enlightening brilliance

C Programming

INDEX

	· Union in C	
	· Difference in structure & union	
8	File Handling & Preprocessor Directives	**2**
	· File Handling goals	
	· File handling functions	
9	Command Line Arguments	**2**
10	Programming Practice	

1. **What is C ?**

A **computer** is an electronic device that manipulates information, or data. It has the ability to **store**, **retrieve**, and **process** data. You probably already know that you can use a computer to **type documents**, **send email**, **play games**, and **browse the Web**. You can also use it to edit or create **spreadsheets**, **presentations**, and even **videos**.

Computer Consist of Hardware and Software

1. Hardware:

Hardware represents the physical components of a computer i.e. the components that can be seen and touched.

Examples of Hardware are following:

- Input devices -- keyboard, mouse etc.
- Output devices -- printer, monitor etc.
- Secondary storage devices -- Hard disk, CD, DVD etc.
- Internal components -- CPU, motherboard, RAM etc.

2. Software:

Software is a collection of instructions that enable the user to interact with a computer, its hardware, or perform tasks. Without software, computers would be useless. For example, without your Internet browser, you could not surf the Internet or read this page and without an operating system, the browser could not run on your computer.
Types of Software are ass follows,

1. Application Software
e.g: Application Packages, Special purpose software like Word, Excel and PowerPoint.

2. System Software:
OS, Packages(Linker, loader,editor),Translators(Compiler & Interpreter)

Communication Between User and Machine:

User want to interact with machine but machine only knows language in the form of 0’s and 1’s i.e. Bit (Binary Language)
So Binary language is also known as Machine level language but it is difficult to understand because it is in the form of bits. So user want language that is close to user

and we know it as High level language .High level language is just like English language, all instructions are in friendly approach.

So, To convert Machine level language to High Level language and high level language to machine level language we need Translators. We will discuss now a translators.

Translators:

Programming languages (compiler or Interpreter) that converts a computer program written in one language to another at that time we need Translators.

Translators are of types

1.Compiler
2. Interpreter

1.1. Compiler

- Compiler Takes **Entire** program as input
- Intermediate Object Code is Generated
- Conditional Control Statements are Executes faster
- Memory Requirement : More (Since Object Code is Generated)
- Program need not be compiled every time
- Errors are displayed after entire program is checked
- Example : C Compiler

1.2. Interpreter

- Interpreter Takes Single instruction as input
- No Intermediate Object Code is Generated
- Conditional Control Statements are Executes slower
- Memory Requirement is Less
- Every time higher level program is converted into lower level program
- Errors are displayed for every instruction interpreted (if any)
- Example: Java

Need of C Language:

C is called as Middle level language. Behaves as High Level Language through Functions - gives a modular programming It gives access to the low level memory through Pointers As its a combination of these two aspects, its neither a High Level nor a Low level language ,so C is Middle Level Language.

History of C Language:

Developed at Bell Laboratories in 1972 by Dennis Ritchie.

Features were derived from earlier language called “B” (Basic Combined Programming Language – BCPL)which is developed by Ken Thompson in 1970’s.

Invented for implementing UNIX operating system

Features of C:

- Simple
- Machine Independent or Portable
- Mid-level programming language
- structured programming language
- Rich Library
- Memory Management
- Fast Speed
- Pointers
- Recursion
- Extensible

C programming Basic:

Each language is made up of two parts,

Vocabulary

Grammar

In programming language,

Vocabulary means Tokens

Grammar means Syntax

Tokens:

Tokens are **basic building blocks** of C Programming.

Tokens consist of Identifiers,Variables,Constants,Character Set,Keywords,Data Types,Operators.

No	Token Type	Example 1	Example 2
1	Keyword	Do	while
2	Constants	number	sum
3	Identifier	-76	89
4	String	“HTF”	“PRIT”
5	Special Symbol	*	@
6	Operators	++	/

Token	Meaning
Keyword	A variable is a meaningful name of data storage location in computer memory. When using a variable you refer to memory address of computer
Constant	Constants are expressions with a fixed value
Identifier	The term identifier is usually used for variable names
String	Sequence of characters
Special Symbol	Symbols other than the Alphabets and Digits and white-spaces
Operators	A symbol that represent a specific mathematical or non mathematical action

Data Types:

- C data types are defined as the data storage format that a variable can store a data to perform a specific operation.
- Data types are used to define a variable before to use in a program.
- Size of variable, constant and array are determined by data types.
- Data Types are of Three Types

