Unit-2 By Dr Saroj Kumar Nanda

C Variables

A **variable in C** is a named piece of memory which is used to store data and access it whenever required. It allows us to use the memory without having to memorize the exact memory address.
To create a variable in C, we have to specify a **name** and the **type of data** it is going to store in the syntax.

```
data_type name;
```

C provides different **data types** that can store almost all kinds of data. For example, int, char, float, double, etc.

```
int num;
char letter;
float decimal;
```

In C, every variable must be declared before it is used. We can also declare multiple variables of same data type in a single statement by separating them using comma as shown:

```
data_type name1, name2, name3, ...;
```

Rules for Naming Variables in C

We can assign any name to a C variable as long as it follows the following rules:

- A variable name must only contain **letters**, **digits**, and **underscores**.
- It must **start with an alphabet** or an **underscore** only. It cannot start with a digit.
- **No white space** is allowed within the variable name.
- A variable name must **not** be any reserved word or **keyword**.
- The name must be unique in the program.

C Variable Initialization

Once the variable is declared, we can store useful values in it. The first value we store is called initial value and the process is called **Initialization**. It is done using assignment operator **(=)**.

```
int num;
num = 3;
```

It is important to initialize a variable because a C variable only contains garbage value when it is declared. We can also initialize a variable along with declaration.

```
int num = 3;
```

Example:

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

   // Create integer variable
   int num = 3;

   // Access the value stored in
   // variable
   printf("%d", num);
   return 0;
}
```

Output

```
3
```

How to use variables in C?

Variables act as name for memory locations that stores some value. It is valid to use the variable wherever it is valid to use its value. It means that a variable name can be used anywhere as a substitute in place of the value it stores.

Example: An integer variable can be used in a mathematical expression in place of numeric values.

```
#include <stdio.h>
int main() {
      // Expression that uses values
   int sum1 = 20 + 40;
      // Defining variables
   int a = 20, b = 40;
      // Expression that uses variables
   int sum2 = a + b;
      printf("%d\n%d", sum1, sum2);
   return 0;
}
```

Output

```
60

60
```

Memory Allocation of C Variables

When a variable is **declared**, the compiler is told that the variable with the given name and type exists in the program. But no memory is allocated to it yet. Memory is allocated when the variable is **defined**.

Most programming languages like C generally declare and define a variable in the single step. For example, in the above part where we create a variable, variable is declared and defined in a single statement.

The size of memory assigned for variables depends on the type of variable. We can check the size of the variables using **sizeof operator.**

Example:

```
#include <stdio.h>
int main() {
   int num = 22;
     // Finding size of num
   printf("%d bytes", sizeof(num));
   return 0;
}
```

Output

```
4 bytes
```

Constants in C

C also provides some variables whose value cannot be changed. These variables are called **constants** and are created simply by prefixing **const** keyword in variable declaration.

Syntax:

```
const data_type name = value;
```
Constants must be initialized at the time of declaration.

Data Types in C

Each variable in C has an associated data type. It specifies the type of data that the variable can store like integer, character, floating, double, etc.

Example:
```
int number;
```
The above statement declares a variable with name **number** that can store **integer** values.

C is a statically type language where each variable's type must be specified at the declaration and once specified, it cannot be changed.

In this article, we will discuss the basic (primary) data types in C.

Please note that the Ranges and Sizes of different data types mentioned below are the most commonly used values. The actual values may vary from compiler to compiler.

Integer Data Type

The integer datatype in C is used to store the integer numbers (any number including positive, negative and zero without decimal part). Octal values, hexadecimal values, and decimal values can also be stored in int data type in C.

- **Range:** -2,147,483,648 to 2,147,483,647
- **Size:** 4 bytes
- **Format Specifier:** %d

Format specifiers are the symbols that are used for printing and scanning values of given data types.

Example:

We use **int keyword** to declare the integer variable:
```
int val;
```
We can store the integer values (literals) in this variable.
```
#include <stdio.h>

int main() {
   int var = 22;

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

Output
```
var = 22
```
A variable of given data type can only contains the values of the same type. So, **var** can only store numbers, not text or anything else.

The integer data type can also be used as:

1. **unsigned int:** It can store the data values from zero to positive numbers, but it can't store negative values
2. **short int:** It is lesser in size than the int by 2 bytes so can only store values from -32,768 to 32,767.

3. **long int:** Larger version of the int datatype so can store values greater than int.
4. **unsigned short int:** Similar in relationship with short int as unsigned int with int.

Note: *The size of an integer data type is compiler dependent. We can use* ***sizeof operator*** *to check the actual size of any data type. In this article, we are discussing the sizes according to 64-bit compilers.*

Character Data Type

Character data type allows its variable to store only a single character. The size of the character is **1 byte**. It is the most basic data type in C. It stores a single character and requires a single byte of memory in almost all compilers.

- **Range:** (-128 to 127) or (0 to 255)
- **Size:** 1 byte
- **Format Specifier:** %c

Example:

```
#include <stdio.h>
int main() {
   char ch = 'A';

   printf("ch = %c", ch);
   return 0;
}
```

Output

```
ch = A
```

Float Data Type

In C programming, **float data type** is used to store single precision floating-point values. These values are decimal and exponential numbers.

- **Range:** 1.2E-38 to 3.4E+38
- **Size:** 4 bytes
- **Format Specifier:** %f

Example:

```
#include <stdio.h>
int main() {
   float val = 12.45;

   printf("val = %f", val);
   return 0;
}
```

Output

```
val = 12.450000
```

Double Data Type

The **double data type** in C is used to store decimal numbers (numbers with floating point values) with double precision. It can easily accommodate about 16 to 17 digits after or before a decimal point.

- **Range:** 1.7E-308 to 1.7E+308
- **Size:** 8 bytes

- **Format Specifier:** %lf

Example:

```
#include <stdio.h>
int main() {
    double val = 1.4521;
        printf("val = %lf", val);
    return 0;
}
```

Output

```
val = 1.452100
```

Void Data Type

The **void data** type in C is used to indicate the absence of a value. Variables of void data type are not allowed. It can only be used for pointers and function return type and parameters.

Example:

```
void fun(int a, int b){
    // function body
}
```

where function **fun** is a void type of function means it doesn't return any value.

Size of Data Types in C

The size of the data types in C is dependent on the size of the architecture, so we cannot define the universal size of the data types. For that, the C language provides the **sizeof()** operator to check the size of the data types.

