SRINIVASAN COLLEGE OF ARTS AND SCIENCE

PERAMBALUR

SUBJECT	**: PROGRAMMING IN C**
SUBJECT CODE	**: 16SCCIT2**
CLASS	**: I B.Sc IT**
SEMESTER	**: II**

PROGRAMMING IN C

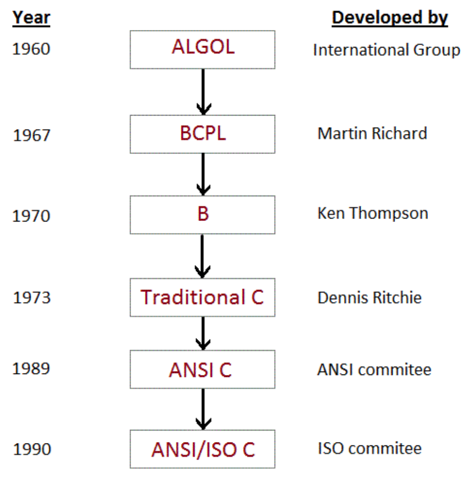
UNIT I

INTRODUCTION OF C LANGUAGE

C is a structured programming language developed by Dennis Ritchie in 1973 at Bell Laboratories. It is one of the most popular computer languages today because of its structure, high-level abstraction, machine independent feature etc. C language was developed to write the UNIX operating system, hence it is strongly associated with UNIX, which is one of the most popular network operating system in use today and heart of internet data superhighway.

History of C language

C language has evolved from three different structured language ALGOL, BCPL and B Language. It uses many concepts from these languages while introduced many new concepts such as datatypes, struct, pointer etc. In 1988, the language was formalised by **American National Standard Institute**(ANSI). In 1990, a version of C language was approved by the **International Standard Organisation**(ISO) and that version of C is also referred to as C89.



C Program and its Structure

- Pre-processor
- Header file
- Function
- Comments

(i) PREPROCESSOR

#include is the first word of any C program. It is also known as a **pre-processor**. The task of a pre-processor is to initialize the environment of the program, i.e to link the program with the header files required.So, when we say #include <stdio.h>, it is to inform the compiler to include the **stdio.h** header file to the program before executing it.

(ii) Header file

A Header file is a collection of built-in(readymade) functions. Header files contain definitions of the functions which can be incorporated into any C program by using pre-processor #include statement with the header file. To use any of the standard functions, the appropriate header file must be included. This is done at the beginning of the C source file. to use the printf() function in a program, which is used to display anything on the screen, the line #include <stdio.h> is required because the header file **stdio.h** contains the printf() function. All header files will have an extension .h

(iii) main() function

main() function is a function that must be there in every C program. Everything inside this function in a C program will be executed. In the above example, int written before the main() function is the return type of main() function. we will discuss about it in detail later. The curly braces { } just after the main() function encloses the body of main() function.

(iv) Comments

We can add comments in our program to describe what we are doing in the program. These comments are ignored by the compiler and are not executed.

To add a single line comment, start it by adding two forward slashses // followed by the comment.

To add multiline comment, enclode it between /* .... */

(V) Return statement - return 0;

A return statement is just meant to define the end of any C program.

All the C programs can be written and edited in normal text editors like Notepad or Notepad++ and must be saved with a file name with extension as .c

FIRST C PROGRAM

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

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.

An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.

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.

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).

String Literals

String literals or constants are enclosed in double quotes "". A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.

You can break a long line into multiple lines using string literals and separating them using white spaces.

Variables

In C language we can store it in a memory space and name the memory space so that it becomes easier to access it.

The naming of an address is known as **variable**. Variable is the name of memory location. Unlike constant, variables are changeable, we can change value of a variable during execution of a program. A programmer can choose a meaningful variable name. Example : average, height, age, total etc.

Datatype of Variable

A **variable** in C language must be given a type, which defines what type of data the variable will hold.

It can be:

- char: Can hold/store a character in it.
- int: Used to hold an integer.
- float: Used to hold a float value.
- double: Used to hold a double value.
- void

Rules to name a Variable

1. Variable name must not start with a digit.
2. Variable name can consist of alphabets, digits and special symbols like underscore _.
3. Blank or spaces are not allowed in variable name.
4. Keywords are not allowed as variable name.
5. Upper and lower case names are treated as different, as C is case-sensitive, so it is suggested to keep the variable names in lower case.

Variable declaration and initialization

Syntax:

Datatype variable name;

Example: int a;

int a=10;

Data types

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:

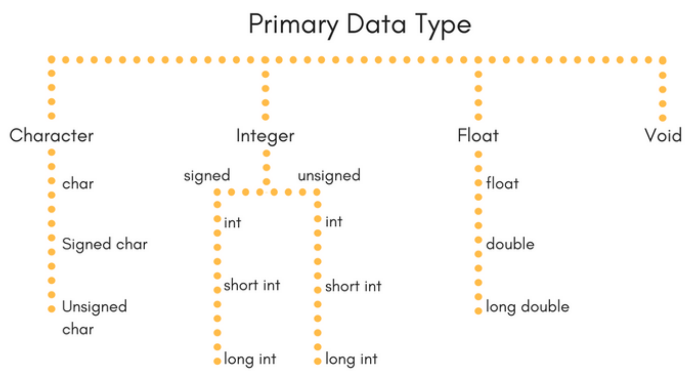
1. **Primary data types**:

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.

Operators

C language supports a rich set of built-in operators. An operator is a symbol that tells the compiler to perform a certain mathematical or logical manipulation. Operators are used in programs to manipulate data and variables.

C operators can be classified into following types:

- Arithmetic operators
- Relational operators
- Logical operators
- Bitwise operators
- Assignment operators
- Conditional operators
- Special operators

Arithmetic operators

C supports all the basic arithmetic operators. The following table shows all the basic arithmetic operators.

Operator	Description
+	adds two operands
-	subtract second operands from first
*	multiply two operand
/	divide numerator by denominator
%	remainder of division

++	Increment operator - increases integer value by one
--	Decrement operator - decreases integer value by one

Relational operators

The following table shows all relation operators supported by C.

Operator	Description
==	Check if two operand are equal
!=	Check if two operand are not equal.
>	Check if operand on the left is greater than operand on the right
<	Check operand on the left is smaller than right operand
>=	check left operand is greater than or equal to right operand
<=	Check if operand on left is smaller than or equal to right operand

Logical operators

C language supports following 3 logical operators. Suppose a = 1 and b = 0,

Operator	Description	Example
&&	Logical AND	(a && b) is false
||	Logical OR	(a || b) is true
!	Logical NOT	(!a) is false

Bitwise operators

Bitwise operators perform manipulations of data at **bit level**. These operators also perform **shifting of bits** from right to left. Bitwise operators are not applied to float or double(These are datatypes, we will learn about them in the next tutorial).

Operator	Description
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
<<	left shift
>>	right shift

Now lets see truth table for bitwise &, | and ^

a	b	a & b	a | b	a ^ b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

The bitwise **shift** operator, shifts the bit value. The left operand specifies the value to be shifted and the right operand specifies the number of positions that the bits in the value have to be shifted. Both operands have the same precedence.

Assignment Operators

Assignment operators supported by C language are as follows.

Operator	Description	Example
=	assigns values from right side operands to left side operand	a=b
+=	adds right operand to the left operand and assign the result to left	a+=b is same as a=a+b
-=	subtracts right operand from the left operand and assign the result to left operand	a-=b is same as a=a-b
*=	mutiply left operand with the right operand and assign the result to left operand	a*=b is same as a=a*b
/=	divides left operand with the right operand and assign the result to left operand	a/=b is same as a=a/b
%=	calculate modulus using two operands and assign the result to left operand	a%=b is same as a=a%b

---

Conditional operator

The conditional operators in C language are known by two more names

1. **Ternary Operator**
2. **? : Operator**

It is actually the if condition that we use in C language decision making, but using conditional operator, we turn the if condition statement into a short and simple operator.

The syntax of a conditional operator is :

expression 1 ? expression 2: expression 3

Explanation:

- The question mark **"?"** in the syntax represents the **if** part.
- The first expression (expression 1) generally returns either true or false, based on which it is decided whether (expression 2) will be executed or (expression 3)
- If (expression 1) returns true then the expression on the left side of **" : "** i.e (expression 2) is executed.
- If (expression 1) returns false then the expression on the right side of **" : "** i.e (expression 3) is executed.

Special operator

Operator	Description	Example
sizeof	Returns the size of an variable	**sizeof(x)** return size of the variable **x**
&	Returns the address of an variable	**&x ;** return address of the variable **x**
*	Pointer to a variable	***x ;** will be pointer to a variable **x**

Expressions

An expression is a combination of variables constants and operators written according to the syntax of C language. In C every expression evaluates to a value i.e., every expression results in some value of a certain type that can be assigned to a variable. Some examples of C expressions are shown in the table given below.

Example

a*b-c/d

Evaluation of Expressions

Expressions are evaluated using an assignment statement of the form

Variable = expression;

Variable is any valid C variable name. When the statement is encountered, the expression is evaluated first and then replaces the previous value of the variable on the left hand side. All variables used in the expression must be assigned values before evaluation is attempted. recedence in Arithmetic Operators

An arithmetic expression without parenthesis will be evaluated from left to right using the rules of precedence of operators. There are two distinct priority levels of arithmetic operators in C.

High priority * / %

Low priority + -

Rules for evaluation of expression

• First parenthesized sub expression left to right are evaluated.
• If parenthesis are nested, the evaluation begins with the innermost sub expression.
• The precedence rule is applied in determining the order of application of operators in evaluating sub expressions.
• The associability rule is applied when two or more operators of the same precedence level appear in the sub expression.
• Arithmetic expressions are evaluated from left to right using the rules of precedence.
• When Parenthesis are used, the expressions within parenthesis assume highest priority.

Operator precedence and associativity

Each operator in C has a precedence associated with it. The precedence is used to determine how an expression involving more than one operator is evaluated. There are distinct levels of precedence and an operator may belong to one of these levels. The operators of higher precedence are evaluated first. The operators of same precedence are evaluated from right to left or from left to right depending on the level. This is known as associativity property of an operator.

The table given below gives the precedence of each operator.

