive OR operator. If the corresponding bit of any of the operand is 1 then the output would be 1, otherwise 0.

For example,

- We consider two variables a and b,

a = 12;

b = 10;

- The binary representation of the above two variables would be:

a = 0000 1100

b = 0000 1010

- When we apply the bitwise exclusive OR operator in the above two variables ($a \oplus b$), then the result would be:

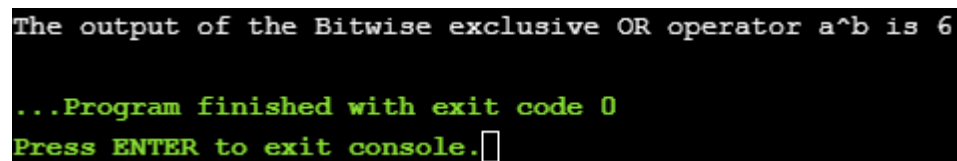
Result = 0000 1110

- As we can observe from the above result that the bits of both the operands are compared one by one; if the corresponding bit value of any of the operand is 1, then the output would be 1 otherwise 0.

Let's understand the bitwise exclusive OR operator through a program.

```
#include <stdio.h>
int main()
{
    int a=12,b=10; // variable declarations
    printf("The output of the Bitwise exclusive OR operator a^b is %d",a^b);
    return 0;
}
```

Output



```
The output of the Bitwise exclusive OR operator a^b is 6
...Program finished with exit code 0
Press ENTER to exit console.█
```

Bitwise complement operator

The bitwise complement operator is also known as one's complement operator. It is represented by the symbol tilde (~). It takes only one operand or variable and performs complement operation on an operand. When we apply the complement operation on any bits, then 0 becomes 1 and 1 becomes 0.

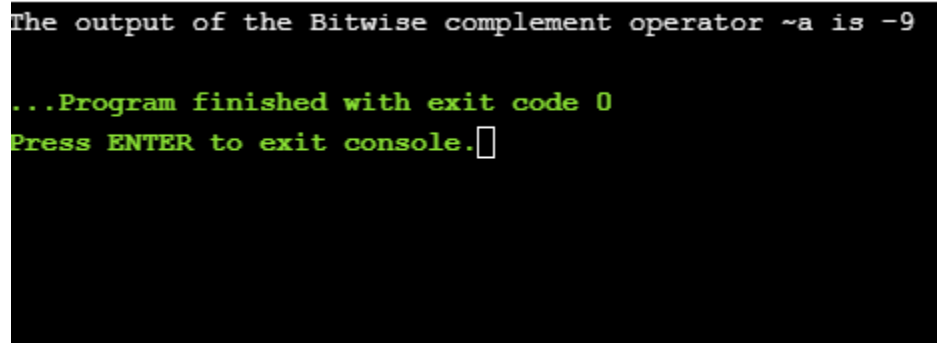
For example,

- If we have a variable named 'a',
a = 8;
- The binary representation of the above variable is given below:
a = 1000
- When we apply the bitwise complement operator to the operand, then the output would be:
Result = 0111
- As we can observe from the above result that if the bit is 1, then it gets changed to 0 else 1.

Let's understand the complement operator through a program.

```
#include <stdio.h>
int main()
{
    int a=8; // variable declarations
    printf("The output of the Bitwise complement operator ~a is %d",~a);
    return 0;
}
```

Output



```
The output of the Bitwise complement operator ~a is -9
...Program finished with exit code 0
Press ENTER to exit console.█
```

Bitwise shift operators

Two types of bitwise shift operators exist in C programming. The bitwise shift operators will shift the bits either on the left-side or right-side. Therefore, we can say that the bitwise shift operator is divided into two categories:

- Left-shift operator
- Right-shift operator

Left-shift operator

It is an operator that shifts the number of bits to the left-side.

Syntax of the left-shift operator is given below:

operand << n

Where,

Operand is an integer expression on which we apply the left-shift operation.

n is the number of bits to be shifted.

In the case of Left-shift operator, 'n' bits will be shifted on the left-side. The 'n' bits on the left side will be popped out, and 'n' bits on the right-side are filled with 0.

For example,

- Suppose we have a statement:

int a = 5;

- The binary representation of 'a' is given below:

a = 0101

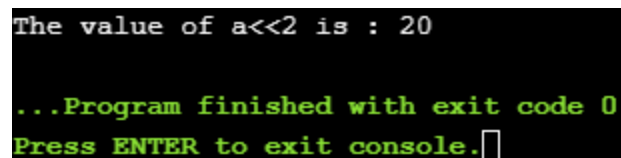
- If we want to left-shift the above representation by 2, then the statement would be:

a << 2;

- $0101 \ll 2 = 00010100$

```
#include <stdio.h>
int main()
{
    int a=5; // variable initialization
    printf("The value of a<<2 is : %d ", a<<2);
    return 0;
}
```

Output



```
The value of a<<2 is : 20

...Program finished with exit code 0
Press ENTER to exit console.█
```

Right-shift operator

It is an operator that shifts the number of bits to the right side.

Syntax of the right-shift operator is given below:

1. Operand >> n;

Where,

Operand is an integer expression on which we apply the right-shift operation.

N is the number of bits to be shifted.

In the case of the right-shift operator, 'n' bits will be shifted on the right-side. The 'n' bits on the right-side will be popped out, and 'n' bits on the left-side are filled with 0.

For example,

Suppose we have a statement,
int a = 7;

The binary representation of the above variable would be:

a = 0111

If we want to right-shift the above representation by 2, then the statement would be:

a>>2;

0000 0111 >> 2 = 0000 0001

```
#include <stdio.h>
int main()
{
    int a=7; // variable initialization
    printf("The value of a>>2 is : %d ", a>>2);
    return 0;
}
```

Output

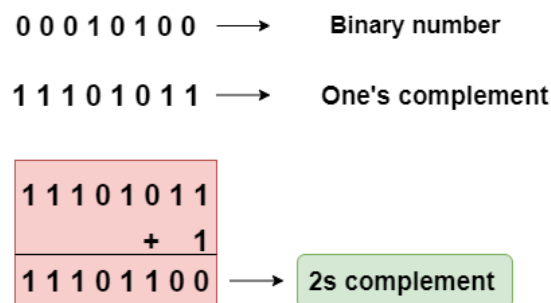
```
The value of a>>2 is : 1

...Program finished with exit code 0
Press ENTER to exit console.
```

What is the 2s complement in C?

