UNIT – III

Types, Operators, and Expressions: Variable names, data types and sizes, constants, declarations, arithmetic operators, relational and logical operators, type conversions, increment and decrement operators, bitwise operators, assignment operators and expressions, conditional expressions precedence and order of evaluation.

Input and output: standard input and output, formatted output-Printf, formatted input-Scanf.

Control Flow: Statements and blocks, if-else, else-if, switch, Loops-while and for, Loops-Dowhile, break and continue, Goto and labels.

Functions and Program Structure: Basics of functions, functions returning non-integers, external variables, scope variables, header variables, register variables, block structure, initialization, recursion, the C processor.

3.1 VARIABLE NAMES

Each and every language in the world requires alphabets to form words. Likewise, a programming language also needs a set of characters to write a program.

The set of characters used in a language is known as its Character Set.

Every language makes use of its own character set to form words or symbols that make up the vocabulary of the language.

C language is case sensitive. By case sensitive, we mean that the C compiler treats lowercase and uppercase characters differently. For example, the variable name num1 is different from Num1.

C language has its own character set. The character set for ANSI Standard C (ANSI C) is as follows:

Uppercase alphabets: A to Z

Lowercase alphabets: a to z

Decimal digits: 0 to 9

Special characters: + - * / % = < > : ; , . ’ “ ?! # \ ( ) { } _ [ ] & | ^ ~

Escape sequences: \b \t \v \r \f \n \\ \' \"\? \0 \a

A variable is a name given to a memory location to store some value. Since the memory location can store different values during execution of a program, the name used to refer to it, is called

a variable.
Since variables are a part of identifiers, they follow the same naming conventions. Like identifiers, a valid variable name can start with an alphabet or an underscore (_) and later have a combination of one or more letters, digits and underscores.

A few examples of valid variable names are : sum,total,average_marks, etc.

When creating a variable, we should mention the type of data (for example, integer or character) that it would store. This is called the data type of that variable.

In a C program, variables should be declared before their usage.

The format for declaring a variable is data_type variable_name;.

For example:

```
int count; // int is the data type and count is the variable name
```

The above declaration can also be combined with initialisation. In such a case, the format for declaring a variable is data_type variable_name = constant_value;.

There are certain predefined words as part of C programming language that have a special meaning and purpose. They are called reserved words or keywords.

The user (programmer) cannot redefine these keywords, i.e.,the user cannot change their spellings or add new ones to the C programming language.

All reserved words in C are formed using only the lowercase letters.

Keywords have special meaning and purpose, so they cannot be used as variable names.

C language has 32 reserved words as per ANSI standards. They are as 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
```

3.2 DATA TYPES AND SIZES

Data types specify how we enter data into our programs and what type of data we enter. C language has some predefined set of data types to handle various kinds of data that we can use in our program. These datatypes have different storage capacities.

C language supports 2 different type of data types:

1. Primary data types: These are fundamental data types in C namely integer(int), floating point(float), character(char) and void.
2. Derived data types: Derived data types are nothing but primary datatypes but a little twisted or grouped together like array, stucture, union and pointer. These are discussed in details later.

Data type determines the type of data a variable will hold. If a variable x is declared as int. it means x can hold only integer values. Every variable which is used in the program must be declared as what data-type it is.

Integer type : Integers are used to store whole numbers.

Size and range of Integer type on 16-bit machine:

Type	Size(bytes)	Range
int or signed int	2	-32,768 to 32767
unsigned int	2	0 to 65535
short int or signed short int	1	-128 to 127
unsigned short int	1	0 to 255
long int or signed long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295

Floating point type

Floating types are used to store real numbers.

Size and range of Integer type on 16-bit machine

Type	Size(bytes)	Range
Float	4	3.4E-38 to 3.4E+38
double	8	1.7E-308 to 1.7E+308
long double	10	3.4E-4932 to 1.1E+4932

Character type

Character types are used to store characters value.

Size and range of Integer type on 16-bit machine

Type	Size(bytes)	Range
char or signed char	1	-128 to 127
unsigned char	1	0 to 255

void type

void type means no value. This is usually used to specify the type of functions which returns nothing. We will get acquainted to this datatype as we start learning more advanced topics in C language, like functions, pointers etc.

3.3 CONSTANTS :

Constants refer to fixed values that the program may not alter during its execution. These fixed values are also called literals.

Constants can be of any of the basic data types like an integer constant, a floating constant, a character constant, or a string literal. There are enumeration constants as well.

Constants are treated just like regular variables except that their values cannot be modified after their definition.

Integer Literals

An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal.

Following are other examples of various types of integer literals −

```
85       /* decimal */
0213     /* octal */
0x4b     /* hexadecimal */
30       /* int */
30u      /* unsigned int */
30l      /* long */
30ul     /* unsigned long */
```

Floating-point Literals

A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.

While representing decimal form, you must include the decimal point, the exponent, or both; and while representing exponential form, you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

Here are some examples of floating-point literals −

```
3.14159       /* Legal */
314159E-5L    /* Legal */
510E          /* Illegal: incomplete exponent */
210f          /* Illegal: no decimal or exponent */
.e55          /* Illegal: missing integer or fraction */
```

Character Constants

Character literals are enclosed in single quotes, e.g., 'x' can be stored in a simple variable of char type.

A character literal can be a plain character (e.g., 'x'), an escape sequence (e.g., '\t'), or a universal character (e.g., '\u02C0').

There are certain characters in C that represent special meaning when preceded by a backslash for example, newline (\n) or tab (\t).

Defining Constants

There are two simple ways in C to define constants −

- Using #define preprocessor.

- Using const keyword.

The #define Preprocessor

Given below is the form to use #define preprocessor to define a constant −

#define identifier value

example

```
#include <stdio.h>

#define LENGTH 10
#define WIDTH  5
```

The const Keyword

You can use const prefix to declare constants with a specific type as follows −

const type variable = value;

The following example explains it in detail

```
#include <stdio.h>

int main() {
   const int  LENGTH = 10;
   const int  WIDTH = 5;
   const char NEWLINE = '\n';
   int area;

   area = LENGTH * WIDTH;
   printf("value of area : %d", area);
   printf("%c", NEWLINE);

   return 0;
}
```

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

value of area : 50

3.4 DECLARATIONS

In C programming, variables which are to be used later in different parts of the functions have to be declared. Variable declaration tells the compiler two things:

- The name of the variable
- The type of data the variable will hold

There are two ways of declaring variable in C programming.

1. Primary Type Declaration
2. User Defined Type Declaration

Primary Type Declaration

A variable can store any data type in C programming. The name of the variable has nothing to do with its type. The general syntax of declaring a variable primarily is

```
data_type var1,var2,...varn;
```

Here, var1, var2,...varn are the names of valid variables.
Variables can also be defined in multiple lines instead of on the same line.

