"C" Programming

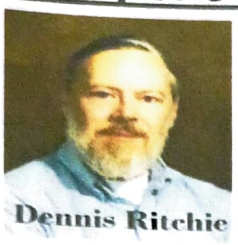
Milan Kumar Nayak
Lect.In Comp.Sc

UNIT-1

C Programming Language

C language Tutorial with programming approach for beginners and professionals, helps you to understand the C language tutorial easily. Our C tutorial explains each topic with programs.
The C Language is developed for creating system applications that directly interact with the hardware devices such as drivers, kernels, etc.
C programming is considered as the base for other programming languages, that is why it is known as mother language

History of C Language



History of C language is interesting to know. Here we are going to discuss a brief history of the c language.**C programming language** was developed in 1972 by Dennis Ritchie at bell laboratories of AT&T (American Telephone & Telegraph), located in the U.S.A.
Dennis Ritchie is known as the **founder of the c language.**It was developed to overcome the problems of previous languages such as B, BCPL, etc.Initially, C language was developed to be used in **UNIX operating system**. It inherits many features of previous languages such as B and BCPL. Let's see the programming languages that were developed before C language.

Language	Year	Developed By
Algol	1960	International Group
BCPL	1967	Martin Richard
B	1970	Ken Thompson
Traditional C	1972	Dennis Ritchie
K & R C	1978	Kernighan & Dennis Ritchie
ANSI C	1989	ANSI Committee
ANSI/ISO C	1990	ISO Committee
C99	1999	Standardization Committee

next>><<prev

Features of C Language

C is the widely used language. It provides many **features** that are given below.

1. Simple
2. Machine Independent or Portable
3. Mid-level programming language
4. structured programming language
5. Rich Library
6. Memory Management
7. Fast Speed
8. Pointers

9. Recursion
10. Extensible

1) Simple
C is a simple language in the sense that it provides a **structured approach** (to break the problem into parts), **the rich set of library functions**, **data types**, etc.

2) Machine Independent or Portable
Unlike assembly language, c programs **can be executed on different machines** with some machine specific changes. Therefore, C is a machine independent language

3) Mid-level programming language
Although, C is **intended to do low-level programming**. It is used to develop system applications such as kernel, driver, etc. It **also supports the features of a high-level language**. That is why it is known as mid-level language.

4) Structured programming language
C is a structured programming language in the sense that **we can break the program into parts using functions**. So, it is easy to understand and modify. Functions also provide code reusability.

5) Rich Library
C **provides a lot of inbuilt functions** that make the development fast.

6) Memory Management
It supports the feature of **dynamic memory allocation**. In C language, we can free the allocated memory at any time by calling the **free()** function.

7) Speed
The compilation and execution time of C language is fast since there are lesser inbuilt functions and hence the lesser overhead.

8) Pointer
C provides the feature of pointers. We can directly interact with the memory by using the pointers. We **can use pointers for memory, structures, functions, array**, etc.

9) Recursion
In C, we **can call the function within the function**. It provides code reusability for every function. Recursion enables us to use the approach of backtracking.

10) Extensible
C language is extensible because it **can easily adopt new features**.

How to install C

There are many compilers available for c and c++. You need to download any one. Here, we are going to use **Turbo C++**. It will work for both C and C++. To install the Turbo C software, you need to follow following steps.

1. Download Turbo C++
2. Create turboc directory inside c drive and extract the tc3.zip inside c:\turboc
3. Double click on install.exe file
4. Click on the tc application file located inside c:\TC\BIN to write the c program

Keywords in C

A keyword is a **reserved word**. You cannot use it as a variable name, constant name, etc. There are only 32 reserved words (keywords) in the C language.

A list of 32 keywords in the c language is given below:

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void	volatile	while

Identifiers

Identifiers are names for entities in a C program, such as variables, arrays, functions, structures, unions and labels. An identifier can be composed only of uppercase, lowercase letters, underscore and digits, but should start only with an alphabet or an underscore. If the identifier is not used in an external link process, then it is called as **internal**. Example: Local variable. If the identifier is used in an external link process, then it is called as **external.** Example: Global variable

An identifier is a string of alphanumeric characters that begins with an alphabetic character or an underscore character that are used to represent various programming elements such as variables, functions, arrays, structures, unions and so on. Actually, an identifier is a user-defined word. There are 53 characters, to represent identifiers. They are 52 alphabetic characters (i.e., both uppercase and lowercase alphabets) and the underscore character. The underscore character is considered as a letter in identifiers. The underscore character is usually used in the middle of an identifier. There are 63 alphanumeric characters, i.e., 53 alphabetic characters and 10 digits (i.e., 0-9).

Rules for constructing identifiers

1. The first character in an identifier must be an alphabet or an underscore and can be followed only by any number alphabets, or digits or underscores.
2. They must not begin with a digit.
3. Uppercase and lowercase letters are distinct. That is, identifiers are case sensitive.
4. Commas or blank spaces are not allowed within an identifier.
5. Keywords cannot be used as an identifier.
6. Identifiers should not be of length more than 31 characters.
7. Identifiers must be meaningful, short, quickly and easily typed and easily read.

Valid identifiers: total sum average _x y_ mark_1 x1

Invalid identifiers

1x - begins with a digit
char - reserved word
x+y - special character

Variables in C

A **variable** is a name of the memory location. It is used to store data. Its value can be changed, and it can be reused many times.

It is a way to represent memory location through symbol so that it can be easily identified.