Order	**Category**	**Operator**	**Operation**	**Associativity**
1	Highest precedence	( ) [ ] ? : : .	Function call	L ? R Left to Right
2	Unary	! ~ + - ++ - - & * Size of	Logical negation (NOT) Bitwise 1's complement Unary plus Unary minus Pre or post increment Pre or post decrement Address Indirection Size of operant in bytes	R ? L Right -> Left
3	Member Access	.* ?*	Dereference Dereference	L ? R
4	Multiplication	* / %	Multiply Divide Modulus	L ? R
5	Additive	+ -	Binary Plus Binary Minus	L ? R
6	Shift	<< >>	Shift Left Shift Right	L ? R
7	Relational	< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	L ? R
8	Equality	== !=	Equal to Not Equal to	L ? R
9	Bitwise AAND	&	Bitwise AND	L ? R
10	Bitwise XOR	^	Bitwise XOR	L ? R
11	Bitwise OR	|	Bitwise OR	L ? R
12	Logical AND	&&	Logical AND	L ? R
14	Conditional	? :	Ternary Operator	R ? L
15	Assignment	= *= %= /= += -=	Assignment Assign product Assign reminder Assign quotient Assign sum Assign difference	R ? L

		&= ^= |= <<= >>=	Assign bitwise AND Assign bitwise XOR Assign bitwise OR Assign left shift Assign right shift	
16	Comma	,	Evaluate	L ? R

UNIT – II

MANAGING INPUT AND OUTPUT OPERATIONS

C Input and Output

Input means to provide the program with some data to be used in the program and **Output** means to display data on screen or write the data to a printer or a file.

C programming language provides many built-in functions to read any given input and to display data on screen when there is a need to output the result.

scanf() and printf() functions

The standard input-output header file, named stdio.h contains the definition of the functions printf() and scanf(), which are used to display output on screen and to take input from user respectively. scanf() or printf() functions. It is known as **format string** and this informs the scanf() function, what type of input to expect and in printf() it is used to give a heads up to the compiler, what type of output to expect.

Format String	Meaning
%d	Scan or print an integer as signed decimal number
%f	Scan or print a floating point number
%c	To scan or print a character
%s	To scan or print a character string. The scanning ends at whitespace.

SYNTAX:

printf("studytonight");

SYNTAX:

Scanf(“control string”, & variable name);

getchar() & putchar() functions

This function reads only single character at a time. You can use this method in a loop in case you want to read more than one character. The putchar() function displays the character passed to it on the screen and returns the same character. This function too displays only a single character at a time.

```
c = getchar();
putchar(c);
```

gets() & puts() functions

The gets() function reads a line from **stdin**(standard input) into the buffer pointed to by str pointer, until either a terminating newline or EOF (end of file) occurs. The puts() function writes the string str and a trailing newline to **stdout**.

str → This is the pointer to an array of chars where the C string is stored.

Decision making statements

Decision making is about deciding the order of execution of statements based on certain conditions or repeat a group of statements until certain specified conditions are met. C language handles decision-making by supporting the following statements,

- if statement
- switch statement
- conditional operator statement (? : operator)
- goto statement

Decision making with if statement

The if statement may be implemented in different forms depending on the complexity of conditions to be tested. The different forms are,

1. Simple if statement
2. if....else statement
3. Nested if....else statement
4. Using else if statement

Simple if statement

The general form of a simple if statement is,

```
if(expression)
{
   statement inside;
}
   statement outside;
```

If the *expression* returns true, then the **statement-inside** will be executed, otherwise **statement-inside** is skipped and only the **statement-outside** is executed.

Example:

```
#include <stdio.h>

void main( )
{
   int x, y;
   x = 15;
   y = 13;
   if (x > y )
   {
      printf("x is greater than y");
   }
}
```

```
x is greater than y
```

if...else statement

The general form of a simple if...else statement is,

```
if(expression)
{
   statement block1;
}
else
{
   statement block2;
}
```

If the *expression* is true, the **statement-block1** is executed, else **statement-block1** is skipped and **statement-block2** is executed.

Example:

```
#include <stdio.h>

void main( )
{
   int x, y;
   x = 15;
   y = 18;
   if (x > y )
   {
      printf("x is greater than y");
```

```
    }
    else
    {
        printf("y is greater than x");
    }
}
```

```
y is greater than x
```

Nested if....else statement

The general form of a nested if...else statement is,

```
if( expression )
{
    if( expression1 )
    {
        statement block1;
    }
    else
    {
        statement block2;
    }
}
else
{
    statement block3;
}
```

if *expression* is false then **statement-block3** will be executed, otherwise the execution continues and enters inside the first if to perform the check for the next if block, where if *expression 1* is true the **statement-block1** is executed otherwise **statement-block2** is executed.

Example:

```
#include <stdio.h>

void main( )
{
    int a, b, c;
    printf("Enter 3 numbers...");
    scanf("%d%d%d",&a, &b, &c);
    if(a > b)
    {
        if(a > c)
        {
            printf("a is the greatest");
        }
        else
        {
            printf("c is the greatest");
        }
    }
    else
    {
        if(b > c)
        {
            printf("b is the greatest");
        }
        else
        {
            printf("c is the greatest");
        }
    }
}
```

else if ladder

The general form of else-if ladder is,

```
if(expression1)
{
    statement block1;
}
else if(expression2)
{
    statement block2;
}
else if(expression3 )
{
    statement block3;
}
else
    default statement;
```

The expression is tested from the top(of the ladder) downwards. As soon as a **true** condition is found, the statement associated with it is executed.

Example :

```
#include <stdio.h>

void main( )
{
    int a;
    printf("Enter a number...");
    scanf("%d", &a);
    if(a%5 == 0 && a%8 == 0)
    {
        printf("Divisible by both 5 and 8");
    }
    else if(a%8 == 0)
    {
        printf("Divisible by 8");
    }
    else if(a%5 == 0)
    {
        printf("Divisible by 5");
    }
    else
    {
        printf("Divisible by none");
    }
}
```

Switch statement

switch statement is a control statement that allows us to choose only one choice among the many given choices. The expression in switch evaluates to return an integral value, which is then compared to the values present in different cases. It executes that block of code which matches the case value. If there is no match, then **default** block is executed(if present). The general form of switch statement is,

```
switch(expression)
{
    case value-1:
              block-1;
              break;
    case value-2:
              block-2;
              break;
    case value-3:
              block-3;
              break;
    case value-4:
              block-4;
                  break;
    default:
                  default-block;
              break;
}
```

Rules for using switch statement

1. The expression (after switch keyword) must yield an **integer** value i.e the expression should be an integer or a variable or an expression that evaluates to an integer.
2. The case **label** values must be unique.
3. The case label must end with a colon(:)
4. The next line, after the **case** statement, can be any valid C statement.

Looping statements

loops are used to execute a set of statements repeatedly until a particular condition is satisfied.

Types of Loop

There are 3 types of Loop in C language, namely:

1. while loop
2. for loop
3. do while loop

while loop

while loop can be addressed as an **entry control** loop. It is completed in 3 steps.

- Variable initialization.(e.g int x = 0;)
- condition(e.g while(x <= 10))
- Variable increment or decrement (x++ or x-- or x = x + 2)

Syntax :

```
variable initialization;
while(condition)
{
   statements;
   variable increment or decrement;
}
```

for loop

for loop is used to execute a set of statements repeatedly until a particular condition is satisfied. We can say it is an **open ended loop.**. General format is,

```
for(initialization; condition; increment/decrement)
{
   statement-block;
}
```

In for loop we have exactly two semicolons, one after initialization and second after the condition. In this loop we can have more than one initialization or increment/decrement, separated using comma operator. But it can have only one **condition**.

The for loop is executed as follows:

1. It first evaluates the initialization code.
2. Then it checks the condition expression.
3. If it is **true**, it executes the for-loop body.
4. Then it evaluate the increment/decrement condition and again follows from step 2.
5. When the condition expression becomes **false**, it exits the loop.

Example: Program to print first 10 natural numbers

```
#include<stdio.h>

void main( )
{
   int x;
   for(x = 1; x <= 10; x++)
   {
      printf("%d\t", x);
   }
}
```

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

Nested for loop

We can also have nested for loops, i.e one for loop inside another for loop. Basic syntax is,

```
for(initialization; condition; increment/decrement)
{
   for(initialization; condition; increment/decrement)
   {
      statement ;
   }
}
```

do while loop

In some situations it is necessary to execute body of the loop before testing the condition. Such situations can be handled with the help of do-while loop. do statement evaluates the body of the loop first and at the end, the condition is checked using while statement. It means that the body of the loop will be executed at least once, even though the starting condition inside while is initialized to be **false**. General syntax is,

```
do
{
   .....
   .....
}
while(condition)
```

Example: Program to print first 10 multiples of 5.

```
#include<stdio.h>

void main()
{
   int a, i;
   a = 5;
   i = 1;
   do
   {
      printf("%d\t", a*i);
      i++;
   }
   while(i <= 10);
}
```

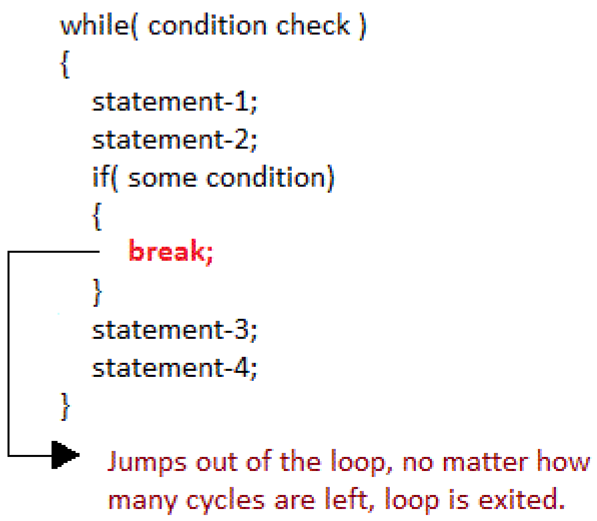
```
5 10 15 20 25 30 35 40 45 50
```

Jumping Out of Loops

Sometimes, while executing a loop, it becomes necessary to skip a part of the loop or to leave the loop as soon as certain condition becomes **true**. This is known as jumping out of loop.

1) break statement

When break statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop.

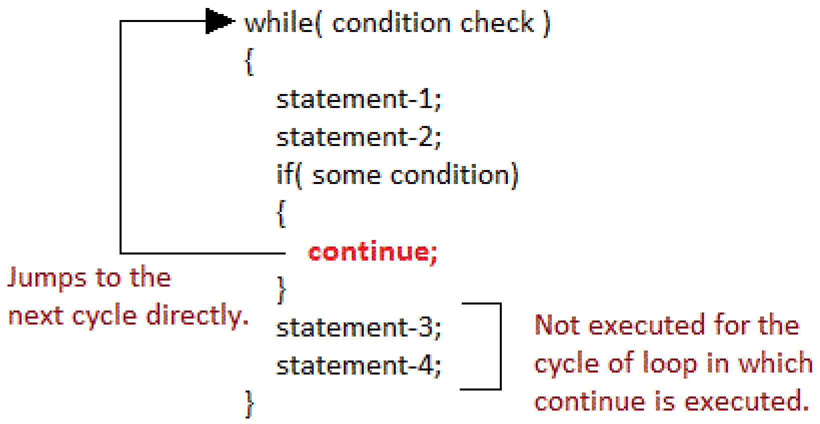


```
while( condition check )
{
   statement-1;
   statement-2;
   if( some condition)
   {
      break;
   }
   statement-3;
   statement-4;
}
```

Jumps out of the loop, no matter how many cycles are left, loop is exited.