The 2s complement in C is generated from the 1s complement in C. As we know that the 1s complement of a binary number is created by transforming bit 1 to 0 and 0 to 1; the 2s complement of a binary number is generated by adding one to the 1s complement of a binary number.

In short, we can say that the 2s complement in C is defined as the sum of the one's complement in C and one.



In the above figure, the binary number is equal to 00010100, and its one's complement is calculated by transforming the bit 1 to 0 and 0 to 1 vice versa. Therefore, one's complement becomes 11101011. After calculating one's complement, we calculate the two's complement by adding 1 to the one's complement, and its result is 11101100.

Let's create a program of 2s complement.

```
#include <stdio.h>
int main()
{
    int n; // variable declaration
    printf("Enter the number of bits do you want to enter :");
    scanf("%d",&n);
    char binary[n+1]; // binary array declaration;
```

```

char onescomplement[n+1]; // onescomplement array declaration
char twoscomplement[n+1]; // twoscomplement array declaration
int carry=1; // variable initialization
printf("\nEnter the binary number : ");
scanf("%s", binary);
printf("%s", binary);
printf("\nThe ones complement of the binary number is :");

// Finding onescomplement in C
for(int i=0;i<n;i++)
{
    if(binary[i]=='0')
        onescomplement[i]='1';
    else if(binary[i]=='1')
        onescomplement[i]='0';
}
onescomplement[n]='\0';
printf("%s",onescomplement);

printf("\nThe twos complement of a binary number is : ");

// Finding twoscomplement in C
for(int i=n-1; i>=0; i--)
{
    if(onescomplement[i] == '1' && carry == 1)
    {
        twoscomplement[i] = '0';
    }
    else if(onescomplement[i] == '0' && carry == 1)
    {
        twoscomplement[i] = '1';
        carry = 0;
    }
    else
    {
        twoscomplement[i] = onescomplement[i];
    }
}
twoscomplement[n]='\0';
printf("%s",twoscomplement);
return 0;
}

```

Output

```

Enter the number of bits do you want to enter :8

Enter the binary number : 01001010
01001010
The ones complement of the binary number is :10110101
The twos complement of a binary number is : 10110110

...Program finished with exit code 0
Press ENTER to exit console.

```

Analysis of the above program,

- First, we input the number of bits, and it gets stored in the 'n' variable.
- After entering the number of bits, we declare character array, i.e., char binary[n+1], which holds the binary number. The 'n' is the number of bits which we entered in the previous step; it basically defines the size of the array.
- We declare two more arrays, i.e., onescomplement[n+1], and twoscomplement[n+1]. The onescomplement[n+1] array holds the ones complement of a binary number while the twoscomplement[n+1] array holds the two's complement of a binary number.
- Initialize the carry variable and assign 1 value to this variable.
- After declarations, we input the binary number.
- Now, we simply calculate the one's complement of a binary number. To do this, we create a loop that iterates throughout the binary array, for(int i=0;i<n;i++). In for loop, the condition is checked whether the bit is 1 or 0. If the bit is 1 then onescomplement[i]=0 else onescomplement[i]=1. In this way, one's complement of a binary number is generated.
- After calculating one's complement, we generate the 2s complement of a binary number. To do this, we create a loop that iterates from the last element to the starting element. In for loop, we have three conditions:
 - If the bit of onescomplement[i] is 1 and the value of carry is 1 then we put 0 in twocomplement[i].
 - If the bit of onescomplement[i] is 0 and the value of carry is 1 then we put 1 in twoscomplement[i] and 0 in carry.

- If the above two conditions are false, then `onescomplement[i]` is equal to `twoscomplement[i]`.

Introduction to Control Statements in C

In C, the control flows from one instruction to the next instruction until now in all programs. This control flow from one command to the next is called sequential control flow. Nonetheless, in most C programs the programmer may want to skip instructions or repeat a set of instructions repeatedly when writing logic. This can be referred to as sequential control flow. The declarations in C let programmers make such decisions which are called decision-making or control declarations.

Types of Control Statements in C

C also supports an unconditional set of branching statements that transfer the control to another location in the program. Selection declarations in C.

- If statements
- Switch Statement
- Conditional Operator Statement
- Goto Statement
- Loop Statements

1. If Statements

If statement enables the programmer to choose a set of instructions, based on a condition. When the condition is evaluated to true, a set of instructions will be executed and a different set of instructions will be executed when the condition is evaluated to false. We have 4 types of if Statement which are:

- If..else
- Nested if
- Else if ladder
- Simple if or null else
- Null else or Simple else

if else Statement

The if-else statement in C is used to perform the operations based on some specific condition. The operations specified in if block are executed if and only if the given condition is true.

There are the following variants of if statement in C language.

- If statement
- If-else statement
- If else-if ladder
- Nested if

If Statement

The if statement is used to check some given condition and perform some operations depending upon the correctness of that condition. It is mostly used in the scenario where we need to perform the different operations for the different conditions. The syntax of the if statement is given below.

```
if(expression){  
  
    //code to be executed  
  
}
```

Example

```
#include<stdio.h>  
  
int main(){  
    int number=0;  
    printf("Enter a number:");  
    scanf("%d",&number);  
    if(number%2==0){  
        printf("%d is even number",number);  
    }  
    return 0;  
}
```

Output

Enter a number:4

4 is even number

enter a number:5

Program to find the largest number of the three.

```
#include <stdio.h>
int main()
{
    int a, b, c;
    printf("Enter three numbers?");
    scanf("%d %d %d",&a,&b,&c);
    if(a>b && a>c)
    {
        printf("%d is largest",a);
    }
    if(b>a && b > c)
    {
        printf("%d is largest",b);
    }
    if(c>a && c>b)
    {
        printf("%d is largest",c);
    }
    if(a == b && a == c)
    {
        printf("All are equal");
    }
}
```

Output

Enter three numbers?