1. Primitive Data Types: int,float,char etc.

2. Derived Data Types: array, Function,etc

3. User Defined Data types: Structure, Union

.No	C Data types	storage Size	Range
1	Char	1	–127 to 127
2	Int	2	–32,767 to 32,767
3	Float	4	1E–37 to 1E+37 with six digits of precision
4	Double	8	1E–37 to 1E+37 with ten digits of precision
5	long double	10	1E–37 to 1E+37 with ten digits of precision
6	long int	4	–2,147,483,647 to 2,147,483,647
7	short int	2	–32,767 to 32,767
8	unsigned short int	2	0 to 65,535
9	signed short int	2	–32,767 to 32,767
10	long long int	8	–(2power(63) –1) to 2(power)63 –1
11	signed long int	4	–2,147,483,647 to 2,147,483,647

12	unsigned long int	4	0 to 4,294,967,295
13	unsigned long long int	8	2(power)64 –1

Operator in C:
Symbol that tells the compiler to perform specific mathematical or logical operations.

Types of operators-

- Arithmetic Operators
- Relational Operators
- Increment Operator
- Decrement Operator
- Logical Operators
- Conditional Operator

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:

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

Increment & Decrement Operator:

Increment operator increase the value of subsequent by 1. value may be increase according to the programmer.

Increment operator are two types as follows :

Post increment(e.g A++)

Pre increment(e.g ++A)

Decrement operator decrease the value of subsequent by 1. value may be decrease according to the programmer.

Decrement operator are of two types as follows :

Post decrement(e.g A--)
Pre decrement(e.g --A)

Logical Operators:

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

Conditional Operator:

Conditional operator is also called as Ternary operator

Syntax:

condition ? result1 : result2

If the condition is true, result1 is returned else result2 is returned.

Structure of C Program:

1.1. Documentation Section
1.2. Link Section
1.3. Definition Section
1.4. Global Declaration Section
1.5. main(){ }
1.6. Subprogram Section-function1,function2, etc

Link Section and Main() section is Mandatory part, remaining all are optional sections.
Simple Examples:

Problem Statement: To print “Welcome to Spark”
Solution:

```
// Program to print “Welcome to Spark”          --------------Documentation Section
#include<stdio.h>                               ---------------Link Section
#include<conio.h>                               ---------------Link Section
main()                                          ---------------main()        Function
Section
{
        printf(“Welcome to Spark”);
        getch();
}
```

Note: #include<stdio.h> and #include<conio.h> these are header Files which is to be included in a program because whatever the inbuilt functions (printf(), scanf(), getch())are there, that are already defined in these header File. getch() is used to get a character from console but does not echo to the screen.

Syntax of printf():

```
printf(“ any string”);
```

Syntax of scanf():

```
scanf(“ formate specifier”,&var_name);
```

Exercise:

1. Implement a program to Calculate Area of Circle by taking input from user.
2. Implement a program to Calculate Simple interest.
3. Implement a program to Calculate Area of Equilateral Triangle.
4. Implement c program to multiply given number by 4 using bitwise Operations
5. Implement a program to check maximum number among three numbers.

2. Control Statements in C

As the name suggests the ‘Control Instructions’ enable us to specify the order in which the various instructions in a program are to be executed by the computer. In other words the control instructions determine the ‘flow of control’ in a program. There are four types of control instructions in C. They are:

(a) Sequence Control Instruction
(b) Selection or Decision Control Instruction
(c) Repetition or Loop Control Instruction
(d) Case Control Instruction

The Sequence control instruction ensures that the instructions are executed in the same order in which they appear in the program. Decision and Case control instructions allow the computer to take a decision as to which instruction is to be executed next. The Loop control instruction helps computer to execute a group of statements repeatedly. In the following chapters we are going to learn these instructions in detail. Try your hand at the Exercise presented on the following pages before proceeding to the next chapter, which discusses the decision control instruction.