```
data_type var1;

data_type var2;

data_type varn;
```

When the variables are declared in single line, then the variables must be separated by commas.

Note: All declaration statements must end with a semi-colon(;).

For example:

```
int age;

float weight;

char gender;
```

In these examples, age, weight and gender are variables which are declared as integer data type, floating data type and character data type respectively.

User-Defined Type Declaration

In C programming, a feature known as "type definition" is available which allows a programmer to define an identifier that represents an existing data type. The user defined identifier can be used later in the program to declare variables. The general syntax of declaring a variable by user-defined type declaration is:

```
typedef type identifier;
```

Note: typedef cannot create a new type

Consider an example:

```
typedef int age;
```

```
typedef float weight;
```

Here, age represents int and weight represent float which can be used later in the program to declare variables as follows:

```
age boy1,boy2;

weight b1,b2;
```

Here, boy1 and boy2 are declared as as integer data type and b1 & b2 are declared as floating integer data type.
The main advantage of using user-defined type declaration is that we can create meaningful data type names for increasing the readability of a program.

Another user-defined data type is enumerated data type. The general syntax of enumerated data type is:

```
enum identifier {value 1,value 2,...value n};
```

Here, identifier is a user-defined enumerated data type which can be used to declare variables that can have one of the values enclosed within the braces. The values inside the braces are known as enumeration constants. After this declaration, we can declare variables to be of this 'new' type as:

```
enum identifier v1, v2, ... vn;
```

The enumerated variables v1, v2, ... vn can only have one of the values value1, value2, ... valuen. The following kinds of declarations are valid:

```
v1=value5;

v3=value1;
```

User-defined Type Declaration Example

```
enum mnth {January, February, ..., December};

enum mnth day_st, day_end;

day_st = January;

day_end = December;

if (day_st == February)
```

```
day_end = November;
```

The compiler automatically assigns integer digits begriming with 0 to all the enumeration constants. That is, the enumeration constant value1 is assigned 0, value2 is assigned 1, and so on. However, the automatic assignments can be overridden by assigning values explicitly to thee enumeration constants.
For example:

```
enum mnth {January = 1, February, ..., December};
```

Here, the constant January is assigned value 1. The remaining values are assigned values that increase successively by 1.
The definition and declaration of enumerated variables can be combined in one statement. For example;

```
enum mnth {January, ... December} day_st, day_end;
```

3.5 ARITHMETIC OPERATORS

The following table shows all the arithmetic operators supported by the C language. Assume variable A holds 10 and variable B holds 20, then −

Operator	Description	Example
+	Adds two operands.	A + B = 30
−	Subtracts second operand from the first.	A − B = -10
*	Multiplies both operands.	A * B = 200
/	Divides numerator by de-numerator.	B / A = 2
%	Modulus Operator and remainder of after an integer division.	B % A = 0
++	Increment operator increases the integer value by one.	A++ = 11
--	Decrement operator decreases the integer value by one.	A-- = 9

3.6RELATIONAL AND LOGICAL OPERATORS

The following table shows all the relational operators supported by C. Assume variable A holds 10 and variable B holds 20 then −

Show Examples

Operator	Description	Example
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	(A <= B) is true.

Logical Operators

Following table shows all the logical operators supported by C language. Assume variable A holds 1 and variable B holds 0, then −

<table>
<tr><th>Operator</th><th>Description</th><th>Example</th></tr>
<tr><td>&&</td><td>Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.</td><td>(A && B) is false.</td></tr>
<tr><td>||</td><td>Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.</td><td>(A || B) is true.</td></tr>
</table>

!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

3.7 TYPE CONVERSIONS

Converting one datatype into another is known as type casting or, type-conversion. For example, if you want to store a 'long' value into a simple integer then you can type cast 'long' to 'int'. You can convert the values from one type to another explicitly using the cast operator as follows −

(type_name) expression

Consider the following example where the cast operator causes the division of one integer variable by another to be performed as a floating-point operation −

```
#include <stdio.h>

main() {

   int sum = 17, count = 5;
   double mean;

   mean = (double) sum / count;
   printf("Value of mean : %f\n", mean );
}
```

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

Value of mean : 3.400000

It should be noted here that the cast operator has precedence over division, so the value of sum is first converted to type double and finally it gets divided by count yielding a double value.

Type conversions can be implicit which is performed by the compiler automatically, or it can be specified explicitly through the use of the cast operator. It is considered good programming practice to use the cast operator whenever type conversions are necessary.

3.8 INCREMENT AND DECREMENT OPERATORS

Increment Operators are used to increased the value of the variable by one and Decrement Operators are used to decrease the value of the variable by one in C programs.

Both increment and decrement operator are used on a single operand or variable, so it is called as a unary operator. Unary operators are having higher priority than the other operators it means unary operators are executed before other operators.

Syntax

```
++  // increment operator
--   // decrement operator
```

Note: Increment and decrement operators are can not apply on constant.

Example

```
x= 4++;  // gives error, because 4 is constant
```

Type of Increment Operator

- pre-increment
- post-increment

pre-increment (++ variable)

In pre-increment first increment the value of variable and then used inside the expression (initialize into another variable).

Syntax

```
++ variable;
```

Example pre-increment

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

void main()
{
int x,i;
i=10;
x=++i;
printf("x: %d",x);
printf("i: %d",i);
getch();
}
```

Output

```
x: 11
i: 11
```

In above program first increase the value of i and then used value of i into expression.

post-increment (variable ++)

In post-increment first value of variable is used in the expression (initialize into another variable) and then increment the value of variable.

Syntax

```
variable ++;
```

Example post-increment

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

void main()
{
int x,i;
i=10;
x=i++;
printf("x: %d",x);
printf("i: %d",i);
getch();
}
```

Output

```
x: 10
i: 11
```

In above program first used the value of i into expression then increase value of i by 1.

Type of Decrement Operator

- pre-decrement
- post-decrement

Pre-decrement (-- variable)

In pre-decrement first decrement the value of variable and then used inside the expression (initialize into another variable).

Syntax

```
-- variable;
```

Example pre-decrement

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

```
void main()
{
int x,i;
i=10;
x=--i;
printf("x: %d",x);
printf("i: %d",i);
getch();
}
```

Output

```
x: 9
i: 9
```

In above program first decrease the value of i and then value of i used in expression.

post-decrement (variable --)

In Post-decrement first value of variable is used in the expression (initialize into another variable) and then decrement the value of variable.

Syntax

```
variable --;
```

Example post-decrement

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

void main()
{
int x,i;
i=10;
x=i--;
printf("x: %d",x);
printf("i: %d",i);
getch();
}
```

Output

```
x: 10
```

```
i: 9
```

In above program first used the value of x in expression then decrease value of i by 1.

3.9 BITWISE OPERATORS

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ is as follows –

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume A = 60 and B = 13 in binary format, they will be as follows −

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

The following table lists the bitwise operators supported by C. Assume variable 'A' holds 60 and variable 'B' holds 13, then −

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100

<table>
<tr><td>|</td><td>Binary OR Operator copies a bit if it exists in either operand.</td><td>(A | B) = 61, i.e., 0011 1101</td></tr>
<tr><td>^</td><td>Binary XOR Operator copies the bit if it is set in one operand but not both.</td><td>(A ^ B) = 49, i.e., 0011 0001</td></tr>
<tr><td>~</td><td>Binary One's Complement Operator is unary and has the effect of 'flipping' bits.</td><td>(~A) = ~(60), i.e,. -0111101</td></tr>
<tr><td><<</td><td>Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.</td><td>A << 2 = 240 i.e., 1111 0000</td></tr>
<tr><td>>></td><td>Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.</td><td>A >> 2 = 15 i.e., 0000 1111</td></tr>
</table>

3.10 ASSIGNMENT OPERATORS AND EXPRESSIONS

Assignment Operators

The following table lists the assignment operators supported by the C language −

Show Examples

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand	C = A + B will assign the value of A + B to C
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	C += A is equivalent to C = C + A

-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

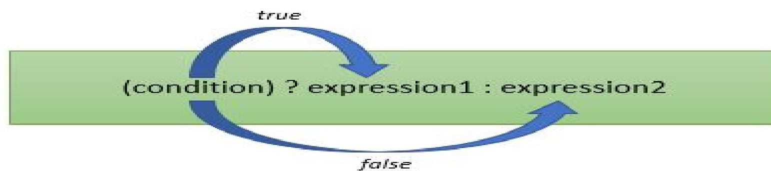
3.11 CONDITIONAL EXPRESSIONS

C programming conditional operator is also known as a **ternary operator**. It takes three operands. Conditional operator is closely related with if..else statement.