12 23 34

34 is largest

If-else Statement

The if-else statement is used to perform two operations for a single condition. The if-else statement is an extension to the if statement using which, we can perform two different operations, i.e., one is for the correctness of that condition, and the other is for the incorrectness of the condition. Here, we must notice that if and else block cannot be executed simultaneously. Using if-else statement is always

preferable since it always invokes an otherwise case with every if condition. The syntax of the if-else statement is given below.

```
if(expression){  
  
    //code to be executed if condition is true  
  
}else{  
  
    //code to be executed if condition is false  
  
}
```

Let's see the simple example to check whether a number is even or odd using if-else statement in C language.

```
#include<stdio.h>  
int main(){  
    int number=0;  
    printf("enter a number:");  
    scanf("%d",&number);  
    if(number%2==0){  
        printf("%d is even number",number);  
    }  
    else{  
        printf("%d is odd number",number);  
    }  
    return 0;  
}
```

Output

enter a number:4

4 is even number

enter a number:5

5 is odd number

Program to check whether a person is eligible to vote or not.

```
#include <stdio.h>
int main()
{
    int age;
    printf("Enter your age?");
    scanf("%d",&age);
    if(age>=18)
    {
        printf("You are eligible to vote...");
    }
    else
    {
        printf("Sorry ... you can't vote");
    }
}
```

Output

Enter your age?18

You are eligible to vote...

Enter your age?13

Sorry ... you can't vote

If else-if ladder Statement

The if-else-if ladder statement is an extension to the if-else statement. It is used in the scenario where there are multiple cases to be performed for different conditions. In if-else-if ladder statement, if a condition is true then the statements defined in the if block will be executed, otherwise if some other condition is true then the statements defined in the else-if block will be executed, at the last if none of the condition is true then the statements defined in the else block will be executed. There are multiple else-if blocks possible. It is similar to the switch case statement where the default is executed instead of else block if none of the cases is matched.

```
if(condition1){
    //code to be executed if condition1 is true
}else if(condition2){
    //code to be executed if condition2 is true
}
```

```
else if(condition3){  
    //code to be executed if condition3 is true  
}  
...  
else{  
    //code to be executed if all the conditions are false  
}
```

Example

```
#include<stdio.h>  
  
int main(){  
    int number=0;  
    printf("enter a number:");  
    scanf("%d",&number);  
    if(number==10){  
        printf("number is equals to 10");  
    }  
    else if(number==50){  
        printf("number is equal to 50");  
    }  
    else if(number==100){  
        printf("number is equal to 100");  
    }  
    else{  
        printf("number is not equal to 10, 50 or 100");  
    }  
    return 0;  
}
```

Output

enter a number:4

number is not equal to 10, 50 or 100

enter a number:50

number is equal to 50

Program to calculate the grade of the student according to the specified marks.

```
#include <stdio.h>

int main()
{
    int marks;
    printf("Enter your marks?");
    scanf("%d",&marks);
    if(marks > 85 && marks <= 100)
    {
        printf("Congrats ! you scored grade A ...");
    }
    else if (marks > 60 && marks <= 85)
    {
        printf("You scored grade B + ...");
    }
    else if (marks > 40 && marks <= 60)
    {
        printf("You scored grade B ...");
    }
    else if (marks > 30 && marks <= 40)
    {
        printf("You scored grade C ...");
    }
    else
    {
        printf("Sorry you are fail ...");
    }
}
```

Output

Enter your marks?10

Sorry you are fail ...

Enter your marks?40

You scored grade C ...

Enter your marks?90

Congrats ! you scored grade A ...

If...else Statement

In this statement, there are two types of statements execute. First, if the condition is true first statement will execute if the condition is false second condition will be executed.

Syntax:

```
if(condition)
{
    statement(s);
}
else
{
    statement(s)
}
statement
```

- **Nested if**

If the condition is evaluated to true in the first if statement, then the condition in the second if statement is evaluated and so on.

Syntax:

```
if(condition)
{
    if(condition)
    {
        statement(s);
    }
    else
    {
        statement(s)
    }
}
```



```
}
```

else if Ladder

The corresponding array of instructions is executed when the first condition is correct. If the condition is incorrect, the next condition will be verified. If all the specifications fail, the default block statements will be executed. The remainder of the ladder can be shown as shown below.

Syntax:

```
if(condition)
{
statement(s);
}
else if(condition)
{
statement(s);
}
else if(condition)
{
statement(s)
}
...
else
{
statement(s)
}
statement(s);
```

Null else or Simple else

If the programmer can execute or skip a set of instructions based on the condition value. The simple one-way statement is selected. A set of statements is carried out if the condition is true. If the condition is false, the control will proceed with the following declaration after the if declaration. Simple else statement:

Syntax:

```
if(condition)
{
statement(s);
```

```
}  
statement(s);
```

2. Switch Statement

C offers a selection statement in several ways as if the program becomes less readable when the number of conditions increases. C has a multi-way selection statement called the switch statement that is easy to understand to resolve this problem. The switch declaration is easy to understand if more than 3 alternatives exist. The command switches between the blocks based on the expression value. Each block will have a corresponding value.

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. Here, We can define various statements in the multiple cases for the different values of a single variable.

The syntax of switch statement in c language is given below:

Syntax:

```
switch(expression)  
{  
  
    case label1:  
        statement(S);  
        break;  
  
    case label2:  
        statement(S);  
        break;  
  
    case label3;  
        statement(s);  
        break;  
    ....  
    case labelN:  
        statement(s);  
        break;  
  
    default:  
        statement(s);  
        break;  
}
```

Using the case keyword every block is shown and the block label follows the case keyword. The default block and the break statement are optional in a switch statement.

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.

Let's try to understand it by the examples. We are assuming that there are following variables.

int x,y,z;

char a,b;

float f;

Valid Switch	Invalid Switch	Valid Case	Invalid Case
switch(x)	switch(f)	case 3;	case 2.5;
switch(x>y)	switch(x+2.5)	case 'a';	case x;
switch(a+b-2)		case 1+2;	case x+2;
switch(func(x,y))		case 'x'>'y';	case 1,2,3;

Functioning of switch case statement

First, the integer expression specified in the switch statement is evaluated. This value is then matched one by one with the constant values given in the different cases. If a match is found, then all the statements specified in that case are executed along with the all the cases present after that case including the default statement. No two cases can have similar values. If the matched case contains a break statement, then all the cases present after that will be skipped, and the control comes out of the switch. Otherwise, all the cases following the matched case will be executed.