Let's see the syntax to declare a variable:

```
type variable_list;
```

The example of declaring the variable is given below:

```
int a;
float b;
char c;
```

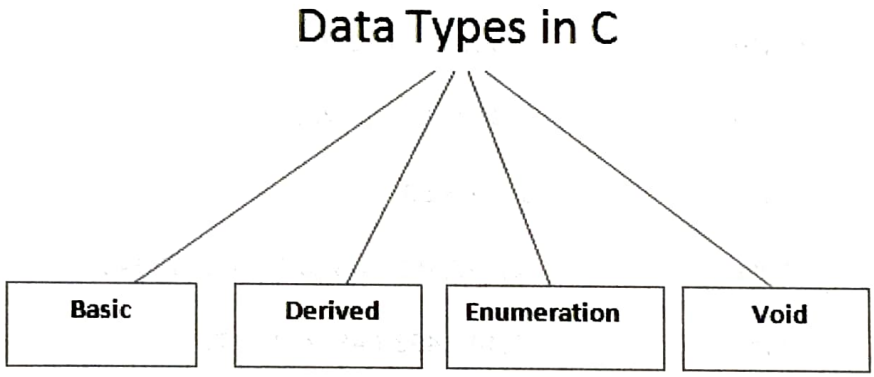
Here, a, b, c are variables. The int, float, char are the data types.

We can also provide values while declaring the variables as given below:

```
int a=10,b=20;//declaring 2 variable of integer type
float f=20.8;
char c='A';
```

Data Types in C

A data type specifies the type of data that a variable can store such as integer, floating, character, etc.



There are the following data types in C language.

Types	Data Types
Basic Data Type	int, char, float, double
Derived Data Type	array, pointer, structure, union
Enumeration Data Type	enum
Void Data Type	void

The memory size of the basic data types may change according to 32 or 64-bit operating system.

Let's see the basic data types. Its size is given **according to 32-bit architecture**.

Data Types	Memory Size	Range
char	1 byte	−128 to 127
signed char	1 byte	−128 to 127
unsigned char	1 byte	0 to 255
short	2 byte	−32,768 to 32,767
signed short	2 byte	−32,768 to 32,767
unsigned short	2 byte	0 to 65,535
int	2 byte	−32,768 to 32,767
signed int	2 byte	−32,768 to 32,767
unsigned int	2 byte	0 to 65,535
short int	2 byte	−32,768 to 32,767
signed short int	2 byte	−32,768 to 32,767
unsigned short int	2 byte	0 to 65,535
long int	4 byte	-2,147,483,648 to 2,147,483,647
signed long int	4 byte	-2,147,483,648 to 2,147,483,647
unsigned long int	4 byte	0 to 4,294,967,295
float	4 byte	
double	8 byte	
long double	10 byte	

Constants in C

A constant is a value or variable that can't be changed in the program, for example: 10, 20, 'a', 3.4, "c programming" etc.
There are different types of constants in C programming.

List of Constants in C

Constant	Example
Decimal Constant	10, 20, 450 etc.
Real or Floating-point Constant	10.3, 20.2, 450.6 etc.
Octal Constant	021, 033, 046 etc.
Hexadecimal Constant	0x2a, 0x7b, 0xaa etc.
Character Constant	'a', 'b', 'x' etc.
String Constant	"c", "c program", "c in javatpoint" etc.

2 ways to define constant in C

There are two ways to define constant in C programming.

1. const keyword
2. #define preprocessor

1) C const keyword

The const keyword is used to define constant in C programming.

1. `const float PI=3.14;`

Now, the value of PI variable can't be changed.

```
1. #include<stdio.h>
2. int main(){
3.     const float PI=3.14;
4.     printf("The value of PI is: %f",PI);
5.     return 0;
6. }
```

Output:

```
The value of PI is: 3.140000
```

If you try to change the the value of PI, it will render compile time error.

```
1. #include<stdio.h>
2. int main(){
3. const float PI=3.14;
4. PI=4.5;
5. printf("The value of PI is: %f",PI);
6.     return 0;
```

```
7. }
```

Output:

```
Compile Time Error: Cannot modify a const object
```

2) C #define preprocessor

The #define preprocessor is also used to define constant. We will learn about #define preprocessor directive later.

First C Program

Before starting the abcd of C language, you need to learn how to write, compile and run the first c program.

To write the first c program, open the C console and write the following code:

```
1. #include <stdio.h>
2. int main(){
3. printf("Hello C Language");
4. return 0;
5. }
```

#include <stdio.h> includes the **standard input output** library functions. The printf() function is defined in stdio.h .

Addition program in C

```
#include<stdio.h>

int main()
{
   int a, b, c;

   printf("Enter two numbers to add\n");
   scanf("%d%d", &a, &b);

   c = a + b;

   printf("Sum of the numbers = %d\n", c);

   return 0;
}
```

How to compile and run the c program

There are 2 ways to compile and run the c program, by menu and by shortcut.

By menu

Now **click on the compile menu then compile sub menu** to compile the c program.

Then **click on the run menu then run sub menu** to run the c program.

By shortcut

Or, press ctrl+f9 keys compile and run the program directly.

Comments in C

Comments in C language are used to provide information about lines of code. It is widely used for documenting code. There are 2 types of comments in the C language.

1. Single Line Comments
2. Multi-Line Comments

Single Line Comments

Single line comments are represented by double slash \\. Let's see an example of a single line comment in C.

```
1. #include<stdio.h>
2. int main(){
3.    //printing information
4.    printf("Hello C");
5. return 0;
6. }
```

Mult Line Comments

Multi-Line comments are represented by slash asterisk * ... *\. It can occupy many lines of code, but it can't be nested. Syntax:

```
/*  code to be commented */
```

Let's see an example of a multi-Line comment in C.

```
#include<stdio.h>
int main(){
   /*printing information
     Multi-Line Comment*/     printf("Hello C");
return 0;
]}
```

Input/Output Operation

The printf() and scanf() functions are used for input and output in C language. Both functions are inbuilt library functions, defined in stdio.h (header file).

printf() function

The **printf() function** is used for output. It prints the given statement to the console.
The syntax of printf() function is given below:

C Operators

An operator is simply a symbol that is used to perform operations. There can be many types of operations like arithmetic, logical, bitwise, etc.
There are following types of operators to perform different types of operations in C language.