Syntax of C programming conditional operator

```
(condition) ? expression1 : expression2
```

If the **condition** is true then **expression1** is executed else **expression2** is executed.




```
#include <stdio.h>

int main()
{
  int x=1, y ;
  y = ( x ==1 ? 2 : 0 ) ;
  printf("x value is %d\n", x);
  printf("y value is %d", y);
}
```

Output: x value is 1
y value is 2

3.12 PRECEDENCE AND ORDER OF EVALUATION

Operator precedence determines the grouping of terms in an expression and decides how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has a higher precedence than the addition operator.

For example, x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has a higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

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

Example

```
#include <stdio.h>

main() {

   int a = 20;
   int b = 10;
   int c = 15;
```

```
  int d = 5;
  int e;
  e = (a + b) * c / d;      // ( 30 * 15 ) / 5
  printf("Value of (a + b) * c / d is : %d\n",  e );

  e = ((a + b) * c) / d;    // (30 * 15 ) / 5
  printf("Value of ((a + b) * c) / d is  : %d\n" ,  e );

  e = (a + b) * (c / d);   // (30) * (15/5)
  printf("Value of (a + b) * (c / d) is  : %d\n",  e );

  e = a + (b * c) / d;     //  20 + (150/5)
  printf("Value of a + (b * c) / d is  : %d\n" ,  e );

  return 0;
}
```

INPUT AND OUTPUT

3.13 STANDARD INPUT AND OUTPUT

C provides three different types of functions in the stdio.h header file for reading and writing data as given below:

Different Types of Functions	Function for Reading	Function for Printing
Character Oriented I/O Functions Used to read/write a single character at a time	getchar()	putchar()
Formatted I/O Functions Used to read/write a data of different formats	scanf()	printf()
String I/O Functions Used to read/write strings	gets()	puts()

The getchar() function is used to read a single character from the standard input.

The getchar() function returns an integer value of the character representation. If the system uses the ASCII character set, then the ASCII value of the character is returned.

The general format for using this function is character_variable = getchar();

Consider the following example given below:

```
char ch;
ch = getchar();
```

The variable ch is declared as a char data type (which occupies only one byte in memory) to store the single character read from the standard input.

The getchar() function will read only one character at a time and stores it in the variable ch.

The putchar() function transmits a single character to a standard output.

The general format for using this function is putchar(variable);, where the variable can either be an integer or a character.

When an integer is passed to the putchar() function, the ASCII character for the given integer value is printed.

Consider the following example using putchar():

```
char ch = 'A';
putchar(ch);  //Displays the character A
putchar(97);  //Displays the character a because 97 is the value of ASCII character 'a'
```

In C, a string is a sequence of characters ending with a '\0' character.

Given below is an example of a string declaration in C:

```
char arr[] = "CodeTantra";
```

Here,
arr[] is called a character array and
"CodeTantra" is called the string literal.

Note that the compiler automatically appends the above string literal "CodeTantra" wiith a '\0' character at the end. '\0' is called a NULL character.

Strings can be declared using character arrays or character pointers. (We shall learn more about arrays and pointers in later sections).

Like printf() and scanf(), the stdio.h file provides special methods like gets() and puts() to read and write strings from standard I/O.

The gets(string_variable) function takes the target character array into which it reads the string from standard input as a parameter.
Similarly, puts(string_variable or string_literal) function accepts character array or a string literal to print it to the standard output.

The scanf() function with a format character %s reads all the characters until a whitespace character is encountered.

The gets() function reads all the characters as a string until a newline character ('\n') is encountered (i.e., when the user presses the enter key). The string may include whitespace characters.

The puts() function writes the given string date to the standard output.

3.14 FORMATTED OUTPUT-PRINTF & FORMATTED INPUT-SCANF

The scanf() function in the stdio.h header file is used to read data of any data type using format characters.

The scanf() function can be used to read multiple data items at a time into variables and it returns the total number of data items that have been read successfully in the end.

The general format of scanf() function is scanf("control_string", argument_list);br/> where control_string contains the required format specifiers enclosed within double quotes and argument_list contains the addresses of the memory locations where the input data is stored. They are separated by commas.

The control_string can contain different format characters (used for reading different data types). Each format character must be prefixed with a % character.

The frequently used format/conversion characters used to read data in different formats are listed below:

Conversion Character	Data format
%d	decimal integer
%o	octal integer
%x	hex integer
%u	unsigned decimal integer
%h	short integer value

%f %e %g	floating point value
%c	single character
%s	string

The code given below demonstrates the usage of format characters in scanf() function:

```
char ch;
int a;
float b;
scanf("%d %f %c", &a, &b, &ch );
```

Here, the scanf() function reads three data items of types integer, float and char (using %d, %f and %c) into &a, &b and &ch respectively.

The printf() function in the stdio.h header file is used to print data of any data type using format characters to the standard output.

The printf() function can be used to print multiple data items stored in variables. In the end, it eturns the total number of characters printed.

The general format of printf() function is printf("control_string", argument_list);. Note that the arguments_list is not required when there are no format specifiers.

The control_string can contain any of the format specifiers enclosed within double quotes, similar to that of the scanf() function.

When format characters are used in the control_string, the argument_list should contain the corresponding variables separated by commas.

The code given below demonstrates the two different ways in which printf() can be used:

```
#include<stdio.h>
void main() {
        int a = 20, b = 30;
        printf("The sum of two given values = %d\n", a+b);
        printf("Hello CodeTantra!");
}
```

The code given above shows the usage of the two formats:
printf("control_string", argument_list)
printf("plain text")

CONTROL FLOW

3.15 STATEMENTS AND BLOCKS

Statements

C has three types of statement.

Assignment =

selection (branching)

```
if (expression)
else
switch
```

iteration (looping)

```
while (expression)
for (expression;expression;expression)
do {block}
```

Blocks

These statements are grouped into blocks, a block is identified by curly brackets...There are two types of block.

statement blocks

```
if ( i == j)
{
 printf("martin \n");
}
```

The statement block containing the printf is only executed if the i == j expression evaluates to TRUE.

function blocks

```
int add( int a, int b)      /* Function definition */
```

```
{
  int c;
  c = a + b;
  return c;
}
```

The statements in this block will only be executed if the add function is called. Complete function example

3.16 IF-ELSE

An if statement can be followed by an optional else statement, which executes when the Boolean expression is false.