1. The if Statement

2. **Like most languages, C uses the keyword if** to implement the decision control instruction. The general form of **if** statement looks like this:

```
if ( this condition is true )
        execute this statement ;
```

The keyword **if** tells the compiler that what follows is a decision control instruction. The condition following the keyword **if** is always enclosed within a pair of parentheses. If the condition, whatever it is, is true, then the statement is executed. If the condition is not true then the statement is not executed;

example:

If you want to verify the person is eligible for voting or not then in this case we are using if statements.

```
main()
{
    int age;
    printf(“Please Enter the age:”);
    scanf(“%d”,&age);
    if(age>=18)
```

```
        {
                Printf("Person is Eligible for voting");
        }
        else
        {
                printf("Person is not eligible forvoting");
        }
}
```

else-if-Ladder:

If you want to check multiple condition then we are using else-if-ladder

syntax:

```
if(condition_1)
{
}
else if(condition_2)
{
}
else if(condition_2)
{
}
else
{
}
```

Branching Statements: (Switch Statement)

Unlike if-then and if-then-else statements, the switch statement can have a number of possible execution paths. A switch works with the byte, short, char, and int primitive data types.

Syntax:

```
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.

Example:

```
main()
{
        int roll=2;
        switch(roll)
        {
        case 1: printf("I am in case 1");
                break;
        case 2: printf("I am in Case 2");
                break;
```

```
            case 3: printf("I am in case 3");
                        break;
            default: printf("wrong choice");
                        break;
            }
}
```

output:

```
I am in case 2
```

3. **Iteration/Looping Statements:**

You may encounter situations, when a block of code needs to be executed several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. Given below is the general form of a loop statement in most of the programming languages −

1. While Loop

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);
            }
```

Example:

```
 main () {

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

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

}
```

Output:

```
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
```

2. **do-While 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 );
```

example:

```
 main () {

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

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


}
```

```
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. 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.
-

Example:

```
main () {

  int a;

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

output:

```
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
```

Nested Loops in c:

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

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

```
for ( init; condition; increment ) {

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

  statement(s);
}
```

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

```
while(condition) {

  while(condition) {
    statement(s);
  }

  statement(s);
}
```

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

```
do {

  statement(s);

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

}while( condition );
```

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

Jumping Statements

There are three Jumping Statements
1. Break statement
2. Continue statements
3. goto statements

The **break** statement in C programming has the following two usages −

- When a **break** statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
- It can be used to terminate a case in the **switch** statement (covered in the next chapter).

If you are using nested loops, the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

Exercise
1. Implement a program to check Person is eligible for voting or not by using if-else

2. Implement a program to check Even and odd number using if-else
3. Implement a program to check whether entered character is vowel or consonant using switch case.
4. Implement a menu driven program for Arithmetic operation using Switch case
5. Implement a program to calculate grade of student according to following condition.

Above 70	Distinction
Between 60 to 70	I class
Between 50 to 60	II class
Between 40 to 50	Pass
Below 40	Fail

3. Functions

- What is Functions?
- Why we need Function?
- Types of functions
- Categories of functions
- Example

What is Functions?

A function is a self-contained block of statements that perform a coherent task of some kind. Every C program can be thought of as a collection of these functions. As we noted earlier, using a function is something like hiring a person to do a specific job for you. Sometimes the interaction with this person is very simple; sometimes it's complex.

A function is a group of statements that together perform a task. Every C program has at least one function, which is **main ()**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.

Why we need a function?

Why write separate functions at all? Why not squeeze the entire logic into one function, **main ( )**? Two reasons:

(a) Writing functions avoids rewriting the same code over and over. Suppose you have a section of code in your program that calculates area of a triangle. If later in the program you want to calculate the area of a different triangle, you won't like it if you are required to write the same instructions all over again. Instead, you would prefer to jump to a 'section of code' that calculates area and then jump back to the place from where you left off. This section of code is nothing but a function.

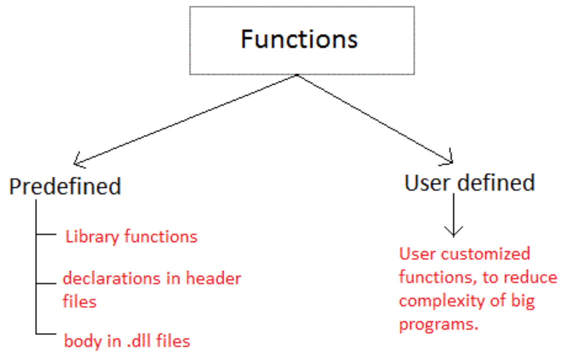
(b) Using functions it becomes easier to write programs and keep track of what they are doing. If the operation of a program can be divided into separate activities, and each activity placed in a different function, then each could be written and checked more or less independently. Separating the code into modular functions also makes the program easier to design and understand.

A function can also be referred as a method or a sub-routine or a procedure, etc.

What is the moral of the story? Don't try to cram the entire logic in one function. It is a very bad style of programming. Instead, break a program into small units and write functions for each of these isolated subdivisions. Don't hesitate to write functions that are called only once. What is important is that these functions perform some logically isolated task.

Functions Types:

1. **Library function**
2. **User defined Function**


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

Library functions are those functions which are defined by 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.

User-defined functions are those functions which are defined by the user at the time of writing program. Functions are made for code reusability and for saving time and space.

Programmers have to follow three goals to write a function.

1. Function definition
2. Function Declaration
3. Function Calling

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function. Function **calling** calls the particular function at any given time.

Function declaration

General syntax of function declaration is,
return-type **function-name (parameter-list) ;**
Like variable and an array, a function must also be declared before its called. A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately. A function declaration consist of 4 parts.

- return-type
- function name
- parameter list
- terminating semicolon

Function definition Syntax
General syntax of function definition is,