- Arithmetic Operators
- Relational Operators
- Shift Operators
- Logical Operators
- Bitwise Operators

- Ternary or Conditional Operators
- Assignment Operator
- Misc Operator

Precedence of Operators in C

The precedence of operator species that which operator will be evaluated first and next. The associativity specifies the operator direction to be evaluated; it may be left to right or right to left.

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<=	Right to left
Comma	,	Left to right

Escape Sequence in C

An escape sequence in C language is a sequence of characters that doesn't represent itself when used inside string literal or character.
It is composed of two or more characters starting with backslash \. For example: \n represents new line.

List of Escape Sequences in C

Escape Sequence	Meaning
\a	Alarm or Beep
\b	Backspace
\f	Form Feed
\n	New Line
\r	Carriage Return
\t	Tab (Horizontal)
\v	Vertical Tab
\\	Backslash
\'	Single Quote
\"	Double Quote
\?	Question Mark
\nnn	octal number
\xhh	hexadecimal number
\0	Null

C if...else Statement

In this tutorial, you will learn about if statement (including if...else and nested if..else) in C programming with the help of examples.

C if Statement

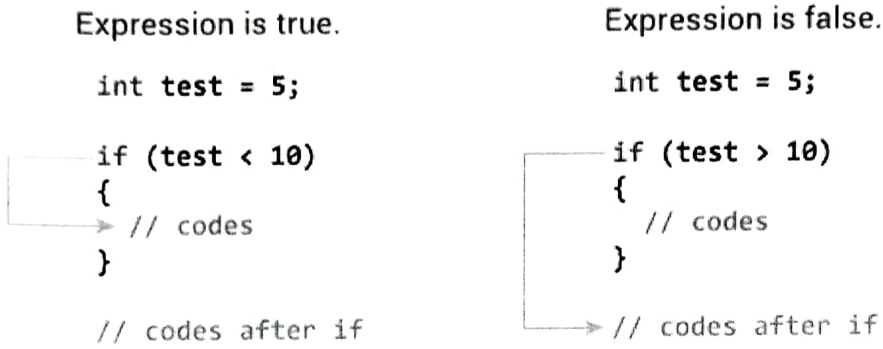
The syntax of the `if` statement in C programming is:

```
if (test expression)
{
   // statements to be executed if the test expression is true
}
```

How if statement works?

The `if` statement evaluates the test expression inside the parenthesis `()`.

- If the test expression is evaluated to true, statements inside the body of `if` are executed.
- If the test expression is evaluated to false, statements inside the body of `if` are not executed.



Example 1: if statement

```
// Program to display a number if it is negative

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

    printf("Enter an integer: ");
    scanf("%d", &number);

    // true if number is less than 0
    if (number < 0) {
        printf("You entered %d.\n", number);
    }

    printf("The if statement is easy.");

    return 0;
}
```

Output 1

```
Enter an integer: -2
You entered -2.
The if statement is easy.
```

When the user enters -2, the test expression `number<0` is evaluated to true. Hence, `You entered -2` is displayed on the screen.

Output 2

```
Enter an integer: 5
The if statement is easy.
```

When the user enters 5, the test expression `number<0` is evaluated to false and the statement inside the body of `if` is not executed

C if...else Statement

The `if` statement may have an optional `else` block. The syntax of the `if..else` statement is:

```
if (test expression) {
    // statements to be executed if the test expression is true
}
else {
    // statements to be executed if the test expression is false
}
```

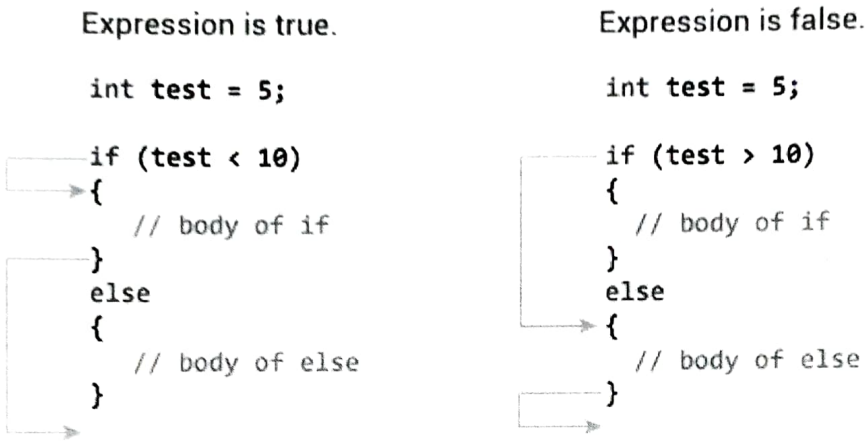
How if...else statement works?

If the test expression is evaluated to true,

- statements inside the body of `if` are executed.
- statements inside the body of `else` are skipped from execution.

If the test expression is evaluated to false,

- statements inside the body of `else` are executed
- statements inside the body of `if` are skipped from execution.