Syntax

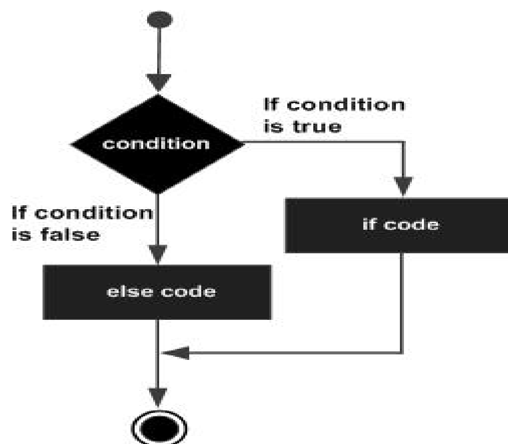
The syntax of an if...else statement in C programming language is −

```
if(boolean_expression)
 {
  /* statement(s) will execute if the boolean expression is true */
}
else
 {
  /* statement(s) will execute if the boolean expression is false */
}
```

If the Boolean expression evaluates to true, then the if block will be executed, otherwise, the else block will be executed.

C programming language assumes any non-zero and non-null values as true, and if it is either zero or null, then it is assumed as false value.

Flow Diagram



Example

```
#include <stdio.h>
 int main ()
{
      int a = 100;
       if( a < 20 )
        {
          printf("a is less than 20\n" );
        }
     else
        {
            printf("a is not less than 20\n" );
        }
      printf("value of a is : %d\n", a);
    return 0;
}
```

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

a is not less than 20;
value of a is : 100

3.17 ELSE-IF

If...else if...else Statement

An if statement can be followed by an optional else if...else statement, which is very useful to test various conditions using single if...else if statement.

When using if...else if..else statements, there are few points to keep in mind −

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.

Syntax

The syntax of an if...else if...else statement in C programming language is −

```
if(boolean_expression 1) {
   /* Executes when the boolean expression 1 is true */
} else if( boolean_expression 2) {
   /* Executes when the boolean expression 2 is true */
} else if( boolean_expression 3) {
   /* Executes when the boolean expression 3 is true */
} else {
   /* executes when the none of the above condition is true */
}
```

Example

```
#include <stdio.h>

int main () {

   /* local variable definition */
   int a = 100;

   /* check the boolean condition */
   if( a == 10 ) {
      /* if condition is true then print the following */
      printf("Value of a is 10\n" );
   } else if( a == 20 ) {
      /* if else if condition is true */
      printf("Value of a is 20\n" );
   } else if( a == 30 ) {
      /* if else if condition is true  */
      printf("Value of a is 30\n" );
   } else {
      /* if none of the conditions is true */
      printf("None of the values is matching\n" );
   }

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

   return 0;
}
```

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

None of the values is matching
Exact value of a is: 100

3.18 SWITCH

A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each switch case.

Syntax

The syntax for a switch statement in C programming language is as follows −

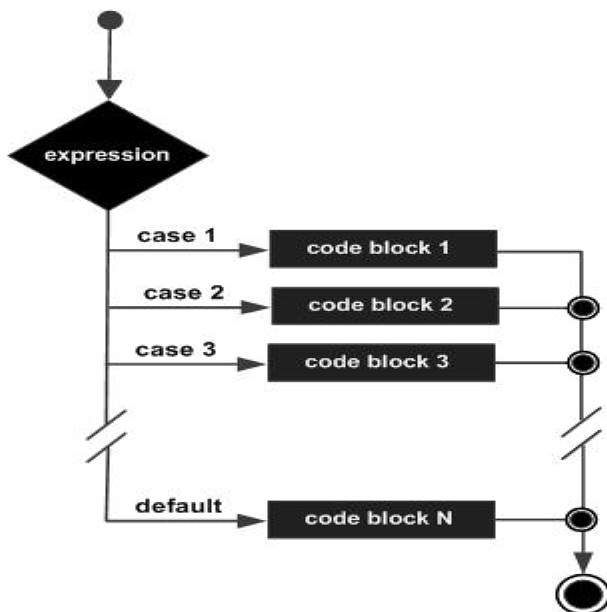
```
switch(expression) {

   case constant-expression  :
      statement(s);
      break; /* optional */

   case constant-expression  :
      statement(s);
      break; /* optional */
```

```
   /* you can have any number of case statements */
   default : /* Optional */
   statement(s);
}
```

The following rules apply to a switch statement −

- The expression used in a switch statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The constant-expression for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.
- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.
- A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Flow Diagram



Example

Live Demo

```
#include <stdio.h>

int main () {

   /* local variable definition */
   char grade = 'B';

   switch(grade) {
      case 'A' :
         printf("Excellent!\n" );
         break;
      case 'B' :
      case 'C' :
         printf("Well done\n" );
         break;
      case 'D' :
         printf("You passed\n" );
         break;
      case 'F' :
         printf("Better try again\n" );
         break;
      default :
         printf("Invalid grade\n" );
   }

   printf("Your grade is  %c\n", grade );

   return 0;
}
```

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

```
Well done
Your grade is B
```

3.19 WHILE

A while loop in C programming repeatedly executes a target statement as long as a given condition is true.

Syntax

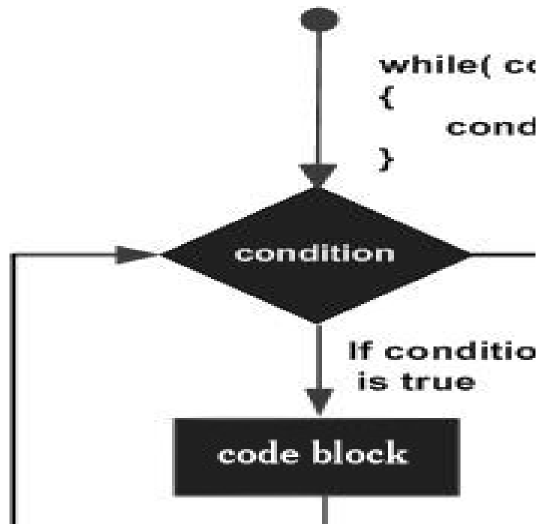
The syntax of a while loop in C programming language is −

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

Here, statement(s) may be a single statement or a block of statements. The condition may be any expression, and true is any nonzero value. The loop iterates while the condition is true.

When the condition becomes false, the program control passes to the line immediately following the loop.

Flow Diagram



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

Example

Live Demo

```
#include <stdio.h>

int main () {

   /* local variable definition */
   int a = 10;

   /* while loop execution */
   while( a < 20 ) {
      printf("value of a: %d\n", a);
      a++;
   }

   return 0;
}
```

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

value of a: 10
value of a: 11
value of a: 12
value of a: 13

value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19

3.20 FOR LOOP

A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax

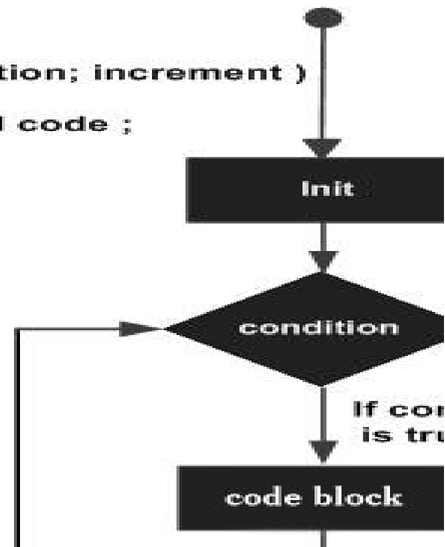
The syntax of a for loop in C programming language is −

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

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

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

Flow Diagram



Example

Live Demo

```
#include <stdio.h>

int main () {

   int a;

   /* for loop execution */
   for( a = 10; a < 20; a = a + 1 ){
      printf("value of a: %d\n", a);
   }

   return 0;
}
```

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

value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19

3.30 DOWHILE

Unlike for and while loops, which test the loop condition at the top of the loop, the do...while loop in C programming checks its condition at the bottom of the loop.