```
return-type function-name (parameter-list)
{
  function-body ;
}
```

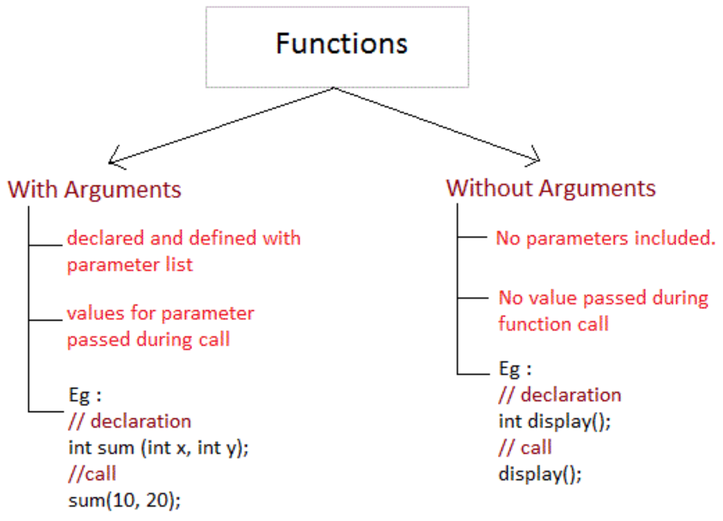
The first line return-type **function-name(parameter)** is known as **function header** and the statement within curly braces is called **function body**.
return-type
return type specifies the type of value(int,float,char,double) that function is expected to return to the program calling the function.
function-name
function name specifies the name of the function. The function name is any valid C identifier and therefore must follow the same rule of formation as other variables in C.
parameter-list
The parameter list declares the variables that will receive the data sent by calling program. They often referred to as formal parameters. These parameters are also used to send values to calling program.
function-body
The function body contains the declarations and the statement(algorithm) necessary for performing the required task. The body is enclosed within curly braces { } and consists of three parts.

- **local** variable declaration.
- **function statement** that performs the tasks of the function.
- a **return** statement that return the value evaluated by the function.

Functions and Arguments
Arguments are the values specified during the function call, for which the formal parameters are declared in the function. Functions are of four categories

1. Function with parameter with return type
2. Function without parameter with return type
3. Function with parameter without return type

4. Function without parameter without return type



Nesting of Functions

C language also allows nesting of functions, one function using another function inside its body. We must be careful while using nested functions, because it may lead to infinte nesting.

```
function1()
{
  function2() ;
  //statements
}
```

If function2 calls function1 inside it, then in this case it will lead to infinite nesting, they will keep calling each other. Hence we must be careful.

Recursion

Recursion is a special of nesting functions, where a function calls itself inside it. We must have certain condition to break out of the recursion, otherwise recursion is infinite.

```
function1()
{
  function1() ;
  //statements
```

Example : Factorial of a number using Recursion

```
#include<stdio.h>
#include<conio.h>
int factorial(int x);

void main()
{
 int a,b;
 clrscr();
 printf("Enter no.");
 scanf("%d",&a);
 b=factorial(a);
 printf("%d",b);
 getch();
}

int factorial(int x)
{
 int r=1;
 if(x==1) return 1;
 else r=x*factorial(x-1);
 return r;
}
```

Types of Function calls in C

Functions are called by their names. If the function is without argument, it can be called directly using its name. But for functions with arguments, we have two ways to call them,

1. Call by Value
2. Call by Reference

Call by Value

In this calling technique we pass the values of arguments which are stored or copied into the formal parameters of functions. Hence, the original values are unchanged only the parameters inside function changes.

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

Output : 10

In this case the actual variable x is not changed, because we pass argument by value, hence a copy of x is passed, which is changed, and that copied value is destroyed as the function ends(goes out of scope). So the variable **x** inside main() still has a value 10.

But we can change this program to modify the original **x**, by making the function **calc()** return a value, and storing that value in x.