Let's see a simple example of c language switch statement.

```
#include<stdio.h>

int main(){
int number=0;
printf("enter a number:");
scanf("%d",&number);
switch(number){
case 10:
printf("number is equals to 10");
break;
case 50:
printf("number is equal to 50");
break;
case 100:
printf("number is equal to 100");
break;
default:
printf("number is not equal to 10, 50 or 100");
}
return 0;
}
```

Output

enter a number:4

number is not equal to 10, 50 or 100

enter a number:50

number is equal to 50

Switch case example 2

```
#include <stdio.h>
int main()
{
    int x = 10, y = 5;
    switch(x>y && x+y>0)
    {
        case 1:
            printf("hi");
            break;
        case 0:
            printf("bye");
            break;
        default:
            printf(" Hello bye ");
    }
}
```

Output

hi

C Switch statement is fall-through

In C language, the switch statement is fall through; it means if you don't use a break statement in the switch case, all the cases after the matching case will be executed.

Let's try to understand the fall through state of switch statement by the example given below.

```
#include<stdio.h>
int main(){
    int number=0;

    printf("enter a number:");
    scanf("%d",&number);
```

```

switch(number){
case 10:
printf("number is equal to 10\n");
case 50:
printf("number is equal to 50\n");
case 100:
printf("number is equal to 100\n");
default:
printf("number is not equal to 10, 50 or 100");
}
return 0;
}

```

Output

```

enter a number:10
number is equal to 10
number is equal to 50
number is equal to 100
number is not equal to 10, 50 or 100

```

Output

```

enter a number:50
number is equal to 50
number is equal to 100
number is not equal to 10, 50 or 100

```

Nested switch case statement

We can use as many switch statement as we want inside a switch statement. Such type of statements is called nested switch case statements. Consider the following example.

```

#include <stdio.h>
int main () {

```

```

int i = 10;
int j = 20;

switch(i) {

    case 10:
        printf("the value of i evaluated in outer switch: %d\n",i);
    case 20:
        switch(j) {
            case 20:
                printf("The value of j evaluated in nested switch: %d\n",j);
            }
        }

    printf("Exact value of i is : %d\n", i );
    printf("Exact value of j is : %d\n", j );

    return 0;
}

```

Output

```

the value of i evaluated in outer switch: 10

The value of j evaluated in nested switch: 20

Exact value of i is : 10

Exact value of j is : 20

```

if-else vs switch

What is an if-else statement?

An if-else statement in C programming is a conditional statement that executes a different set of statements based on the condition that is true or false. The 'if' block will be executed only when the specified condition is true, and if the specified condition is false, then the else block will be executed.

Syntax of if-else statement is given below:

```
if(expression)
{
    // statements;
}
else
{
    // statements;
}
```

What is a switch statement?

A switch statement is a conditional statement used in C programming to check the value of a variable and compare it with all the cases. If the value is matched with any case, then its corresponding statements will be executed. Each case has some name or number known as the identifier. The value entered by the user will be compared with all the cases until the case is found. If the value entered by the user is not matched with any case, then the default statement will be executed.

Syntax of the switch statement is given below:

```
switch(expression)
{
    case constant 1:
        // statements;
        break;
    case constant 2:
        // statements;
        break;
    case constant n:
        // statements;
        break;
    default:
        // statements;
}
```

Similarity b/w if-else and switch

Both the if-else and switch are the decision-making statements. Here, decision-making statements mean that the output of the expression will decide which statements are to be executed.

Differences b/w if-else and switch statement

The following are the differences between if-else and switch statement are:



Definition

- **if-else** : Based on the result of the expression in the 'if-else' statement, the block of statements will be executed. If the condition is true, then the 'if' block will be executed otherwise 'else' block will execute.
- **Switch statement** : The switch statement contains multiple cases or choices. The user will decide the case, which is to execute.

Expression

- **If-else** : It can contain a single expression or multiple expressions for multiple choices. In this, an expression is evaluated based on the range of values or conditions. It checks both equality and logical expressions.
- **Switch** : It contains only a single expression, and this expression is either a single integer object or a string object. It checks only equality expression.

Evaluation

- **If-else** : An if-else statement can evaluate almost all the types of data such as integer, floating-point, character, pointer, or Boolean.
- **Switch** : A switch statement can evaluate either an integer or a character.

Sequence of Execution

- **If-else** : In the case of 'if-else' statement, either the 'if' block or the 'else' block will be executed based on the condition.
- **Switch** : In the case of the 'switch' statement, one case after another will be executed until the break keyword is not found, or the default statement is executed.

Default Execution

- **If-else** : If the condition is not true within the 'if' statement, then by default, the else block statements will be executed.
- **Switch** : If the expression specified within the switch statement is not matched with any of the cases, then the default statement, if defined, will be executed.

Values

- **If-else** : Values are based on the condition specified inside the 'if' statement. The value will decide either the 'if' or 'else' block is to be executed.
- **Switch** : In this case, value is decided by the user. Based on the choice of the user, the case will be executed.

Use

- **If-else** : It evaluates a condition to be true or false.
- **Switch** : A switch statement compares the value of the variable with multiple cases. If the value is matched with any of the cases, then the block of statements associated with this case will be executed.

Editing

- **If-else** : Editing in 'if-else' statement is not easy as if we remove the 'else' statement, then it will create the havoc.
- **Switch** : Editing in switch statement is easier as compared to the 'if-else' statement. If we remove any of the cases from the switch, then it will not interrupt the execution of other cases. Therefore, we can say that the switch statement is easy to modify and maintain.

Speed

- **If-else** : If the choices are multiple, then the speed of the execution of 'if-else' statements is slow.
- **Switch** : The case constants in the switch statement create a jump table at the compile time. This jump table chooses the path of the execution based on the value of the expression. If we have a multiple choice, then the execution of the switch statement will be much faster than the equivalent logic of 'if-else' statement.

Let's summarize the above differences in a tabular form.

	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	If we have multiple choices then the switch statement is the best option as the speed of the

	the speed of the execution will be slow.	execution will be much higher than 'if-else'.
--	--	---

3. Conditional Operator Statement

C language provides an unusual operator, which is represented as a conditional operator.

Syntax:

(condition)? expr1: expr2

Expr1 is executed when the condition is valid. Then Expr2 will be executed if the statement is incorrect.

4. goto Statement

goto statement is known for jumping control statements. It is used to transfer the control of the program from one block to another block. goto keyword is used to declare the goto statement.

Syntax:

goto labelname;

labelname;

In the above syntax, goto is a keyword that is used to transfer the control to the labelname. labelname is a variable name. In this case, the goto will transfer the control of the program to the labelname and statements followed by the labelname will be executed.