A do...while loop is similar to a while loop, except the fact that it is guaranteed to execute at least one time.

Syntax

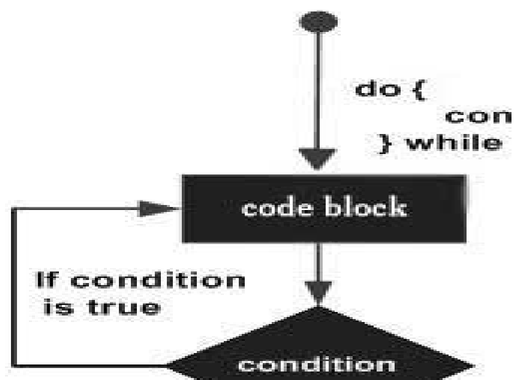
The syntax of a do...while loop in C programming language is −

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

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop executes once before the condition is tested.

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop executes again. This process repeats until the given condition becomes false.

Flow Diagram



Example

```
#include <stdio.h>

int main () {

   /* local variable definition */
   int a = 10;

   /* do loop execution */
   do {
      printf("value of a: %d\n", a);
      a = a + 1;
   }while( a < 20 );

   return 0;
}
```

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

value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19

3.31 BREAK AND CONTINUE

The continue statement in C programming works somewhat like the break statement. Instead of forcing termination, it forces the next iteration of the loop to take place, skipping any code in between.

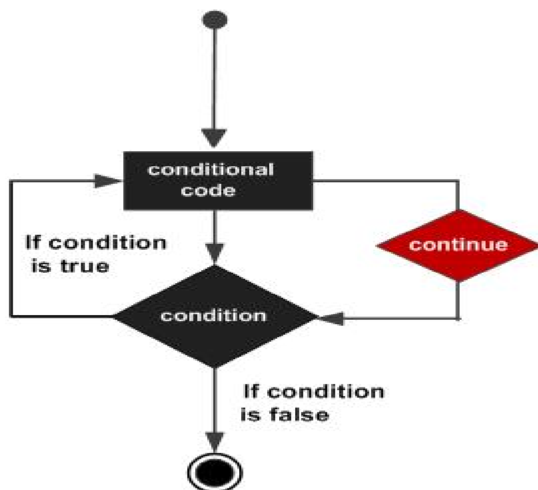
For the for loop, continue statement causes the conditional test and increment portions of the loop to execute. For the while and do...while loops, continue statement causes the program control to pass to the conditional tests.

Syntax

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

```
continue;
```

Flow Diagram



Example

Live Demo

```
#include <stdio.h>

int main () {

   /* local variable definition */
   int a = 10;

   /* do loop execution */
   do {

      if( a == 15) {
         /* skip the iteration */
         a = a + 1;
         continue;
      }

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

   } while( a < 20 );

   return 0;
}
```

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

value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19

3.32 GOTO AND LABELS

A goto statement in C programming provides an unconditional jump from the 'goto' to a labeled statement in the same function.

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

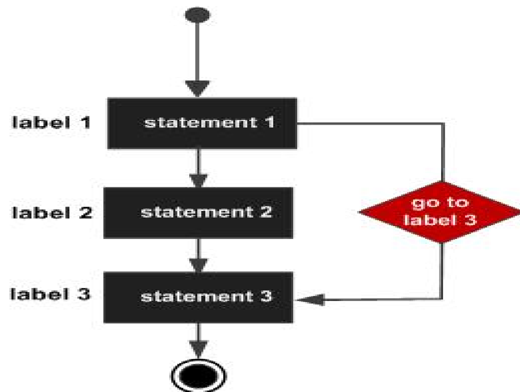
Syntax

The syntax for a goto statement in C is as follows −

goto label;
..
.
label: statement;

Here label can be any plain text except C keyword and it can be set anywhere in the C program above or below to goto statement.

Flow Diagram



Example

```
#include <stdio.h>

int main () {

   /* local variable definition */
   int a = 10;

   /* do loop execution */
   LOOP:do {

      if( a == 15) {
         /* skip the iteration */
         a = a + 1;
         goto LOOP;
      }

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

   }while( a < 20 );

   return 0;
}
```

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

value of a: 10
value of a: 11

value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19

FUNCTIONS AND PROGRAM STRUCTURE

3.33 BASICS OF FUNCTIONS

A program can be used to solve a simple problem or a large complex problem.

Programs solving simple problems are very easier to understand and identify mistakes, if any, in them.

The steps involved in programs solving large complex problems are difficult to understand. So, large programs are subdivided into a number of smaller programs called subprograms or modules.

Each subprogram specifies one or more actions to be performed for the larger program, such subprograms are called as subroutines or functions.

In some times we need to write a particular block of code for more than once in our program. This may lead to bugs and irritation for the programmer.

C language provides an approach in which you need to declare and define a group of statements once and that can be called and used whenever required. This saves both time and space.

A Function is a self-contained block of statements that specifies one or more actions to be performed for the large program.

The main reasons for using functions are:

- to improve the readability of code.
- improves the re-usability of the code, same function can be used in any program rather than writing the same code.
- debugging of the code would be easier if you use functions (errors are easy to be traced).
- reduces the size of the code, duplicate set of statements are replaced by function calls.

A C program is made up of one or more functions. Functions are classified into 2 types, they are Library functions and User-defined functions.

Built–in functions are predefined functions, supplied along with the compiler and these can be used in any C program. They are also known as library functions and all these functions are available in C library.

Some of the examples of library functions are scanf(), printf(), gets(), sqrt() and so on.

The functions defined by the users are called as user-defined functions. The main() function is an example of user-defined function because the code with in the main() is written by the user.

Let us consider the following code.

```
#include <stdio.h>
#include <math.h>
void main() {
        int x, y;
        printf("Enter the value : ");
        scanf("%d", &x);
        y = sqrt(x);
        printf ("The square root of %d is : %d", x, y);
}
```

Here main() is user-defined function and printf(), scanf(), sqrt() are library functions.

The main() function invokes other functions with in it and printf(), scanf(), sqrt() are invoked by the main() function.

Here main() is calling function and remaining are known as called functions.

A function that invokes another function is known as calling function. A function which is invoked by another function is known as called function. A called function may be a calling function for another function.

User-defined functions are the functions which are defined by the user at the time of writing program.

Functions are made for code re-usability and for saving time and space.

In C, main() is the user-defined function and first calling function in any program.

main() is a special function which tells the compiler to start the execution of a C program from the beginning of the function main().

It is not possible to have more than one main() function because the compiler will not know where to start the execution in such a situation.

An identifier other than keywords followed by parenthesis is recognized as a function name by the compiler.

To make use of the user-defined function the programmer must be able to know the following 3 concepts.