```
int calc(int x);
int main()
{
 int x = 10;
 x = calc(x);
 printf("%d", x);
}

int calc(int x)
{
 x = x + 10 ;
 return x;
}
```

Output : 20

Call by Reference

In this we pass the address of the variable as arguments. In this case the formal parameter can be taken as a reference or a pointer, in both the case they will change the values of the original variable.

```
void calc(int *p);
int main()
{
 int x = 10;
 calc(&x);     // passing address of x as argument
 printf("%d", x);
}

void calc(int *p)
{
 *p = *p + 10;
}
```

Output : 20

Exercise:

1. Implement a program for a function to check whether number is Armstrong or not.
2. Implement a program for prime number using function.
3. Implement a program for arithmetic operation using all categories of function.
4. Implement a program for factorial using Recursion.
5. Implement a program for fibonacci series using Recursion.

4.Introduction to Pointers

Pointers are variables that hold address of another variable of same data type.

Pointers are one of the most distinct and exciting features of C language. It provides power and flexibility to the language. Although pointer may appear little confusing and complicated in the beginning, but trust me its a powerful tool and handy to use once its mastered.

Benefit of using pointers

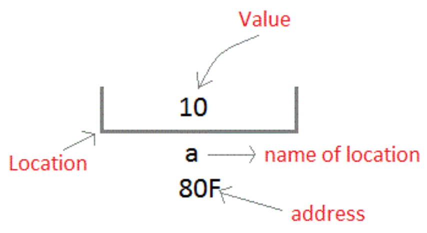
- Pointers are more efficient in handling Array and Structure.
- Pointer allows references to function and thereby helps in passing of function as arguments to other function.
- It reduces length and the program execution time.
- It allows C to support dynamic memory management.

Concept of Pointer

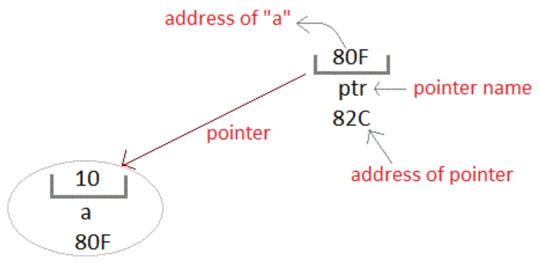
Whenever a **variable** is declared, system will allocate a location to that variable in the memory, to hold value. This location will have its own address number.

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

int a = 10 ;



We can access the value 10 by either using the variable name **a** or the address 80F. Since the memory addresses are simply numbers they can be assigned to some other variable. The variable that holds memory address are called **pointer variables**. A **pointer** variable is therefore nothing but a variable that contains an address, which is a location of another variable. Value of **pointer variable** will be stored in another memory location.



Declaring a pointer variable

General syntax of pointer declaration is,

data-type *pointer_name;

Data type of pointer must be same as the variable, which the pointer is pointing. **void** type pointer works with all data types, but isn't used oftenly.

Initialization of Pointer variable

Pointer Initialization is the process of assigning address of a variable to **pointer** variable. Pointer variable contains address of variable of 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.

```
int a = 10 ;
int *ptr ;       //pointer declaration
ptr = &a ;       //pointer initialization
or,
int *ptr = &a ;     //initialization and declaration together
```

Pointer variable always points to same type of data.

```
float a;
int *ptr;
ptr = &a;    //ERROR, type mismatch
```

Dereferencing of Pointer

Once a pointer has been assigned the address of a variable. To access the value of variable, pointer is dereferenced, using the **indirection operator** *.

