Unit-2

Introduction to C: Introduction – Structure of C Program – Writing the first C Program – File used in C Program – Compiling and Executing C Programs – Using Comments –Keywords – Identifiers – Basic Data Types in C – Variables – Constants – I/O Statements in C- Operators in C- Programming Examples.
Decision Control and Looping Statements: Introduction to Decision Control Statements– Conditional Branching Statements – Iterative Statements – Nested Loops – Break and Continue Statement – Goto Statement

2.1 Introduction

- Developed at Bell Laboratories in the early seventies by Dennis Ritchie.
- Born out of two other languages – BCPL (Basic Control Programming Language) and B.
- C introduced such things as character types, floating point arithmetic, structures, unions and the preprocessor.
- The principal objective was to devise a language that was easy enough to understand to be "high-level" – i.e. understood by general programmers, but low-level enough to be applicable to the writing of systems-level software.
- The language should abstract the details of how the computer achieves its tasks in such a way as to ensure that C could be portable across different types of computers, thus allowing the UNIX operating system to be compiled on other computers with a minimum of re-writing.
- C as a language was in use by 1973, although extra functionality, such as new types, was introduced up until 1980.
- In 1978, Brian Kernighan and Dennis M. Ritchie wrote the seminal work *The C Programming Language,* which is now the standard reference book for C.
- A formal ANSI standard for C was produced in 1989.
- In 1986, a descendant of C, called C++ was developed by Bjarne Stroustrup, which is in wide use today. Many modern languages such as C#, Java and Perl are based on C and C++.
- Using C language scientific, business and system-level applications can be developed easily.

The C Character set:

C uses :

Upper case letters- A to Z
Lowercase letters- a to z
Digits: 0 to 9
Special characters: **+, -, *, /, =, %, &, #, !, ?, ^, “, ‘, ~, \, |, <, >, (,), [,], {, }, : (colon), ; (semi colon), .(dot), _, (blank space)**

C uses certain combinations of these characters to represent special conditions such as newline, horizontal tab ..etc.

2.2 Structure of C Program

The structure of a C program looks as follows:

Header Files (.h):

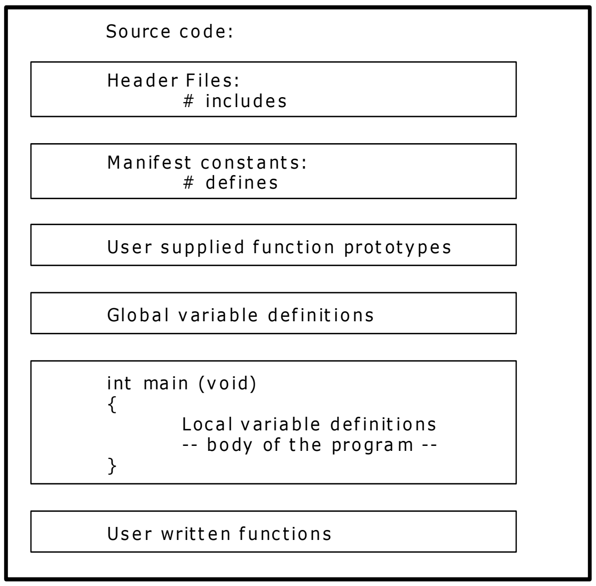
Header files contain declaration information for function or constants that are referred in programs. They are used to keep source-file size to a minimum and to reduce the amount of redundant information that must be coded.

# includes:

An include directive tells the preprocessor to include the contents of the specified file at the point in the program. Path names must either be enclosed by double quotes or angle brackets.

Example:

```
1:      # include <stdio.h>
2:      # include "mylib.h"
3:      # include "mine\include\mylib.h"
```


Source code:
Header Files:
# includes
Manifest constants:
# defines
User supplied function prototypes
Global variable definitions
int main (void)
{
Local variable definitions
-- body of the program --
}
User written functions

In the example (1) above, the <> tells the preprocessor to search for the included file in a special known \include directory or directories.

In the example (2), the double quotes (" ") indicate that the current directory should be checked for the header file first. If it is not found, the special directory (or directories) should be checked.

The example (3) is similar, but the named relative directory \mine\include is checked for the header file mylib.h.

Relative paths can also be proceeded by the .\ or ..\ notation; absolute paths always begin with a \.

# defines:

ANSI C allows you to declare ***constants.*** The # define directive is used to tell the preprocessor to perform a search-and-replace operation.

Example:

```
# define Pi 3.14159
# define Tax-rate 0.0735
```

In the example above, the preprocessor will search through the source file and replace every instance of the token Pi with 3.14159

After performing the search and replace operation, the preprocessor removes the # define line.

User supplied function prototypes:

Declares the user-written functions actually defined later in the source file. A function prototype is a statement (rather than an entire function declaration) that is placed a head of a calling function in the source code to define the function's label before it is used. A function prototype is simply a reproduction of a function header (the first line of a function declaration) that is terminated with a semi-colon.

Global variable definitions:
Create the Global variables before main ().

The main function:

main () The main function is the entry point in the C program. All C programs begin execution by calling the main () function. When the main function returns, your program terminates execution and control passes back to the operating system.
Every C/C++ program must have one and only one main function.

{ The next line consists of a single curly brace which signifies the start of main () function.

Next we need to write the code for the program.

} which signifies the end of main () function.

User-written functions:

Divide the application into logical procedural units and factor out commonly used code to eliminate repetition.

2.3 Writing the first C Program

Every C program consists of one or more modules called functions. One of these functions is called main. The program begins by executing main function and accesses other functions, if any. Functions are written after or before main function separately. A function has (1) heading consists of name with list of arguments (optional) enclosed in parenthesis, (2) argument declaration (if any) and (3) compound statement enclosed in two braces { } such that each statement ends with a semicolon. Comments, which are not executable statement, of necessary can be placed in between /* and */.

Example1:

/*program to display hello message

```
#include<stdio.h>
#include<conio.h>
main( )
{
printf(" hello C");
}
```

Program simply display the message: " hello C".

Example2:

```
/* program to find the area pf a circle */
#include<stdio.h>
#include<conio.h>
main( )
```

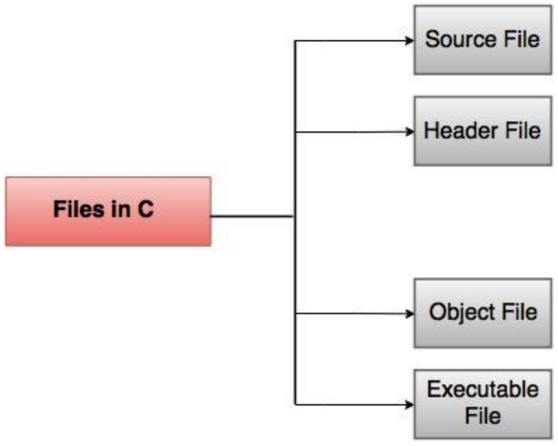
```
{
        float r, a;
        printf("radius");
        scanf("%f", &r);
        a=3.145*r*r;
        printf("area of circle=%f", area);
}
```

```
Ouput of the program:
radius:4
area of the circle:50.32
```

this program is used to display the area of the triangle

2.4 File used in C Program

A C program uses four types of files as follows:



Source Code File

- This file includes the source code of the program.
- The extension for these kind of files are '.c'. It defines the main and many more functions written in C.
- main() is the starting point of the program. It may also contain other source code files.

Header Files

- They have an extension '.h'. They contain the C function declarations and macro definitions that are shared between various source files.

Object files

- They are the files that are generated by the compiler as the source code file is processed.
- These files generally contain the binary code of the function definitions.
- The object file is used by the linker for producing an executable file for combining the object files together. It has a '.o' extension.

Executable file

- This file is generated by the linker.
- Various object files are linked by the linker for producing a binary file which will be executed directly.
- They have an '.exe' extension.

2.5Compiling and Executing C Programs

Step 1: Type the program in C editor and save with .c extension (Press F2 to save).

Step 2: The **compilation** is the process of converting high-level language instructions into low-level language instructions. We use the shortcut key **Alt + F9(or Alt+C)** to compile a C program in **Turbo C**. Whenever we press **Alt + F9**, the source file is going to be submitted to the Compiler. On receiving a source file, the compiler first checks for the Errors. If there are any Errors then compiler returns List of Errors, if there are no errors then the source code is converted into **object code** and stores it as a file with **.obj** extension. Then the object code is given to the **Linker**. The Linker combines both the **object code** and specified **header file** code and generates an **Executable file** with a **.exe** extension.

Step 3: Executing / Running Executable File (Ctrl + F9) or(Alt+R)

After completing compilation successfully, an executable file is created with a **.exe** extension. The processor can understand this **.exe** file content so that it can perform the task specified in the source file.We use a shortcut key **Ctrl + F9** to run a C program. Whenever we press **Ctrl + F9**, the **.exe** file is submitted to the **CPU**. On receiving **.exe** file, **CPU** performs the task according to the instruction written in the file. The result generated from the execution is placed in a window called **User Screen**.

2.6 Using Comments

In C, comments begin with the sequence /* and are terminated by */. Any thing that is between the beginning and ending comments symbols is ignored by the compiler.

Example:

```
/* sample program */
```

2.7 Keywords

Here are the 32 keywords:
Flow control (6) – if, else, return, switch, case, default
Loops (5) – for, do, while, break, continue
Common *types* (5) – int, float, double, char, void
For dealing with *structures* (3) – struct, typedef, union
Counting and sizing things (2) – enum, sizeof
Rare but still useful *types* (7) – extern, signed, unsigned, long, short, static, const
Keywords that is undiscouraged and which we NEVER use (1) – goto
We don't use unless we're doing something strange (3) – auto, register, volatile

2.8 Identifiers

The C language defines the names that are used to reference variables, functions, labels, and various other user-defined objects as identifiers.

An identifier can vary from one to several characters. The first character must be a letter or an underscore with subsequent characters being letters, numbers or under-score.