- Define a Function (or) Function Definition
- Function Prototype (or) Function Declaration
- Calling a Function (or) Invoke a Function

A function definition describes what a function does, how its actions are achieved and how it is used. It consists of a function header and a function body.

The general format of defining a function is

```
return_type function_name(parameters list) {
        //Local variable declarations
        //Executable statements
        return(expression);
}
```

The first line which heads the function is known as function header. The function header should not end with a semicolon (;) in defining a function.

The function body follows the function header and it is always enclosed in braces. The body of the function is combination of local variable declarations and executable statements.

Here the statements describe the actions to be performed by the function. The body of function definition is also known as a block or compound statement.

The elements specified within the parenthesis of the function name are known as parameters or arguments.

The general format of defining a function is

```
return_type function_name(parameters list) {
  //Local variable declarations
  //Executable statements
  return(expression);
```

```
}
```

The return_type specifies the data type of the value returned by the function. The return_value may be of the primitive data type or empty data type.

If return_type is omitted then the default return_type of any function is int.

void is specified in the place of return_type if the function returns no value.

function_name is any valid identifier. It can't begin with underscore because such names are reserved for the use of C library.

The arguments in the parameter_list are known as formal parameters. Zero or more parameters may be used. Each parameter must be preceded by its data type.

More than one parameter must be separated by commas. Parameters are used to pass the values into the function definition. For parameter less functions the keyword void is placed within the parenthesis of the function name.

It consists of the key word return followed by an expression within the block and it returns only a single value to the calling function when the function returns a value.

A value or an expression may follow return if the function returns a value; otherwise nothing follows return.

The syntax of the return statement is return(expression); (Or) return expression;.

The following points must be kept in mind while defining a function.

- A function cannot be defined more than once in a program.
- One function cannot be defined within another function definition.
- Function definitions may appear in any order.
- Built-in functions are predefined and they are available in C library that is supplied along the compiler.

Whenever a function is invoked with in another function it must be declared before use. Such declaration is known as function declaration or function prototype.

Function declaration always ends with a semicolon (;). The general format of the function prototype is

```
return_type function_name(parameter_list);
```

In the above function declaration, the parameter names in the parameter_list are optional.

Hence, it is possible to have the data type of each parameter without mentioning the parameter name as shown below.

```
return_type function_name(data_type, data_type ... data_type);
```

A parameter less function is declared by using void inside the parenthesis as

```
return_type function_name(void);
```

For user understandability all the function declarations are specified before the main() function.

A function is invoked to make use of it. The general format of a function call is

```
function_name(var1, var2 .... varn);
```

Where var1, var2,..., varn are argument expressions. For a parameter less function, there is no argument in the function call also.

The arguments var1, var2,..., varn in a function call are called as actual arguments.

If a function returns a value, the function call may appear in any expression and the returned value used as an operand in the evaluation of the expression.

3.34 FUNCTIONS RETURNING NON-INTEGERS:

An argument is an expression which is passed to a function by its caller in order for the function to perform its task. It is an expression in the comma-separated list bound by the parentheses in a function call expression.

A function may be called by the portion of the program with some arguments and these arguments are known as actual arguments (or) original arguments.

Actual arguments are local to the particular function. These variables are placed in the function declaration and function call. These arguments are defined in the calling function.

The parameters are variables defined in the function to receive the arguments.

Formal parameters are those parameters which are present in the function definition.

Formal parameters are available only with in the specified function. Formal parameters belong to the called function.

Formal parameters are also the local variables to the function. So, the formal parameters are

occupied memory when the function execution starts and they are destroyed when the function execution completed.

Let us consider the below example:

```
#include <stdio.h>
int add(int, int);
void main() {
	int a = 10, b = 20;
	printf("Sum of two numbers = %d\n", add(a, b)); // variables a, b are called actual arguments
}
int add(int x, int y) { // variables x, y are called formal parameters
	return(x + y);
}
```

In the above code whenever the function call add(a, b) is made, the execution control is transferred to the function definition of add().

The values of actual arguments a and b are copied in to the formal arguments x and y respectively.

The formal parameters x and y are available only with in the function definition of add(). After completion of execution of add(), the control is transferred back to the main().

A local variable is declared inside a function.

A local variable is visible only inside their function, only statements inside function can access that local variable.

Local variables are declared when the function execution started and local variables gets destroyed when control exits from function.

Let us consider an example:

```
#include <stdio.h>
void test();
void main() {
	int a = 22, b = 44;
	test();
	printf("Values in main() function a = %d and b = %d\n", a, b);
}
```

```
void test() {
        int a = 50, b = 80;
        printf("Values in test() function a = %d and b = %d\n", a, b);
}
```

In the above code we have 2 functions main() and test(), in these functions local variables are declared with same variable names a and b but they are different.

Operating System calls the main() function at the time of execution. the local variables with in the main() are created when the main() starts execution.

when a call is made to test() function, first the control is transferred from main() to test(), next the local variables with in the test() are created and they are available only with in the test() function.

After completion of execution of test() function, the local variables are destroyed and the control is transferred back to the main() function.

Global variables are declared outside of any function.

A global variable is visible to any every function and can be used by any piece of code.

Unlike local variable, global variables retain their values between function calls and throughout the program execution.

Let us consider an example:

```
#include <stdio.h>
int a = 20; // Global declaration
void test();
void main() {
        printf("In main() function a = %d\n", a); // Prints 20
        test();
        a = a + 15; // Uses global variable
        printf("In main() function a = %d\n", a); // Prints 55
}
void test() {
        a = a + 20; // Uses global variable
        printf("In test() function a = %d\n", a); // Prints 40
}
```

In the above code the global variable a is declared outside of all the functions. So, the variable a can be accessed in every function.

Operating System calls the main() function at the time of execution. the variable a has no local declaration, so it access the global variable a.

In test() function also there is no local declaration of variable a, the variable a gets access from the global.

The global variables are destroyed only after completion of execution of entire program.

All the C functions can be called either with arguments or without arguments in a C program. These functions may or may not return values to the calling function.

Depending on the arguments and return values functions are classified into 4 categories.

1. Function without arguments and without return value
2. Function with arguments and without return value
3. Function without arguments and with return value
4. Function with arguments and with return value

When a function has no arguments, it does not receive any data from the calling function.

Similarly, when a function does not return a value, the calling function does not receive any data from the called function.

In effect, there is no data transfer between the calling function and the called function in the category function without arguments and without return value.

Let us consider an example of a function without arguments and without return value:

```
#include <stdio.h>
void india_capital(void);
void main() {
        india_capital();
}
void india_capital() {
        printf("New Delhi is the capital of India\n");
}
```

In the above sample code the function void india_capital(void); specifies that the function does not receive any arguments and does not return any value to the main() function.

When a function definition has arguments, it receives data from the calling function.

The actual arguments in the function call must correspond to the formal parameters in the function definition, i.e. the number of actual arguments must be the same as the number of formal parameters, and each actual argument must be of the same data type as its corresponding formal parameter.

The formal parameters must be valid variable names in the function definition and the actual arguments may be variable names, expressions or constants in the function call.

The variables used in actual arguments must be assigned values before the function call is made. When a function call is made, copies of the values of actual arguments are passed to the called function.

What occurs inside the function will have no effect on the variables used in the actual argument list. There may be several different calls to the same function from various places with a program.