```
int a,*p;
a = 10;
p = &a;

printf("%d",*p);    //this will print the value of a.

printf("%d",*&a);  //this will also print the value of a.
```

SPARK
enlightening brilliance

```
printf("%u",&a);  //this will print the address of a.

printf("%u",p);  //this will also print the address of a.

printf("%u",&p);  //this will also print the address of p.
```

Exercise:
1. What are the Types of pointer and Explain each type with program.
2. Write a swapping program by using pointer concept.
3. Write a function which returns a pointer variable.
4. write a program for arithmetic operation using pointer.
5. Write a program to check whether number is palindrome or not by using pointer.

5. 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 a students,
- or to store list of numbers or characters etc.

Since arrays provide an easy way to represent data, it is classified amongst the data structures in C. Other data structures in c are **structure**, **lists**, **queues** and **trees**. Array can be used to represent not only simple list of data but also table of data in two or three dimensions.

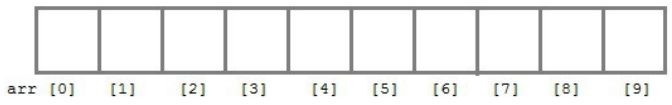
Declaring an Array

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

data-type variable-name[size];

for example :

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

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

```
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 things to remember is that when you will give more initializer than declared array size than the **compiler** will give an error.

```
#include<stdio.h>
#include<conio.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]);
 }
 getch();
}
```

Output :

```
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 array, or to initialize array with user specified values. Example,

```
#include<stdio.h>
#include<conio.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]);
 }
 getch();
}
```

Two dimensional Arrays

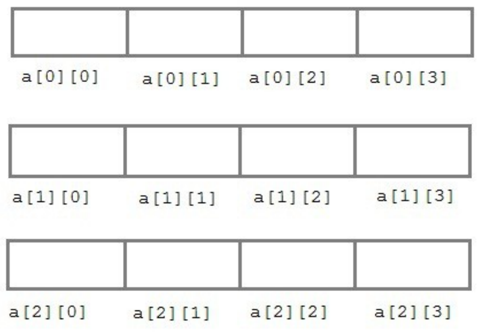
C language supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.

Two-dimensional array is declared as follows,

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

Example :

int a[3][4];



The above array can also be declared and initialized together. Such as,

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

Run-time initialization of two dimensional Array

```
#include<stdio.h>
#include<conio.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]);
  }
 }
getch();
}
```

Exercise:

1. Implement a program to Reverse the array elements.
2. Implement a program to search a given element by using array.
3. Implement a program to sort array in descending order.
4. Implement a program which shows even and odd position elements.

6. String

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[10]="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 listings all 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 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<conio.h>
#include<string.h>
void main()
{
 char str[20];
 clrscr();
 printf("Enter a string");
 scanf("%[^\n]",&str);
 printf("%s",str);
 getch();
}
```

Another method to read character string with white spaces from terminal is **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 program to use these functions.

The following are the most commonly used string handling functions.

Method	**Description**
strcat()	It is used to concatenate(combine) two string
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"**.

strlen() function

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

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

output :

```
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);
```

output:

```
-1
```

Exercise:

4.1. Implement a program to merge two string without using Library function.

4.2. Implement a program for the following output:

i/p: s1:{‘1’,’3’,’5’,’7’,’9’} s2= s1:{‘2’,’4’,’6’,’8’,’10’}

o/p: s3={‘1’,’2’,’3’,’4’,’5’,’6’,’7’,’8’,’9’,’10’}

4. Implement a program to reverse the string without using Library function
5. Implement a program to count no of character occure in string.
6. Implement a program to sort a string alphabetically order.

7. Structure & Union

Introduction to Structure

Structure is a user-defined data type in C which allows you to combine different data types to store a particular type of record. Structure helps to construct a complex data type in more meaningful way. It is somewhat similar to an Array. The only difference is that array is used to store collection of similar datatypes while structure can store collection of any type of data.

Structure is used to represent a record. Suppose you want to store record of **Student** which consists of student name, address, roll number and age. You can define a structure to hold this information.

Defining a structure

struct keyword is used to define a structure. **struct** define a new data type which is a collection of different type of data.

Syntax :

```
struct structure_name
{
 //Statements
};
```

Example of Structure

```
struct Book
{
 char name[15];
 int price;
 int pages;
};
```

Here the **struct Book** declares a structure to hold the details of book which consists of three data fields, namely name, price and pages. These fields are called **structure elements or members**. Each member can have different data type,like in this case, **name** is of char type and **price** is of int type etc. **Book** is the name of the structure and is called structure tag.

Declaring Structure Variables

It is possible to declare variables of a **structure**, after the structure is defined. **Structure** variable declaration is similar to the declaration of variables of any other data types. Structure variables can be declared in following two ways.