Example 2: if...else statement

```
// Check whether an integer is odd or even

#include <stdio.h>
int main() {
    int number;
    printf("Enter an integer: ");
    scanf("%d", &number);

    // True if the remainder is 0
    if  (number%2 == 0) {
        printf("%d is an even integer.",number);
    }
    else {
     printf("%d is an odd integer.",number);
   }

    return 0;
}
```

Output

```
Enter an integer: 7
7 is an odd integer.
```

When the user enters 7, the test expression `number%2==0` is evaluated to false. Hence, the statement inside the body of `else` is executed.

C if...else Ladder

The `if...else` statement executes two different codes depending upon whether the test expression is true or false. Sometimes, a choice has to be made from more than 2 possibilities. The if...else ladder allows you to check between multiple test expressions and execute different statements.

Syntax of nested if...else statement.

```
if (test expression1) {
   // statement(s)
}
else if(test expression2) {
   // statement(s)
 }
 else if (test expression3) {
    // statement(s)
 }
 .
 .
 else {
    // statement(s)
 }
```

Example 3: C if...else Ladder

```
// Program to relate two integers using =, > or < symbol

#include <stdio.h>
int main() {
    int number1, number2;
    printf("Enter two integers: ");
    scanf("%d %d", &number1, &number2);

    //checks if the two integers are equal.
    if(number1 == number2) {
        printf("Result: %d = %d",number1,number2);
    }

    //checks if number1 is greater than number2.
    else if (number1 > number2) {
        printf("Result: %d > %d", number1, number2);
    }

    //checks if both test expressions are false
    else {
        printf("Result: %d < %d",number1, number2);
    }

    return 0;
}
```

Output

```
Enter two integers: 12
23
Result: 12 < 23
```

Nested if...else

It is possible to include an `if...else` statement inside the body of another `if...else` statement.

Example 4: Nested if...else

This program given below relates two integers using either <, > and = similar to the `if...else` ladder's example. However, we will use a nested `if...else` statement to solve this problem.

```
#include <stdio.h>
int main() {
    int number1, number2;
    printf("Enter two integers: ");
    scanf("%d %d", &number1, &number2);

    if (number1 >= number2) {
      if (number1 == number2) {
        printf("Result: %d = %d",number1,number2);
      }
      else {
        printf("Result: %d > %d", number1, number2);
      }
    }
    else {
        printf("Result: %d < %d",number1, number2);
    }

     return 0;
 }
```

If the body of an `if...else` statement has only one statement, you do not need to use brackets `{}`.

For example, this code

```
1. if (a > b) {
2.      print("Hello");
3. }
4. print("Hi");
```

is equivalent to

```
if (a > b)
     print("Hello");
print("Hi");
```

UNIT-2

What are Loops?

In looping, a program executes the sequence of statements many times until the stated condition becomes false. A loop consists of two parts, a body of a loop and a control statement. The control statement is a combination of some conditions that direct the body of the loop to execute until the specified condition becomes false.

Types of Loops

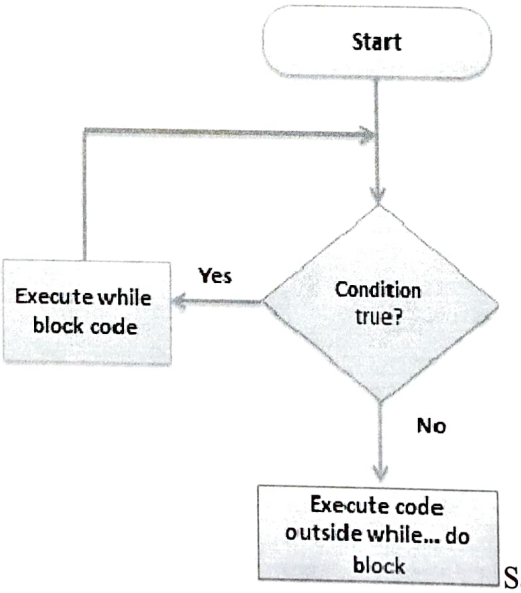
Depending upon the position of a control statement in a program, a loop is classified into two types:

1. Entry controlled loop

2. Exit controlled loop

In an **entry controlled loop,** a condition is checked before executing the body of a loop. It is also called as a pre-checking loop.

In an **exit controlled loop**, a condition is checked after executing the body of a loop. It is also called as a post-checking loop.



Sample Loop

The control conditions must be well defined and specified otherwise the loop will execute an infinite number of times. The loop that does not stop executing and processes the statements number of times is called as an **infinite loop**. An infinite loop is also called as an "**Endless loop**." Following are some characteristics of an infinite loop:

1. No termination condition is specified.

2. The specified conditions never meet.

The specified condition determines whether to execute the loop body or not.

'C' programming language provides us with three types of loop constructs:

1. The while loop

2. The do-while loop

3. The for loop

While Loop

A while loop is the most straightforward looping structure. The basic format of while loop is as follows:

```
while (condition) {
            statements;
}
```

It is an entry-controlled loop. In while loop, a condition is evaluated before processing a body of the loop. If a condition is true then and only then the body of a loop is executed. After the body of a loop is executed then control again goes back at the beginning, and the condition is checked if it is true, the same process is executed until the condition becomes false. Once the condition becomes false, the control goes out of the loop.

After exiting the loop, the control goes to the statements which are immediately after the loop. The body of a loop can contain more than one statement. If it contains only one statement, then the curly braces are not compulsory. It is a good practice though to use the curly braces even we have a single statement in the body.

In while loop, if the condition is not true, then the body of a loop will not be executed, not even once. It is different in do while loop which we will see shortly.