Example:

Correct: Count, test23, High_balances, _name

In-correct: 1 count, hil there, high..balance

In turbo C, the first 32 characters of an identifier name are significant. In C, upper and lower case characters are treated as different and distinct.

Example:

count, Count, COUNT are three separate identifiers.

Rules for naming identifiers are:

1) Name should only consist of alphabets (both upper and lower case), digits and underscore (_) sign.
2) First characters should be alphabet or underscore
3) Name should not be a keyword
4) Since C is a case sensitive, the upper case and lower case considered differently, for example code, Code, CODE etc. are different identifiers.
5) Identifiers are generally given in some meaningful name such as value, net_salary, age, data etc. An identifier name may be long, some implementation recognizes only first eight characters, most recognize 31 characters. ANSI standard compiler recognizes 31 characters. Some *invalid identifiers* are 5cb, int, res#, avg no etc.

2.9 Basic Data Types in C

Data types are used for declaring variables or functions of different types before its use. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted. The value of a variable can be changed any time.
C has the following 4 types of data types

basic built-in data types: int, float, double, char
Enumeration data type: enum
Derived data type: pointer, array, structure, union
Void data type: void

A variable declared to be of type **int** can be used to contain integral values only—that is, values that do not contain decimal places. A variable declared to be of type **float** can be used for storing floating- point numbers (values containing decimal places). The **double** type is the same as type float, only with roughly twice the precision. The **char** data type can be used to store a single character, such as the letter *a*, the digit character *6*, or a semicolon similarly A variable declared char can only store character type value.
There are two types of type qualifier in c

Size qualifier: short, long
Sign qualifier: signed, unsigned

When the qualifier unsigned is used the number is always positive, and when signed is used number may be positive or negative. If the sign qualifier is not mentioned, then by default sign qualifier is assumed. The range of values for signed data types is less than that of unsigned data type. Because in signed type, the left most bit is used to represent sign, while in unsigned type this bit is also used to represent the value. The size and range of the different data types on a 16 bit machine is given below:

Type	**Size**	**Range**	**Precision for real numbers**
char	1 byte	-128 to 127	
unsigned char	1 byte	0 to 255	
signed char	1 byte	-128 to 127	
short int or short	2 bytes	-32,768 to 32,767	

unsigned short or unsigned short int	2 bytes	0 to 65535	
int	2 bytes	-32,768 to 32,767	
unsigned int	2 bytes	0 to 65535	
Long or long int	4 bytes	-2147483648 to 2147483647 (2.1 billion)	
unsigned long or unsigned long int	4 bytes	0 to 4294967295	
float	4 bytes	3.4 E–38 to 3.4 E+38	6 digits of precision
double	8 bytes	1.7 E-308 to 1.7 E+308	15 digits of precision
long double	10 bytes	+3.4 E-4932 to 1.1 E+4932	provides between 16 and 30 decimal places

The following table lists the most common conversion specifiers and the types of data that they convert:

Specifier	**Data Type**
%c	char
%f	float
%d or %i	signed int (decimal)
%h	short int
%p	Pointer (Address)
%s	String of Characters
Qualified Data Types	
%lf	long float or double
%o	unsigned int (octal)
%u	unsigned int (decimal)
%x	unsigned int (hexadecimal)
%X	Unsigned Hexadecimal (Upper Case)
%e	Scientific Notation (Lower case e)
%E	Scientific Notation (Upper case E)
%g	Uses %e or %f which ever is shorter
%ho	short unsigned int (octal)
%hu	short unsigned int (decimal)

%hx	short unsigned int (hexadecimal)
%lo	long unsigned int (octal)
%lu	long unsigned int (decimal)
%lx	long unsigned int (hexadecimal)
%Lf	long double
%n	The associated argument is an integer pointer into which the number of characters written so far is placed.
%%	Prints a % sign

2.10 Variables

Variables are declared in three basic places: inside functions, in the definition of function parameters and outside of the functions. If they are declared inside functions, they are called as *local variables*, if in the definition of functions then formal parameters and out side of all functions, then global variables.

2.10.1. Global variables:

Global variables are known throughout the program. The variables hold their values throughout the programs execution. Global variables are created by declaring them outside of any function.
Global variables are defined above main () in the following way:

```
short number, sum;
int bignumber, bigsum;
char letter;
main()
{

}
```

It is also possible to pre-initialise global variables using the = operator for assignment.

2.10.2. Local variables:

Variable that are declared inside a function are called "local variables". These variables are referred to as "automatic variables". Local variables may be referenced only by statements that are inside the block in which the variables are declared.

Local variables exist only while the block of code in which they are declared is executing, i.e. a local variable is created upon entry into its block & destroyed upon exit.

Example:
Consider the following two functions:

```
void func1 (void)
{
        int x;
        x = 10;
}
```

```
        void func2 (void)
        {
                int x;
                x = -199;
        }
```
The integer variable x is declared twice, once in func1 () & once in func2 (). The x in func1 () has no bearing on or relationship to the x in func2 (). This is because each x is only known to the code within the same block as variable declaration.

2.11 Constants

1.9. Manifest Constants:

The # define directive is used to tell the preprocessor to perform a search-and-replace operation.

```
Example:    # define Pi 3.14159
            # define Tax-rate 0.0735
```
In the example above, the preprocessor will search through the source file and replace every instance of the token Pi with 3.14159

After performing the search and replace operation, the preprocessor removes the # define line.

The following are two purposes for defining and using manifest constants:

(1) They improve source-code readability
(2) They facilitate program maintenance.

1.10. Constants:

Constant is a any value that cannot be changed during program execution. In C, any number, single character, or character string is known as a *constant*. A constant is an entity that doesn’t change whereas a variable is an entity that may change. For example, the number 50 represents a constant integer value. The character string "Programming in C is fun.\n" is an example of a constant character string. There are 4 basic types of constants . they are integer constants, floating-point constants, character constants and string constants. Various types of constants are:

(a) **integer constants**: It is an integer valued numbers, written in three different number system, decimal (base 10) , octal(base8), and hexadecimal(base 16). A decimal integer constant consists of 0,1,…..,9..

Example: 75 6,0,32, etc…..

An octal integer constant consists of digits 0,1,…,7. with 1st digit 0 to indicate that it is an octal integer.

Example : 0, 01, 0756, 032, etc…..

A hexadecimal integer constant consists of 0,1, …,9,A, B, C, D, E, F. It begins with 0x.

Example: 0x7AA2, 0xAB, etc……

Usually negative integer constant begin with (-) sign. An unsigned integer constant is identified by appending U to the end of the constant like 673U, 098U, 0xACLFU etc. Note that 1234560789LU is an unsigned integer constant.

(b) floating point constants : It is a decimal number (ie: base 10) with a decimal point or an exponent or both. Ex; 32.65, 0.654, 0.2E-3, 2.65E10 etc. These numbers have greater range than integer constants.

(c) character constants : It is a single character enclosed in single quotes like ‘a’. ‘3’, ‘?’, ‘A’ etc. each character has an ASCII to identify. For example ‘A’ has the ASCII code 65, ‘3’ has the code 51 and so on.

(d) escape sequences: An escape sequence is used to express non printing character like a new line, tab etc. it begin with the backslash (\ ) followed by letter like a, n, b, t, v, r, etc. the commonly used escape sequence are
\a : for alert \n : new line \0 : null
\b : backspace \f : form feed \? : question mark
\f : horizontal tab \r : carriage return \' : single quote
\v : vertical tab \" : quotation mark
(e) string constants : it consists of any number of consecutive characters enclosed in double quotes .Ex : " C program" , "mathematics" etc……

2.12 I/O Statements in C

Data Input and Output:
For inputting and outputting data we use library functions. The important of these functions are getch( ), putchar( ), scanf( ), printf( ), gets( ), puts( ),. For using these functions in a C-program there should be preprocessor statement #include<stdio.h>.

Single Character Input and Output
Single character input output functions are getch(), getche(), getc(), getchar(), putch(), putc(), putchar(), fgetc(), fputc(). These functions are also known as unformatted Input and Output functions.

getch():
getch() is used to get a character from console(keyboard) but does not echo to the screen.
Syntax:
```
        int getch(void);
```
Example:
```
void main()
{
    char ch;
    ch = getch();
    printf("Input Char Is :%c",ch);
}
```
During the program execution, a single character is get or read through *the* **getch()**. The given value is not displayed on the screen and the compiler does not wait for another character to be typed. And then, the given character is printed through the **printf** function.

getche():
This function is used to get a character from the console, and echos to the screen.
Syntax :
```
        int getche(void)
```
Example:
```
Void main()
{
char ch;
Ch=getche();
Printf("Input character is::%c",ch);
}
```
During the program execution, a single character is get or read through the **getche()**. The given value is displayed on the screen and the compiler does not wait for another character to be typed. Then,after wards the character is printed through the **printf** function.

getc():

Characters are read using getc() or its equivalent fgetc(). The prototype for getc() is:

```
        int getc(FILE *fp);
```

getc() returns an integer. getc() returns an EOF when the end of file has been reached. The following code reads a text file to the end:

```
do
        {
                ch = getc (fp);
        } while(ch != EOF);
```

getchar():
It is used to read a single character (char type) from keyboard. The syntax is

```
        char variable name = getchar( );
```

Example:

```
char c;
c = getchar( );
```

For reading an array of characters or a string we can use getchar( ) function.
Example:

```
#include<stdio.h>
main( )
{
        char place[80];
        int i;
        for(i = 0;( place [i] = getchar( ))! = '\n', ++i);
}
```

This program reads a line of text.

putch():
putch displays any alphanumeric characters to the standard output device. It displays only one character at a time.
Declaration:

```
        Int putch (variable_name);
```

Example Declaration:

```
                          char ch = 'a';
                        putch(ch)
```

Remarks:
putch print only one character at a time screen.
Example Program:

```
void main()
{
        char ch='a';
        putch(ch)
}
```

putc():
The prototype for putc() is:

```
        int putc (int ch, FILE *fp);
```

where ch is the character to be output. For historical reasons, ch is defined as an int, but only the low order byte is used.
If the putc() operation is successful, it returns the character written, otherwise it returns EOF. Characters are written using putc() or its equivalent fputc().
putchar():
It is used to display single character. The syntax is

```
        putchar(char c);
```

Example:
```
char c;
c = ‘a’;
putchar(c);
```
Using these two functions, we can write a very basic program to copy the input, a character at a time, to the output:
```
#include <stdio.h>
/* copy input to output */
main()
{
        int c;
        c = getchar();
        while(c != EOF)
        {
                putchar(c);
                c = getchar();
        }
return 0;
}
```

fgetc():
The C library function **int fgetc(FILE *stream)** gets the next character (an unsigned char) from the specified stream and advances the position indicator for the stream.
Following is the declaration for fgetc() function.
```
        int fgetc(FILE *stream)
```
Parameters
- **stream** -- This is the pointer to a FILE object that identifies the stream on which the operation is to be performed.