Example

```
#include <stdio.h>
int main(){
     // Use sizeof() to know size
    // the data types
    printf("The size of int: %d\n",
        sizeof(int));
    printf("The size of char: %d\n",
        sizeof(char));
    printf("The size of float: %d\n",
        sizeof(float));
    printf("The size of double: %d",
        sizeof(double));
    return 0;
}
```

Output

```
The size of int: 4

The size of char: 1

The size of float: 4

The size of double: 8
```

Different data types also have different ranges up to which can vary from compiler to compiler. Below is a list of ranges along with the memory requirement and format specifiers on the **64-bit GCC compiler**.

Data Type	Size (bytes)	Range	Format Specifier
short int	2	-32,768 to 32,767	**%hd**
unsigned short int	2	0 to 65,535	**%hu**
unsigned int	4	0 to 4,294,967,295	**%u**
int	4	-2,147,483,648 to 2,147,483,647	**%d**
long int	4	-2,147,483,648 to 2,147,483,647	**%ld**
unsigned long int	4	0 to 4,294,967,295	**%lu**
long long int	8	-(2^63) to (2^63)-1	**%lld**
unsigned long long int	8	0 to 18,446,744,073,709,551,615	**%llu**
signed char	1	-128 to 127	**%c**
unsigned char	1	0 to 255	**%c**
float	4	1.2E-38 to 3.4E+38	**%f**
double	8	1.7E-308 to 1.7E+308	**%lf**
long double	16	3.4E-4932 to 1.1E+4932	**%Lf**

Note: *The l**ong, short, signed and unsigned** are **datatype modifier** that can be used with some primitive data types to change the size or length of the datatype.*

Literals in C

In C, **literals** are constant values assigned to variables. They represent fixed values that cannot be changed. Literals occupy memory but do not have references like variables. Often, the terms constants and literals are used interchangeably.

Printing Variables Along with String

```
#include <stdio.h>
int main() {
    int age = 22;
     // Prints Age
    printf("The value of the variable age is %d\n", age);

    return 0;
}
```

Output

```
The value of the variable age is 22
```

Reading an Integer

```
#include <stdio.h>
int main() {
    int age;
    printf("Enter your age: ");
      // Reads an integer
    scanf("%d", &age);
      // Prints the age
    printf("Age is: %d\n", age);
    return 0;
}
```

Output

```
Enter your age:
25 (Entered by the user)
Age is: 25
```

Explanation: %d is used to read an integer; and &age provides the address of the variable where the input will be stored.

Reading a Character

```
#include <stdio.h>
int main() {
    int ch;
    printf("Enter a character: \n");
      // Reads an integer
    scanf("%c", &ch);

    // Prints the age
    printf("Entered character is: %d\n", ch);
    return 0;
}
```

Output

```
Enter a character:
a (Entered by the user)
Entered character is: a
```

Reading a string

The scanf() function can also be used to read string input from users. But it can only read single words.

```
#include <stdio.h>
int main() {
    char str[100];  // Declare an array to hold the input string
    printf("Enter a string: ");
    scanf("%s", str);  // Reads input until the first space or newline
    printf("You entered: %s\n", str);
    return 0;
}
```

Output:

```
Enter a String:
Geeks (Entered by the user)
Entered string is: Geeks
```

The scanf() function can not handle spaces and stops at the first blanksspace. to handle this situation we can use fgets() which is a better alternative as it can handle spaces and prevent buffer overflow.

fgets()

fgets() reads the given number of characters of a line from the input and stores it into the specified string. It can read multiple words at a time.

Syntax

```
fgets(str, n, stdin);
```

where **buff** is the string where the input will be stored and **n** is the maximum number of characters to read. **stdin** represents input reading from the keyboard.

Example:

```
#include <stdio.h>
#include <string.h>
int main() {
      // String variable
    char name[20];

    printf("Enter your name: \n");
    fgets(name, sizeof(name), stdin);
        printf("Hello, %s", name);
    return 0;
}
```

Output

```
Enter your name:
John (Entered by User)
Hello, John
```

For reading a single character, we use getchar() in C.

Operators in C

Operators are the basic components of C programming. They are symbols that represent some kind of operation, such as mathematical, relational, bitwise, conditional, or logical computations, which are to be performed on values or variables. The values and variables used with operators are called **operands**.

Example:

```
#include <stdio.h>
int main() {
      // Expression for getting sum
   int sum = 10 + 20;
      printf("%d", sum);
   return 0;
}
```

Output

```
30
```

In the above expression, **'+'** is the **addition operator** that tells the compiler to add both of the operands 10 and 20. To dive deeper into how operators are used with data structures, the C Programming Course Online with Data Structures covers this topic thoroughly.

Unary, Binary and Ternary Operators

On the basis of the number of operands they work on, operators can be classified into three types :

1. **Unary Operators:** Operators that work on single operand. Example: Increment(++) , Decrement(--)
2. **Binary Operators:** Operators that work on two operands. Example: Addition (+), Subtraction(-) , Multiplication (*)
3. **Ternary Operators:** Operators that work on three operands. Example**:** Conditional Operator(? :)

Types of Operators in C

C language provides a wide range of built in operators that can be classified into 6 types based on their functionality:

Table of Content

Arithmetic Operators

The **arithmetic operators** are used to perform arithmetic/mathematical operations on operands. There are **9 arithmetic** operators in C language**:**

Symbol	Operator	Description	Syntax
+	**Plus**	Adds two numeric values.	**a + b**
-	**Minus**	Subtracts right operand from left operand.	**a - b**

Symbol	Operator	Description	Syntax
*	**Multiply**	Multiply two numeric values.	**a * b**
/	**Divide**	Divide two numeric values.	**a / b**
%	**Modulus**	Returns the remainder after diving the left operand with the right operand.	**a % b**
+	**Unary Plus**	Used to specify the positive values.	**+a**
-	**Unary Minus**	Flips the sign of the value.	**-a**
++	**Increment**	Increases the value of the operand by 1.	**a++**
--	**Decrement**	Decreases the value of the operand by 1.	**a--**

Example of C Arithmetic Operators

```
#include <stdio.h>
int main() {
    int a = 25, b = 5;
    // using operators and printing results
    printf("a + b = %d\n", a + b);
    printf("a - b = %d\n", a - b);
    printf("a * b = %d\n", a * b);
    printf("a / b = %d\n", a / b);
    printf("a % b = %d\n", a % b);
    printf("+a = %d\n", +a);
    printf("-a = %d\n", -a);
    printf("a++ = %d\n", a++);
    printf("a-- = %d\n", a--);
    return 0;
}
```

Output

```
a + b = 30

a - b = 20

a * b = 125

a / b = 5

a % b = 0

+a = 25
```

```
-a = -25

a++ = 25

a-- = 26
```

Relational Operators

The **relational operators** in C are used for the comparison of the two operands. All these operators are binary operators that return true or false values as the result of comparison.