Following program illustrates a while loop:

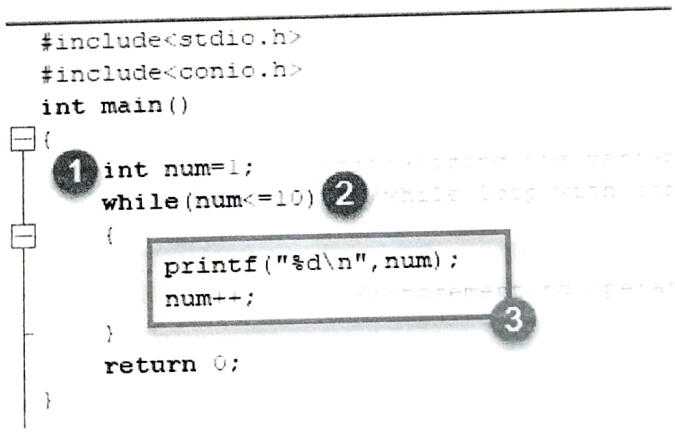
```
#include<stdio.h>
#include<conio.h>
int main()
{
        int num=1;       //initializing the variable
```

```
        while(num<=10)   //While loop with condition
        {
                printf("%d\n",num);
                num++;          //incrementing operation
        }
        return 0;
}
```

Output:

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

The above program illustrates the use of while loop. In the above program, we have printed series of numbers from 1 to 10 using a while loop.



1. We have initialized a variable called num with value 1. We are going to print from 1 to 10 hence the variable is initialized with value 1. If you want to print from 0, then assign the value 0 during initialization.
2. In a while loop, we have provided a condition (num<=10), which means the loop will execute the body until the value of num becomes 10. After that, the loop will be terminated, and control will fall outside the loop.
3. In the body of a loop, we have a print function to print our number and an increment operation to increment the value per execution of a loop. An initial value of num is 1, after the execution, it will become 2, and during the next

execution, it will become 3. This process will continue until the value becomes 10 and then it will print the series on console and terminate the loop.

\n is used for formatting purposes which means the value will be printed on a new line.

Do-While loop

A do-while loop is similar to the while loop except that the condition is always executed after the body of a loop. It is also called an exit-controlled loop.

The basic format of while loop is as follows:

```
do {
  statements
} while (expression);
```

As we saw in a while loop, the body is executed if and only if the condition is true. In some cases, we have to execute a body of the loop at least once even if the condition is false. This type of operation can be achieved by using a do-while loop.

In the do-while loop, the body of a loop is always executed at least once. After the body is executed, then it checks the condition. If the condition is true, then it will again execute the body of a loop otherwise control is transferred out of the loop.

Similar to the while loop, once the control goes out of the loop the statements which are immediately after the loop is executed.

The critical difference between the while and do-while loop is that in while loop the while is written at the beginning. In do-while loop, the while condition is written at the end and terminates with a semi-colon (;)

The following program illustrates the working of a do-while loop:

We are going to print a table of number 2 using do while loop.

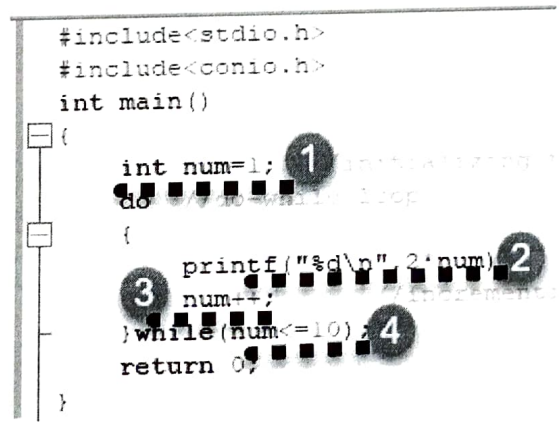
```
#include<stdio.h>
#include<conio.h>
int main()
{
        int num=1;      //initializing the variable
        do      //do-while loop
        {
                printf("%d\n",2*num);
                num++;          //incrementing operation
```

```
        }while(num<=10);
        return 0;
}
```

Output:

```
2
4
6
8
10
12
14
16
18
20
```

In the above example, we have printed multiplication table of 2 using a do-while loop. Let's see how the program was able to print the series.



1. First, we have initialized a variable 'num' with value 1. Then we have written a do-while loop.
2. In a loop, we have a print function that will print the series by multiplying the value of num with 2.
3. After each increment, the value of num will increase by 1, and it will be printed on the screen.
4. Initially, the value of num is 1. In a body of a loop, the print function will be executed in this way: 2*num where num=1, then 2*1=2 hence the value two will be printed. This will go on until the value of num becomes 10. After that loop will be terminated and a statement which is immediately after the loop will be executed. In this case return 0.

For loop

A for loop is a more efficient loop structure in 'C' programming. The general structure of for loop is as follows:

```
for (initial value; condition; incrementation or decrementation )
{
  statements;
}
```

- The initial value of the for loop is performed only once.
- The condition is a Boolean expression that tests and compares the counter to a fixed value after each iteration, stopping the for loop when false is returned.
- The incrementation/decrementation increases (or decreases) the counter by a set value.

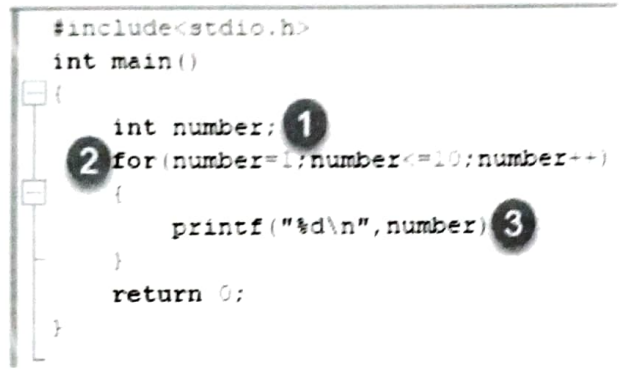
Following program illustrates the use of a simple for loop:

```
#include<stdio.h>
int main()
{
        int number;
        for(number=1;number<=10;number++) //for loop to print 1-10 numbers
        {
                printf("%d\n",number);            //to print the number
        }
        return 0;
}
```

Output:

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

The above program prints the number series from 1-10 using for loop.