2) continue statement

It causes the control to go directly to the test-condition and then continue the loop process. On encountering continue, cursor leave the current cycle of loop, and starts with the next cycle.



```
while( condition check )
{
   statement-1;
   statement-2;
   if( some condition)
   {
      continue;
   }
   statement-3;
   statement-4;
}
```

Jumps to the next cycle directly.
Not executed for the cycle of loop in which continue is executed.

UNIT – III

ARRAYS

Arrays:

In C language, arrays are reffered to as structured data types. An array is defined as **finite ordered collection of homogenous** data, stored in contiguous memory locations.

Here the words,

- **finite** *means* data range must be defined.
- **ordered** *means* data must be stored in continuous memory addresses.
- **homogenous** *means* data must be of similar data type.

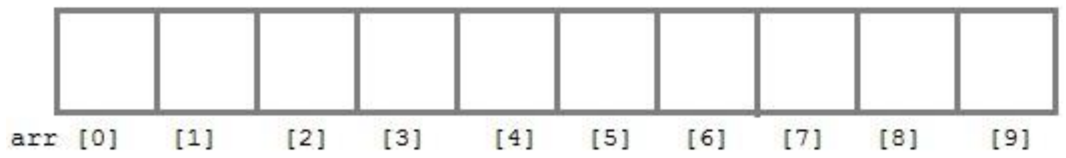
Example where arrays are used,

- to store list of Employee or Student names,
- to store marks of students,
- or to store list of numbers or characters etc.

Declaring an Array

Like any other variable, arrays must be declared before they are used. General form of array declaration is,

```
data-type variable-name[size];

/* Example of array declaration */

int arr[10];
```



Here int is the data type, arr is the name of the array and 10 is the size of array. It means array arr can only contain 10 elements of int type.

Index of an array starts from 0 to **size-1** i.e first element of arr array will be stored at arr[0] address and the last element will occupy arr[9].

Initialization of an Array

After an array is declared it must be initialized. Otherwise, it will contain **garbage** value(any random value). An array can be initialized at either **compile time** or at **runtime**.

Compile time Array initialization

Compile time initialization of array elements is same as ordinary variable initialization. The general form of initialization of array is,

```
data-type array-name[size] = { list of values };

/* Here are a few examples */
int marks[4]={ 67, 87, 56, 77 };    // integer array initialization

float area[5]={ 23.4, 6.8, 5.5 };   // float array initialization

int marks[4]={ 67, 87, 56, 77, 59 };    // Compile time error
```

One important thing to remember is that when you will give more initializer(array elements) than the declared array size than the **compiler** will give an error.

```
#include<stdio.h>

void main()
{
   int i;
   int arr[] = {2, 3, 4};      // Compile time array initialization
   for(i = 0 ; i < 3 ; i++)
   {
      printf("%d\t",arr[i]);
   }
}
```

```
2 3 4
```

Runtime Array initialization

An array can also be initialized at runtime using scanf() function. This approach is usually used for initializing large arrays, or to initialize arrays with user specified values. Example,

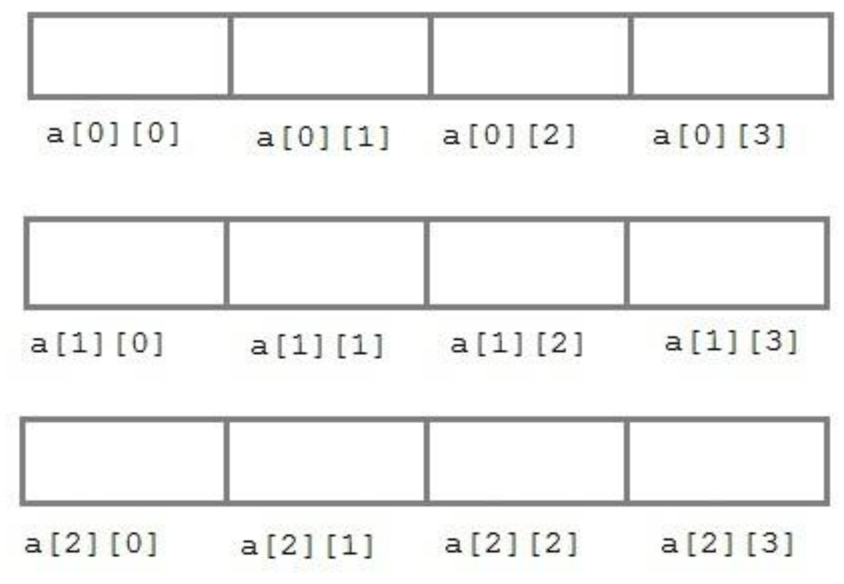
```
#include<stdio.h>

void main()
{
   int arr[4];
   int i, j;
   printf("Enter array element");
   for(i = 0; i < 4; i++)
   {
      scanf("%d", &arr[i]);    //Run time array initialization
   }
   for(j = 0; j < 4; j++)
   {
      printf("%d\n", arr[j]);
   }
}
```

<u>**Two dimensional Arrays**</u>

C language supports multidimensional arrays also. The simplest form of a multidimensional array is the two-dimensional array. Both the row's and column's index begins from 0.

Two-dimensional arrays are declared as follows,

```
data-type array-name[row-size][column-size]

/* Example */
int a[3][4];
```



An array can also be declared and initialized together. For example,

```
int arr[][3] = {
   {0,0,0},
   {1,1,1}
};
```

Note: We have not assigned any row value to our array in the above example. It means we can initialize any number of rows. But, we must always specify number of columns, else it will give a compile time error. Here, a 2*3 multi-dimensional matrix is created.

Runtime initialization of a two dimensional Array

```
#include<stdio.h>

void main()
{
   int arr[3][4];
   int i, j, k;
   printf("Enter array element");
   for(i = 0; i < 3;i++)
   {
      for(j = 0; j < 4; j++)
      {
         scanf("%d", &arr[i][j]);
      }
   }
   for(i = 0; i < 3; i++)
   {
      for(j = 0; j < 4; j++)
      {
         printf("%d", arr[i][j]);
      }
   }
}
```

String and Character Array

String is a sequence of characters that is treated as a single data item and terminated by null character '\0'. Remember that C language does not support strings as a data type. A **string** is actually one-dimensional array of characters in C language. These are often used to create meaningful and readable programs.

For example: The string "hello world" contains 12 characters including '\0' character which is automatically added by the compiler at the end of the string.

Declaring and Initializing a string variables:

There are different ways to initialize a character array variable.

```
char name[13] = "StudyTonight";       // valid character array initialization
```

```
char name[10] = {'L','e','s','s','o','n','s','\0'};     // valid initialization
```

Remember that when you initialize a character array by listing all of its characters separately then you must supply the '\0' character explicitly.

Some examples of illegal initialization of character array are,

```
char ch[3] = "hell";    // Illegal
```

```
char str[4];
str = "hell";   // Illegal
```

String Input and Output:

Input function scanf() can be used with **%s** format specifier to read a string input from the terminal. But there is one problem with scanf() function, it terminates its input on the first white space it encounters. Therefore if you try to read an input string "Hello World" using scanf() function, it will only read **Hello** and terminate after encountering white spaces.

However, C supports a format specification known as the **edit set conversion code %[..]** that can be used to read a line containing a variety of characters, including white spaces.

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

void main()
{
   char str[20];
   printf("Enter a string");
   scanf("%[^\n]", &str);  //scanning the whole string, including the white spaces
   printf("%s", str);
}
```

Another method to read character string with white spaces from terminal is by using the gets() function.

```
char text[20];
gets(text);
printf("%s", text);
```

String Handling Functions:

C language supports a large number of string handling functions that can be used to carry out many of the string manipulations. These functions are packaged in **string.h** library. Hence, you must include **string.h** header file in your programs to use these functions.

The following are the most commonly used string handling functions.

Method	Description
strcat()	It is used to concatenate(combine) two strings
strlen()	It is used to show length of a string
strrev()	It is used to show reverse of a string
strcpy()	Copies one string into another
strcmp()	It is used to compare two string

strcat() function

strcat("hello", "world");
strcat() function will add the string **"world"** to **"hello"** i.e it will ouput helloworld.

strlen() function

strlen() function will return the length of the string passed to it.

```
int j;
```

```
j = strlen("studytonight");
printf("%d",j);
```

```
12
```

strcmp() function

strcmp() function will return the ASCII difference between first unmatching character of two strings.

```
int j;
j = strcmp("study", "tonight");
printf("%d",j);
```

```
-1
```

strcpy() function

It copies the second string argument to the first string argument.

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

int main()
{
   char s1[50];
   char s2[50];

   strcpy(s1, "StudyTonight");     //copies "studytonight" to string s1
   strcpy(s2, s1);     //copies string s1 to string s2

   printf("%s\n", s2);

   return(0);
}
```

```
StudyTonight
```

strrev() function

It is used to reverse the given string expression.

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

```
int main()
{
   char s1[50];

   printf("Enter your string: ");
   gets(s1);
   printf("\nYour reverse string is: %s",strrev(s1));
   return(0);
}
```

```
Enter your string: studytonight Your reverse string is: thginotyduts
```

Functions

A **function** is a block of code that performs a particular task.

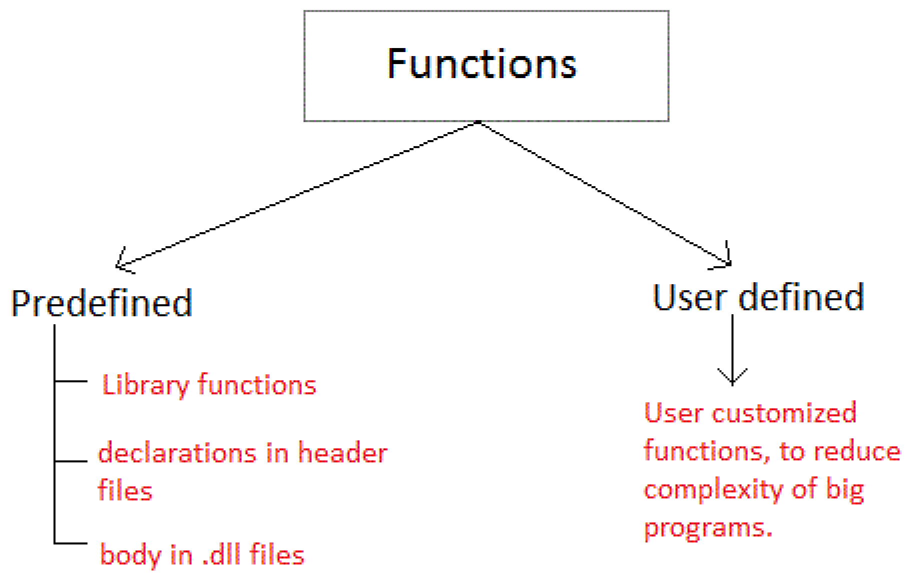
There are many situations where we might need to write same line of code for more than once in a program.