This function returns the character read as an unsigned char cast to an int or EOF on end of file or error.
Example
The following example shows the usage of fgetc() function.
```
#include <stdio.h>

int main ()
{
  FILE *fp;
  int c;
  int n = 0;
    fp = fopen("file.txt","r");
  if(fp == NULL)
  {
    perror("Error in opening file");
    return(-1);
  }
  do
  {
    c = fgetc(fp);
    if( feof(fp) )
    {
      break ;
    }
    printf("%c", c);
```

```
   }while(1);
   fclose(fp);
   return(0);
}
```

Let us assume, we have a text file **file.txt**, which has the following content. This file will be used as an input for our example program:

```
        We are in 2012
```

Now, let us compile and run the above program that will produce the following result:

```
        We are in 2012
```

fputc():
The value returned by the fputc() is the value of the character written . if an error occurs, EOF is returned.
For example, the following program writes to the specified stream.

```
#include <stdio.h>
#include<stdlib.h>
int main()
      {
      FILE *fptr;
      char text[100];
      int i=0;
      clrscr();
      printf("Enter a text:\n");
      gets(text);
      if((fptr = fopen("TEST","w"))==NULL)
                {
                printf("Cannot open file\n");
                exit(1);
                }
       while(text[i]!='\0')
                  fputc(text[i++],fptr);
       if(fclose(fptr))
                  pritf("File close error\n");
       getch();
       return 0;
       }
```

Input Data & Output data
There are some functions which can read and write more than a single character. They are known as formated input and output functions. for example, scanf(), printf(), sscanf(), sprintf(), fscanf(),fprintf().

Printf():
The C library function **printf()** sends formatted output to stdout.
Following is the declaration for printf() function.

int printf(**"Control string", argument list**)

Parameters

- control string − This is the string that contains the text to be written to stdout. It can optionally contain embedded format tags that are replaced by the values specified in subsequent additional arguments and formatted as requested. Control string prototype is **%[flags][width][.precision][length]specifier**, which is explained below −

specifier	Output
c	Character
d or i	Signed decimal integer
e	Scientific notation (mantissa/exponent) using e character
E	Scientific notation (mantissa/exponent) using E character
f	Decimal floating point
g	Uses the shorter of %e or %f
G	Uses the shorter of %E or %f
o	Signed octal
s	String of characters
u	Unsigned decimal integer
x	Unsigned hexadecimal integer
X	Unsigned hexadecimal integer (capital letters)
p	Pointer address
n	Nothing printed
%	Character

flags	Description
-	Left-justify within the given field width; Right justification is the default (see width sub-specifier).
+	Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a -ve sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with o, x or X specifiers the value is preceded with 0, 0x or 0X respectively for values different than zero. Used with e, E and f, it forces the written output to contain a decimal point even if no digits would follow. By default, if no digits follow, no decimal point is written. Used with g or G the result is the same as with e or E but trailing zeros are not removed.
0	Left-pads the number with zeroes (0) instead of spaces, where padding is specified (see width sub-specifier).

width	Description
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The width is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

.precision	Description
.number	For integer specifiers (d, i, o, u, x, X) − precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of 0 means that no character is written for the value 0. For e, E and f specifiers − this is the number of digits to be printed after the decimal point. For g and G specifiers − This is the maximum number of significant digits to be printed. For s − this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. For c type − it has no effect. When no precision is specified, the default is 1. If the period is specified without an explicit value for precision, 0 is assumed.
.*	The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

length	Description
h	The argument is interpreted as a short int or unsigned short int (only applies to integer specifiers: i, d, o, u, x and X).
l	The argument is interpreted as a long int or unsigned long int for integer specifiers (i, d, o, u, x and X), and as a wide character or wide character string for specifiers c and s.
L	The argument is interpreted as a long double (only applies to floating point specifiers: e, E, f, g and G).

- additional arguments − Depending on the format string, the function may expect a sequence of additional arguments, each containing one value to be inserted instead of each %-tag specified in the format parameter (if any). There should be the same number of these arguments as the number of %-tags that expect a value.

Return Value

If successful, the total number of characters written is returned. On failure, a negative number is returned.

Example
The following example shows the usage of printf() function.

```
#include <stdio.h>
int main ()
{
  int ch;
  for( ch = 75 ; ch <= 100; ch++ )
  {
    printf("ASCII value = %d, Character = %c\n", ch , ch );
  }

  return(0);
}
```

Let us compile and run the above program to produce the following result −

```
ASCII value = 75, Character = K
ASCII value = 76, Character = L
ASCII value = 77, Character = M
ASCII value = 78, Character = N
ASCII value = 79, Character = O
ASCII value = 80, Character = P
ASCII value = 81, Character = Q
ASCII value = 82, Character = R
ASCII value = 83, Character = S
ASCII value = 84, Character = T
ASCII value = 85, Character = U
ASCII value = 86, Character = V
ASCII value = 87, Character = W
ASCII value = 88, Character = X
ASCII value = 89, Character = Y
ASCII value = 90, Character = Z
ASCII value = 91, Character = [
ASCII value = 92, Character = \
ASCII value = 93, Character = ]
ASCII value = 94, Character = ^
ASCII value = 95, Character = _
ASCII value = 96, Character = `
ASCII value = 97, Character = a
ASCII value = 98, Character = b
ASCII value = 99, Character = c
ASCII value = 100, Character = d
```

sprintf()
The C library function **sprintf()** sends formatted output to a string pointed to, by **str**.
Declaration
Following is the declaration for sprintf() function.
int sprintf(char *str, "Control string","argumentlist")
Parameters

- **str** − This is the pointer to an array of char elements where the resulting C string is stored.
- **Control string** − This is the String that contains the text to be written to buffer. It can optionally contain embedded format tags that are replaced by the values specified in subsequent additional arguments and formatted as requested. Control

string prototype: **%[flags][width][.precision][length]specifier**, as explained above.

If successful, the total number of characters written is returned excluding the null-character appended at the end of the string, otherwise a negative number is returned in case of failure.
Example
The following example shows the usage of sprintf() function.
```
#include <stdio.h>
#include <math.h>
int main()
{
  char str[80];
  sprintf(str, "Value of Pi = %f", M_PI);
  puts(str);
    return(0);
}
```
Let us compile and run the above program, this will produce the following result −
```
Value of Pi = 3.141593
```

fprintf()
The C library function fprintf()sends formatted output to a stream.
Declaration
Following is the declaration for fprintf() function.
int fprintf(FILE *stream, "Control string","argumentlist") Parameters

- stream − This is the pointer to a FILE object that identifies the stream.
- format − This is the C string that contains the text to be written to the stream. It can optionally contain embedded format tags that are replaced by the values specified in subsequent additional arguments and formatted as requested. Format tags prototype is **%[flags][width][.precision][length]specifier**, which is explained above.

If successful, the total number of characters written is returned otherwise, a negative number is returned.
Example
The following example shows the usage of fprintf() function.
```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  FILE * fp;
  fp = fopen ("file.txt", "w+");
  fprintf(fp, "%s %s %s %d", "We", "are", "in", 2012);
    fclose(fp);
   return(0);
}
```
Let us compile and run the above program that will create a file file.txt with the following content −
```
We are in 2012
```
Now let's see the content of the above file using the following program −
```
#include <stdio.h>
int main ()
{
  FILE *fp;
  int c;
  fp = fopen("file.txt","r");
  while(1)
  {
    c = fgetc(fp);
    if( feof(fp) )
    {
      break;
    }
    printf("%c", c);
  }
  fclose(fp);
  return(0);
}
```
Let us compile and run above program to produce the following result.

```
We are in 2012
```

scanf()

Description

The C library function **int scanf()** reads formatted input from stdin.

Declaration

Following is the declaration for scanf() function.

int **scanf**(**"Control string", argument list**)

Parameters

- **Control string** -- This is the C string that contains one or more of the following items:*Whitespace character, Non-whitespace character and Format specifiers*. A format specifier will be like **[=%[*][width][modifiers]type=]** as explained above.

If successful, the total number of characters written is returned, otherwise a negative number is returned.

Example

The following example shows the usage of scanf() function.

```
#include <stdio.h>
int main()
{
  char str1[20], str2[30];
  printf("Enter name: ");
  scanf("%s", str1);
  printf("Enter your website name: ");
  scanf("%s", str2);
  printf("Entered Name: %s\n", str1);
  printf("Entered Website:%s", str2);
  return(0);
}
```
Let us compile and run the above program that will produce the following result in interactive mode:

```
Enter name: admin
Enter your website name: www.vidyanikethan.com
Entered Name: admin
Entered Website: www.vidyanikethan.com
```

sscanf()

The C library function **int sscanf()**reads formatted input from a string.