These are a total of 6 relational operators in C:

Symbol	Operator	Description	Syntax
<	**Less than**	Returns true if the left operand is less than the right operand. Else false	**a < b**
>	**Greater than**	Returns true if the left operand is greater than the right operand. Else false	**a > b**
<=	**Less than or equal to**	Returns true if the left operand is less than or equal to the right operand. Else false	**a <= b**
>=	**Greater than or equal to**	Returns true if the left operand is greater than or equal to right operand. Else false	**a >= b**
==	**Equal to**	Returns true if both the operands are equal.	**a == b**
!=	**Not equal to**	Returns true if both the operands are NOT equal.	**a != b**

Example of C Relational Operators

```
#include <stdio.h>
int main() {
    int a = 25, b = 5;
    // using operators and printing results
    printf("a < b  : %d\n", a < b);
    printf("a > b  : %d\n", a > b);
    printf("a <= b: %d\n", a <= b);
    printf("a >= b: %d\n", a >= b);
    printf("a == b: %d\n", a == b);
    printf("a != b : %d\n", a != b);
    return 0;
}
```

Output

```
a < b  : 0
```

```
a > b  : 1

a <= b: 0

a >= b: 1

a == b: 0

a != b : 1
```

Here, 0 means false and 1 means true.

Logical Operator

Logical Operators are used to combine two or more conditions/constraints or to complement the evaluation of the original condition in consideration. The result of the operation of a logical operator is a Boolean value either **true** or **false**.

There are **3** logical operators in C**:**

Symbol	Operator	Description	Syntax
&&	**Logical AND**	Returns true if both the operands are true.	**a && b**
||	**Logical OR**	Returns true if both or any of the operand is true.	**a || b**
!	**Logical NOT**	Returns true if the operand is false.	**!a**

Example of Logical Operators in C

```
#include <stdio.h>
int main() {
    int a = 25, b = 5;

    // using operators and printing results
    printf("a && b : %d\n", a && b);
    printf("a || b : %d\n", a || b);
    printf("!a: %d\n", !a);
    return 0;
}
```

Output

```
a && b : 1

a || b : 1

!a: 0
```

Bitwise Operators

The Bitwise operators are used to perform bit-level operations on the operands. The operators are first converted to bit-level and then the calculation is performed on the operands.

Note: Mathematical operations such as addition, subtraction, multiplication, etc. can be performed at the bit level for faster processing.
There are 6 bitwise operators in C:

Symbol	Operator	Description	Syntax
&	**Bitwise AND**	Performs bit-by-bit AND operation and returns the result.	**a & b**
|	**Bitwise OR**	Performs bit-by-bit OR operation and returns the result.	**a | b**
^	**Bitwise XOR**	Performs bit-by-bit XOR operation and returns the result.	**a ^ b**
~	**Bitwise First Complement**	Flips all the set and unset bits on the number.	**~a**
<<	**Bitwise Leftshift**	Shifts bits to the left by a given number of positions; multiplies the number by 2 for each shift.	**a << b**
>>	**Bitwise Rightshilft**	Shifts bits to the right by a given number of positions; divides the number by 2 for each shift.	**a >> b**

Example of Bitwise Operators

```
#include <stdio.h>
int main() {
    int a = 25, b = 5;
    // using operators and printing results
    printf("a & b: %d\n", a & b);
    printf("a | b: %d\n", a | b);
    printf("a ^ b: %d\n", a ^ b);
    printf("~a: %d\n", ~a);
    printf("a >> b: %d\n", a >> b);
    printf("a << b: %d\n", a << b);
    return 0;
}
```

Output

```
a & b: 1

a | b: 29
```

```
a ^ b: 28

~a: -26

a >> b: 0

a << b: 800
```

Assignment Operators

Assignment operators are used to assign value to a variable. The left side operand of the assignment operator is a variable and the right side operand of the assignment operator is a value. The value on the right side must be of the same data type as the variable on the left side otherwise the compiler will raise an error.

The assignment operators can be combined with some other operators in C to provide multiple operations using single operator. These operators are called compound operators.

In C, there are 11 assignment operators:

Symbol	Operator	Description	Syntax
=	**Simple Assignment**	Assign the value of the right operand to the left operand.	**a = b**
+=	**Plus and assign**	Add the right operand and left operand and assign this value to the left operand.	**a += b**
-=	**Minus and assign**	Subtract the right operand and left operand and assign this value to the left operand.	**a -= b**
*=	**Multiply and assign**	Multiply the right operand and left operand and assign this value to the left operand.	**a *= b**
/=	**Divide and assign**	Divide the left operand with the right operand and assign this value to the left operand.	**a /= b**
%=	**Modulus and assign**	Assign the remainder in the division of left operand with the right operand to the left operand.	**a %= b**
&=	**AND and assign**	Performs bitwise AND and assigns this value to the left operand.	**a &= b**
|=	**OR and assign**	Performs bitwise OR and assigns this value to the left operand.	**a |= b**

Symbol	Operator	Description	Syntax
^=	**XOR and assign**	Performs bitwise XOR and assigns this value to the left operand.	**a ^= b**
>>=	**Rightshift and assign**	Performs bitwise Rightshift and assign this value to the left operand.	**a >>= b**
<<=	**Leftshift and assign**	Performs bitwise Leftshift and assign this value to the left operand.	**a <<= b**

Example of C Assignment Operators

```
#include <stdio.h>
int main() {
    int a = 25, b = 5;
    // using operators and printing results
    printf("a = b: %d\n", a = b);
    printf("a += b: %d\n", a += b);
    printf("a -= b: %d\n", a -= b);
    printf("a *= b: %d\n", a *= b);
    printf("a /= b: %d\n", a /= b);
    printf("a %%= b: %d\n", a %= b);
    printf("a &= b: %d\n", a &= b);
    printf("a |= b: %d\n", a |= b);
    printf("a ^= b: %d\n", a ^= b);
    printf("a >>= b: %d\n", a >>= b);
    printf("a <<= b: %d\n", a <<= b);
    return 0;
}
```

Output

```
a = b: 5

a += b: 10

a -= b: 5

a *= b: 25

a /= b: 5

a %= b: 0

a &= b: 0

a |= b: 5

a ^= b: 0

a >>= b: 0
```

```
a <<= b: 0
```

Other Operators

Apart from the above operators, there are some other operators available in C used to perform some specific tasks. Some of them are discussed here:

sizeof Operator

- **sizeof** is much used in the C programming language.
- It is a compile-time unary operator which can be used to compute the size of its operand.
- The result of sizeof is of the unsigned integral type which is usually denoted by size_t.
- Basically, the sizeof the operator is used to compute the size of the variable or datatype.