1) Declaring Structure variables separately

```
struct Student
{
 char[20] name;
 int age;
 int rollno;
} ;

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

2) Declaring Structure Variables with Structure definition

```
struct Student
{
 char[20] name;
 int age;
 int rollno;
} S1, S2 ;
```

Here **S1** and **S2** are variables of structure **Student**. However this approach is not much recommended.

Accessing Structure Members

Structure members can be accessed and assigned values in number of ways. Structure member has no meaning independently. In order to assign a value to a structure member, the member name must be linked with the **structure** variable using dot . operator also called **period** or **member access** operator.

```
struct Book
{
 char name[15];
 int price;
 int pages;
} b1 , b2 ;

b1.price=200;      //b1 is variable of Book type and price is member of Book
```

We can also use scanf() to give values to structure members through terminal.

```
scanf(" %s ", b1.name);
scanf(" %d ", &b1.price);
```

Structure Initialization

Like any other data type, structure variable can also be initialized at compile time.

```
struct Patient
```

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

struct Patient p1 = { 180.75 , 73, 23 };    //initialization
or,
struct patient p1;
p1.height = 180.75;     //initialization of each member separately
p1.weight = 73;
p1.age = 23;
```

Array of Structure

We can also declare an array of **structure**. Each element of the array representing a **structure** variable. **Example :** struct employee emp[5];

The above code define an array **emp** of size 5 elements. Each element of array **emp** is of type **employee**

```
#include<stdio.h>
#include<conio.h>
struct employee
{
 char ename[10];
 int sal;
};

struct employee emp[5];
int i,j;
void ask()
{
 for(i=0;i<3;i++)
  {
   printf("\nEnter %dst employee record\n",i+1);
   printf("\nEmployee name\t");
   scanf("%s",emp[i].ename);
   printf("\nEnter employee salary\t");
   scanf("%d",&emp[i].sal);
  }
 printf("\nDisplaying Employee record\n");
 for(i=0;i<3;i++)
  {
   printf("\nEmployee name is %s",emp[i].ename);
   printf("\nSlary is %d",emp[i].sal);
  }
}
void main()
{
 clrscr();
 ask();
 getch();
}
```

Nested Structures

Nesting of structures, is also permitted in C language.

Example :

```
struct student
{
 char[30] name;
 int age;
   struct address
    {
     char[50] locality;
     char[50] city;
     int pincode;
    };
};
```

Structure as function arguments

We can pass a structure as a function argument in similar way as we pass any other variable or array.

Example :

```
#include<stdio.h>
#include<conio.h>
struct student
{
 char name[10];
 int roll;
};
void show(struct student st);
void main()
{
 struct student std;
 clrscr();
 printf("\nEnter student record\n");
 printf("\nstudent name\t");
 scanf("%s",std.name);
 printf("\nEnter student roll\t");
 scanf("%d",&std.roll);
 show(std);
 getch();
}

void show(struct student st)
{
 printf("\nstudent name is %s",st.name);
 printf("\nroll is %d",st.roll);
}
```

typedef

typedef is a keyword used in C language to assign alternative names to existing types. Its mostly used with user defined data types, when names of data types get slightly complicated. Following is the general syntax for using typedef,

typedef existing_name alias_name

Lets take an example and see how typedef actually works.

typedef unsigned long ulong;

The above statement define a term **ulong** for an unsigned long type. Now this **ulong** identifier can be used to define unsigned long type variables.

ulong i, j ;

Application of typedef

typedef can be used to give a name to user defined data type as well. Lets see its use with structures.

```
typedef struct
{
  type member1;
  type member2;
  type member3;
} type_name ;
```

Here **type_name** represents the stucture definition associated with it. Now this **type_name** can be used to declare a variable of this stucture type.

type_name t1, t2 ;

Example of structure definition using typedef

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

typedef struct employee
{
 char  name[50];
 int   salary;
} emp ;