Declaration

Following is the declaration for sscanf() function.
int sscanf(const char *str, "Control string",arguments);
Parameters

- **str** -- This is the C string that the function processes as its source to retrieve the data.
- **Control string** -- This is the C string that contains one or more of the following items: *Whitespace character, Non-whitespace character and Format specifiers*

A format specifier follows this prototype: **[=%[*][width][modifiers]type=]** as discussed above in scanf() function.
On success, the function returns the number of variables filled. In the case of an input failure before any data could be successfully read, EOF is returned.
Example
The following example shows the usage of sscanf() function.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
  int day, year;
  char weekday[20], month[20], dtm[100];
  strcpy( dtm, "Saturday March 25 1989" );
  sscanf( dtm, "%s %s %d  %d", weekday, month, &day, &year );
  printf("%s %d, %d = %s\n", month, day, year, weekday );
     return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
March 25, 1989 = Saturday
```

fscanf()
The C library function **int fscanf(FILE *stream)**reads formatted input from a stream.
Declaration
Following is the declaration for fscanf() function.
int fscanf(FILE *stream, "Control string",arguments)
Parameters

- **stream** − This is the pointer to a FILE object that identifies the stream.
- **Control Sring** − This is the C string that contains one or more of the following items − *Whitespace character, Non-whitespace character* and *Format specifiers*. A format specifier will be as **[=%[*][width][modifiers]type=]**, which is explained above in scanf() function.

  This function returns the number of input items successfully matched and assigned, which can be fewer than provided for, or even zero in the event of an early matching failure.

Example
The following example shows the usage of fscanf() function.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  char str1[10], str2[10], str3[10];
  int year;
  FILE * fp;
  fp = fopen ("file.txt", "w+");
  fputs("We are in 2012", fp);
  rewind(fp);
  fscanf(fp, "%s %s %s %d", str1, str2, str3, &year);
  printf("Read String1 |%s|\n", str1 );
  printf("Read String2 |%s|\n", str2 );
  printf("Read String3 |%s|\n", str3 );
  printf("Read Integer |%d|\n", year );
  fclose(fp);
  return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Read String1 |We|
Read String2 |are|
Read String3 |in|
Read Integer |2012|
```

The gets and puts Function

gets():

The easiest way to input a string from the keyboard is with the gets() library function. The general form gets() is:

```
        gets(array_name);
```

To read a string, call gets() with the name of the array, with out any index, as its arguments. Upon return form gets() the array will hold the string input. The gets() function will continue to read characters until you enter a carriage return. The header file used for gets() is stdio.h

Example:

```
# include <stdio.h>
main()
{
        char str[80];
        printf ("\nEnter a string:");
        gets (str);
        printf ("%s", str);
}
```

the carriage return does not become part of the string instead a null terminator is placed at the end.

puts():

The puts() functions writes its string argument to the screen followed by a newline. Its prototype is:

```
        puts(string);
```

It recognizes the same back slash code as printf(), such as "\t" for tab. As puts() can output a string of characters – It cannot output numbers or do format conversions it required faster overhead than printf(). Hence, puts() function is often used when it is important to have highly optimized code.

For example, to display "hello" on the screen:

```
        puts("hello");
```

fgets():

The C library function char *fgets(char *str, int n, FILE *stream) reads a line from the specified stream and stores it into the string pointed to by str. It stops when either (n-1) characters are read, the newline character is read, or the end-of-file is reached, whichever comes first.
Following is the declaration for fgets() function.

```
char *fgets(char *str, int n, FILE *stream)
```

Parameters

- str -- This is the pointer to an array of chars where the string read is stored.
- n -- This is the maximum number of characters to be read (including the final null-character). Usually, the length of the array passed as str is used.
- stream -- This is the pointer to a FILE object that identifies the stream where characters are read from.

On success, the function returns the same str parameter. If the End-of-File is encountered and no characters have been read, the contents of str remain unchanged and a null pointer is returned. If an error occurs, a null pointer is returned.
Example
The following example shows the usage of fgets() function.

```
#include <stdio.h>
int main()
{
  FILE *fp;
  char str[60];
    /* opening file for reading */
  fp = fopen("file.txt" , "r");
  if(fp == NULL)
  {
    perror("Error opening file");
    return(-1);
  }
  if( fgets (str, 60, fp)!=NULL )
  {
   /* writing content to stdout */
    puts(str);
  }
  fclose(fp);
  return(0);
}
```

Let us assume, we have a text file file.txt, which has the following content. This file will be used as an input for our example program:

```
We are in 2012
```

Now, let us compile and run the above program that will produce the following result:

```
We are in 2012
```

fputs():

The C library function int fputs(const char *str, FILE *stream) writes a string to the specified stream up to but not including the null character.
Following is the declaration for fputs() function.

```
int fputs(const char *str, FILE *stream)
```

Parameters

- str -- This is an array containing the null-terminated sequence of characters to be written.

- stream -- This is the pointer to a FILE object that identifies the stream where the string is to be written.

This function returns a non-negative value, or else on error it returns EOF.

Example

The following example shows the usage of fputs() function.

```
#include <stdio.h>
int main ()
{
  FILE *fp;
  fp = fopen("file.txt", "w+");
  fputs("This is c programming.", fp);
  fputs("This is a system programming language.", fp);
  fclose(fp);
    return(0);
}
```

Let us compile and run the above program, this will create a file file.txt with the following content:

```
This is c programming.This is a system programming language.
```

Now let's see the content of the above file using the following program:

```
#include <stdio.h>

int main ()
{
  FILE *fp;
  int c;
  fp = fopen("file.txt","r");
  while(1)
  {
    c = fgetc(fp);
    if( feof(fp) )
    {
      break ;
    }
    printf("%c", c);
  }
  fclose(fp);
  return(0);
}
```

Let us compile and run the above program to produce the following result.

```
This is c programming. This is a system programming language.
```

2.13 Operators in C

There are three general classes of operators: arithmetic, relational and logical and bitwise.

1.10.1. Arithmetic Operators:

Operation	Symbol	Meaning
Add	+	addition
Subtract	-	subtraction
Multiply	*	Multiplication

Division	/	Remainder lost
Modulus	%	Gives the remainder on integer division, so 7 % 3 is 1.
--	Decrement	
++	Increment	

As well as the standard arithmetic operators (+ - * /) found in most languages, C provides some more operators.
Assignment is = ***i.e.*** *i* = 4; ch = \`y';
Increment ++, Decrement -- which are more efficient than their long hand equivalents.
For example, X = X + 1 can be written as ++X or as X++. There is however a difference when they are used in expression.

The ++ and -- operators can be either in post-fixed or pre-fixed. A pre-increment operation such as ++a, increments the value of a by 1, before a is used for computation, while a post increment operation such as a++, uses the current value of a in the calculation and then increments the value of a by 1. Consider the following:

```
X = 10;
Y = ++X;
```

In this case, Y will be set to 11 because X is first incremented and then assigned to Y.
However if the code had been written as

```
X = 10;
Y = X++;
```

Y would have been set to 10 and then X incremented. In both the cases, X is set to 11; the difference is when it happens.

The % (modulus) operator only works with integers.

Division / is for both integer and float division. So be careful.

The answer to: *x* = 3 / 2 is 1 even if *x* is declared a float!!

RULE: If both arguments of / are integer then do integer division. So make sure you do this. The correct (for division) answer to the above is *x* = 3.0 / 2 or *x*= 3 / 2.0 or (better) *x* = 3.0 / 2.0.

1.10.2. Relational Operators:

The relational operators are used to determine the relationship of one quantity to another. They always return 1 or 0 depending upon the outcome of the test. The relational operators are as follows:

Operator	Action
= =	equal to
! =	not equal to
<	Less than
<=	less than or equal to
>	Greater than

>=	Greater than or equal to

To test for equality is ==

If the values of x and y, are 1 and 2 respectively then the various expressions and their results are:

Expression	**Result**	**Value**
X != 2	False	0
X == 2	False	0
X == 1	True	1
Y != 3	True	1

A warning: Beware of using "='' instead of "= ='', such as writing accidentally

```
if (i = j) .....
```

This is a perfectly **LEGAL** C statement (syntactically speaking) which copies the value in "j" into "i", and delivers this value, which will then be interpreted as TRUE if j is non-zero. This is called **assignment by value** -- a key feature of C.

Not equals is: !=

Other operators < (less than), > (grater than), <= (less than or equals), >= (greater than or equals) are as usual.

1.10.3. Logical (Comparison) Operators:

Logical operators are usually used with conditional statements. The three basic logical operators are && for logical AND, || for logical OR and ! for not.

The truth table for the logical operators is shown here using one's and zero's. (the idea of true and false under lies the concepts of relational and logical operators). In C true is any value other than zero, false is zero. Expressions that use relational or logical operators return zero for false and one for true.

P	Q	P && q	P || q	! p
0	0	0	0	1
0	1	0	1	1
1	1	1	1	0
1	0	0	1	0

Example:
(i) x == 6 && y == 7

This while expression will be TRUE (1) if both x equals 6 and y equals 7, and FALSE (0) otherwise.

(ii) x < 5 || x > 8

This whole expression will be TRUE (1) if either x is less than 5 or x is greater than 8 and FALSE (0) other wise.

1.10.4. Bit wise Operators:

The ***bit wise*** operators of C are summarised in the following table:

Bitwise operators	
&	AND
|	OR
^	XOR
~	One's Compliment
<<	Left shift
>>	Right Shift

The truth table for Bitwise operators AND, OR, and XOR is shown below. The table uses 1 for true and 0 for false.

P	Q	P AND q	P OR q	P XOR q
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

DO NOT confuse & with &&: & is bit wise AND, && logical AND. Similarly for | and ||.
~ is a unary operator: it only operates on one argument to right of the operator. It finds 1’s compliment (unary). It translates all the 1 bits into O’s and all O’s into 1’s

Example:

12 = 00001100
~12 = 11110011 = 243

The shift operators perform appropriate shift by operator on the right to the operator on the left. The right operator must be positive. The vacated bits are filled with zero (***i.e.*** when shift operation takes places any bits shifted off are lost).

Example:

X << 2 shifts the bits in X by 2 places to the left.