Let us consider an example of a function with arguments and without return value:

```
#include <stdio.h>
void largest(int, int);
void main() {
        int a, b;
        printf("Enter two numbers : ");
        scanf("%d%d" , &a, &b);
        largest(a, b);
}
void largest(int x, int y) {
        if (x > y) {
                printf("Largest element = %d\n", x);
        } else {
                printf("Largest element = %d\n", y);
        }
}
```

In the above sample code the function void largest(int, int); specifies that the function receives two integer arguments from the calling function and does not return any value to the called function.

When the function call largest(a, b) is made in the main() function, the values of actual arguments a and b are copied in to the formal parameters x and y.

After completion of execution of largest(int x, int y) function, it does not return any value to the main() function. Simply the control is transferred to the main() function.

When a function has no arguments, it does not receive any data from the calling function.

When a function return a value, the calling function receives data from the called function.

Let us consider an example of a function without arguments and with return value:

```
#include <stdio.h>
int sum(void);
void main() {
		printf("\nSum of two given values = %d\n", sum());
}
int sum() {
		int a, b, total;
		printf("Enter two numbers : ");
		scanf("%d%d", &a, &b);
		total = a + b;
		return total;
}
```

In the above sample code the function int sum(void); specifies that the function does not receive any arguments but return a value to the calling function.

3.35 EXTERNAL VARIABLES

External variables are also known as global variables. These variables are defined outside the function. These variables are available globally throughout the function execution. The value of global variables can be modified by the functions. “extern” keyword is used to declare and define the external variables.

Scope − They are not bound by any function. They are everywhere in the program i.e. global.

Default value − Default initialized value of global variables are Zero.

Lifetime − Till the end of the execution of the program.

Here are some important points about extern keyword in C language,

- External variables can be declared number of times but defined only once.
- “extern” keyword is used to extend the visibility of function or variable.

- By default the functions are visible throughout the program, there is no need to declare or define extern functions. It just increase the redundancy.
- Variables with “extern” keyword are only declared not defined.
- Initialization of extern variable is considered as the definition of the extern variable.

Here is an example of extern variable in C language

Example

```
#include <stdio.h>
extern int x = 32;
int b = 8;
int main() {
   auto int a = 28;
   extern int b;
   printf("The value of auto variable : %d\n", a);
   printf("The value of extern variables x and b : %d,%d\n",x,b);
   x = 15;
   printf("The value of modified extern variable x : %d\n",x);
   return 0;
}
```

Output

```
The value of auto variable : 28
The value of extern variables x and b : 32,8
The value of modified extern variable x : 15
```

3.36 SCOPE VARIABLES

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed. There are three places where variables can be declared in C programming language −

- Inside a function or a block which is called local variables.
- Outside of all functions which is called global variables.
- In the definition of function parameters which are called formal parameters.

Let us understand what are local and global variables, and formal parameters.

Local Variables

Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. The following example shows how local variables are used. Here all the variables a, b, and c are local to main() function.

Live Demo

```
#include <stdio.h>

int main () {

  /* local variable declaration */
  int a, b;
  int c;

  /* actual initialization */
  a = 10;
  b = 20;
  c = a + b;

  printf ("value of a = %d, b = %d and c = %d\n", a, b, c);

  return 0;
}
```

Global Variables

Global variables are defined outside a function, usually on top of the program. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. The following program show how global variables are used in a program.

Live Demo

```
#include <stdio.h>

/* global variable declaration */
int g;

int main () {

  /* local variable declaration */
  int a, b;

  /* actual initialization */
  a = 10;
  b = 20;
  g = a + b;

  printf ("value of a = %d, b = %d and g = %d\n", a, b, g);
```

```
  return 0;
}
```

A program can have same name for local and global variables but the value of local variable inside a function will take preference. Here is an example −

Live Demo

```
#include <stdio.h>

/* global variable declaration */
int g = 20;

int main () {

  /* local variable declaration */
  int g = 10;

  printf ("value of g = %d\n",  g);

  return 0;
}
```

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

value of g = 10

3.37 HEADER FILES

A header file is a file with extension .h which contains C function declarations and macro definitions to be shared between several source files. There are two types of header files: the files that the programmer writes and the files that comes with your compiler.

You request to use a header file in your program by including it with the C preprocessing directive #include, like you have seen inclusion of stdio.h header file, which comes along with your compiler.

Including a header file is equal to copying the content of the header file but we do not do it because it will be error-prone and it is not a good idea to copy the content of a header file in the source files, especially if we have multiple source files in a program.

A simple practice in C or C++ programs is that we keep all the constants, macros, system wide global variables, and function prototypes in the header files and include that header file wherever it is required.

Include Syntax

Both the user and the system header files are included using the preprocessing directive #include. It has the following two forms −

#include <file>

This form is used for system header files. It searches for a file named 'file' in a standard list of system directories. You can prepend directories to this list with the -I option while compiling your source code.

#include "file"

This form is used for header files of your own program. It searches for a file named 'file' in the directory containing the current file. You can prepend directories to this list with the -I option while compiling your source code.

3.38 REGISTER VARIABLES

In C language, each variable has a storage class which decides scope and lifetime of that variable.

The storage class of a variable determines whether the variable has a global or local scope and lifetime.

All variables defined in a C program are allocated some physical location in memory where variable's value is stored. Memory and CPU registers are examples of memory locations where a variable's value can be stored.

Storage classes provide the following information:

- Where the variable would be stored.
- What will be the initial value of the variable; i.e., the default value for the variable.
- What is the scope of the variable; i.e., if the variable is accessible globally or is local to a function.
- What is the life of the variable; i.e., how long would the variable exist in memory.

The scope determines the parts of a program in which a variable is available for use.

The lifetime refers to the period during which a variable retains a given value during execution of the program. It is also referred as longevity.

There are four storage class specifiers in C language, they are:

- auto
- extern
- static
- register

The general form of a variable declaration that uses a storage class is:

```
storage_class_specifier data_type variable_name;
```

At most one storage_class_specifier may be given in a declaration. If no storage_class_specifier is specified then following rules are used:

- Variables declared inside a function are taken to be auto.
- Variables declared outside a function are taken to be global.
- Automatic variables are declared inside a block/function in which they are to be utilized.

  These variables are created when a function is called and destroyed automatically when a function is exited.

  Automatic variables are private to the function where they are declared. Because of this property, automatic variables are also referred to as local or internal variables.

  A variable declared inside a function without any storage class specification, is by default an automatic variable.

  Auto variables can be only accessed within the block/function they have been declared and not outside them.

  We seldom use the auto keyword in declarations because all block-scoped variables which are not explicitly declared with other storage classes are implicitly auto.

  The format for specifying the automatic variables are

- auto data_type variable_1, variable_2,..., variable_n;
- Some of the important points regarding auto variables are:

- Storage: CPU memory
- Initial value: undefined (can be any value)
- Keyword: auto
- Scope: local to the block in which it is defined
- Lifetime: till control remains within the block in which it is defined

- A normal global variable can be made extern when it is being used much before its declaration.

  The format for specifying the external variables is

- extern data_type variable_1, variable_2,..., variable_n;
- Some of the important points regarding extern variables are:

- Storage: Memory
- Initial value: zero (0)
- Keyword: extern
- Scope: entire program
- Lifetime: throughout the program execution

- Auto variables are stored in the memory. CPU also has a small storage area called registers which are used for quicker storage and retrieval.