5. Loop Statements

The programmer may want to repeat several instructions when writing C programs until some requirements are met. To that end, C makes looping declarations for decision-making. The looping can be defined as repeating the same process multiple times until a specific condition satisfies. There are three types of loops used in the C language.

Why use loops in C language?

The looping simplifies the complex problems into the easy ones. It enables us to alter the flow of the program so that instead of writing the same code again and again, we can repeat the same code for a finite number of times. For example, if we need to print the first 10 natural numbers then, instead of using the printf statement 10 times, we can print inside a loop which runs up to 10 iterations.

Advantage of loops in C

- 1) It provides code reusability.
- 2) Using loops, we do not need to write the same code again and again.
- 3) Using loops, we can traverse over the elements of data structures (array or linked lists).

Types of C Loops

There are three types of loops in C language that is given below:

- For Loop
- While Loop
- Do While Loop

For Loop

In the For loop, the initialization statement is executed only one time. After that, the condition is checked and if the result of condition is true it will execute the loop. If it is false, then for loop is terminated. However, the result of condition evaluation is true, statements inside the body of for loop gets executed, and the expression is updated. After that, the condition is checked again. This process goes on until the result of the condition becomes false. When the condition is false, the loop terminates. The for loop in C language is used to iterate the statements or a part of the program several times. It is frequently used to traverse the data structures like the array and linked list.

Syntax of for loop in C

The syntax of for loop in c language is given below:

```
for(Expression 1; Expression 2; Expression 3){  
  
//code to be executed  
  
}
```

C for loop Examples

Let's see the simple program of for loop that prints table of 1.

```
#include<stdio.h>
int main(){
int i=0;
for(i=1;i<=10;i++){
printf("%d \n",i);
}
return 0;
}
```

Output

```
1
2
3
4
5
6
7
8
9
10
```

C Program: Print table for the given number using C for loop

```
#include<stdio.h>
int main(){
int i=1,number=0;
printf("Enter a number: ");
scanf("%d",&number);
for(i=1;i<=10;i++){
printf("%d \n",(number*i));
}
return 0;
}
```

Output

Enter a number: 2

2

4

6

8

10

12

14

16

18

20

Enter a number: 1000

1000

2000

3000

4000

5000

6000

7000

8000

9000

10000

Properties of Expression 1

The expression represents the initialization of the loop variable.

We can initialize more than one variable in Expression 1.

Expression 1 is optional.

In C, we can not declare the variables in Expression 1. However, It can be an exception in some compilers.

Example

```
#include <stdio.h>
int main()
{
    int a,b,c;
    for(a=0,b=12,c=23;a<2;a++)
    {
        printf("%d ",a+b+c);
    }
}
```

Output

35 36

Example

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

Output

1 2 3 4

Properties of Expression 2

- Expression 2 is a conditional expression. It checks for a specific condition to be satisfied. If it is not, the loop is terminated.

- Expression 2 can have more than one condition. However, the loop will iterate until the last condition becomes false. Other conditions will be treated as statements.
- Expression 2 is optional.
- Expression 2 can perform the task of expression 1 and expression 3. That is, we can initialize the variable as well as update the loop variable in expression 2 itself.
- We can pass zero or non-zero value in expression 2. However, in C, any non-zero value is true, and zero is false by default.

Example

```
#include <stdio.h>

int main()
{
    int i;
    for(i=0;i<=4;i++)
    {
        printf("%d ",i);
    }
}
```

output

0 1 2 3 4

Example

```
#include <stdio.h>

int main()
{
    int i,j,k;

    for(i=0,j=0,k=0;i<4,k<8,j<10;i++)
    {
        printf("%d %d %d\n",i,j,k);

        j+=2;
    }
}
```

```
        k+=3;
    }
}
```

Output

```
0 0 0
1 2 3
2 4 6
3 6 9
4 8 12
```

Example

```
#include <stdio.h>

int main()
{
    int i;

    for(i=0;;i++)
    {
        printf("%d",i);
    }
}
```

Output

infinite loop

- Properties of Expression 3
- Expression 3 is used to update the loop variable.
- We can update more than one variable at the same time.
- Expression 3 is optional.

Example

```
#include<stdio.h>
```

```

void main ()
{
    int i=0,j=2;

    for(i = 0;i<5;i++,j=j+2)
    {
        printf("%d %d\n",i,j);
    }
}

```

Output

```

0 2
1 4
2 6
3 8
4 10

```

Loop body

The braces {} are used to define the scope of the loop. However, if the loop contains only one statement, then we don't need to use braces. A loop without a body is possible. The braces work as a block separator, i.e., the value variable declared inside for loop is valid only for that block and not outside. Consider the following example.

```

#include<stdio.h>

void main ()
{
    int i;

    for(i=0;i<10;i++)
    {
        int i = 20;
    }
}

```

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

Output

20 20 20 20 20 20 20 20 20 20

Infinite for loop in C

To make a for loop infinite, we need not give any expression in the syntax. Instead of that, we need to provide two semicolons to validate the syntax of the for loop. This will work as an infinite for loop.

```
#include<stdio.h>  
  
void main ()  
{  
    for(;;)  
    {  
        printf("welcome to ICS");  
    }  
}
```

If you run this program, you will see above statement infinite times.

while Loop

In C, the while loop is a guided entry loop. The body of the while loops is only performed if the condition is valid. The loop structure is not executed if the condition scores to incorrect. The while loops are usually used when several instructions have to be repeated for an indefinite time.

While loop is also known as a pre-tested loop. In general, a while loop allows a part of the code to be executed multiple times depending upon a given boolean condition. It can be viewed as a repeating if statement. The while loop is mostly used in the case where the number of iterations is not known in advance.

Syntax of while loop in C language

The syntax of while loop in c language is given below:

```
while(condition){  
  
//code to be executed  
  
}
```

Example of the while loop in C language

Let's see the simple program of while loop that prints table of 1.

```
#include<stdio.h>
```

```
int main(){
```

```
int i=1;
```

```
while(i<=10){
```

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

```
i++;
```

```
}
```

```
return 0;
```

```
}
```

Output

1

2

3

4

5

6

7

8

9

10

Program to print table for the given number using while loop in C

```
#include<stdio.h>

int main(){

int i=1,number=0,b=9;

printf("Enter a number: ");

scanf("%d",&number);

while(i<=10){

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

i++;

}

return 0;

}
```

Output

Enter a number: 50

50

100

150

200

250

300

350

400

450

500

Enter a number: 100

100

200

300

400

500

600

700

800

900

1000

Properties of while loop

- A conditional expression is used to check the condition. The statements defined inside the while loop will repeatedly execute until the given condition fails.
- The condition will be true if it returns 0. The condition will be false if it returns any non-zero number.
- In while loop, the condition expression is compulsory.
- Running a while loop without a body is possible.
- We can have more than one conditional expression in while loop.
- If the loop body contains only one statement, then the braces are optional.