So:

if X = 00000010 (binary) or 2 (decimal)

then:
X >>= 2 implies X = 00000000 or 0 (decimal)
Also: if X = 00000010 (binary) or 2 (decimal)
X <<= 2 implies X = 00001000 or 8 (decimal)
Therefore a shift left is equivalent to a multiplication by 2.
Similarly, a shift right is equal to division by 2.

NOTE: Shifting is much faster than actual multiplication (*) or division (/) by 2. So if you want fast multiplications or division by 2 ***use shifts***.
The bit wise AND operator (&) returns 1 if both the operands are one, otherwise it returns zero. For example, if y = 29 and z = 83, x = y & z the result is:

```
0  0  0  1  1  1  0  1   29 in binary
                                    &
0  1  0  1  0  0  1  1   83 in binary

0  0  0  1  0  0  0  1   Result
```

The bit wise or operator (|) returns 1 if one or more bits have a value of 1, otherwise it returns zero. For example if, y = 29 and z = 83, x = y | z the result is:

```
0  0  0  1  1  1  0  1   29 in binary
                                    |
0  1  0  1  0  0  1  1   83 in binary

0  1  0  1  1  1  1  1   Result
```

The bit wise XOR operator (^) returns 1 if one of the operand is 1 and the other is zero, otherwise if returns zero. For example, if y = 29 and z = 83, x = y ^ z the result is:

```
0  0  0  1  1  1  0  1   29 in binary
                                    ^
0  1  0  1  0  0  1  1   83 in binary

0  1  0  0  1  1  1  0   Result
```

1.10.5. Conditional Operator:

Conditional expression use the operator symbols question mark (?)
(x > 7) ? 2 : 3
What this says is that if x is greater than 7 then the expression value is 2. Otherwise the expression value is 3.
In general, the format of a conditional expression is:
    a ? b : c
Where, a, b & c can be any C expressions.
Evaluation of this expression begins with the evaluation of the sub-expression 'a'. If the value of 'a' is true then the while condition expression evaluates to the value of the sub-expression 'b'. If the value of 'a' is FALSE then the conditional expression returns the value of the sub-expression 'C'.

1.11. sizeof Operator:

In situation where you need to incorporate the size of some object into an expression and also for the code to be portable across different machines the size of unary operator will be useful. The size of operator computes the size of any object at compile time. This can be used for dynamic memory allocation.
    Usage: sizeof (object)
The object itself can be the name of any sort of variable or the name of a basic type (like int, float, char etc).

Example:
        sizeof (char) = 1

```
sizeof (int) = 2
sizeof (float) = 4
sizeof (double) = 8
```

1.12. Special Operators:

Some of the special operators used in C are listed below. These are reffered as separators or punctuators.

Ampersand (&)	Comma (,)	Asterick (*)
Ellipsis (…)	Braces ({ })	Hash (#)
Brackets ([ ])	Parenthesis (())	Colon (:)
Semicolon (;)		

Ampersand:
Ampersand (&) also referred as address of operator usually precedes the identifier name, which indicates the memory allocation (address) of the identifier.

Comma:
Comma (,) operator is used to link the related expressions together. Comma used expressions are linked from left to right and the value of the right most expression is the value of the combined expression. The comma operator has the lowest precedence of all operators. For example:

```
Sum = (x = 12, y = 8, x + y);
```

The result will be sum = 20.

The comma operator is also used to separate variables during declaration. For example:

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

Asterick:
Asterick (*) also referred as an indirection operator usually precedes the identifier name, which identifies the creation of the pointer operator. It is also used as an unary operator.

Ellipsis:
Ellipsis (…) are three successive periods with no white space in between them. It is used in function prototypes to indicate that this function can have any number of arguments with varying types. For example:

```
void fun (char c, int n, float f, . . . . )
```

The above declaration indicates that fun () is a function that takes at least three arguments, a char, an int and a float in the order specified, but can have any number of additional arguments of any type.

Hash:
Hash (#) also referred as pound sign is used to indicate preprocessor directives, which is discussed in detail already.

Parenthesis:

Parenthesis () also referred as function call operator is used to indicate the opening and closing of function prototypes, function calls, function parameters, etc., Parenthesis are also used to group expressions, and there by changing the order of evaluation of expressions.

Semicolon:
Semicolon (;) is a statement terminator. It is used to end a C statement. All valid C statements must end with a semicolon, which the C compiler interprets as the end of the statement. For example:

```
c = a + b;
b = 5;
```

1.13. Order of Precedence of C Operators:

It is necessary to be careful of the meaning of such expressions as `a + b * c`.
We may want the effect as either

```
(a + b) * c
    or
a + (b * c)
```

All operators have a priority, and high priority operators are evaluated before lower priority ones. Operators of the same priority are evaluated from left to right, so that:

```
a - b - c
```

is evaluated as: `(a - b) - c`
as you would expect.
From high priority to low priority the order for all C operators (we have not met all of them yet) is:

```
Highest         ( )   [ ]   -> .
                 !   ~ - (type) * &  sizeof  ++ --
                 * / %
                + -
                << >>
                <  <=  >=  >
                == !=
                &
                ^
                |
                &&
                ||
                ?:
                = +=  -=  *=  /=  etc.
Lowest          ,  (comma)
```

Thus: `a < 10 && 2 * b < c`
is interpreted as: `(a < 10) && ((2 * b) < c)`

Operators	Description	Precedence level	Associativity
()	function call	**1**	left to right
[]	array subscript		
→	arrow operator		
.	dot operator		
+	unary plus	2	right to left
-	unary minus		
++	increment		
- -	decrement		
!	logical not		
~	1's complement		
*	indirection		
&	address		
(data type)	type cast		
sizeof	size in byte		
*	multiplication	3	left to right
/	division		
%	modulus		
+	addition	4	left to right
-	subtraction		
<<	left shift	**5**	left to right
>>	right shift		
<=	less than equal to	6	left to right
>=	greater than equal to		
<	less than		
>	greater than		
==	equal to	7	left to right
!=	not equal to		
&	bitwise AND	8	left to right

^	bitwise XOR	9	left to right
|	bitwise OR	10	left to right
&&	logical AND	11	
||	logical OR	12	
?:	conditional operator	13	
=, *=, /=, %= &=, ^=, <<= >>=	assignment operator	14	right to left
,	comma operator	15	

2.14 Type Conversion and Type Casting.

Mixing *types* can cause problems. For example:

```
int a = 3;
int b = 2;
float c;
c = b * (a / b);
printf ("2 * (3 / 2) = %f\n", c);
```

doesn't behave as you might expect. Because the first (a/b) is performed with integer arithmetic it gets the answer 1 not 1.5. Therefore the program prints:

```
2 * (3/2) = 2.000
```

The best way round this is what is known as a *cast*. We can *cast* a variable of one type to another type like so:

```
int a = 3;
int b = 2;
float c;
c = b * ( (float) a / b);
```

The (float) a construct tells the compiler to switch the type of variable to be a float. The value of the expression is automatically cast to float . The main use of *casting* is when you have written a routine which takes a variable of one type and you want to call it with a variable of another type. For example, say we have written a power function with a prototype like so:

```
int pow (int n, int m);
```

We might well have a float that we want to find an approximate power of a given number ***n***. Your compiler should complain bitterly about your writing:

```
float n= 4.0;
int squared;
squared= pow (n, 2);
```

The compiler will not like this because it expects ***n*** to be of type int but not float. However, in this case, we want to tell the compiler that we do know what we're doing and have good reason for passing it a float when it expects an int (whatever that reason might be). Again, a cast can rescue us:

```
float n = 4.0;
int squared;
squared = pow ((int) n, 2);      /* We cast the float down to an int*/
```

IMPORTANT RULE: To move a variable from one type to another then we use a *cast* which has the form (*variable_type*) *variable_name*.

CAUTION: It can be a problem when we *downcast* – that is cast to a type which has less precision than the type we are casting from. For example, if we cast a double to a float we will lose some bits of precision. If we cast an int to a char it is likely to overflow [recall that a char is basically an int which fits into 8 binary bits].

Decision Control and Looping Statements: Introduction to Decision Control Statements– Conditional Branching Statements – Iterative Statements – Nested Loops – Break and Continue Statement – Goto Statement

2.15 Introduction to Decision Control Statements:

In any programming language, there is a need to perform different tasks based on the condition. For example, consider an online website, when you enter wrong id or password it displays error page and when you enter correct credentials then it displays welcome page. So there must be a logic in place that checks the condition (id and password) and if the condition returns true it performs a task (displaying welcome page) else it performs a different task (displaying error page).

Using decision control statements we can control the flow of program in such a way so that it executes certain statements based on the outcome of a condition (i.e. true or false).

2.16 Decision Control Statements:

In C Programming language we have following decision control statements.

1. **if statement**
2. **if-else & else-if statement**
3. **switch-case statements**

Conditional Statements C

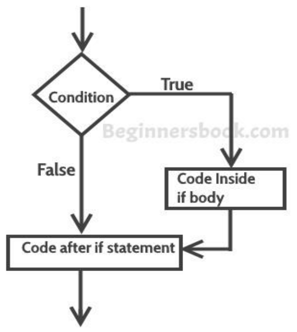
If statement: The statements inside if body executes only when the condition defined by if statement is true. If the condition is false then compiler skips the statement enclosed in if's body. We can have any number of if statements in a C program.

Syntax of if statement:

The statements inside the body of "if" only execute if the given condition returns true. If the condition returns false then the statements inside "if" are skipped.

```
if (condition)
{
    //Block of C statements here
    //These statements will only execute if the condition is true
}
```

Flow Diagram of if statement



Example of if statement

```
#include <stdio.h>
int main()
{
   int x = 20;
   int y = 22;
   if (x<y)
   {
      printf("Variable x is less than y");
   }
   return 0;
}
```
Output:

```
Variable x is less than y
```
Explanation: The condition (x<y) specified in the "if" returns true for the value of x and y, so the statement inside the body of if is executed.