Syntax

```
sizeof (operand)
```

Comma Operator (,)

The comma operator (represented by the token) is a binary operator that evaluates its first operand and discards the result, it then evaluates the second operand and returns this value (and type).

The comma operator has the lowest precedence of any C operator. It can act as both operator and separator.

Syntax

```
operand1 , operand2
```

Conditional Operator (? :)

The **conditional operator** is the only ternary operator in C++. It is a conditional operator that we can use in place of if..else statements.

Syntax

```
expression1 ? Expression2 : Expression3;
```

Here, **Expression1** is the condition to be evaluated. If the condition(**Expression1**) is ***True*** then we will execute and return the result of **Expression2** otherwise if the condition(**Expression1**) is ***false*** then we will execute and return the result of **Expression3**.

dot (.) and arrow (->) Operators

Member operators are used to reference individual members of classes, structures, and unions.

- The dot operator is applied to the actual object.
- The arrow operator is used with a pointer to an object.

Syntax

```
structure_variable . member;
structure_pointer -> member;
```

Cast Operators

Casting operators convert one data type to another. For example, int(2.2000) would return 2.

- A cast is a special operator that forces one data type to be converted into another.

Syntax

```
(new_type) operand;
```

addressof (&) and Dereference (*) Operators

Addressof operator & returns the address of a variable and the dereference operator * is a pointer to a variable. For example *var; will pointer to a variable var.

Example of Other C Operators

```
// C Program to demonstrate the use of Misc operators
#include <stdio.h>
int main()
```

```
{
    // integer variable
    int num = 10;
    int* add_of_num = &num;
    printf("sizeof(num) = %d bytes\n", sizeof(num));
    printf("&num = %p\n", &num);
    printf("*add_of_num = %d\n", *add_of_num);
    printf("(10 < 5) ? 10 : 20 = %d\n", (10 < 5) ? 10 : 20);
    printf("(float)num = %f\n", (float)num);
    return 0;
}
```

Output

```
sizeof(num) = 4 bytes

&num = 0x7ffdb58c037c

*add_of_num = 10

(10 < 5) ? 10 : 20 = 20

(float)num = 10.000000
```

Operator Precedence and Associativity

Operator Precedence and Associativity is the concept that decides which operator will be evaluated first in the case when there are multiple operators present in an expression to avoid ambiguity. As, it is very common for a C expression or statement to have multiple operators and in this expression.

The below table describes the precedence order and associativity of operators in C. The precedence of the operator decreases from top to bottom.

Precedence	Operator	Description	Associativity
1	**()**	Parentheses (function call)	left-to-right
	[]	Brackets (array subscript)	left-to-right
	.	Member selection via object name	left-to-right
	->	Member selection via a pointer	left-to-right
	a++ , a--	Postfix increment/decrement (a is a variable)	left-to-right
2	**++a , --a**	Prefix increment/decrement (a is a variable)	right-to-left

Precedence	Operator	Description	Associativity
	+ , -	Unary plus/minus	right-to-left
	! , ~	Logical negation/bitwise complement	right-to-left
	(type)	Cast (convert value to temporary value of type)	right-to-left
	*****	Dereference	right-to-left
	&	Address (of operand)	right-to-left
	sizeof	Determine size in bytes on this implementation	right-to-left
3	*** , / , %**	Multiplication/division/modulus	left-to-right
4	**+ , -**	Addition/subtraction	left-to-right
5	<< , >>	Bitwise shift left, Bitwise shift right	left-to-right
	< , <=	Relational less than/less than or equal to	left-to-right
6	> , >=	Relational greater than/greater than or equal to	left-to-right
7	**== , !=**	Relational is equal to/is not equal to	left-to-right
8	**&**	Bitwise AND	left-to-right
9	^	Bitwise XOR	left-to-right

Precedence	Operator	Description	Associativity
10	|	Bitwise OR	left-to-right
11	**&&**	Logical AND	left-to-right
12	||	Logical OR	left-to-right
13	**?:**	Ternary conditional	right-to-left
	=	Assignment	right-to-left
	+= , -=	Addition/subtraction assignment	right-to-left
	***= , /=**	Multiplication/division assignment	right-to-left
	%= , &=	Modulus/bitwise AND assignment	right-to-left
	^= , |=	Bitwise exclusive/inclusive OR assignment	right-to-left
14	<<=, >>=	Bitwise shift left/right assignment	right-to-left
15	,	expression separator	left-to-right

Decision Making in C (if , if..else, Nested if, if-else-if)

In C, programs can choose which part of the code to execute based on some condition. This ability is called **decision making** and the statements used for it are called **conditional statements.** These statements evaluate one or more conditions and make the decision whether to execute a block of code or not.

For example, consider that there is a show that starts only when certain number of people are present in the audience. So, you can write a program like as shown:

```
#include <stdio.h>
int main() {
   // Number of people in the audience
   int num = 100;
      // Conditional code inside decision making statement
   if (num > 50) {
      printf("Start the show");
   }
```

```
    return 0;
}
```
Output

```
Start the show
```

In the above program, the show only starts when the number of people is greater than **50**. It is specified in the **if statement** (a type of conditional statement) as a condition **(num > 50)**. You can decrease the value of **num** to less than **50** and try rerunning the code.

Types of Conditional Statements in C

In the above program, we have used if statement, but there are many different types of conditional statements available in C language:

1. if in C

The if statement is the simplest decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statements is executed otherwise not.

A **condition** is any expression that evaluates to either a true or false (or values convertible to true or flase).

Example

```
#include <stdio.h>
int main() {
        int i = 10;
    // If statement
    if (i < 18) {
        printf("Eligible for vote");
    }
}
```
Output

```
Eligible for vote
```

The expression inside **() parenthesis** is the **condition** and set of statements inside **{} braces** is its **body**. If the condition is true, only then the body will be executed.