Example

```
#include<stdio.h>
void main ()
{
    int j = 1;
    while(j+=2,j<=10)
    {
        printf("%d ",j);
```

```

    }
    printf("%d",j);
}

```

Output

3 5 7 9 11

Example

```

#include<stdio.h>
void main ()
{
    while()
    {
        printf("hello ICS");
    }
}

```

Output

compile time error: while loop can't be empty

Example

```

#include<stdio.h>
void main ()
{
    int x = 10, y = 2;
    while(x+y-1)
    {
        printf("%d %d",x--,y--);
    }
}

```

infinite loop

Infinitive while loop in C

If the expression passed in while loop results in any non-zero value then the loop will run the infinite number of times.

```

while(1){

//statement

}

```


do while Loop

Unlike while loop, the body of the do is the difference between while and ... while loop is guaranteed to be done once at a time.

Syntax:

```
do
{
//statements inside the loop

}
while(condition);
```

do-while loop in C

The do-while loop continues until a given condition satisfies. It is also called post tested loop. It is used when it is necessary to execute the loop at least once (mostly menu driven programs).

The syntax of do-while loop in c language is given below:

```
do{

//code to be executed

}while(condition);
```

Example 1

```
#include<stdio.h>
#include<stdlib.h>
void main ()
{
    char c;
    int choice,dummy;
    do{
        printf("\n1. Print Hello\n2. Print ICS\n3. Exit\n");
        scanf("%d",&choice);
        switch(choice)
```

```

{
    case 1 :
        printf("Hello");
        break;
    case 2:
        printf("ICS");
        break;
    case 3:
        exit(0);
        break;
    default:
        printf("please enter valid choice");
}
printf("do you want to enter more?");
scanf("%d",&dummy);
scanf("%c",&c);
}while(c=='y');
}

```

Output

1. Print Hello
2. Print ICS
3. Exit

1

Hello

do you want to enter more?

y

1. Print Hello
2. Print ICS
3. Exit

2

ICS

do you want to enter more?

n

do while example

There is given the simple program of c language do while loop where we are printing the table of 1.

```
#include<stdio.h>
int main(){
int i=1;
do{
printf("%d \n",i);
i++;
}while(i<=10);
return 0;
}
```

Output

```
1
2
3
4
5
6
7
8
9
10
```

Program to print table for the given number using do while loop

```
#include<stdio.h>
int main(){
int i=1,number=0;
printf("Enter a number: ");
scanf("%d",&number);
do{
```

```
printf("%d \n", (number*i));  
  
i++;  
  
}while(i<=10);  
  
return 0;  
  
}
```

Output

Enter a number: 5

5

10

15

20

25

30

35

40

45

50

Enter a number: 10

10

20

30

40

50

60

70

80

90

100

Infinitive do while loop

The do-while loop will run infinite times if we pass any non-zero value as the conditional expression.

```
do{  
  
//statement  
  
}while(1);
```

Nested Loops in C

C supports nesting of loops in C. Nesting of loops is the feature in C that allows the looping of statements inside another loop. Let's observe an example of nesting loops in C.

Any number of loops can be defined inside another loop, i.e., there is no restriction for defining any number of loops. The nesting level can be defined at n times. You can define any type of loop inside another loop; for example, you can define 'while' loop inside a 'for' loop.

Syntax of Nested loop

```
Outer_loop  
  
{  
  
    Inner_loop  
  
    {  
  
        // inner loop statements.  
  
    }  
  
    // outer loop statements.  
  
}
```

Outer_loop and Inner_loop are the valid loops that can be a 'for' loop, 'while' loop or 'do-while' loop.

Nested for loop

The nested for loop means any type of loop which is defined inside the 'for' loop.

```

for (initialization; condition; update)
{
    for(initialization; condition; update)
    {
        // inner loop statements.
    }
    // outer loop statements.
}

```

Example of nested for loop

```

#include <stdio.h>

int main()
{
    int n;// variable declaration

    printf("Enter the value of n :");

    // Displaying the n tables.
    for(int i=1;i<=n;i++) // outer loop
    {
        for(int j=1;j<=10;j++) // inner loop
        {
            printf("%d\t",i*j)); // printing the value.
        }

        printf("\n");
    }
}

```

Explanation of the above code

- First, the 'i' variable is initialized to 1 and then program control passes to the $i \leq n$.
- The program control checks whether the condition ' $i \leq n$ ' is true or not.

- If the condition is true, then the program control passes to the inner loop.
- The inner loop will get executed until the condition is true.
- After the execution of the inner loop, the control moves back to the update of the outer loop, i.e., $i++$.
- After incrementing the value of the loop counter, the condition is checked again, i.e., $i \leq n$.
- If the condition is true, then the inner loop will be executed again.
- This process will continue until the condition of the outer loop is true.

Output:

```
Enter the value of n : 3
1      2      3      4      5      6      7      8      9      10
2      4      6      8      10     12     14     16     18     20
3      6      9      12     15     18     21     24     27     30

...Program finished with exit code 0
Press ENTER to exit console.
```

Nested while loop

The nested while loop means any type of loop which is defined inside the 'while' loop.

```
while(condition)
{
    while(condition)
    {
        // inner loop statements.
    }
    // outer loop statements.
}
```

Example of nested while loop

```
#include <stdio.h>

int main()
{
    int rows; // variable declaration
```

```

int columns; // variable declaration

int k=1; // variable initialization

printf("Enter the number of rows :"); // input the number of rows.

scanf("%d",&rows);

printf("\nEnter the number of columns :"); // input the number of columns.

scanf("%d",&columns);

int a[rows][columns]; //2d array declaration

int i=1;

while(i<=rows) // outer loop
{
    int j=1;

    while(j<=columns) // inner loop
    {
        printf("%d\t",k); // printing the value of k.

        k++; // increment counter

        j++;
    }

    i++;

    printf("\n");
}
}

```

Explanation of the above code.