Example of multiple if statements

We can use multiple if statements to check more than one conditions.

```
#include <stdio.h>
int main()
{
   int x, y;
   printf("enter the value of x:");
   scanf("%d", &x);
   printf("enter the value of y:");
   scanf("%d", &y);
   if (x>y)
   {
        printf("x is greater than y\n");
   }
   if (x<y)
   {
        printf("x is less than y\n");
   }
   if (x==y)
   {
        printf("x is equal to y\n");
   }
   printf("End of Program");
   return 0;
}
```
In the above example the output depends on the user input.

Output:

```
enter the value of x:20
enter the value of y:20
x is equal to y
End of Program
```

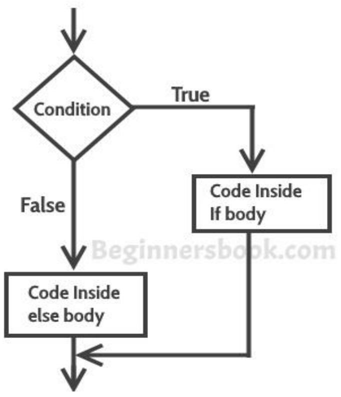
If-else statement: In this decision control statement, we have two block of statements. If condition results true then if block gets executed else statements inside else block executes. Else cannot exist without if statement. In this tutorial, I have covered else-if statements as well.

Syntax of if else statement:

If condition returns true then the statements inside the body of "if" are executed and the statements inside body of "else" are skipped. If condition returns false then the statements inside the body of "if" are skipped and the statements in "else" are executed.

```
if(condition) {
  // Statements inside body of if
}
else {
  //Statements inside body of else
}
```

Flow diagram of if else statement



1.1.1 Example of if else statement

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

```
#include <stdio.h>
int main()
{
  int age;
  printf("Enter your age:");
  scanf("%d",&age);
  if(age >=18)
  {
       /* This statement will only execute if the
        * above condition (age>=18) returns true
        */
       printf("You are eligible for voting");
  }
  else
  {
       /* This statement will only execute if the
        * condition specified in the "if" returns false.
        */
       printf("You are not eligible for voting");
  }
  return 0;
}
```

Output:

```
Enter your age:14
You are not eligible for voting
```

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

```
#include <stdio.h>
int main()
{
  int age;
  printf("Enter your age:");
  scanf("%d",&age);
  if(age >=18)
        printf("You are eligible for voting");
  else
        printf("You are not eligible for voting");
  return 0;
}
```

Nested If..else statement

When an if else statement is present inside the body of another "if" or "else" then this is called nested if else.

Syntax of Nested if else statement:

```
if(condition)
{
   //Nested if else inside the body of "if"
   if(condition2)
{
     //Statements inside the body of nested "if"
   }
   else
{
     //Statements inside the body of nested "else"
   }
}
else {
   //Statements inside the body of "else"
}
```

Example of nested if..else

```
#include <stdio.h>
int main()
{
  int var1, var2;
  printf("Input the value of var1:");
  scanf("%d", &var1);
  printf("Input the value of var2:");
  scanf("%d",&var2);
  if (var1 != var2)
  {
        printf("var1 is not equal to var2\n");
        //Nested if else
        if (var1 > var2)
        {
                printf("var1 is greater than var2\n");
        }
        else
        {
                printf("var2 is greater than var1\n");
        }
  }
  else
  {
        printf("var1 is equal to var2\n");
  }
  return 0;
}
```

Output:

```
Input the value of var1:12
Input the value of var2:21
var1 is not equal to var2
var2 is greater than var1
```

else..if statement

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

Syntax of else..if statement:

```
if (condition1)
{
  //These statements would execute if the condition1 is true
}
else if(condition2)
{
  //These statements would execute if the condition2 is true
}
else if (condition3)
{
  //These statements would execute if the condition3 is true
}
.
.
else
{
  //These statements would execute if all the conditions return false.
}
```

Example of else..if statement

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

```
#include <stdio.h>
int main()
{
  int var1, var2;
  printf("Input the value of var1:");
  scanf("%d", &var1);
  printf("Input the value of var2:");
  scanf("%d",&var2);
  if (var1 !=var2)
  {
       printf("var1 is not equal to var2\n");
  }
  else if (var1 > var2)
  {
       printf("var1 is greater than var2\n");
  }
  else if (var2 > var1)
  {
       printf("var2 is greater than var1\n");
  }
  else
  {
       printf("var1 is equal to var2\n");
  }
  return 0;
}
```

Output:

```
Input the value of var1:12
Input the value of var2:21
var1 is not equal to var2
```

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

Important Points:

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

Switch-case statement: This is very useful when we need to evaluate multiple conditions. The switch block defines an expression (or condition) and case has a block of statements, based on the result of expression, corresponding case block gets executed. A switch can have any number of cases; however there should be only one default handler.

The **switch case statement** is used when we have multiple options and we need to perform a different task for each option.

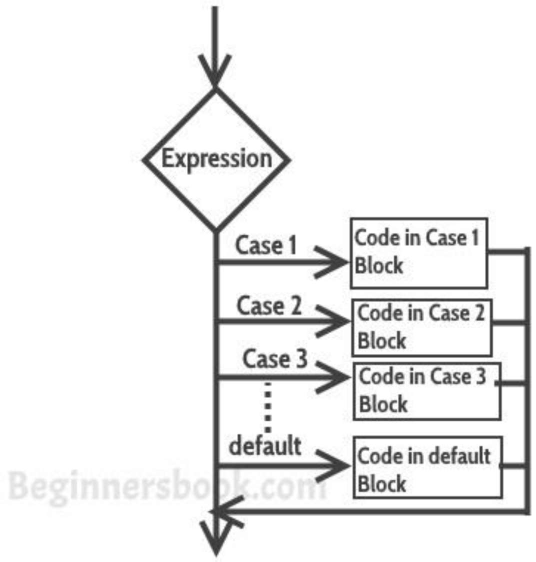
Switch Case Statement

Before we see how a switch case statement works in a C program, let's check out the syntax of it.

```
switch (variable or an integer expression)
{
    case constant:
    //C Statements
    ;
    case constant:
```

```
    //C Statements
    ;
    default:
    //C Statements
    ;
}
```

Flow Diagram of Switch Case



Example of Switch Case in C

Let's take a simple example to understand the working of a switch case statement in C program.

```
#include <stdio.h>
int main()
{
    int num=2;
    switch(num+2)
    {
       case 1:
         printf("Case1: Value is: %d", num);
       case 2:
         printf("Case1: Value is: %d", num);
       case 3:
         printf("Case1: Value is: %d", num);
       default:
         printf("Default: Value is: %d", num);
   }
   return 0;
}
Output:
Default: value is: 2
```

Explanation: In switch I gave an expression, you can give variable also. I gave num+2, where num value is 2 and after addition the expression resulted 4. Since there is no case defined with value 4 the default case is executed.

Twist in a story – Introducing Break statement

Before we discuss more about break statement, guess the output of this C program.

```
#include <stdio.h>
int main()
{
    int i=2;
    switch (i)
    {
      case 1:
        printf("Case1 ");
      case 2:
        printf("Case2 ");
      case 3:
        printf("Case3 ");
      case 4:
        printf("Case4 ");
      default:
        printf("Default ");
    }
   return 0;
}
```

Output:

Case2 Case3 Case4 Default

I passed a variable to switch, the value of the variable is 2 so the control jumped to the case 2, However there are no such statements in the above program which could break the flow after the execution of case 2. That’s the reason after case 2, all the subsequent cases and default statements got executed.

How to avoid this situation?

We can use break statement to break the flow of control after every case block.

Break statement in Switch Case

Break statements are useful when you want your program-flow to come out of the switch body. Whenever a break statement is encountered in the switch body, the control comes out of the switch case statement.

Example of Switch Case with break

I’m taking the same above that we have seen above but this time we are using break.

```
#include <stdio.h>
int main()
{
    int i=2;
    switch (i)
    {
        case 1:
          printf("Case1 ");
          break;
        case 2:
          printf("Case2 ");
          break;
        case 3:
          printf("Case3 ");
          break;
        case 4:
          printf("Case4 ");
          break;
        default:
          printf("Default ");
    }
    return 0;
}
```
Output:

```
Case 2
```
Why didn't I use break statement after default?
The control would itself come out of the switch after default so I didn't use it, however if you want to use the break after default then you can use it, there is no harm in doing that.
Few Important points regarding Switch Case
1) Case doesn't always need to have order 1, 2, 3 and so on. They can have any integer value after case keyword. Also, case doesn't need to be in an ascending order always, you can specify them in any order as per the need of the program.
2) You can also use characters in switch case. for example –
```
#include <stdio.h>
int main()
{
    char ch='b';
    switch (ch)
    {
       case 'd':
          printf("CaseD ");
          break;
       case 'b':
          printf("CaseB");
          break;
       case 'c':
          printf("CaseC");
          break;
       case 'z':
          printf("CaseZ ");
          break;
       default:
          printf("Default ");
   }
   return 0;
}
```
Output:

```
CaseB
```
3) The expression provided in the switch should result in a constant value otherwise it would not be valid.
For example:
Valid expressions for switch –
```
switch(1+2+23)
switch(1*2+3%4)
```
Invalid switch expressions –

```
switch(ab+cd)
switch(a+b+c)
```

4) Nesting of switch statements are allowed, which means you can have switch statements inside another switch. However nested switch statements should be avoided as it makes program more complex and less readable.

5) Duplicate case values are not allowed. For example, the following program is incorrect: This program is **wrong** because we have two case 'A' here which is wrong as we cannot have duplicate case values.

```
#include <stdio.h>
int main()
{
    char ch='B';
    switch (ch)
    {
       case 'A':
          printf("CaseA");
          break;
       case 'A':
          printf("CaseA");
          break;
       case 'B':
          printf("CaseB");
          break;
       case 'C':
          printf("CaseC ");
          break;
       default:
          printf("Default ");
   }
   return 0;
}
```

6) The **default** statement is optional, if you don't have a default in the program, it would run just fine without any issues. However it is a good practice to have a default statement so that the default executes if no case is matched. This is especially useful when we are taking input from user for the case choices, since user can sometime enter wrong value, we can remind the user with a proper error message that we can set in the default statement.