*If there is only a single statement in the body, **{} braces** can be omitted.*

2. if-else in C

The **if** statement alone tells us that if a condition is true, it will execute a block of statements and if the condition is false, it won’t. But what if we want to do something else when the condition is false? Here comes the C **else** statement. We can use the **else** statement with the **if** statement to execute a block of code when the condition is false. The if-else statement consists of two blocks, one for false expression and one for true expression.

Example

```
#include <stdio.h>
int main() {
    int i = 10;

    if (i > 18) {
        printf("Eligible for vote");
    }
    else {
        printf("Not Eligible for vote");
    }
    return 0;
}
```
Output
```
Not Eligible for vote
```
The block of code following the *else* statement is executed as the condition present in the *if* statement is false.

3. Nested if-else in C

A nested if in C is an if statement that is the target of another if statement. Nested if statements mean an if statement inside another if statement. Yes, C allow us to nested if statements within if statements, i.e, we can place an if statement inside another if statement.

Example
```
#include <stdio.h>
int main(){
    int i = 10;
    if (i == 10) {
        if (i < 18)
            printf("Still not eligible for vote");
        else
            printf("Eligible for vote\n");
    }
    else {
        if (i == 20) {
            if (i < 22)
                printf("i is smaller than 22 too\n");
            else
                printf("i is greater than 25");
        }
    }
    return 0;
}
```
Output
```
Still not eligible for vote
```
4. if-else-if Ladder in C

The if else if statements are used when the user has to decide among multiple options. The C if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the C else-if ladder is bypassed. If none of the conditions is true, then the final else statement will be executed. if-else-if ladder is similar to the switch statement.

Example
```
#include <stdio.h>
int main() {
    int i = 20;
    // If else ladder with three conditions
    if (i == 10)
        printf("Not Eligible");
    else if (i == 15)
        printf("wait for three years");
    else if (i == 20)
        printf("You can vote");
    else
        printf("Not a valid age");
            return 0;
}
```

Output

```
You can vote
```

5. switch Statement in C

The switch case statement is an alternative to the if else if ladder that can be used to execute the conditional code based on the value of the variable specified in the switch statement. The switch block consists of cases to be executed based on the value of the switch variable.

Example

```
#include <stdio.h>
int main() {
       // variable to be used in switch statement
    int var = 18;

    // declaring switch cases
    switch (var) {
    case 15:
        printf("You are a kid");
        break;
    case 18:
        printf("Eligible for vote");
        break;
    default:
        printf("Default Case is executed");
        break;
    }
    return 0;
}
```

Output

```
Eligible for vote
```

Note: The switch expression should evaluate to either integer or character. It cannot evaluate any other data type.

6. Conditional Operator in C

The conditional operator is used to add conditional code in our program. It is similar to the if-else statement. It is also known as the ternary operator as it works on three operands.

Example:

```
#include <stdio.h>
int main() {
    int var;
    int flag = 0;
    // using conditional operator to assign the value to var
    // according to the value of flag
    var = flag == 0 ? 25 : -25;
    printf("Value of var when flag is 0: %d\n", var);
    return 0;
}
```

Output

```
Value of var when flag is 0: 25
```

7. Jump Statements in C

These statements are used in C for the unconditional flow of control throughout the functions in a program. They support four types of jump statements:

A) break

This loop control statement is used to terminate the loop. As soon as the break statement is encountered from within a loop, the loop iterations stop there, and control returns from the loop immediately to the first statement after the loop.

Example

```
#include <stdio.h>
int main() {
    int arr[] = { 1, 2, 3, 4, 5, 6 };
    int key = 3;
    int size = 6;
    for (int i = 0; i < size; i++) {
        if (arr[i] == key) {
            printf("Element found at position: %d",
                   (i + 1));
            break;
        }
    }
    return 0;
}
```

Output

```
Element found at position: 3
```

B) continue

This loop control statement is just like the break statement. The *continue* statement is opposite to that of the break *statement*, instead of terminating the loop, it forces to execute the next iteration of the loop.

As the name suggests the continue statement forces the loop to continue or execute the next iteration. When the continue statement is executed in the loop, the code inside the loop following the continue statement will be skipped and the next iteration of the loop will begin.

Example:

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

```
int main() {
    for (int i = 1; i <= 10; i++) {
        // If i is equals to 6,
        // continue to next iteration
        // without printing
        if (i == 6)
            continue;
        else
            printf("%d ", i);
    }
    return 0;
}
```

Output

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

C) goto

The goto statement in C also referred to as the unconditional jump statement can be used to jump from one point to another within a function.

Examples:

```
#include <stdio.h>
int main() {
    int n = 1;
label:
    printf("%d ", n);
    n++;
    if (n <= 10)
        goto label;
    return 0;
}
```

Output

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

D) return

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

Example:

```
#include <stdio.h>
int sum(int a, int b) {
    int s1 = a + b;
    return s1;
}
int main()
```

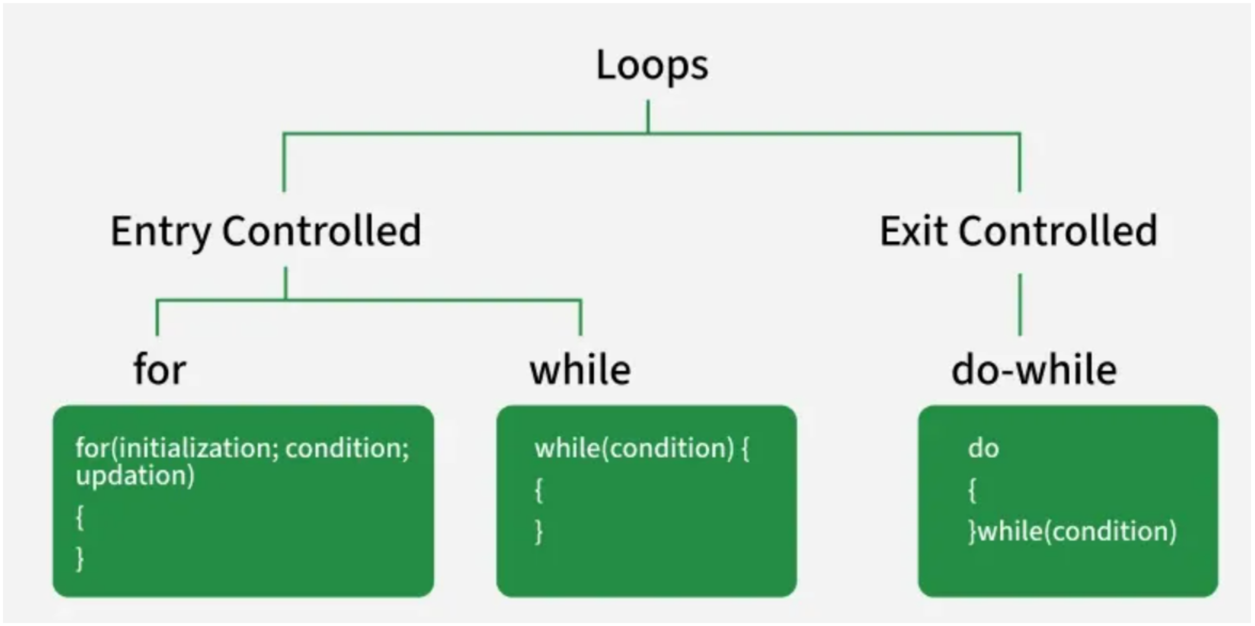
```
{
    int num1 = 10;
    int num2 = 10;
    int sumOf = sum(num1, num2);
    printf("%d", sumOf);
    return 0;
}
```
Output
```
20
```