void main( )
{
 emp e1;
 printf("\nEnter Employee record\n");
 printf("\nEmployee name\t");
 scanf("%s",e1.name);
 printf("\nEnter Employee salary \t");
 scanf("%d",&e1.salary);
 printf("\nstudent name is %s",e1.name);
 printf("\nroll is %d",e1.salary);
 getch();
}
```

typedef and Pointers

typedef can be used to give an alias name to pointers also. Here we have a case in which use of typedef is beneficial during pointer declaration.

In Pointers * binds to the right and not the left.

int* x, y ;

By this declaration statement, we are actually declaring **x** as a pointer of type int, whereas **y** will be declared as a plain integer.

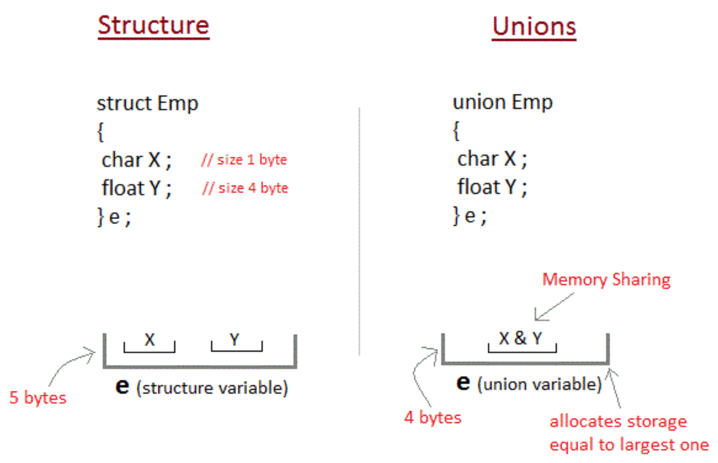
typedef int* **IntPtr** ;

IntPtr x, y, z;

But if we use **typedef** like in above example, we can declare any number of pointers in a single statement.

Unions in C Language

Unions are conceptually similar to **structures**. The syntax of **union** is also similar to that of 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 same time**. A **union** is declared using **union** keyword.

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

This declares a variable **It1** of type union **item**. 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 a **union** variable, irrespective of its size. The compiler allocates the storage that is large enough to hold largest variable type in the **union**. In the **union** declared above the member **x** requires 4 bytes which is largest among the members in 16-bit machine. Other members of **union** will share the same address.

Accessing a Union Member

Syntax for accessing **union** member is similar to accessing structure member,

```
union test
{
 int a;
 float b;
 char c;
}t;

t.a ;    //to access members of union t
t.b ;
t.c ;
```

Complete Example for Union

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

```
#include <conio.h>

union item
{
 int a;
 float b;
 char ch;
};

int main( )
{
 union item it;
 it.a = 12;
 it.b = 20.2;
 it.ch='z';
 clrscr();
 printf("%d\n",it.a);
 printf("%f\n",it.b);
 printf("%c\n",it.ch);
 getch();
 return 0;
}
```

Output :

```
-26426
20.1999
z
```

As you can see here, the values of **a** and **b** get corrupted and only variable **c** prints the expected result. Because in **union**, the only member whose value is currently stored will have the memory.

Exercise:

1. Implement a program to display a records of 10 student by using structure array variable.
2. Implement a program for Nested Structure for collage and department.
3. Implement a program which differentiate structure and union.
4. Implement a program of structure using typedef.
5. Implement a table which shows a field like this,sr.no, Player name, Total matches,Total Run,and avg.

8. File Handling in C Language

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 readymade 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 **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,

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

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 simplest functions used to read and write individual characters to a file.

```
#include<stdio.h>
#include<conio.h>
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()) != EOF)
    printf("%c",ch);
 fclose(fp);
}
```

Reading and Writing from File using fprintf() and fscanf()

```
#include<stdio.h>
#include<conio.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) );
  getch();
}
```

In this program, we have create 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 console usinf standard **printf()** function.

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.

Reading and Writing in a Binary File

A Binary file is similar to the 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.

9. Command Line Argument

Command line argument is a parameter supplied to the program when it is invoked. Command line argument is an important concept in C programming. It is mostly used when you need to control your program from outside. Command line arguments are passed to **main()** method.

Syntax :

int main(int argc, char *argv[])

Here **argc** counts the number of arguments on the command line and **argv[ ]** is a pointer array which holds pointers of type char which points to the arguments passed to the program.

Example for Command Line Argument

```
#include <stdio.h>
#include <conio.h>
int main( int argc, char *argv[] )
{
  int i;
  if( argc >= 2 )
   {
    printf("The arguments supplied are:\n");
    for(i=1;i< argc;i++)
     {
      printf("%s\t",argv[i]);
     }
   }
   else
   {
     printf("argument list is empty.\n");
   }
 getch();
 return 0;
}
```

Remember that **argv[0]** holds the name of the program and **argv[1]** points to the first command line argument and argv[n] gives the last argument. If no argument is supplied, argc will be one.

Exercise:

5.1. Implement a program which take input from user and display into a file.
5.2. Implement a program which count no of character, digits, vowel, consonants and symbol from a file.
5.3. Implement a program which copy the content of file into multiple files.
5.4. Implement a program to append the data into a file.
5.5. Implement a program to erase the content of file.

****