C language provides an approach in which you can declare and define a group of statements once in the form of a function and it can be called and used whenever required.

These functions defined by the user are also know as **User-defined Functions**

C functions can be classified into two categories,

1. **Library functions**
2. **User-defined functions**


Functions
Predefined
User defined
Library functions
declarations in header files
body in .dll files
User customized functions, to reduce complexity of big programs.

Library functions are those functions which are already defined in C library, example printf(), scanf(), strcat() etc. You just need to include appropriate header files to use these functions. These are already declared and defined in C libraries.

A **User-defined functions** on the other hand, are those functions which are defined by the user at the time of writing program. These functions are made for code reusability and for saving time and space.

Benefits of Using Functions

1. It provides modularity to your program's structure.
2. It makes your code reusable. You just have to call the function by its name to use it, wherever required.
3. In case of large programs with thousands of code lines, debugging and editing becomes easier if you use functions.
4. It makes the program more readable and easy to understand.

Function Declaration

General syntax for function declaration is,

returntype functionName(type1 parameter1, type2 parameter2,...);

Like any variable or an array, a function must also be declared before its used. Function declaration informs the compiler about the function name, parameters is accept, and its return type. The actual body of the function can be defined separately. It's also called as **Function Prototyping**. Function declaration consists of 4 parts.

- returntype

- function name
- parameter list
- terminating semicolon

returntype :

When a function is declared to perform some sort of calculation or any operation and is expected to provide with some result at the end, in such cases, a return statement is added at the end of function body. Return type specifies the type of value(int, float, char, double) that function is expected to return to the program which called the function.

Note: In case your function doesn't return any value, the return type would be void.

functionName

Function name is an identifier and it specifies the name of the function. The function name is any valid C identifier and therefore must follow the same naming rules like other variables in C language.

parameter list

The parameter list declares the type and number of arguments that the function expects when it is called. Also, the parameters in the parameter list receives the argument values when the function is called. They are often referred as **formal parameters**.

Function definition Syntax:

the general syntax of function definition is,

```
returntype functionName(type1 parameter1, type2 parameter2,...)
{
   // function body goes here
}
```

The first line *returntype* **functionName(type1 parameter1, type2 parameter2,...)** is known as **function header** and the statement(s) within curly braces is called **function body**.

functionbody

The function body contains the declarations and the statements(algorithm) necessary for performing the required task. The body is enclosed within curly braces { ... } and consists of three parts.

- **local** variable declaration(if required).
- **function statements** to perform the task inside the function.
- a **return** statement to return the result evaluated by the function(if return type is void, then no return statement is required).

Calling a function

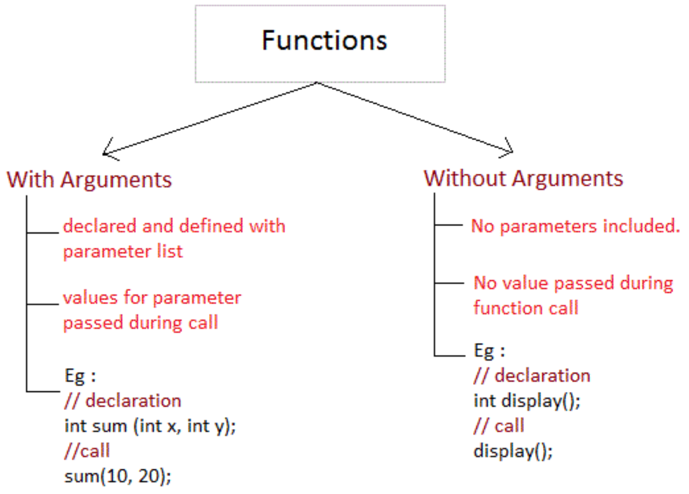
When a function is called, control of the program gets transferred to the function.

functionName(argument1, argument2,...);

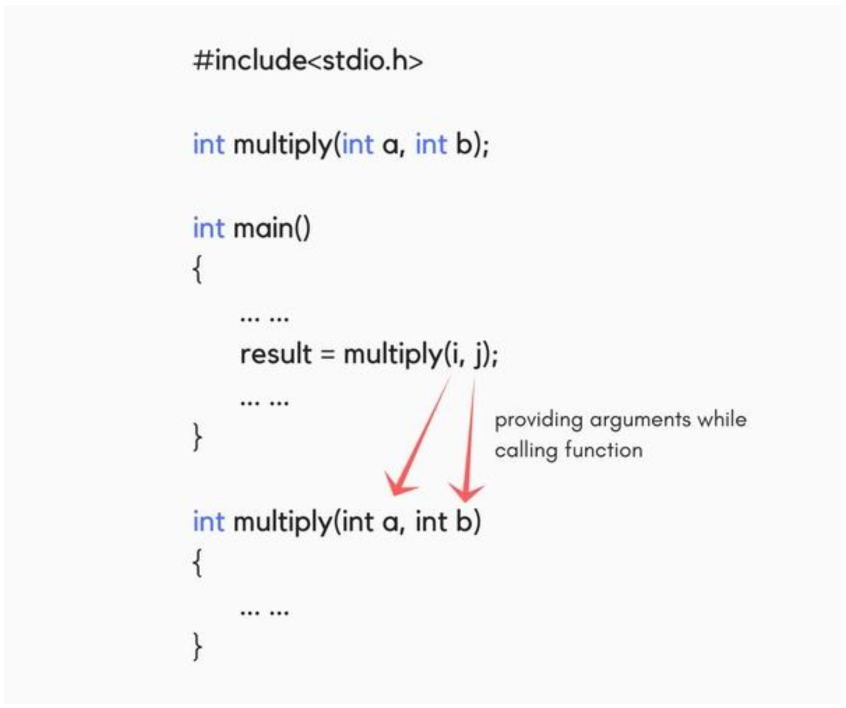
In the example above, the statement multiply(i, j); inside the main() function is function call.

Passing Arguments to a function

Arguments are the values specified during the function call, for which the formal parameters are declared while defining the function.



It is possible to have a function with parameters but no return type. It is not necessary, that if a function accepts parameter(s), it must return a result too.

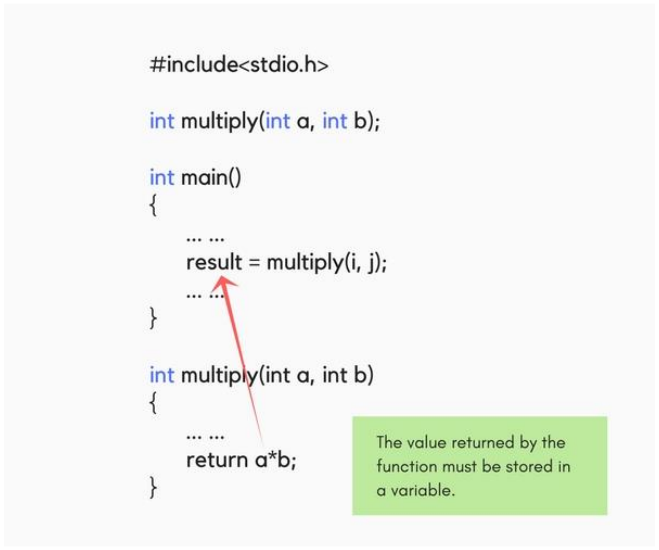


```
#include<stdio.h>

int multiply(int a, int b);

int main()
{
    ... ...
    result = multiply(i, j);
    ... ...
}

int multiply(int a, int b)
{
    ... ...
}
```

While declaring the function, we have declared two parameters a and b of type int. Therefore, while calling that function, we need to pass two arguments, else we will get compilation error. And the two arguments passed should be received in the function definition, which means that the function header in the function definition should have the two parameters to hold the argument values. These received arguments are also known as **formal parameters**. The name of the variables while declaring, calling and defining a function can be different.

Returning a value from function

A function may or may not return a result. But if it does, we must use the return statement to output the result. return statement also ends the function execution, hence it must be the last statement of any function. If you write any statement after the return statement, it won't be executed.



```
#include<stdio.h>

int multiply(int a, int b);

int main()
{
    ... ...
    result = multiply(i, j);
    ... ...
}

int multiply(int a, int b)
{
    ... ...
    return a*b;
}
```

The value returned by the function must be stored in a variable.

The datatype of the value returned using the return statement should be same as the return type mentioned at function declaration and definition. If any of it mismatches, you will get compilation error.

In the next tutorial, we will learn about the different types of user defined functions in C language and the concept of Nesting of functions which is used in recursion.

Type of User-defined Functions :

There can be 4 different types of user-defined functions, they are:

1. Function with no arguments and no return value
2. Function with no arguments and a return value
3. Function with arguments and no return value
4. Function with arguments and a return value

Below, we will discuss about all these types, along with program examples.

1.Function with no arguments and no return value

Such functions can either be used to display information or they are completely dependent on user inputs.

Below is an example of a function, which takes 2 numbers as input from user, and display which is the greater number.

```
#include<stdio.h>

void greatNum();      // function declaration

int main()
{
   greatNum();       // function call
   return 0;
}

void greatNum()       // function definition
{
   int i, j;
   printf("Enter 2 numbers that you want to compare...");
   scanf("%d%d", &i, &j);
   if(i > j) {
      printf("The greater number is: %d", i);
   }
   else {
      printf("The greater number is: %d", j);
   }
}
```

2.Function with no arguments and a return value

We have modified the above example to make the function greatNum() return the number which is greater amongst the 2 input numbers.

```
#include<stdio.h>

int greatNum();       // function declaration

int main()
{
   int result;
   result = greatNum();        // function call
   printf("The greater number is: %d", result);
   return 0;
}

int greatNum()       // function definition
{
   int i, j, greaterNum;
   printf("Enter 2 numbers that you want to compare...");
   scanf("%d%d", &i, &j);
   if(i > j) {
      greaterNum = i;
   }
   else {
      greaterNum = j;
   }
   // returning the result
   return greaterNum;
}
```

3.Function with arguments and no return value

We are using the same function as example again and again, to demonstrate that to solve a problem there can be many different ways.