- We have created the 2d array, i.e., int a[rows][columns].
- The program initializes the 'i' variable by 1.

- Now, control moves to the while loop, and this loop checks whether the condition is true, then the program control moves to the inner loop.
- After the execution of the inner loop, the control moves to the update of the outer loop, i.e., $i++$.
- After incrementing the value of 'i', the condition ($i \leq \text{rows}$) is checked.
- If the condition is true, the control then again moves to the inner loop.
- This process continues until the condition of the outer loop is true.

Output:

```
Enter the number of rows : 5

Enter the number of columns :3
1      2      3
4      5      6
7      8      9
10     11     12
13     14     15

...Program finished with exit code 0
Press ENTER to exit console.
```

Nested do..while loop

The nested do..while loop means any type of loop which is defined inside the 'do..while' loop.

do

{

do

{

// inner loop statements.

}while(condition);

// outer loop statements.

}while(condition);

Example of nested do..while loop.

```
#include <stdio.h>

int main()
{
    /*printing the pattern

    *****

    *****

    *****

    ***** */

    int i=1;
    do    // outer loop
    {
        int j=1;
        do    // inner loop
        {
            printf("*");

            j++;
        }while(j<=8);

        printf("\n");

        i++;
    }while(i<=4);
}
```

Output:

```
*****
*****
*****
*****

...Program finished with exit code 0
Press ENTER to exit console.□
```

Explanation of the above code.

First, we initialize the outer loop counter variable, i.e., 'i' by 1.

As we know that the do..while loop executes once without checking the condition, so the inner loop is executed without checking the condition in the outer loop.

After the execution of the inner loop, the control moves to the update of the i++.

When the loop counter value is incremented, the condition is checked. If the condition in the outer loop is true, then the inner loop is executed.

This process will continue until the condition in the outer loop is true.

Infinite Loop in C

What is infinite loop?

An infinite loop is a looping construct that does not terminate the loop and executes the loop forever. It is also called an indefinite loop or an endless loop. It either produces a continuous output or no output.

When to use an infinite loop

An infinite loop is useful for those applications that accept the user input and generate the output continuously until the user exits from the application manually. In the following situations, this type of loop can be used:

All the operating systems run in an infinite loop as it does not exist after performing some task. It comes out of an infinite loop only when the user manually shuts down the system.

All the servers run in an infinite loop as the server responds to all the client requests. It comes out of an indefinite loop only when the administrator shuts down the server manually.

All the games also run in an infinite loop. The game will accept the user requests until the user exits from the game.

We can create an infinite loop through various loop structures. The following are the loop structures through which we will define the infinite loop:

- for loop
- while loop
- do-while loop
- go to statement
- C macros

For loop

Let's see the infinite 'for' loop. The following is the definition for the infinite for loop:

```
for(;;)
{
    // body of the for loop.
}
```

As we know that all the parts of the 'for' loop are optional, and in the above for loop, we have not mentioned any condition; so, this loop will execute infinite times.

Let's understand through an example.

```
#include <stdio.h>

int main()
{
    for(;;)
    {
        printf("Hello ICS");
    }
    return 0;
}
```

In the above code, we run the 'for' loop infinite times, so "Hello ICS" will be displayed infinitely.

while loop

Now, we will see how to create an infinite loop using a while loop. The following is the definition for the infinite while loop:

```
while(1)
{
    // body of the loop..
}
```

In the above while loop, we put '1' inside the loop condition. As we know that any non-zero integer represents the true condition while '0' represents the false condition.

Let's look at a simple example.

```
#include <stdio.h>

int main()
{
    int i=0;
    while(1)
    {
        i++;
        printf("i is :%d",i);
    }
    return 0;
}
```

In the above code, we have defined a while loop, which runs infinite times as it does not contain any condition. The value of 'i' will be updated an infinite number of times.

Output

do..while loop

```
do
{
    // body of the loop..
}while(1);
```

goto statement

```
infinite_loop;  
  
// body statements.  
  
goto infinite_loop;
```

Macros

```
#include <stdio.h>
```

```
int main()
{
    infinite
    {
        printf("hello");
    }
    return 0;
}
```

Output

[illegible]

Let's understand through an example.

```
int main()
{
    char ch;
    while(1)
```

```

{
    ch=getchar();

    if(ch=='n')
    {
        break;
    }

    printf("hello");
}

return 0;
}

```

In the above code, we have defined the while loop, which will execute an infinite number of times until we press the key 'n'. We have added the 'if' statement inside the while loop. The 'if' statement contains the break keyword, and the break keyword brings control out of the loop.

Unintentional infinite loops

Sometimes the situation arises where unintentional infinite loops occur due to the bug in the code. If we are the beginners, then it becomes very difficult to trace them. Below are some measures to trace an unintentional infinite loop:

We should examine the semicolons carefully. Sometimes we put the semicolon at the wrong place, which leads to the infinite loop.

```

#include <stdio.h>

int main()
{
    int i=1;
    while(i<=10);
    {
        printf("%d", i);
    }
}

```



```

i++;
}
return 0;
}

```

In the above code, we put the semicolon after the condition of the while loop which leads to the infinite loop. Due to this semicolon, the internal body of the while loop will not execute.

We should check the logical conditions carefully. Sometimes by mistake, we place the assignment operator (=) instead of a relational operator (==).

```

#include <stdio.h>

int main()
{
    char ch='n';
    while(ch='y')
    {
        printf("hello");
    }
    return 0;
}

```

In the above code, we use the assignment operator (ch='y') which leads to the execution of loop infinite number of times.

We use the wrong loop condition which causes the loop to be executed indefinitely.

```

#include <stdio.h>

int main()
{
    for(int i=1;i>=1;i++)

```

```

{
    printf("hello");
}
return 0;
}

```

The above code will execute the 'for loop' infinite number of times. As we put the condition ($i \geq 1$), which will always be true for every condition, it means that "hello" will be printed infinitely.

We should be careful when we are using the break keyword in the nested loop because it will terminate the execution of the nearest loop, not the entire loop.

```

#include <stdio.h>

int main()
{
    while(1)
    {
        for(int i=1;i<=10;i++)
        {
            if(i%2==0)
            {
                break;
            }
        }
    }

    return 0;
}

```

In the above code, the while loop will be executed an infinite number of times as we use the break keyword in an inner loop. This break keyword will bring the control out of the inner loop, not from the outer loop.