2.17 Iterative Statements

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

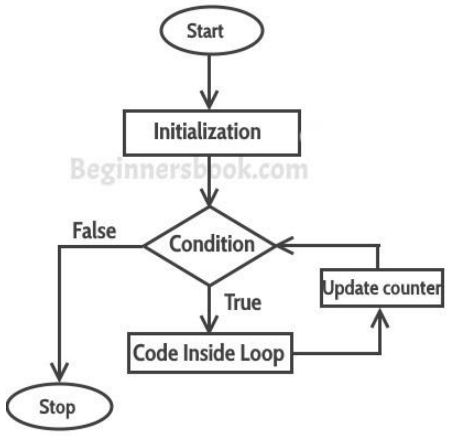
C For loop

This is one of the most frequently used loop in C programming.

Syntax of for loop:

```
for (initialization; condition test; increment or decrement)
{
       //Statements to be executed repeatedly
}
```

Flow Diagram of For loop



Step 1: First initialization happens and the counter variable gets initialized.
Step 2: In the second step the condition is checked, where the counter variable is tested for the given condition, if the condition returns true then the C statements inside the body of for loop gets executed, if the condition returns false then the for loop gets terminated and the control comes out of the loop.

Step 3: After successful execution of statements inside the body of loop, the counter variable is incremented or decremented, depending on the operation (++ or –).

Example of For loop

```
#include <stdio.h>
int main()
{
   int i;
   for (i=1; i<=3; i++)
   {
       printf("%d\n", i);
   }
   return 0;
}
```

Output:

```
1
2
3
```

Various forms of for loop in C

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

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

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

```
int num=10;
for (;num<20;num++)
```

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

```
for (num=10; num<20; )
{
     //Statements
     num++;
}
```

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

```
int num=10;
for (;num<20;)
{
     //Statements
     num++;
}
```

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

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

Nested For Loop in C
Nesting of loop is also possible. Lets take an example to understand this:

```
#include <stdio.h>
int main()
{
  for (int i=0; i<2; i++)
  {
          for (int j=0; j<4; j++)
          {
            printf("%d, %d\n",i ,j);
          }
  }
  return 0;
}
```

Output:

```
0, 0
0, 1
0, 2
0, 3
1, 0
```

```
1, 1
1, 2
1, 3
```

In the above example we have a for loop inside another for loop, this is called nesting of loops. One of the example where we use nested for loop is Two dimensional array.

Multiple initialization inside for Loop in C

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

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

What's the difference between above for loop and a simple for loop?

1. It is initializing two variables. Note: both are separated by comma (,).
2. It has two test conditions joined together using AND (&&) logical operator. Note: You cannot use multiple test conditions separated by comma, you must use logical operator such as && or || to join conditions.
3. It has two variables in increment part. **Note:** Should be separated by comma.

Example of for loop with multiple test conditions

```
#include <stdio.h>
int main()
{
  int i,j;
  for (i=1,j=1 ; i<3 || j<5; i++,j++)
  {
          printf("%d, %d\n",i ,j);
  }
  return 0;
}
```

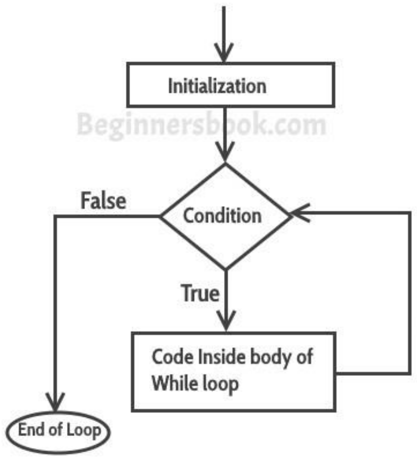
A loop is used for executing a block of statements repeatedly until a given condition returns false. In the previous tutorial we learned for loop. In this guide we will learn while loop in C.

While loop

Syntax of while loop:

```
while (condition test)
{
     //Statements to be executed repeatedly
     // Increment (++) or Decrement (--) Operation
}
```

Flow Diagram of while loop



Example of while loop

```
#include <stdio.h>
int main()
{
  int count=1;
  while (count <= 4)
  {
         printf("%d ", count);
         count++;
  }
  return 0;
}
```

Output:

```
1 2 3 4
```

step1: The variable count is initialized with value 1 and then it has been tested for the condition.
step2: If the condition returns true then the statements inside the body of while loop are executed else control comes out of the loop.
step3: The value of count is incremented using ++ operator then it has been tested again for the loop condition.

Guess the output of this while loop

```
#include <stdio.h>
int main()
{
    int var=1;
    while (var <=2)
    {
      printf("%d ", var);
    }
}
```

The program is an example of **infinite while loop**. Since the value of the variable var is same (there is no ++ or – operator used on this variable, inside the body of loop) the condition var<=2 will be true forever and the loop would never terminate.

1.1.2 Examples of infinite while loop

Example 1:

```
#include <stdio.h>
int main()
{
    int var = 6;
    while (var >=5)
    {
      printf("%d", var);
      var++;
    }
  return 0;
}
```

Infinite loop: var will always have value >=5 so the loop would never end.

Example 2:

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

```
{
   int var =5;
   while (var <=10)
   {
     printf("%d", var);
     var--;
   }
   return 0;
}
```

Infinite loop: var value will keep decreasing because of –- operator, hence it will always be <= 10.

1.2 Use of Logical operators in while loop

Just like relational operators (<, >, >=, <=, ! =, ==), we can also use logical operators in while loop. The following scenarios are valid :

```
while(num1<=10 && num2<=10)
```

-using AND(&&) operator, which means both the conditions should be true.

```
while(num1<=10||num2<=10)
```

– OR(||) operator, this loop will run until both conditions return false.

```
while(num1!=num2 &&num1 <=num2)
```

– Here we are using two logical operators NOT (!) and AND(&&).

```
while(num1!=10 ||num2>=num1)
```

1.2.1 Example of while loop using logical operator

In this example we are testing multiple conditions using logical operator inside while loop.

```
#include <stdio.h>
int main()
{
  int i=1, j=1;
  while (i <= 4 || j <= 3)
  {
          printf("%d %d\n",i, j);
          i++;
          j++;
  }
  return 0;
}
```

Output:

```
1 1
2 2
3 3
4 4
```

A do while loop is similar to while loop with one exception that it executes the statements inside the body of do-while before checking the condition. On the other hand in the while loop, first the condition is checked and then the statements in while loop are executed. So you can say that if a condition is false at the first place then the do while would run once, however the while loop would not run at all.

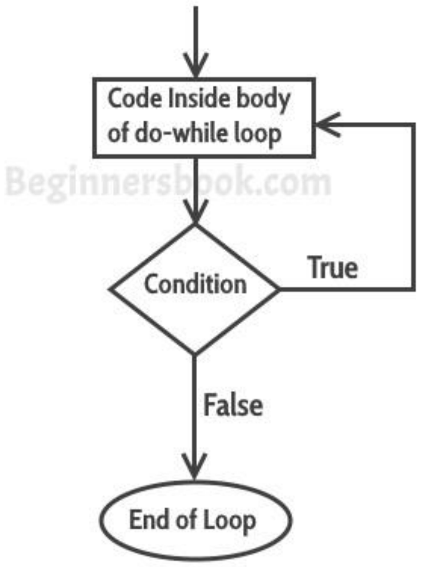
1.1 C – do..while loop

Syntax of do-while loop

```
do
{
   //Statements

}while(condition test);
```

Flow diagram of do while loop



Example of do while loop

```
#include <stdio.h>
int main()
{
		int j=0;
		do
		{
				printf("Value of variable j is: %d\n", j);
				j++;
		}while (j<=3);
		return 0;
}
```

Output:

```
Value of variable j is: 0
Value of variable j is: 1
Value of variable j is: 2
Value of variable j is: 3
```

1.2 While vs do..while loop in C

Using while loop:

```
#include <stdio.h>
int main()
{
   int i=0;
   while(i==1)
   {
	printf("while vs do-while");
   }
   printf("Out of loop");
}
```

Output:

```
Out of loop
```

Same example using do-while loop

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

```
int main()
{
  int i=0;
  do
  {
          printf("while vs do-while\n");
  }while(i==1);
  printf("Out of loop");
}
```

Output:

```
while vs do-while
Out of loop
```

Explanation: As I mentioned in the beginning of this guide that do-while runs at least once even if the condition is false because the condition is evaluated, after the execution of the body of loop.

2.18 Nested Loops

C programming allows to use one loop inside another loop. The following section shows a few examples to illustrate the concept.

Syntax

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

```
for ( init; condition; increment )
{
  for ( init; condition; increment )
  {
    statement(s);
  }
  statement(s);
}
```

The syntax for a **nested while loop** statement in C programming language is as follows −

```
while(condition)
{
  while(condition)
  {
    statement(s);
  }
  statement(s);
}
```

The syntax for a **nested do...while loop** statement in C programming language is as follows −

```
do {
  statement(s);
  do {
    statement(s);
  }while( condition );
}while( condition );
```

A final note on loop nesting is that you can put any type of loop inside any other type of loop. For example, a 'for' loop can be inside a 'while' loop or vice versa.