This time, we have modified the above example to make the function greatNum() take two int values as arguments, but it will not be returning anything.

```
#include<stdio.h>

void greatNum(int a, int b);       // function declaration

int main()
```

```
{
   int i, j;
   printf("Enter 2 numbers that you want to compare...");
   scanf("%d%d", &i, &j);
   greatNum(i, j);       // function call
   return 0;
}

void greatNum(int x, int y)       // function definition
{
   if(x > y) {
      printf("The greater number is: %d", x);
   }
   else {
      printf("The greater number is: %d", y);
   }
}
```

4.Function with arguments and a return value

This is the best type, as this makes the function completely independent of inputs and outputs, and only the logic is defined inside the function body.

```
#include<stdio.h>

int greatNum(int a, int b);      // function declaration

int main()
{
   int i, j, result;
   printf("Enter 2 numbers that you want to compare...");
   scanf("%d%d", &i, &j);
   result = greatNum(i, j); // function call
   printf("The greater number is: %d", result);
   return 0;
}

int greatNum(int x, int y)       // function definition
{
   if(x > y) {
      return x;
   }
   else {
      return y;
   }
}
```

Nesting of Functions

C language also allows nesting of functions i.e to use/call one function inside another function's body. We must be careful while using nested functions, because it may lead to infinite nesting.

```
function1()
{
   // function1 body here

   function2();

   // function1 body here
}
```

If function2() also has a call for function1() inside it, then in that case, it will lead to an infinite nesting. They will keep calling each other and the program will never terminate.

Not able to understand? Lets consider that inside the main() function, function1() is called and its execution starts, then inside function1(), we have a call for function2(), so the control of program will go to the function2(). But as function2() also has a call to function1() in its body, it will call function1(), which will again call function2(), and this will go on for infinite times, until you forcefully exit from program execution.

Recursion

Recursion is a special way of nesting functions, where a function calls itself inside it. We must have certain conditions in the function to break out of the recursion, otherwise recursion will occur infinite times.

```
function1()
{
   // function1 body
   function1();
   // function1 body
}
```

Example: Factorial of a number using Recursion

```
#include<stdio.h>

int factorial(int x);       //declaring the function

void main()
{
   int a, b;

   printf("Enter a number...");
   scanf("%d", &a);
   b = factorial(a);       //calling the function named factorial
   printf("%d", b);
}

int factorial(int x) //defining the function
{
   int r = 1;
   if(x == 1)
      return 1;
   else
      r = x*factorial(x-1);       //recursion, since the function calls itself

   return r;
}
```

Similarly, there are many more applications of recursion in C language. Go to the programs section, to find out more programs using recursion.

Types of Function calls:

Functions are called by their names, we all know that, then what is this tutorial for? Well if the function does not have any arguments, then to call a function you can directly use its name. But for functions with arguments, we can call a function in two different ways, based on how we specify the arguments, and these two ways are:

1. Call by Value
2. Call by Reference

1.Call by Value

Calling a function by value means, we pass the values of the arguments which are stored or copied into the formal parameters of the function. Hence, the original values are unchanged only the parameters inside the function changes.

```
#include<stdio.h>

void calc(int x);

int main()
{
   int x = 10;
   calc(x);
   // this will print the value of 'x'
   printf("\nvalue of x in main is %d", x);
   return 0;
}
```

```
void calc(int x)
{
    // changing the value of 'x'
    x = x + 10 ;
    printf("value of x in calc function is %d ", x);
}
```

```
value of x in calc function is 20 value of x in main is 10
```

2.Call by Reference

In call by reference we pass the address(reference) of a variable as argument to any function. When we pass the address of any variable as argument, then the function will have access to our variable, as it now knows where it is stored and hence can easily update its value.

In this case the formal parameter can be taken as a **reference** or a **pointer**(don't worry about pointers, we will soon learn about them), in both the cases they will change the values of the original variable.

```
#include<stdio.h>

void calc(int *p);      // functin taking pointer as argument

int main()
{
    int x = 10;
    calc(&x);       // passing address of 'x' as argument
    printf("value of x is %d", x);
    return(0);
}

void calc(int *p)       //receiving the address in a reference pointer variable
{
    /*
        changing the value directly that is
        stored at the address passed
    */
    *p = *p + 10;
}
```

```
value of x is 20
```

UNIT – IV

STRUCTURES

UNIT IV

C Structures

Structure is a user-defined datatype in C language which allows us to combine data of different types together. Structure helps to construct a complex data type which is more meaningful. It is somewhat similar to an Array, but an array holds data of similar type only. But structure on the other hand, can store data of any type, which is practical more useful.

Defining a structure

struct keyword is used to define a structure. struct defines a new data type which is a collection of primary and derived datatypes.

Syntax:

```
struct [structure_tag]
{
   //member variable 1
   //member variable 2
   //member variable 3
   ...
}[structure_variables];
```

we start with the struct keyword, then it's optional to provide your structure a name, we suggest you to give it a name, then inside the curly braces, we have to mention all the member variables, which are nothing but normal C language variables of different types like int, float, array etc.

After the closing curly brace, we can specify one or more structure variables, again this is optional.

Note: The closing curly brace in the structure type declaration must be followed by a semicolon(;).

Example of Structure

```
struct Student
{
   char name[25];
   int age;
   char branch[10];
   // F for female and M for male
   char gender;
};
```

Here struct Student declares a structure to hold the details of a student which consists of 4 data fields, namely name, age, branch and gender. These fields are called **structure elements or members**.

Each member can have different datatype, like in this case, name is an array of char type and age is of int type etc. **Student** is the name of the structure and is called as the **structure tag**.

Structure ***variable declaration is similar to the declaration of any normal variable of any other datatype. Structure variables can be declared in following two ways: 1) Declaring Structure variables separately***

```
struct Student
{
   char name[25];
   int age;
   char branch[10];
   //F for female and M for male
   char gender;
};

struct Student S1, S2;      //declaring variables of struct Student
```

2) Declaring Structure variables with structure definition

```
struct Student
{
   char name[25];
   int age;
   char branch[10];
   //F for female and M for male
   char gender;
}S1, S2;
```

Structure Initialization

Like a variable of any other datatype, structure variable can also be initialized at compile time.

```
struct Patient
{
   float height;
   int weight;
   int age;
};

struct Patient p1 = { 180.75 , 73, 23 };
```

Nested Structures

Nesting of structures, is also permitted in C language. Nested structures means, that one structure has another stucture as member variable.

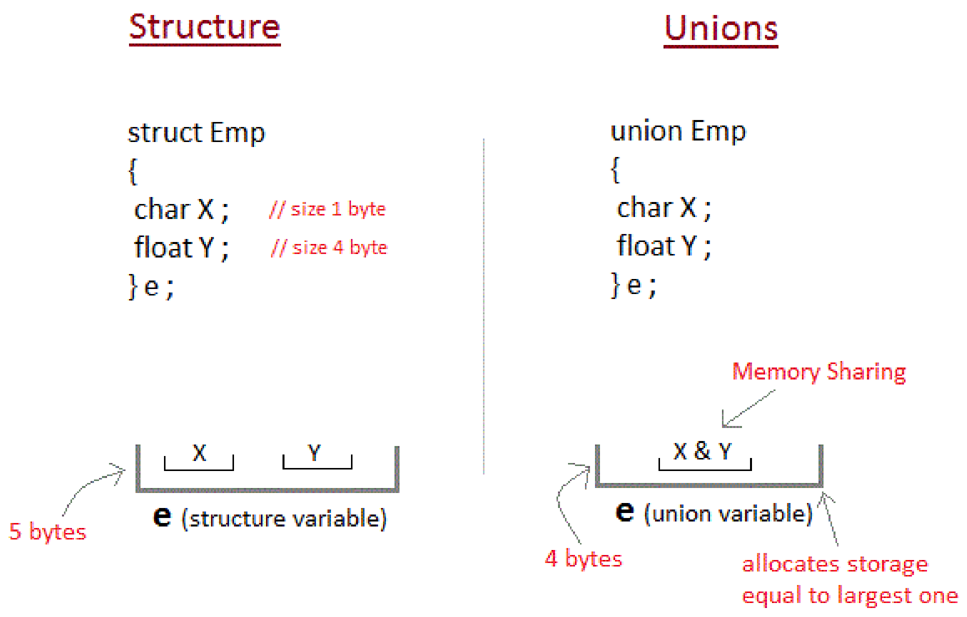
Example:

```
struct Student
{
   char[30] name;
```

```
    int age;
    /* here Address is a structure */
    struct Address
    {
        char[50] locality;
        char[50] city;
        int pincode;
    }addr;
};
```

Unions in C

Unions are conceptually similar to **structures**. The syntax to declare/define a union is also similar to that of a structure. The only differences is in terms of storage. In **structure** each member has its own storage location, whereas all members of **union** uses a single shared memory location which is equal to the size of its largest data member.



This implies that although a **union** may contain many members of different types, **it cannot handle all the members at the same time**. A **union** is declared using the union keyword.

```
union item
{
    int m;
    float x;
    char c;
}It1;
```

This union contains three members each with a different data type. However only one of them can be used at a time. This is due to the fact that only one location is allocated for all the union variables, irrespective of their size.

Pointers

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

Pointers are used to access memory and manipulate the address.

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

Address in C

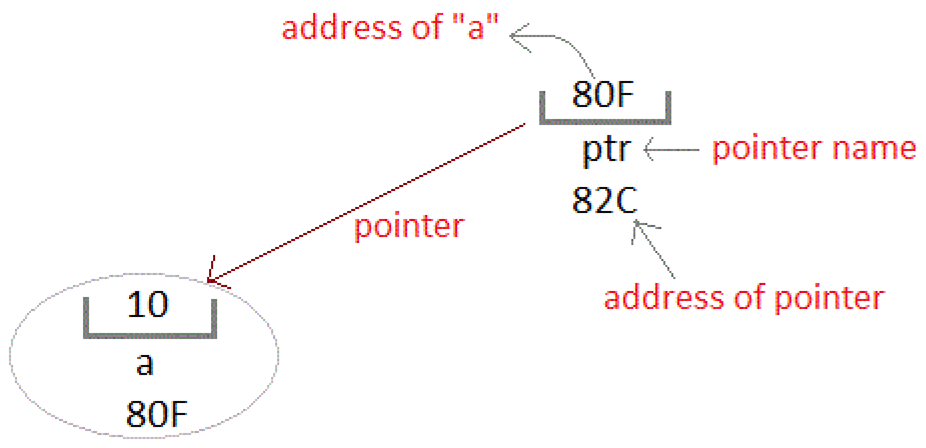
Whenever a variable is defined in C language, a memory location is assigned for it, in which it's value will be stored. We can easily check this memory address, using the & symbol.

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

```
#include<stdio.h>

void main()
{
   int var = 7;
   printf("Value of the variable var is: %d\n", var);
   printf("Memory address of the variable var is: %x\n", &var);
}
```

The variables which are used to hold memory addresses are called **Pointer variables**.

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



Benefits of using pointers

Below we have listed a few benefits of using pointers:

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

Declaring, Initializing and using a pointer variable in C

1. While declaring/initializing the pointer variable, * indicates that the variable is a pointer.
2. The address of any variable is given by preceding the variable name with Ampersand &.
3. The pointer variable stores the address of a variable. The declaration int *a doesn't mean that a is going to contain an integer value. It means that a is going to contain the address of a variable storing integer value.
4. To access the value of a certain address stored by a pointer variable, * is used. Here, the * can be read as **'value at'**.