1. We have declared a variable of an int data type to store values.
2. In for loop, in the initialization part, we have assigned value 1 to the variable number. In the condition part, we have specified our condition and then the increment part.
3. In the body of a loop, we have a print function to print the numbers on a new line in the console. We have the value one stored in number, after the first iteration the value will be incremented, and it will become 2. Now the variable number has the value 2. The condition will be rechecked and since the condition is true loop will be executed, and it will print two on the screen. This loop will keep on executing until the value of the variable becomes 10. After that, the loop will be terminated, and a series of 1-10 will be printed on the screen.

In C, the for loop can have multiple expressions separated by commas in each part.

For example:

```
for (x = 0, y = num; x < y; i++, y--) {
  statements;
}
```

Also, we can skip the initial value expression, condition and/or increment by adding a semicolon.

For example:

```
int i=0;
int max = 10;
for (; i < max; i++) {
  printf("%d\n", i);
}
```

Notice that loops can also be nested where there is an outer loop and an inner loop. For each iteration of the outer loop, the inner loop repeats its entire cycle.

Consider the following example, that uses nested for loops output a multiplication table:

```
#include <stdio.h>
int main() {
int i, j;
int table = 2;
int max = 5;
for (i = 1; i <= table; i++) { // outer loop
  for (j = 0; j <= max; j++) { // inner loop
    printf("%d x %d = %d\n", i, j, i*j);
  }
  printf("\n"); /* blank line between tables */
}}
```

Output:

```
1 x 0 = 0
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
1 x 5 = 5

2 x 0 = 0
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
```

The nesting of for loops can be done up-to any level. The nested loops should be adequately indented to make code readable. In some versions of 'C,' the nesting is limited up to 15 loops, but some provide more.

The nested loops are mostly used in array applications which we will see in further tutorials.

Introduction to C Programming

Arrays

Overview

- An array is a collection of data items, all of the same type, accessed using a common name.
- A one-dimensional array is like a list; A two dimensional array is like a table; The C language places no limits on the number of dimensions in an array, though specific implementations may.
- Some texts refer to one-dimensional arrays as ***vectors***, two-dimensional arrays as ***matrices***, and use the general term ***arrays*** when the number of dimensions is unspecified or unimportant.

Declaring Arrays

- Array variables are declared identically to variables of their data type, except that the variable name is followed by one pair of square [ ] brackets for each dimension of the array.
- Uninitialized arrays must have the dimensions of their rows, columns, etc. listed within the square brackets.
- Dimensions used when declaring arrays in C must be positive integral constants or constant expressions.
    - In C99, dimensions must still be positive integers, but variables can be used, so long as the variable has a positive value at the time the array is declared. (Space is allocated only once, at the time the array is declared. The array does NOT change sizes later if the variable used to declare it changes.)
- Examples:

```
int i, j, intArray[ 10 ], number;
float floatArray[ 1000 ];
int tableArray[ 3 ][ 5 ];     /* 3 rows by 5 columns */

const int NROWS = 100;     // ( Old code would use #define NROWS 100 )
const int NCOLS = 200;     // ( Old code would use #define NCOLS 200 )
float matrix[ NROWS ][ NCOLS ];
```

- C99 Only Example:

```
int numElements;
printf( "How big an array do you want? " );
scanf( "%d", &numElements; )
if( numElements <= 0 ) {
   printf( "Error - Quitting\n" );
   exit( 0 );
}
double data[ numElements ];   // This only works in C99, not in plain C
```

Initializing Arrays

- Arrays may be initialized when they are declared, just as any other variables.
- Place the initialization data in curly {} braces following the equals sign. Note the use of commas in the examples below.

- An array may be partially initialized, by providing fewer data items than the size of the array. The remaining array elements will be automatically initialized to zero.
- If an array is to be completely initialized, the dimension of the array is not required. The compiler will automatically size the array to fit the initialized data. (Variation: Multidimensional arrays - see below.)
- Examples:

```
int i = 5, intArray[ 6 ] = { 1, 2, 3, 4, 5, 6 }, k;
float sum = 0.0f, floatArray[ 100 ] = { 1.0f, 5.0f, 20.0f };
double piFractions[ ] = { 3.141592654, 1.570796327, 0.785398163 };
```

Designated Initializers:

- In C99 there is an alternate mechanism, that allows you to initialize specific elements, not necessarily at the beginning.
- This method can be mixed in with traditional iniitalization
- For example:

```
int numbers[ 100 ] = { 1, 2, 3, [10] = 10, 11, 12, [60] = 50, [42] = 420 };
```

  - In this example,the first three elements are initialized to 1, 2, and 3 respectively.
  - Then element 10 (the 11th element) is initialized to 10
  - The next two elements (12th and 13th) are initialized to 11 and 12 respectively.
  - Element number 60 (the 61st) is initialized to 50, and number 42 (the 43rd) to 420.
    - (Note that the designated initializers do not need to appear in order.)
  - As with traditional methods, all uninitialized values are set to zero.
  - If the size of the array is not given, then the largest initialized position determines the size of the array.