What are loops in C?

Loops in C programming are used to repeat a block of code until the specified condition is met. It allows programmers to execute a statement or group of statements multiple times without writing the code again and again.

Types of Loops in C

There are 3 looping statements in C:



Let's discuss all 3 types of loops in C one by one.

for Loop

for loop is an **entry-controlled** loop, which means that the condition is checked before the loop's body executes.

Syntax
```
for (initialization; condition; updation) {
    // body of for loop
}
```
The various parts of the for loop are:

- **Initialization:** Initialize the variable to some initial value.
- **Test Condition:** This specifies the test condition. If the condition evaluates to true, then body of the loop is executed. If evaluated false, loop is terminated.

- **Update Expression:** After the execution loop's body, this expression increments/decrements the loop variable by some value.
- **Body of Loop**: Statements to repeat. Generally enclosed inside **{} braces.**

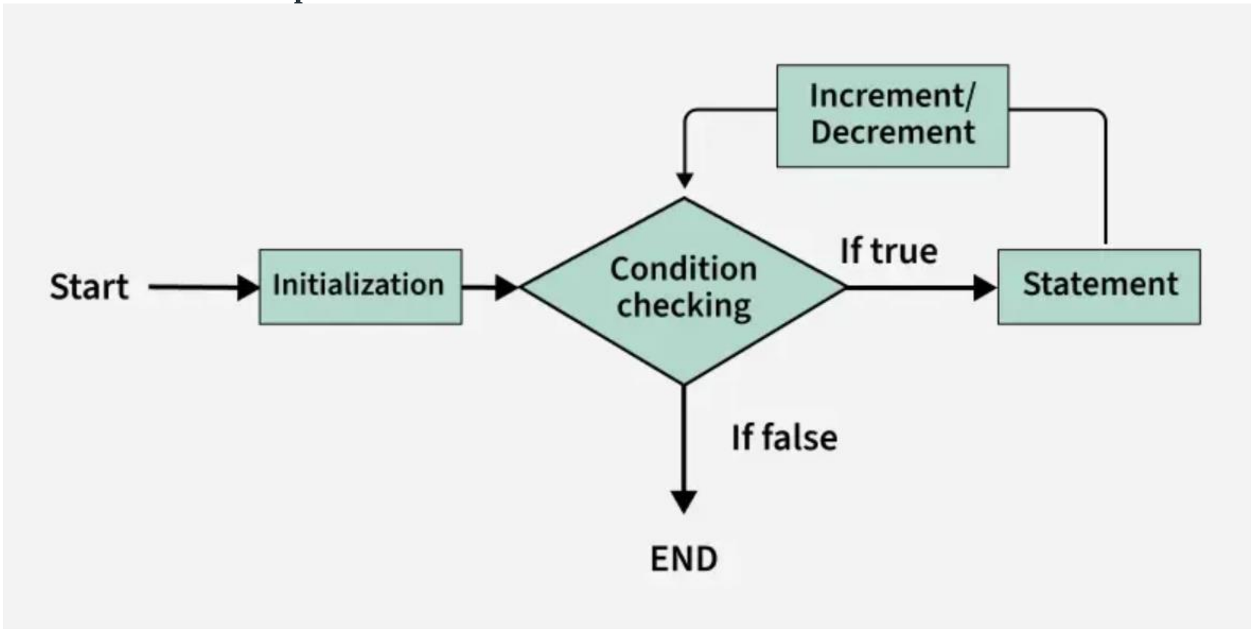
Example

```
#include <stdio.h>
int main() {
      // Loop to print numbers from 1 to 5
   for (int i = 0; i < 5; i++) {
      printf( "%d ", i + 1);
   }
      return 0;
}
```

Output

```
1 2 3 4 5
```

Flowchart of for Loop



while Loop

A **while loop** is also an **entry-controlled loop** in which the condition is checked before entering the body.

Syntax

```
while (condition) {
   // Body of the loop
}
```

Only the **condition** is the part of **while loop** syntax, we have to initialize and update loop variable manually.

Example:

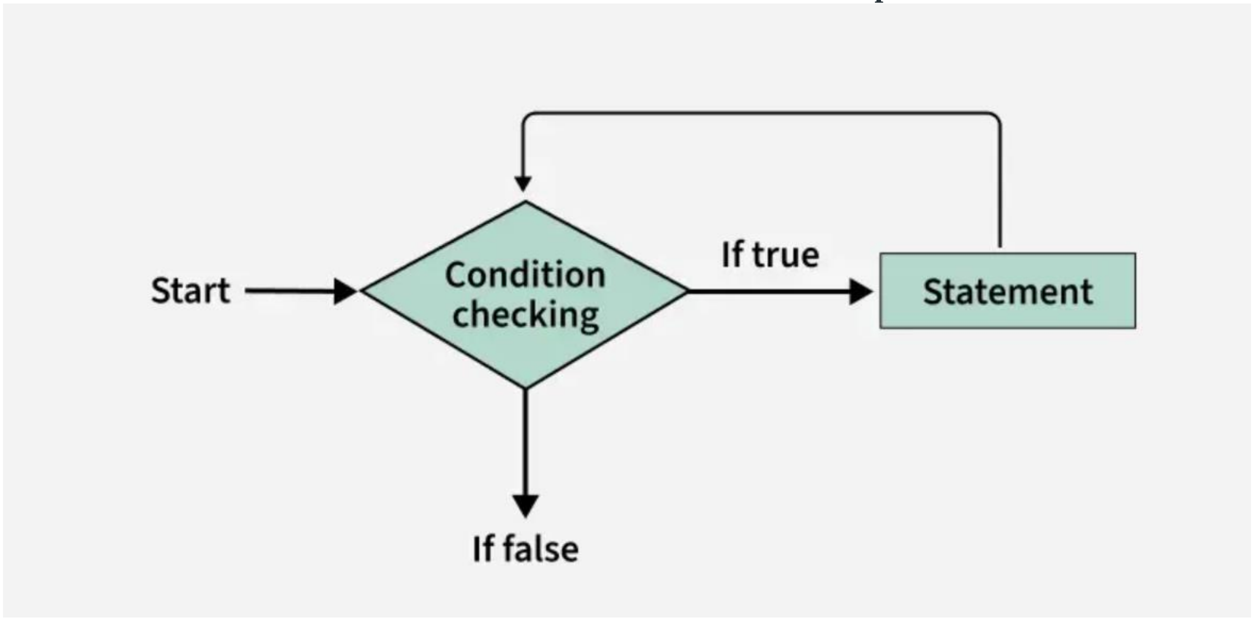
```
#include <stdio.h>
int main() {

  // Initialization expression
  int i = 0;
   // Test expression
  while(i <= 5) {
    printf("%d ", i + 1);
        // update expression
    i++;
  }
    return 0;
}
```