We should be very careful when we are using the floating-point value inside the loop as we cannot underestimate the floating-point errors.

```
#include <stdio.h>

int main()
{
    float x = 3.0;
    while (x != 4.0) {
        printf("x = %f\n", x);
        x += 0.1;
    }
    return 0;
}
```

In the above code, the loop will run infinite times as the computer represents a floating-point value as a real value. The computer will represent the value of 4.0 as 3.999999 or 4.000001, so the condition (x !=4.0) will never be false. The solution to this problem is to write the condition as (k<=4.0).

C 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

Syntax:

```
//loop or switch case
```

```
break;
```

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

C break statement with the nested loop

In such case, it breaks only the inner loop, but not outer loop.

```
#include<stdio.h>
```

```
int main(){
```

```
    int i=1,j=1;//initializing a local variable
```

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

```

for(j=1;j<=3;j++){
printf("%d &d\n",i,j);
if(i==2 && j==2){
break;//will break loop of j only
}
} //end of for loop
return 0;
}

```

Output

```

1 1
1 2
1 3
2 1
2 2
3 1
3 2
3 3

```

As you can see the output on the console, 2 3 is not printed because there is a break statement after printing i==2 and j==2. But 3 1, 3 2 and 3 3 are printed because the break statement is used to break the inner loop only.

break statement with while loop

Consider the following example to use break statement inside while loop.

```

#include<stdio.h>

void main ()
{
    int i = 0;

```

```

while(1)
{
    printf("%d ",i);

    i++;

    if(i == 10)

        break;

}

printf("came out of while loop");
}

```

Output

0 1 2 3 4 5 6 7 8 9 came out of while loop

break statement with do-while loop

Consider the following example to use the break statement with a do-while loop.

```

#include<stdio.h>

void main ()
{
    int n=2,i,choice;

    do
    {
        i=1;

        while(i<=10)

        {

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

            i++;

        }

    }
}

```

```

    printf("do you want to continue with the table of %d , enter any non-zero value to continue.",n+1);

    scanf("%d",&choice);

    if(choice == 0)

    {

        break;

    }

    n++;

}while(1);
}

```

Output

2 X 1 = 2

2 X 2 = 4

2 X 3 = 6

2 X 4 = 8

2 X 5 = 10

2 X 6 = 12

2 X 7 = 14

2 X 8 = 16

2 X 9 = 18

2 X 10 = 20

do you want to continue with the table of 3 , enter any non-zero value to continue.1

3 X 1 = 3

3 X 2 = 6

3 X 3 = 9

3 X 4 = 12

3 X 5 = 15

3 X 6 = 18

3 X 7 = 21

3 X 8 = 24

3 X 9 = 27

3 X 10 = 30

do you want to continue with the table of 4 , enter any non-zero value to continue.0

C continue statement

The continue statement in C language is used to bring the program control to the beginning of the loop. The continue statement skips some lines of code inside the loop and continues with the next iteration. It is mainly used for a condition so that we can skip some code for a particular condition.

Syntax:

```
//loop statements
```

```
continue;
```

```
//some lines of the code which is to be skipped
```

Continue statement example 1

```
#include<stdio.h>
```

```
void main ()
```

```
{
```

```
    int i = 0;
```

```
    while(i!=10)
```

```
    {
```

```
        printf("%d", i);
```

```
        continue;
```

```
        i++;
```

```
    }
```



```
}
```

Output

infinite loop

Continue statement example 2

```
#include<stdio.h>
```

```
int main(){
```

```
int i=1;//initializing a local variable
```

```
//starting a loop from 1 to 10
```

```
for(i=1;i<=10;i++){
```

```
if(i==5){//if value of i is equal to 5, it will continue the loop
```

```
continue;
```

```
}
```

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

```
}//end of for loop
```

```
return 0;
```

```
}
```

Output

1

2

3

4

6

7

8

9

10

As you can see, 5 is not printed on the console because loop is continued at i==5.

C continue statement with inner loop

In such case, C continue statement continues only inner loop, but not outer loop.

```
#include<stdio.h>

int main(){

int i=1,j=1;//initializing a local variable

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

for(j=1;j<=3;j++){

if(i==2 && j==2){

continue;//will continue loop of j only

}

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

}

}

return 0;

}
```

Output

1 1

1 2

1 3

2 1

2 3

3 1

3 2

As you can see, 2 2 is not printed on the console because inner loop is continued at $i==2$ and $j==2$.

C goto statement

The goto statement is known as jump statement in C. As the name suggests, goto is used to transfer the program control to a predefined label. The goto statement can be used to repeat some part of the code for a particular condition. It can also be used to break the multiple loops which can't be done by using a single break statement. However, using goto is avoided these days since it makes the program less readable and complicated.

Syntax:

label:

//some part of the code;

goto label;

goto example

Let's see a simple example to use goto statement in C language.

```
#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

When should we use goto?

The only condition in which using goto is preferable is when we need to break the multiple loops using a single statement at the same time. Consider the following example.

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    int i, j, k;
```

```
    for(i=0;i<10;i++)
```

```
    {
```

```
        for(j=0;j<5;j++)
```

```
        {
```

```
            for(k=0;k<3;k++)
```

```
            {
```

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

```

    if(j == 3)
    {
        goto out;
    }
}
}
}

out:

printf("came out of the loop");
}

0 0 0
0 0 1
0 0 2
0 1 0
0 1 1
0 1 2
0 2 0
0 2 1
0 2 2
0 3 0

```

came out of the loop

Type Casting in C

Typecasting allows us to convert one data type into other. In C language, we use cast operator for typecasting which is denoted by (type).

Syntax:

(type)value;

Note: It is always recommended to convert the lower value to higher for avoiding data loss.

Without Type Casting:

```
int f= 9/4;
```

```
printf("f : %d\n", f );//Output: 2
```

With Type Casting:

```
float f=(float) 9/4;
```

```
printf("f : %f\n", f );//Output: 2.250000
```

Type Casting example

Let's see a simple example to cast int value into the float.

```
#include<stdio.h>
```

```
int main(){
```

```
float f= (float)9/4;
```

```
printf("f : %f\n", f );
```

```
return 0;
```

```
}
```

Output:

```
f : 2.250000
```