Example: The following program uses a nested for loop to find the prime numbers from 2 to 100 –

```c
#include <stdio.h>
 int main () {
   /* local variable definition */
   int i, j;
      for(i = 2; i<100; i++) {
      for(j = 2; j <= (i/j); j++)
      if(!(i%j)) break; // if factor found, not prime
      if(j > (i/j)) printf("%d is prime\n", i);
   }
    return 0;
}
```

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

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
97 is prime
```

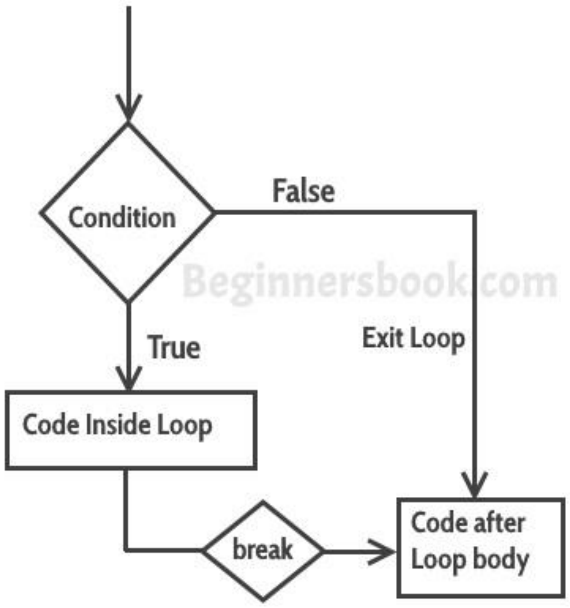
2.19 Break and Continue Statement –

The break statement is used inside loops and switch case.

C – break statement

1. It is used to come out of the loop instantly. When a break statement is encountered inside a loop, the control directly comes out of loop and the loop gets terminated. It is used with if statement, whenever used inside loop.
2. This can also be used in switch case control structure. Whenever it is encountered in switch-case block, the control comes out of the switch-case(see the example below).

Flow diagram of break statement



Syntax:

```
break;
```

1.2.1 Example – Use of break in a while loop

```
#include <stdio.h>
int main()
{
    int num =0;
    while(num<=100)
    {
      printf("value of variable num is: %d\n", num);
      if (num==2)
      {
         break;
      }
      num++;
    }
    printf("Out of while-loop");
    return 0;
}
```

Output:

```
value of variable num is: 0
value of variable num is: 1
value of variable num is: 2
Out of while-loop
```

1.2.2 Example – Use of break in a for loop

```
#include <stdio.h>
int main()
{
     int var;
     for (var =100; var>=10; var --)
     {
         printf("var: %d\n", var);
```

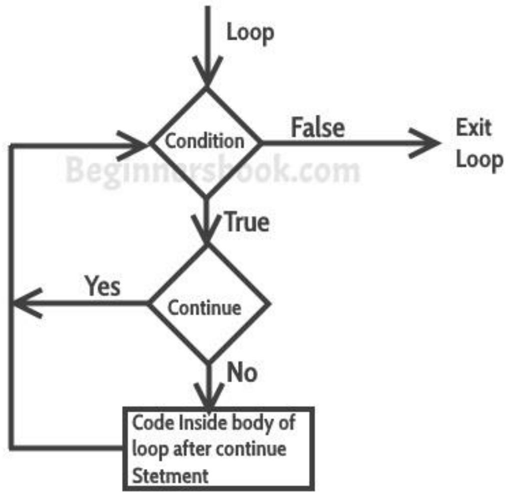
```
        if (var==99)
        {
           break;
        }
     }
    printf("Out of for-loop");
    return 0;
}
```
Output:
```
var: 100
var: 99
Out of for-loop
```
1.2.3 Example – Use of break statement in switch-case
```
#include <stdio.h>
int main()
{
     int num;
     printf("Enter value of num:");
     scanf("%d",&num);
     switch (num)
     {
        case 1:
          printf("You have entered value 1\n");
          break;
        case 2:
          printf("You have entered value 2\n");
          break;
        case 3:
          printf("You have entered value 3\n");
          break;
        default:
          printf("Input value is other than 1,2 & 3 ");
    }
    return 0;
}
```
Output:
```
Enter value of num:2
You have entered value 2
```
You would always want to use break statement in a switch case block, otherwise once a case block is executed, the rest of the subsequent case blocks will execute. For example, if we don't use the break statement after every case block then the output of this program would be:
```
Enter value of num:2
You have entered value 2
You have entered value 3
Input value is other than 1,2 & 3
```

The **continue statement** is used inside loops. When a continue statement is encountered inside a loop, control jumps to the beginning of the loop for next iteration, skipping the execution of statements inside the body of loop for the current iteration.

C – Continue statement

Syntax:
```
continue;
```

1.2.4 Flow diagram of continue statement



1.2.5 Example: continue statement inside for loop

```
#include <stdio.h>
int main()
{
  for (int j=0; j<=8; j++)
  {
    if (j==4)
    {
              /* The continue statement is encountered when
               * the value of j is equal to 4.
               */
              continue;
     }

     /* This print statement would not execute for the
          * loop iteration where j ==4  because in that case
          * this statement would be skipped.
          */
     printf("%d ", j);
  }
  return 0;
}
```

Output:

```
0 1 2 3 5 6 7 8
```

Value 4 is missing in the output, why? When the value of variable j is 4, the program encountered a continue statement, which makes the control to jump at the beginning of the for loop for next iteration, skipping the statements for current iteration (that's the reason printf didn't execute when j is equal to 4).

Example: Use of continue in While loop

In this example we are using continue inside while loop. When using while or do-while loop you need to place an increment or decrement statement just above the continue so that the counter value is changed for the next iteration. For example, if we do not place counter–statement in the body of "if" then the value of counter would remain 7 indefinitely.

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

```
{
   int counter=10;
   while (counter >=0)
   {
            if (counter==7)
            {
                counter--;
                continue;
            }
            printf("%d  ", counter);
            counter--;
   }
   return 0;
}
```

Output:

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

The print statement is skipped when counter value was 7.

Another Example of continue in do-While loop

```
#include <stdio.h>
int main()
{
  int j=0;
  do
  {
    if (j==7)
    {
      j++;
      continue;
    }
    printf("%d ", j);
    j++;
  }while(j<10);
  return 0;
}
```

Output:

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

2.20 Goto Statement

The goto statement is rarely used because it makes program confusing, less readable and complex. Also, when this is used, the control of the program won't be easy to trace, hence it makes testing and debugging difficult.

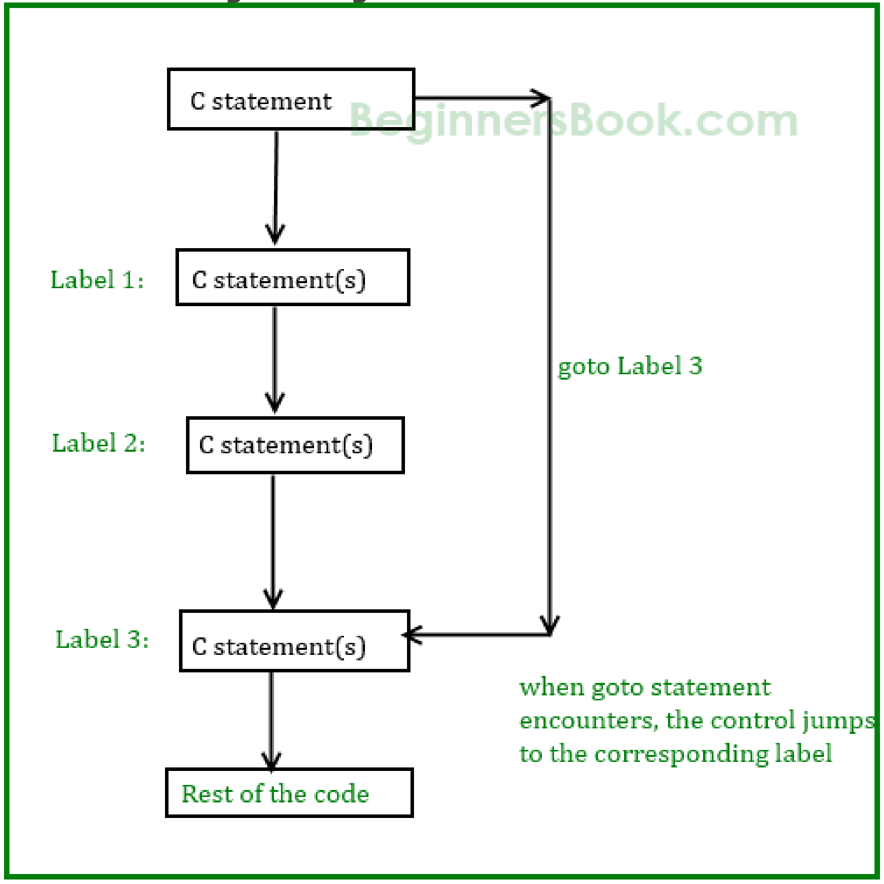
C – goto statement

When a goto statement is encountered in a C program, the control jumps directly to the label mentioned in the goto stateemnt

Syntax of goto statement in C

```
goto label_name;
..
..
label_name: C-statements
```

1.2.6 Flow Diagram of goto



Example of goto statement

```
#include <stdio.h>
int main()
{
  int sum=0;
  for(int i = 0; i<=10; i++){
         sum = sum+i;
         if(i==5){
            goto addition;
         }
  }

  addition:
  printf("%d", sum);

  return 0;
}
```

Output:

```
15
```

Explanation: In this example, we have a label addition and when the value of i (inside loop) is equal to 5 then we are jumping to this label using goto. This is reason the sum is displaying the sum of numbers till 5 even though the loop is set to run from 0 to 10.

Questions:

1. Write the Structure of C Program.
2. Explain about various File used in C Program
3. Discuss how to Compile and Executing C Programs.
4. Write about comments in C.
5. Define Keywords and list them.
6. Define Identifiers and write the rules for naming identifiers.
7. List and explain various Basic Data Types in C.
8. Define Constants and explain about them in detail.
9. Write about various I/O Statements in C.
10. List and explain various Operators in C.
11. Explain in detail about various Decision Control Statements
12. Discuss in detail about Conditional Branching Statements.
13. List and explain various Iterative Statements with examples.
14. Write about Nested Loops.
15. Explain about Break, Continue and Goto Statement.

Important programs.

1. A program to reverse a given number.
2. A program to check given number is prime or not.
3. A program to check a given number is palindrome or not.
4. A program to generate Fibonacci series
5. A program to find factorial of a given number.