SYNTAX

```
datatype *pointer_name;
```

EXAMPLE

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

Initialization of C Pointer variable

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

```
#include<stdio.h>

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

Pointer to a Pointer in C(Double Pointer)

Pointers are used to store the address of other variables of similar datatype. But if you want to store the address of a pointer variable, then you again need a pointer to store it. Thus, when one pointer variable stores the address of another pointer variable, it is known as **Pointer to Pointer** variable or **Double Pointer**.

Syntax:

```
int **p1;
```

Here, we have used two indirection operator(*) which stores and points to the address of a pointer variable i.e, int *. If we want to store the address of this (double pointer) variable p1, then the syntax would become:

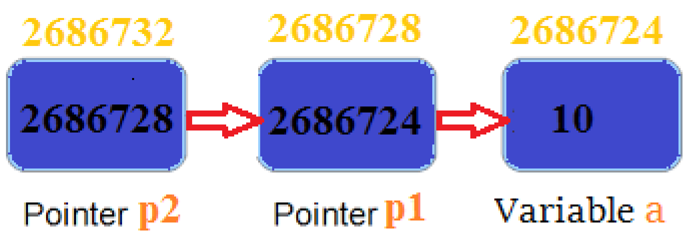
```
int ***p2
```

Simple program to represent Pointer to a Pointer

```
#include <stdio.h>

int main() {

   int  a = 10;
   int  *p1;      //this can store the address of variable a
   int  **p2;
   /*
      this can store the address of pointer variable p1 only.
      It cannot store the address of variable 'a'
   */
   p1 = &a;
   p2 = &p1;

   printf("Address of a = %u\n", &a);
   printf("Address of p1 = %u\n", &p1);
   printf("Address of p2 = %u\n\n", &p2);

   // below print statement will give the address of 'a'
   printf("Value at the address stored by p2 = %u\n", *p2);

   printf("Value at the address stored by p1 = %d\n\n", *p1);

   printf("Value of **p2 = %d\n", **p2); //read this *(*p2)

   /*
      This is not allowed, it will give a compile time error-
      p2 = &a;
      printf("%u", p2);
   */
   return 0;
}
```

```
Address of a = 2686724 Address of p1 = 2686728 Address of p2 = 2686732 Value at the address stored by p2 = 2686724 Value at the address stored by p1 = 10 Value of **p2 = 10
```

Explanation of the above program



- p1 pointer variable can only hold the address of the variable a (i.e Number of indirection operator(*)-1 variable). Similarly, p2 variable can only hold the address of variable p1. It cannot hold the address of variable a.
- *p2 gives us the value at an address stored by the p2 pointer. p2 stores the address of p1 pointer and value at the address of p1 is the address of variable a. Thus, *p2 prints address of a.

- **p2 can be read as *(*p2). Hence, it gives us the value stored at the address *p2. From above statement, you know *p2 means the address of variable a. Hence, the value at the address *p2 is 10. Thus, **p2 prints 10.

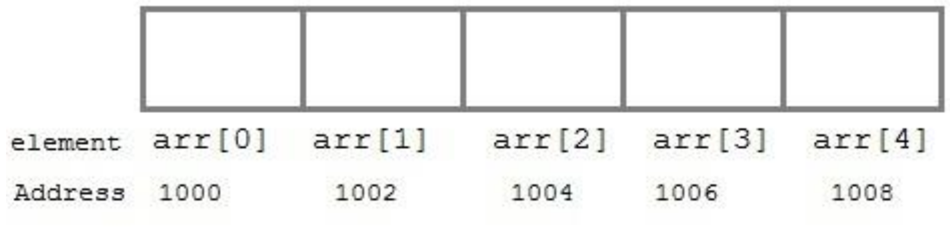
Pointer and Arrays in C

When an array is declared, compiler allocates sufficient amount of memory to contain all the elements of the array. Base address i.e address of the first element of the array is also allocated by the compiler.

Suppose we declare an array arr,

int arr[5] = { 1, 2, 3, 4, 5 };

Assuming that the base address of arr is 1000 and each integer requires two bytes, the five elements will be stored as follows:



Here variable arr will give the base address, which is a constant pointer pointing to the first element of the array, arr[0]. Hence arr contains the address of arr[0] i.e 1000. In short, arr has two purpose - it is the name of the array and it acts as a pointer pointing towards the first element in the array.

arr is equal to &arr[0] by default

We can also declare a pointer of type int to point to the array arr.

```
int *p;
p = arr;
// or,
p = &arr[0];    //both the statements are equivalent.
```

Now we can access every element of the array arr using p++ to move from one element to another.

NOTE: You cannot decrement a pointer once incremented. p-- won't work.

1.Pointer to Array

As studied above, we can use a pointer to point to an array, and then we can use that pointer to access the array elements. Lets have an example,

```
#include <stdio.h>

int main()
{
   int i;
   int a[5] = {1, 2, 3, 4, 5};
   int *p = a;     // same as int*p = &a[0]
   for (i = 0; i < 5; i++)
   {
      printf("%d", *p);
      p++;
   }

   return 0;
}
```

In the above program, the pointer *p will print all the values stored in the array one by one. We can also use the Base address (a in above case) to act as a pointer and print all the values.

he generalized form for using pointer with an array,

```
*(a+i)
```

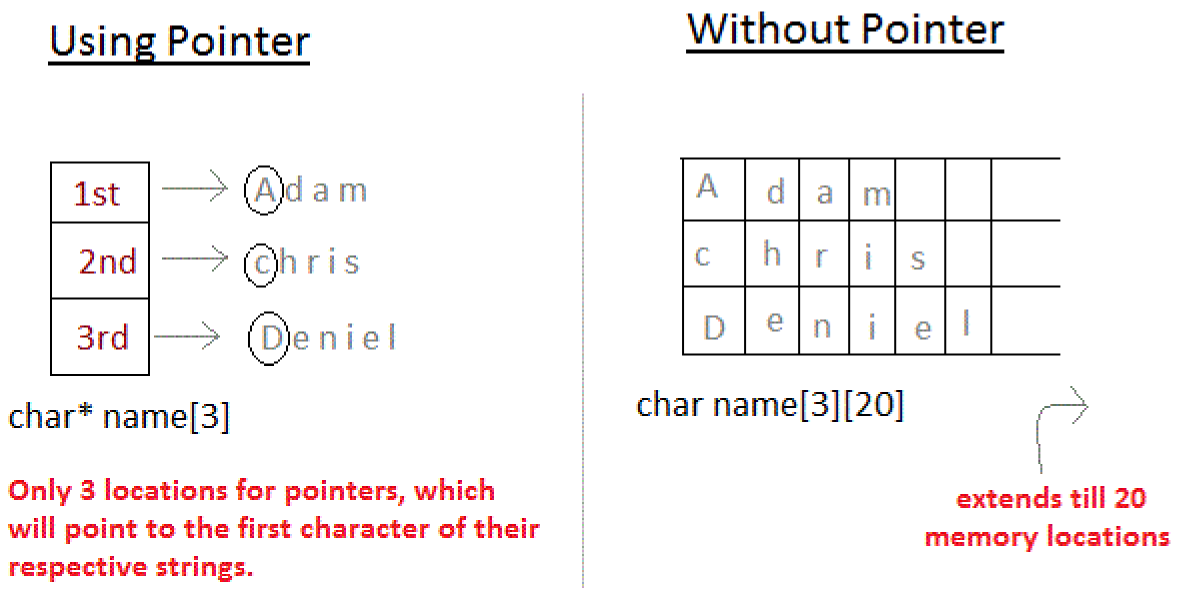
is same as:

```
a[i]
```

2.Array of Pointers

We can also have array of pointers. Pointers are very helpful in handling character array with rows of varying length.

```
char *name[3] = {
   "Adam",
   "chris",
   "Deniel"
};
//Now lets see same array without using pointer
char name[3][20] = {
   "Adam",
   "chris",
    "Deniel"
};
```


Using Pointer
1st
2nd
3rd
Adam
Chris
Deniel
char* name[3]
Only 3 locations for pointers, which will point to the first character of their respective strings.
Without Pointer
A d a m
c h r i s
D e n i e l
char name[3][20]
extends till 20 memory locations

In the second approach memory wastage is more, hence it is prefered to use pointer in such cases.

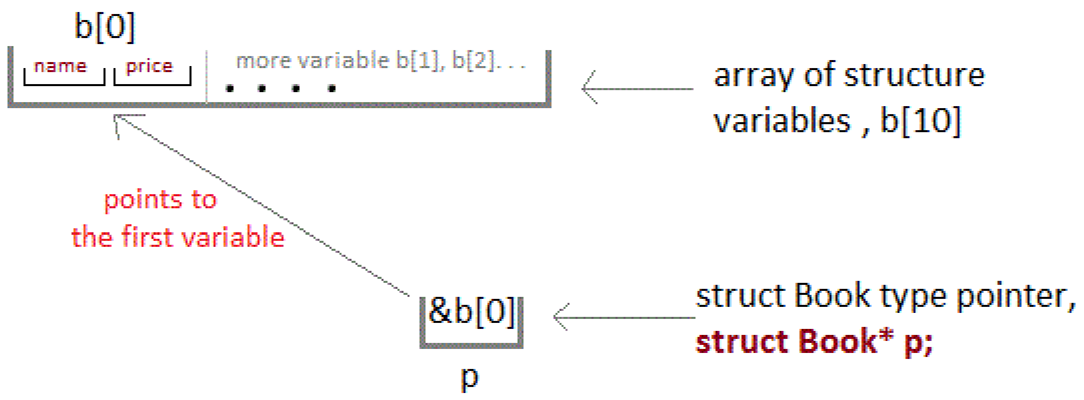
When we say memory wastage, it doesn't means that the strings will start occupying less space, no, characters will take the same space, but when we define array of characters, a contiguos memory space is located equal to the maximum size of the array, which is a wastage, which can be avoided if we use pointers instead.

Pointer to Array of Structures in C

Like we have array of integers, array of pointers etc, we can also have array of structure variables. And to use the array of structure variables efficiently, we use **pointers of structure type**. We can also have pointer to a single structure variable, but it is mostly used when we are dealing with array of structure variables.

```
#include <stdio.h>

struct Book
{
    char name[10];
    int price;
}
```

```
int main()
{
   struct Book a;      //Single structure variable
   struct Book* ptr;   //Pointer of Structure type
   ptr = &a;

   struct Book b[10];  //Array of structure variables
   struct Book* p;     //Pointer of Structure type
   p = &b;

   return 0;
}
```


b[0]
name
price
more variable b[1], b[2]. . .
array of structure variables , b[10]
points to the first variable
&b[0]
p
struct Book type pointer,
struct Book* p;

Accessing Structure Members with Pointer

To access members of structure using the structure variable, we used the dot . operator. But when we have a pointer of structure type, we use arrow -> to access structure members.