In C, we can suggest the compiler to store certain variables in the CPU registers by specifying them with the storage class register.

Apart from their physical storage location, register variables follow the same rules of scope as automatic variables.

The syntax for declaring register variables is

- register data_type variable_1;

- There may be some special cases when we want some frequently accessed variables to be stored in registers to increase the performance of the program. In such cases we use the register storage class while declaring the variable.

It should be noted that there is no guarantee that these variables will always be stored in registers. When there isn't enough space in registers, they will be stored in the memory along with other auto variables.

Care should be taken that we do not specify too many variables as register variables, as it may even degrade the performance. If there are too many, the CPU might end up spending too much time moving data back and forth between registers and temporary storage area, if there are no enough registers to hold all such variables.

Ideally registers should be used only when one has good knowledge of the compiler and the computer architecture being used.

A few points regarding register variables are given below:

- Storage: CPU registers
- Initial value: garbage value
- Keyword: register
- Scope: local to the block in which it is defined
- Lifetime: till control remains within the block in which it is defined

- When a variable is specified as static its value is persisted until the end of program.

A variable can be declared as a static variable by using the keyword static.

Auto variables are created each time they are initialized in a block and are destroyed when they go out of scope.

However, static variables are initialized only once and they remain in existence till the end of program.

A static variable can either be local (to a function) or global.

The format for specifying static variables is

- static data_type variable_1, variable_2,..., variable_n;
- The static variables which are declared inside a function are stored in the statically allocated memory and remains alive through out the execution of the program, but remember that their scope remains local to the function.

The main difference between auto and static is that the static variables do not disappear when the function is no longer active (in scope) and their values persist across multiple calls of the same function.

Some of the important points regarding static variables are:

- Storage: Memory
- Initial value: zero (0)
- Keyword: static
- Scope: local to the block in which it is defined
- Lifetime: throughout the program execution

3.39 BLOCK STRUCTURE

C is not a block-structured language in the sense of Pascal or similar languages, because functions may not be defined within other functions. On the other hand, variables can be defined in a block-structured fashion within a function. Declarations of variables (including initializations) may follow the left brace that introduces *any* compound statement, not just the one that begins a function. Variables declared in this way hide any identically named variables in outer blocks, and remain in existence until the matching right brace. For example, in

```
if (n > 0) {
    int i;  /* declare a new i */

    for (i = 0; i < n; i++)
        ...
}
```

the scope of the variable i is the "true" branch of the if; this i is unrelated to any i outside the block. An automatic variable declared and initialized in a block is initialized each time the block is entered. A static variable is initialized only the first time the block is entered.

Automatic variables, including formal parameters, also hide external variables and functions of the same name. Given the declarations

```
int x;
int y;

f(double x)
{
    double y;
    ...
}
```

then within the function f, occurrences of x refer to the parameter, which is a double; outside of f, they refer to the external int. The same is true of the variable y.

As a matter of style, it's best to avoid variable names that conceal names in an outer scope; the potential for confusion and error is too great.

3.40 INITIALIZATION

First of all we should learn how to declare a variable in c programming language? There are two things which need to be defined while declaring a variable:

- Data type - which type of value is going to be stored in the variable.
- Identifier - Valid variable name (name of the allocated memory blocks for the variable).

Variable initialization

In c programming language, variable can be initialized in the declaration statement of any block (either it may main's block or any other function's block).

While declaring a variable you can provide a value to the variable with assignment operator.

Here is the syntax of the variable initialization

data_type variable_name=value;

Integer variable initialization
int number=10;
Float variable initialization
float value=23.45f;
Character variable initialization
char gender = 'M';
Character array/ string initialization
char country_name[]= "India";
OR
char country_name[10]= "India";
/*here 10 is maximum number of character*/
Integer array initialization
int arr[]={10,20,30,40,50};
OR
int arr[5]={10,20,30,40,50};

3.41 RECURSION

Those functions which are called by themselves are called Recursive functions. It means that the same function is called again within itself. The function itself becomes the calling function of it. Control is cycled within the function until a break point is reached in the program.

Non-Recursive Function
These functions are called only once from the calling function.
Eg:
/* Write C programs that use both recursive and non-recursive functions To find the factorial of a given integer.*/

```
01 #include< stdio.h>

02 #include< conio.h>

03 unsigned long factrec(int );

04 unsigned long factnrec(int );
```

```
05  void main()
06  {
07   int a;
08   unsigned long f1,f2;
09   clrscr();
10   printf("\n\n\t\tENTER VALUE OF A: ");
11   scanf("%d",&a);
12   f1=factrec(a);
13   f2=factnrec(a);
14   printf("\n\n\nFACTORIAL OF %d USING RECURSIVE FUNCTION IS: %d\n",a,f1);
15   printf("\nFACTORIAL OF %d USING NON-RECURSIVE FUNCTION IS :
 %d",a,f2);
16   getch();
17  }
18  /* RECURSIVE FUNCTION*/
19
20  unsigned long factrec(int x)
21  {
22   if(x<=1)
23   return 1;
24   else
25   return x*factrec(x-1);
26  }
```

```
27

28  /* NON-RECURSIVE FUNCTION*/

29  unsigned long factnrec(int x)

30  {

31    int i;

32    unsigned long f=1;

33    if(x<=1)

34    return 1;

35    else

36    {

37     for(i = 1; i <= x; i++)

38     f=f*i;

39   }

40   return f;

41 }
```

OUTPUT:
ENTER VALUE OF A: 6
FACTORIAL OF 6 USING RECURSIVE FUNCTION IS: 720
FACTORIAL OF 6 USING NON-RECURSIVE FUNCTION IS : 720

3.42 THE C PROCESSOR

C language provides a collection of such header files which form the C standard library. These files are usually available in operating systems like Linux by default.

Programmers can also create their own header files which are usually referred to as user defined header files.

As mentioned earlier, there is a file called stdio.h among the header files present in C standard

library, which contains the most commonly used functions to print data to console and to read (also called scan) data.
In order to use the functions available in the header file stdio.h, the following line has to be used in a program:

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

It makes the contents of the header file available to the compiler and the linker during the execution of programs.
The header files are included in a program using #include directive.

The header files can be included using #include in two ways as follows:

1. #include <header_file_name.h>: This variant is used to include system header files made available in C standard library. The compiler searches for named file in the standard list of system directories.
2. #include "header_file_name.h": This variant is commonly used to include user defined header files. The compiler searches for the named files only in the local or project-specific paths.

Given below is the most commonly included header file which contains the standard input/output functions like, printf(), scanf(), etc.

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

Here, the symbol # is called the preprocessor directive, include is called the command and stdio.h is the header file.

In programming terminology, a macro is a pattern or a rule which specifies how a certain sequence of text should be replaced.
C allows us to define two types of macros using the preprocessor directive #define as shown below:

1. #define PI 3.14
2. #define MAX(a, b) ((a) > (b)? (a) : (b))

The first type of substitution has been discussed while learning about symbolic constant.
The second type can be used to define a rule or a function which works on given arguments.
Given below is the general syntax for declaring macros using #define:

```
#define macro_name replacement_text

#define macro_name(arg1, arg2... argn) function_expression
```