Output

```
1 2 3 4 5 6
```

Flowchart of while Loop

The below flowchart demonstrates execution flow of the **while loop.**



do-while Loop

The do-while loop is an **exit-controlled loop**, which means that the condition is checked after executing the loop body. Due to this, the loop body will **execute at least once** irrespective of the test condition.

Syntax

```
do {
    // Body of the loop
} while (condition);
```

Like while loop, only the condition is the part of **do while loop** syntax, we have to do the initialization and updating of loop variable manually.

Example:

```
#include <stdio.h>
 int main() {
      // Initialization expression
  int i = 0;
```

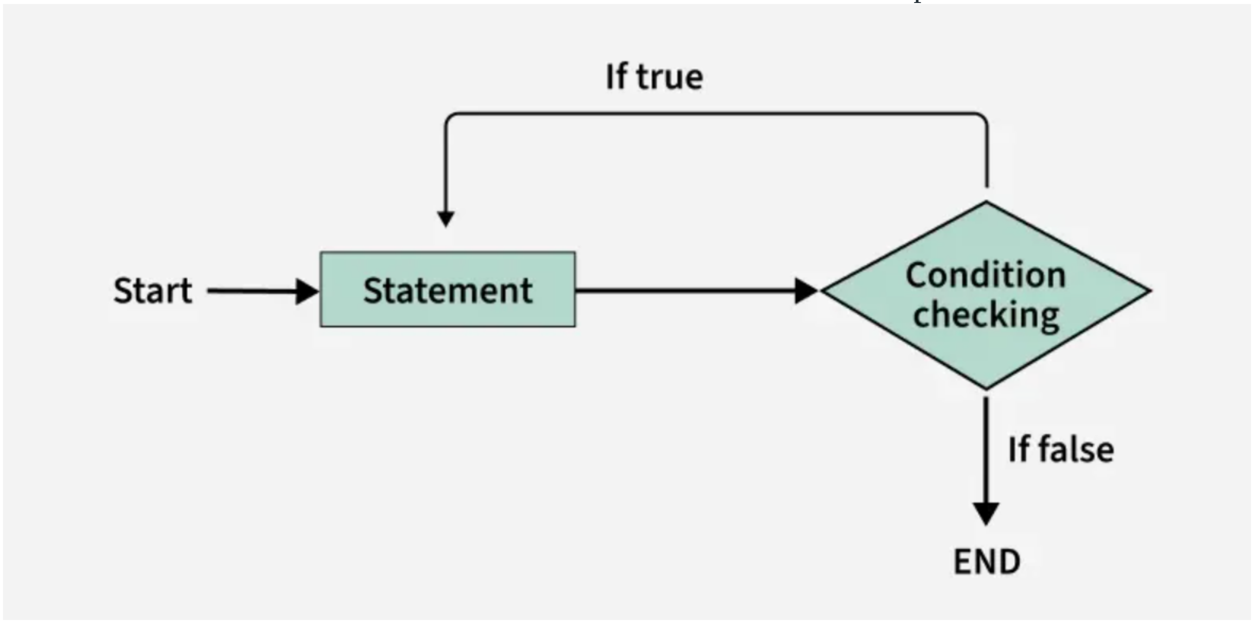
```
   do
  {
   // loop body
   printf( "%d ", i);
    // Update expression
   i++;
      // Condition to check
  }  while (i <= 10);
   return 0;
}
```

Output

```
0 1 2 3 4 5 6 7 8 9 10
```

Flowchart of do-while Loop

The below flowchart demonstrates execution flow of the do while loop.



Infinite Loop

An **infinite loop** is executed when the test expression never becomes false, and the body of the loop is executed repeatedly. A program is stuck in an Infinite loop when the condition is always true. Mostly this is an error that can be resolved by using Loop Control statements.

Using for loop:

```
#include <stdio.h>
int main () {
		// This is an infinite for loop
		// as the condition expression
		// is blank
		for ( ; ; ) {
				printf("This loop will run forever.");
		}
		return 0;
}
```

Output

```
This                    loop                    will                    run                    forever.
This                    loop                    will                    run                    forever.
This                    loop                    will                    run                    forever.
...
```

Using While loop:

```
#include <stdio.h>

int main()  {
        while (1)
                printf("This loop will run forever.\n");
        return 0;
}
```

Output

```
This                    loop                    will                    run                    forever.
This                    loop                    will                    run                    forever.
This                    loop                    will                    run                    forever.
...
```

Using the do-while loop:

```
#include <stdio.h>
int main() {
        do {
                printf("This loop will run forever.");
        } while (1);

        return 0;
}
```

Output

```
This                    loop                    will                    run                    forever.
This                    loop                    will                    run                    forever.
This                    loop                    will                    run                    forever.
...
```

Nested Loops

Nesting loops means placing one loop inside another. The inner loop runs fully for each iteration of the outer loop. This technique is helpful when you need to perform multiple iterations within each cycle of a larger loop, like when working with a two-dimensional array or performing tasks that require multiple levels of iteration.