Using Arrays

- Elements of an array are accessed by specifying the index (offset) of the desired element within square [ ] brackets after the array name.
- Array subscripts must be of integer type. (int, long int, char, etc.)
- **VERY IMPORTANT:** Array indices start at zero in C, and go to one less than the size of the array. For example, a five element array will have indices zero through four. This is because the index in C is actually an offset from the beginning of the array. (The first element is at the beginning of the array, and hence has zero offset.)
- **Landmine:** The most common mistake when working with arrays in C is forgetting that indices start at zero and stop one less than the array size.
- Arrays are commonly used in conjunction with loops, in order to perform the same calculations on all (or some part) of the data items in the array.

Multidimensional Arrays

- Multi-dimensional arrays are declared by providing more than one set of square [ ] brackets after the variable name in the declaration statement.
- One dimensional arrays do not require the dimension to be given if the array is to be completely initialized. By analogy, multi-dimensional arrays do not require **the first** dimension to be given if the array is to be completely initialized. All dimensions after the first must be given in any case.
- For two dimensional arrays, the first dimension is commonly considered to be the number of rows, and the second dimension the number of columns. We will use this convention when discussing two dimensional arrays.
- Two dimensional arrays are considered by C/C++ to be an array of (single dimensional arrays). For example, "int numbers[5][6]" would refer to a single dimensional array of 5 elements, wherein each element is a single dimensional array of 6 integers. By extension, "int numbers[12][5][6]" would refer to an array of twelve elements, each of which is a two dimensional array, and so on.
- Another way of looking at this is that C stores two dimensional arrays by rows, with all elements of a row being stored together as a single unit. Knowing this can sometimes lead to more efficient programs.
- Multidimensional arrays may be completely initialized by listing all data elements within a single pair of curly {} braces, as with single dimensional arrays.
- It is better programming practice to enclose each row within a separate subset of curly {} braces, to make the program more readable. This is required if any row other than the last is to be partially initialized. When subsets of braces are used, the last item within braces is not followed by a comma, but the subsets are themselves separated by commas.
- Multidimensional arrays may be partially initialized by not providing complete initialization data. Individual rows of a multidimensional array may be partially initialized, provided that subset braces are used.
- Individual data items in a multidimensional array are accessed by fully qualifying an array element. Alternatively, a smaller dimensional array may be accessed by partially qualifying the array name. For example, if "data" has been declared as a three dimensional array of floats, then data[1][2][5] would refer to a float, data[1][2] would refer to a one-dimensional array of floats, and data[1] would refer to a two-dimensional array of floats. The reasons for this and the incentive to do this relate to memory-management issues that are beyond the scope of these notes.

Sample Programs Using 1-D Arrays

- The first sample program uses loops and arrays to calculate the first twenty Fibonacci numbers. Fibonacci numbers are used to determine the sample points used in certain optimization methods.

```
/* Program to calculate the first 20 Fibonacci numbers. */

#include <stdlib.h>
#include <stdio.h>

int main( void ) {

   int i, fibonacci[ 20 ];

   fibonacci[ 0 ] = 0;
   fibonacci[ 1 ] = 1;

   for( i = 2; i < 20; i++ )
      fibonacci[ i ] = fibonacci[ i - 2 ] + fibonacci[ i - 1 ];

   for( i = 0; i < 20; i++ )
      printf( "Fibonacci[ %d ] = %f\n", i, fibonacci[ i ] );

} /* End of sample program to calculate Fibonacci numbers */
```

- **Exercise:** What is the output of the following program:

```
/* Sample Program Using Arrays */

#include <stdlib.h>
#include <stdio.h>

int main( void ) {

   int numbers[ 10 ];
   int i, index = 2;

   for( i = 0; i < 10; i++ )
      numbers[ i ] = i * 10;

   numbers[ 8 ] = 25;
   numbers[ 5 ] = numbers[ 9 ] / 3;
   numbers[ 4 ] += numbers[ 2 ] / numbers[ 1 ];
   numbers[ index ] = 5;
   ++numbers[ index ];
   numbers[ numbers[ index++ ] ] = 100;
   numbers[ index ] = numbers[ numbers[ index + 1 ] / 7 ]--;

   for( index = 0; index < 10; index++ )
      printf( "numbers[ %d ] = %d\n" index, numbers[ index ] );

} /* End of second sample program */
```

Sample Program Using 2-D Arrays

```
/* Sample program Using 2-D Arrays */

#include <stdlib.h>
#include <stdio.h>

int main( void ) {

  /* Program to add two multidimensional arrays */
  /* Written May 1995 by George P. Burdell */

  int a[ 2 ][ 3 ] = { { 5, 6, 7 }, { 10, 20, 30 } };
  int b[ 2 ][ 3 ] = { { 1, 2, 3 }, {  3,  2,  1 } };
  int sum[ 2 ][ 3 ], row, column;

  /* First the addition */

  for( row = 0; row < 2; row++ )
    for( column = 0; column < 3; column++ )
      sum[ row ][ column ] =
        a[ row ][ column ] + b[ row ][ column ];

  /* Then print the results */

  printf( "The sum is: \n\n" );

  for( row = 0; row < 2; row++ ) {
    for( column = 0; column < 3; column++ )
      printf( "\t%d",  sum[ row ][ column ] );
    printf( '\n' );   /* at end of each row */
  }

  return 0;
```

Introduction to C Pointers

A Pointer in C language is a variable which holds the address of another variable of same data type.

Pointers are used to access memory and manipulate the address.

Pointers are one of the most distinct and exciting features of C language. It provides power and flexibility to the language. Although pointers may appear a little confusing and complicated in the beginning, but trust me, once you understand the concept, you will be able to do so much more with C language.