```
#include <stdio.h>

struct my_structure {
   char name[20];
   int number;
   int rank;
};

int main()
{
   struct my_structure variable = {"StudyTonight", 35, 1};

   struct my_structure *ptr;
```

```
    ptr = &variable;

    printf("NAME: %s\n", ptr->name);
    printf("NUMBER: %d\n", ptr->number);
    printf("RANK: %d", ptr->rank);

    return 0;
}
```

```
NAME: StudyTonight NUMBER: 35 RANK: 1
```

Pointers as Function Argument in C

Pointer as a function parameter is used to hold addresses of arguments passed during function call. This is also known as **call by reference**. When a function is called by reference any change made to the reference variable will effect the original variable.

Example Time: Swapping two numbers using Pointer

```
#include <stdio.h>

void swap(int *a, int *b);

int main()
{
    int m = 10, n = 20;
    printf("m = %d\n", m);
    printf("n = %d\n\n", n);

    swap(&m, &n);    //passing address of m and n to the swap function
    printf("After Swapping:\n\n");
    printf("m = %d\n", m);
    printf("n = %d", n);
    return 0;
}

/*
    pointer 'a' and 'b' holds and
    points to the address of 'm' and 'n'
*/
void swap(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

```
m = 10 n = 20 After Swapping: m = 20 n = 10
```

1.Functions returning Pointer variables

A function can also return a pointer to the calling function. In this case you must be careful, because local variables of function doesn't live outside the function. They have scope only inside the function. Hence if you return a pointer connected to a local variable, that pointer will be pointing to nothing when the function ends.

```
#include <stdio.h>

int* larger(int*, int*);

void main()
{
   int a = 15;
   int b = 92;
   int *p;
   p = larger(&a, &b);
   printf("%d is larger",*p);
}

int* larger(int *x, int *y)
{
   if(*x > *y)
      return x;
   else
      return y;
}
```

```
92 is larger
```

2.Pointer to functions

It is possible to declare a pointer pointing to a function which can then be used as an argument in another function. A pointer to a function is declared as follows,

```
type (*pointer-name)(parameter);
```

Here is an example :

```
int (*sum)();   //legal declaration of pointer to function
int *sum();     //This is not a declaration of pointer to function
```

A function pointer can point to a specific function when it is assigned the name of that function.

```
int sum(int, int);
int (*s)(int, int);
s = sum;
```

Here s is a pointer to a function sum. Now sum can be called using function pointer s along with providing the required argument values.

```
s (10, 20);
```

Example of Pointer to Function

```
#include <stdio.h>

int sum(int x, int y)
{
   return x+y;
}

int main( )
{
   int (*fp)(int, int);
   fp = sum;
   int s = fp(10, 15);
   printf("Sum is %d", s);

   return 0;
}
```

```
25
```

File Input/Output

A **file** represents a sequence of bytes on the disk where a group of related data is stored. File is created for permanent storage of data. It is a ready made structure.

In C language, we use a structure **pointer of file type** to declare a file.

```
FILE *fp;
```

C provides a number of functions that helps to perform basic file operations. Following are the functions,

Function	description
fopen()	create a new file or open a existing file
fclose()	closes a file
getc()	reads a character from a file
putc()	writes a character to a file
fscanf()	reads a set of data from a file
fprintf()	writes a set of data to a file
getw()	reads a integer from a file
putw()	writes a integer to a file
fseek()	set the position to desire point
ftell()	gives current position in the file
rewind()	set the position to the begining point

Opening a File or Creating a File

The fopen() function is used to create a new file or to open an existing file.

General Syntax:

*fp = FILE *fopen(const char *filename, const char *mode);

Here, *fp is the FILE pointer (FILE *fp), which will hold the reference to the opened(or created) file.

filename is the name of the file to be opened and **mode** specifies the purpose of opening the file. Mode can be of following types,

mode	description
r	opens a text file in reading mode
w	opens or create a text file in writing mode.
a	opens a text file in append mode
r+	opens a text file in both reading and writing mode
w+	opens a text file in both reading and writing mode
a+	opens a text file in both reading and writing mode
rb	opens a binary file in reading mode
wb	opens or create a binary file in writing mode
ab	opens a binary file in append mode
rb+	opens a binary file in both reading and writing mode
wb+	opens a binary file in both reading and writing mode
ab+	opens a binary file in both reading and writing mode

Closing a File

The fclose() function is used to close an already opened file.

General Syntax :

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

Here fclose() function closes the file and returns **zero** on success, or **EOF** if there is an error in closing the file. This **EOF** is a constant defined in the header file **stdio.h**.

Input/Output operation on File

In the above table we have discussed about various file I/O functions to perform reading and writing on file. getc() and putc() are the simplest functions which can be used to read and write individual characters to a file.

```
#include<stdio.h>

int main()
{
   FILE *fp;
   char ch;
   fp = fopen("one.txt", "w");
   printf("Enter data...");
   while( (ch = getchar()) != EOF) {
      putc(ch, fp);
   }
   fclose(fp);
   fp = fopen("one.txt", "r");

   while( (ch = getc(fp)! = EOF)
   printf("%c",ch);

   // closing the file pointer
   fclose(fp);

   return 0;
}
```

1.Reading and Writing to File using fprintf() and fscanf()

```
#include<stdio.h>

struct emp
{
   char name[10];
   int age;
};
```

```
void main()
{
   struct emp e;
   FILE *p,*q;
   p = fopen("one.txt", "a");
   q = fopen("one.txt", "r");
   printf("Enter Name and Age:");
   scanf("%s %d", e.name, &e.age);
   fprintf(p,"%s %d", e.name, e.age);
   fclose(p);
   do
   {
      fscanf(q,"%s %d", e.name, e.age);
      printf("%s %d", e.name, e.age);
   }
   while(!feof(q));
}
```

In this program, we have created two FILE pointers and both are refering to the same file but in different modes.

fprintf() function directly writes into the file, while fscanf() reads from the file, which can then be printed on the console using standard printf() function.

2.Difference between Append and Write Mode

Write (w) mode and Append (a) mode, while opening a file are almost the same. Both are used to write in a file. In both the modes, new file is created if it doesn't exists already.

The only difference they have is, when you **open** a file in the **write** mode, the file is reset, resulting in deletion of any data already present in the file. While in **append** mode this will not happen. Append mode is used to append or add data to the existing data of file(if any). Hence, when you open a file in Append(a) mode, the cursor is positioned at the end of the present data in the file.

3.Reading and Writing in a Binary File

A Binary file is similar to a text file, but it contains only large numerical data. The Opening modes are mentioned in the table for opening modes above.

fread() and fwrite() functions are used to read and write is a binary file.

```
fwrite(data-element-to-be-written, size_of_elements, number_of_elements, pointer-to-file);
```

fread() is also used in the same way, with the same arguments like fwrite() function. Below mentioned is a simple example of writing into a binary file

```
const char *mytext = "The quick brown fox jumps over the lazy dog";
```

```
FILE *bfp= fopen("test.txt", "wb");
if (bfp)
{
   fwrite(mytext, sizeof(char), strlen(mytext), bfp);
   fclose(bfp);
}
```

fseek(), ftell() and rewind() functions

- fseek(): It is used to move the reading control to different positions using fseek function.
- ftell(): It tells the byte location of current position of cursor in file pointer.
- rewind(): It moves the control to beginning of the file.

Some File Handling Program Examples

- List all the Files present in a Directory
- Read Content of a File and Display it on screen
- Finding Size of a File
- Create a File and store Information in it
- Reverse the Content of File and Print it
- Copy Content of one File into Another File

<u>UNIT – V</u>

Dynamic Memory Allocation in C

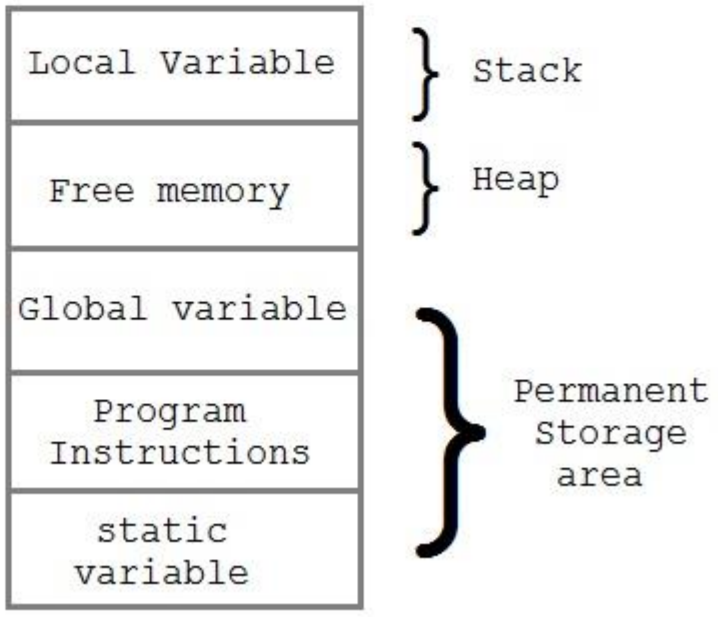
The process of allocating memory at runtime is known as **dynamic memory allocation**. Library routines known as **memory management functions** are used for allocating and freeing memory during execution of a program. These functions are defined in **stdlib.h** header file.

Function	**Description**
malloc()	allocates requested size of bytes and returns a void pointer pointing to the first byte of the allocated space
calloc()	allocates space for an array of elements, initialize them to zero and then returns a void pointer to the memory
free	releases previously allocated memory
realloc	modify the size of previously allocated space

Memory Allocation Process

Global variables, static variables and program instructions get their memory in **permanent** storage area whereas **local** variables are stored in a memory area called **Stack**.

The memory space between these two region is known as **Heap** area. This region is used for dynamic memory allocation during execution of the program. The size of heap keep changing.



Allocating block of Memory

malloc() function is used for allocating block of memory at runtime. This function reserves a block of memory of the given size and returns a **pointer** of type void. This means that we can assign it to any type of pointer using typecasting. If it fails to allocate enough space as specified, it returns a NULL pointer.

Syntax:

```
void* malloc(byte-size)
```

Time for an Example: malloc()

```
int *x;
x = (int*)malloc(50 * sizeof(int));     //memory space allocated to variable x
free(x);    //releases the memory allocated to variable x
```

calloc() is another memory allocation function that is used for allocating memory at runtime. calloc function is normally used for allocating memory to derived data types such as **arrays** and **structures**. If it fails to allocate enough space as specified, it returns a NULL pointer.