Example:

```
#include <stdio.h>
int main() {
  // Outer loop runs 3 times
  for (int i = 0; i < 3; i++) {
        // Inner loop runs 2 times for each
    // outer loop iteration
    for (int j = 0; j < 2; j++) {
            printf("i = %d, j = %d\n", i, j);
        }
    }
    return 0;
}
```
Output

```
i = 0, j = 0

i = 0, j = 1

i = 1, j = 0

i = 1, j = 1

i = 2, j = 0

i = 2, j = 1
```

Loop Control Statements

Loop control statements in C programming are used to change execution from its normal sequence.

Name	Description
break	The break statement is used to terminate the loop statement.
continue	When encountered, the continue statement skips the remaining body and jumps to the next iteration of the loop.
goto	goto statement transfers the control to the labeled statement.

Example:

```
#include <stdio.h>
int main() {
    for (int i = 0; i < 5; i++) {
        if (i == 3) {
                    // Exit the loop when i equals 3
            break;
        }
        printf("%d ", i);
    }
    printf("\n");

    for (int i = 0; i < 5; i++) {
        if (i == 3) {

            // Skip the current iteration
            // when i equals 3
            continue;
        }
        printf("%d ", i);
    }
    printf("\n");
    for (int i = 0; i < 5; i++) {
        if (i == 3) {

            // Jump to the skip label when
            // i equals 3
            goto skip;
        }
        printf("%d ", i);
    }

skip:
    printf("\nJumped to the 'skip' label %s",
    "when i equals 3.");

    return 0;
}
```

Output

```
0 1 2

0 1 2 4

0 1 2

Jumped to the 'skip' label when i equals 3.
```

C Program with Switch Case Statements

We use decision making statements in C to control the order of execution of our program. Switch case in C is one the decision making statements which is mostly used when we have multiple alternatives to choose from.

The syntax of switch statement is:

```
switch(expression)
{
    case value1:
        statement_1;
 break;
case value2:
statement_2;
 break;
    //we can have as many cases as we want
case value_n:
        statement_n;
        break;
   default:
        default statement;     //this is not necessary. It is used only for convenience
}
```
Following is a simple to help you understand how a switch statement works:
The algorithm of switch statement is as follows:

1. Firstly, the expression inside the **switch(expression)** is evaluated.
2. Then, it is matched with the case value of each statement.
3. If it matches with one of the case values, we execute that particular set of statements with which it has matched. If the break statement is used after that case, we break out of switch otherwise keep executing till we reach the end of switch(because there is no break statement) or we reach another break statement.
4. If it does not match, we execute the default condition(if present) and come out of switch.

C Program to check if input character is a vowel using Switch Case
Below is a program to check vowel using switch case.
```
#include<stdio.h>
int main()
{
    printf("\n\n\t\tStudytonight - Best place to learn\n\n\n");
    char ch;
    printf("Input a Character :  ");
    scanf("%c", &ch);
    switch(ch)
    {
        case 'a':
        case 'A':
        case 'e':
        case 'E':
        case 'i':
        case 'I':
        case 'o':
        case 'O':
        case 'u':
        case 'U':
            printf("\n\n%c is a vowel.\n\n", ch);
            break;
        default:
            printf("%c is not a vowel.\n\n", ch);
    }
    printf("\n\n\t\t\tCoding is Fun !\n\n\n");
    return 0;
}
```
Output:
Check Vowel using Switch Case
Explanation:

If break statement is not used for a case then all the cases following the valid case are executed and evaluated. This way you can make your code easier to understand by writing only break statement only once to check multiple conditions in one go.

default is executed only if none of the above cases are true. It is similar to the else statement of the if-else code.

C Program to print the Largest and Smallest using Global Declaration

Some important points about Global variable declaration are:
It can be done anywhere within the program.
Unlike local variables that can be used within the scope of a particular function.
& is used to assign the input value to the variable and store it at that particular location.
%0nd is used to represent numbers in n digit format with leading 0's.
Below is a program to find largest and smallest value using global declaration.
```
#include<stdio.h>
int a,b;
int main()
{
    printf("\n\n\t\tStudytonight - Best place to learn\n\n\n");
    printf("\n\nEnter the two values to find the greatest and smallest number: \n");
    scanf("%d%d", &a, &b);
    if(a == b)
        printf("Both are equal\n");
    else if(a < b)
    {
        printf("\n\nThe largest number is %03d\n", b);
        printf("\nThe smallest number is %03d\n", a);
        printf("\nThe largest number is %03d\n", b);
    }
    else    //Only possibility remaining
    {
        printf("The largest number is %03d\n", a);
        printf("The smallest number is %03d\n", b);
    }
    printf("\n\n\t\t\tCoding is Fun !\n\n\n");
    return 0;
}
```

Output:

Largest and Smallest using Global Declaration

Basic for Loop Program

Every loop consists of three parts in a sequence

1. **Initialization**: Use to initialize the loop variable.
2. **Condition**: It is checked after each iteration as an entry point to the loop.
3. **Updation**: Incrementing the loop variable to eventually terminate the loop not satisfying the loop condition.

Basic do while Loop Program

Every loop consists of three parts in sequence:

1. **Initialization**: Use to initialize the loop variable.
2. **Condition**: It is checked after each iteration as an entry point to the loop.
3. **Updation**: Incrementing the loop variable to eventually terminate the loop not satisfying the loop condition.

Here is the C language tutorial explaining do while Loop → do while Loop in C

Do while loop is used when the actual code must be executed atleast **once**. For example: Incase of menu driven functions.

Below is a simple program on do while loop.

```
#include<stdio.h>
int main()
{
    printf("\n\n\t\tStudytonight - Best place to learn\n\n\n");
    /*  always declare the variables before using them  */
    int i = 10;     // declaration and initialization at the same time
 do // do contains the actual code and the updation
    {


        printf("i = %d\n",i);
        i = i-1;    // updation
    }
    // while loop doesn't contain any code but just the condition
    while(i > 0);

    printf("\n\The value of i after exiting the loop is %d\n\n", i);
    printf("\n\n\t\t\tCoding is Fun !\n\n\n");
    return 0;
}
```

Below is a program for sum of digits of a number.

```
#include<stdio.h>

int main()
{
		printf("\n\n\t\tStudytonight - Best place to learn\n\n\n");

		int n, sum = 0, c, remainder;

		printf("Enter the number you want to add digits of:  ");
		scanf("%d", &n);

		while(n != 0)
		{
		remainder = n%10;
		sum += remainder;
		n = n/10;
		}

		printf("\n\nSum of the digits of the entered number is  =  %d\n\n", sum);
		printf("\n\n\n\n\t\t\tCoding is Fun !\n\n\n");
		return 0;
}
```