Before we start understanding what pointers are and what they can do, let's start by understanding what does "Address of a memory location" means?

Address in C

Whenever a variable is defined in C language, a memory location is assigned for it, in which it's value will be stored. We can easily check this memory address, using the & symbol.

If `var` is the name of the variable, then `&var` will give it's address.

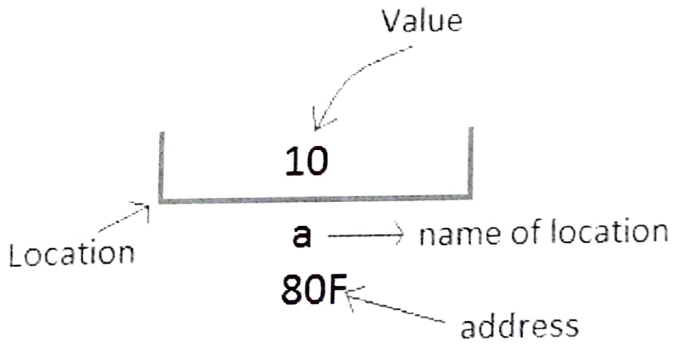
Let's write a small program to see memory address of any variable that we define in our program.

Concept of Pointers

Whenever a **variable** is declared in a program, system allocates a location i.e an address to that variable in the memory, to hold the assigned value. This location has its own address number, which we just saw above.

Let us assume that system has allocated memory location `80F` for a variable `a`.

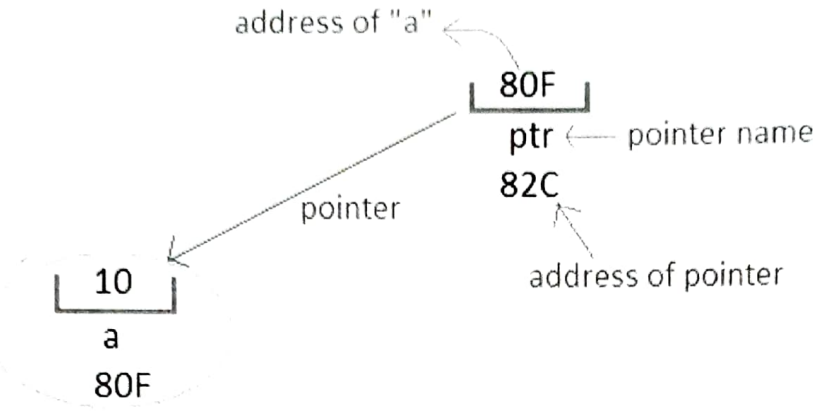
```
int a = 10;
```



We can access the value `10` either by using the variable name `a` or by using its address `80F`.

The question is how we can access a variable using it's address? Since the memory addresses are also just numbers, they can also be assigned to some other variable. The variables which are used to hold memory addresses are called **Pointer variables**.

A **pointer** variable is therefore nothing but a variable which holds an address of some other variable. And the value of a **pointer variable** gets stored in another memory location.



Benefits of using pointers

Below we have listed a few benefits of using pointers:

1. Pointers are more efficient in handling Arrays and Structures.
2. Pointers allow references to function and thereby helps in passing of function as arguments to other functions.
3. It reduces length of the program and its execution time as well.
4. It allows C language to support Dynamic Memory management.

In the next tutorial we will learn syntax of pointers, how to declare and define a pointer, and using a pointer. See you in the next tutorial.

Declaring, Initializing and using a pointer variable in C

In this tutorial, we will learn how to declare, initialize and use a pointer. We will also learn what NULL pointer are and where to use them. Let's start!

Declaration of C Pointer variable

General syntax of pointer declaration is,

```
datatype *pointer_name;
```

Data type of a pointer must be same as the data type of the variable to which the pointer variable is pointing. `void` type pointer works with all data types, but is not often used.

Here are a few examples:

```
int *ip     // pointer to integer variable
float *fp;      // pointer to float variable
double *dp;     // pointer to double variable
char *cp;       // pointer to char variable
```

Initialization of C Pointer variable

Pointer Initialization is the process of assigning address of a variable to a **pointer** variable. Pointer variable can only contain address of a variable of the same data type. In C language **address operator** & is used to determine the address of a variable.
The & (immediately preceding a variable name) returns the address of the variable associated with it.

```
#include<stdio.h>

void main()
{
    int a = 10;
    int *ptr;       //pointer declaration
    ptr = &a;       //pointer initialization
}
```

Pointer variable always point to variables of same datatype. Let's have an example to showcase this:

```
#include<stdio.h>

void main()
{
    float a;
    int *ptr;
    ptr = &a;       // ERROR, type mismatch
}
```

If you are not sure about which variable's address to assign to a pointer variable while declaration, it is recommended to assign a `NULL` value to your pointer variable. A pointer which is assigned a `NULL` value is called a **NULL pointer**.

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

```
int main()
{
    int *ptr = NULL;
    return 0;
}
```

A Simple Example of Pointers in C

This program shows how a pointer is declared and used. There are several other things that we can do with pointers, we have discussed them later in this guide. For now, we just need to know how to link a pointer to the address of a variable.

Important point to note is: The data type of pointer and the variable must match, an int pointer can hold the address of int variable, similarly a pointer declared with float data type can hold the address of a float variable. In the example below, the pointer and the variable both are of int type.

```
#include <stdio.h>
int main()
{
   //Variable declaration
   int num = 10;

   //Pointer declaration
   int *p;

   //Assigning address of num to the pointer p
   p = #

   printf("Address of variable num is: %p", p);
   return 0;
}
```

Output:

```
Address of variable num is: 0x7fff5694dc58
```