Syntax:

```
void *calloc(number of items, element-size)
```

Time for an Example: calloc()

```
struct employee
{
   char *name;
   int salary;
};
typedef struct employee emp;
emp *e1;
e1 = (emp*)calloc(30,sizeof(emp));
```

realloc() changes memory size that is already allocated dynamically to a variable.

Syntax:

```
void* realloc(pointer, new-size)
```

Time for an Example: realloc()

```
int *x;
x = (int*)malloc(50 * sizeof(int));
```

x = (int*)realloc(x,100);   //allocated a new memory to variable x

Diffrence between malloc() and calloc()

calloc()	**malloc()**
calloc() initializes the allocated memory with 0 value.	malloc() initializes the allocated memory with garbage values.
Number of arguments is 2	Number of argument is 1
Syntax : (cast_type *)calloc(blocks , size_of_block);	**Syntax :** (cast_type *)malloc(Size_in_bytes);

Program to represent Dynamic Memory Allocation(using calloc())

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

int main()
{
   int i, n;
   int *element;

   printf("Enter total number of elements: ");
   scanf("%d", &n);

   /*
      returns a void pointer(which is type-casted to int*)
      pointing to the first block of the allocated space
   */
   element = (int*) calloc(n,sizeof(int));

   /*
      If it fails to allocate enough space as specified,
      it returns a NULL pointer.
   */
   if(element == NULL)
   {
      printf("Error.Not enough space available");
      exit(0);
   }

   for(i = 0; i < n; i++)
   {
      /*
         storing elements from the user
```

```
            in the allocated space
        */
       scanf("%d", element+i);
    }
    for(i = 1; i < n; i++)
    {
       if(*element > *(element+i))
       {
            *element = *(element+i);
        }
    }

    printf("Smallest element is %d", *element);

    return 0;
}
```

```
Enter total number of elements: 5 4 2 1 5 3 Smallest element is 1
```

Allocating block of Memory

malloc() function is used for allocating block of memory at runtime. This function reserves a block of memory of the given size and returns a **pointer** of type void. This means that we can assign it to any type of pointer using typecasting. If it fails to allocate enough space as specified, it returns a NULL pointer.

Syntax:

```
void* malloc(byte-size)
```

Time for an Example: malloc()

```
int *x;
x = (int*)malloc(50 * sizeof(int));     //memory space allocated to variable x
free(x);    //releases the memory allocated to variable x
```

calloc() is another memory allocation function that is used for allocating memory at runtime. calloc function is normally used for allocating memory to derived data types such as **arrays** and **structures**. If it fails to allocate enough space as specified, it returns a NULL pointer.

Syntax:

```
void *calloc(number of items, element-size)
```

Time for an Example: calloc()

```
struct employee
```

```
{
   char *name;
   int salary;
};
typedef struct employee emp;
emp *e1;
e1 = (emp*)calloc(30,sizeof(emp));
```

realloc() changes memory size that is already allocated dynamically to a variable.

Syntax:

void* realloc(pointer, new-size)

Time for an Example: realloc()

```
int *x;
x = (int*)malloc(50 * sizeof(int));
x = (int*)realloc(x,100);   //allocated a new memory to variable x
```

Diffrence between malloc() and calloc()

calloc()	**malloc()**
calloc() initializes the allocated memory with 0 value.	malloc() initializes the allocated memory with garbage values.
Number of arguments is 2	Number of argument is 1
Syntax : (cast_type *)calloc(blocks , size_of_block);	**Syntax :** (cast_type *)malloc(Size_in_bytes);

Program to represent Dynamic Memory Allocation(using calloc())

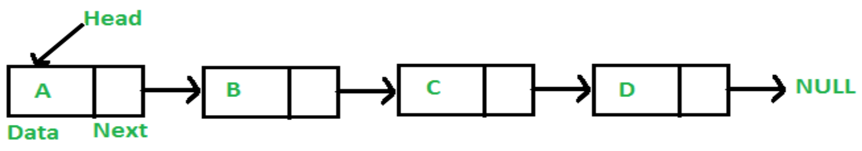
```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int i, n;
   int *element;

   printf("Enter total number of elements: ");
   scanf("%d", &n);

   /*
```

```
        returns a void pointer(which is type-casted to int*)
        pointing to the first block of the allocated space
    */
    element = (int*) calloc(n,sizeof(int));

    /*
        If it fails to allocate enough space as specified,
        it returns a NULL pointer.
    */
    if(element == NULL)
    {
        printf("Error.Not enough space available");
        exit(0);
    }

    for(i = 0; i < n; i++)
    {
        /*
            storing elements from the user
            in the allocated space
        */
       scanf("%d", element+i);
    }
    for(i = 1; i < n; i++)
    {
       if(*element > *(element+i))
       {
            *element = *(element+i);
        }
    }

    printf("Smallest element is %d", *element);

    return 0;
}
```

Enter total number of elements: 5 4 2 1 5 3 Smallest element is 1

Linked List

Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers.



Definition of Linked List

Arrays can be used to store linear data of similar types, but arrays have the following limitations.
1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.
2) Inserting a new element in an array of elements is expensive because the room has to be created for the new elements and to create room existing elements have to be shifted.

Advantages over arrays
1) Dynamic size
2) Ease of insertion/deletion
Drawbacks:
1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists efficiently with its default implementation. Read about it here.
2) Extra memory space for a pointer is required with each element of the list.
3) Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

Representation:

A linked list is represented by a pointer to the first node of the linked list. The first node is called the head. If the linked list is empty, then the value of the head is NULL.
Each node in a list consists of at least two parts:
1) data
2) Pointer (Or Reference) to the next node

Preprocessor:

C Preprocessor is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation. We'll refer to the C Preprocessor as CPP.

All preprocessor commands begin with a hash symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column. The following section lists down all the important preprocessor directives −

Sr.No.	Directive & Description
1	**#define** Substitutes a preprocessor macro.
2	**#include** Inserts a particular header from another file.
3	**#undef** Undefines a preprocessor macro.

4	**#ifdef** Returns true if this macro is defined.
5	**#ifndef** Returns true if this macro is not defined.
6	**#if** Tests if a compile time condition is true.
7	**#else** The alternative for #if.
8	**#elif** #else and #if in one statement.
9	**#endif** Ends preprocessor conditional.
10	**#error** Prints error message on stderr.
11	**#pragma** Issues special commands to the compiler, using a standardized method.

Preprocessors Examples

Analyze the following examples to understand various directives.

#define MAX_ARRAY_LENGTH 20

This directive tells the CPP to replace instances of MAX_ARRAY_LENGTH with 20. Use *#define* for constants to increase readability.

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

These directives tell the CPP to get stdio.h from **System Libraries** and add the text to the current source file. The next line tells CPP to get **myheader.h** from the local directory and add the content to the current source file.

```
#undef  FILE_SIZE
#define FILE_SIZE 42
```

It tells the CPP to undefine existing FILE_SIZE and define it as 42.

```
#ifndef MESSAGE
  #define MESSAGE "You wish!"
#endif
```

It tells the CPP to define MESSAGE only if MESSAGE isn't already defined.

```
#ifdef DEBUG
  /* Your debugging statements here */
```

```
#endif
```

PROGRAMMING GUIDELINES

Coding Standards and Guidelines

Different modules specified in the design document are coded in the Coding phase according to the module specification. The main goal of the coding phase is to code from the design document prepared after the design phase through a high-level language and then to unit test this code.

Good software development organizations want their programmers to maintain to some well-defined and standard style of coding called coding standards. They usually make their own coding standards and guidelines depending on what suits their organization best and based on the types of software they develop. It is very important for the programmers to maintain the coding standards otherwise the code will be rejected during code review.

Purpose of Having Coding Standards:

- A coding standard gives a uniform appearance to the codes written by different engineers.
- It improves readability, and maintainability of the code and it reduces complexity also.
- It helps in code reuse and helps to detect error easily.
- It promotes sound programming practices and increases efficiency of the programmers.

Some of the coding standards are given below:

1. **Limited use of globals:**
   These rules tell about which types of data that can be declared global and the data that can't be.
2. **Standard headers for different modules:**
   For better understanding and maintenance of the code, the header of different modules should follow some standard format and information. The header format must contain below things that is being used in various companies:
   - Name of the module
   - Date of module creation
   - Author of the module
   - Modification history
   - Synopsis of the module about what the module does
   - Different functions supported in the module along with their input output parameters
   - Global variables accessed or modified by the module
3. **Naming conventions for local variables, global variables, constants and functions:**
   Some of the naming conventions are given below:
   - Meaningful and understandable variables name helps anyone to understand the reason of using it.
   - Local variables should be named using camel case lettering starting with small letter (e.g. **localData**) whereas Global variables names should start with a capital letter (e.g. **GlobalData**). Constant names should be formed using capital letters only (e.g. **CONSDATA**).
   - It is better to avoid the use of digits in variable names.
   - The names of the function should be written in camel case starting with small letters.

- The name of the function must describe the reason of using the function clearly and briefly.

4. **Indentation:**
   Proper indentation is very important to increase the readability of the code. For making the code readable, programmers should use White spaces properly. Some of the spacing conventions are given below:
   - There must be a space after giving a comma between two function arguments.
   - Each nested block should be properly indented and spaced.
   - Proper Indentation should be there at the beginning and at the end of each block in the program.
   - All braces should start from a new line and the code following the end of braces also start from a new line.

5. **Error return values and exception handling conventions:**
   All functions that encountering an error condition should either return a 0 or 1 for simplifying the debugging.
   On the other hand, Coding guidelines give some general suggestions regarding the coding style that to be followed for the betterment of understandability and readability of the code. Some of the coding guidelines are given below :

6. **Avoid using a coding style that is too difficult to understand:**
   Code should be easily understandable. The complex code makes maintenance and debugging difficult and expensive.
7. **Avoid using an identifier for multiple purposes:**
   Each variable should be given a descriptive and meaningful name indicating the reason behind using it. This is not possible if an identifier is used for multiple purposes and thus it can lead to confusion to the reader. Moreover, it leads to more difficulty during future enhancements.
8. **Code should be well documented:**
   The code should be properly commented for understanding easily. Comments regarding the statements increase the understandability of the code.
9. **Length of functions should not be very large:**
   Lengthy functions are very difficult to understand. That's why functions should be small enough to carry out small work and lengthy functions should be broken into small ones for completing small tasks.
10. **Try not to use GOTO statement:**
    GOTO statement makes the program unstructured, thus it reduces the understandability of the program and also debugging becomes difficult.

Advantages of Coding Guidelines:

- Coding guidelines increase the efficiency of the software and reduces the development time.
- Coding guidelines help in detecting errors in the early phases, so it helps to reduce the extra cost incurred by the software project.
- If coding guidelines are maintained properly, then the software code increases readability and understandability thus it reduces the complexity of the code.
  - It reduces the hidden cost for developing the software.