**********ALL THE BEST FOR UR EXAMS**********

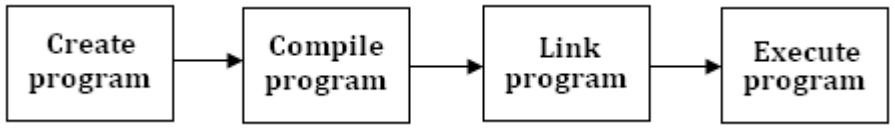
What is C?

C language is a general purpose and structured pragramming langauge developed by **"Dennis Ritchie" at AT &T's Bell Laboratories in the 1972s in USA.**
It is also called as "Procedure oriented programming language."
C is not specially designed for specific applications areas like COBOL (Common Business Oriented Language) or FORTRAN (Formula Translation). It is well suited for business and scietific applications. It has some various features like control structures, looping statements, arrays, macros required for these applications.
The C language has following numorous features as:

1. Portability
2. Flexiblility
3. effectiveness and efficiency
4. Reliability
5. interactivity

Execution of C program:

C program executes in following 4 (four steps).



1. **Creating a program :**

An editor like notepad or wordpad is used to create a C program. This file contains a source code which consists of executable code.The file should be saved as "filename.c" extension only.

2. **Compiling the program :**

The next step is to compile the program. The code is compiled by using compiler. Compiler converts executable code to binary code i.e. Object code. (cc filename.c or gcc filename.c or gcc -o filename filename.c or F9 dependence on plotform)

3. **Linking a program to library :**

The object code of a program is linked with libraries that are needed for execution of a program. The linker is used to link the program with libraries. It creates a file with '*.exe' extension.

4. **Execution of program :**

The final executable file is then run by dos command prompt or by any other software.

History of C:

Year of Establishment	Language Name	Developed by
1960	ALGOL-60	Cambridge University
1963	CPL(Combined Programming Language)	Cambridge University
1967	BCPL(Basic Combined Programming Language)	Martin Richard at Cambridge University
1970	B	Ken Thompson at AT & T's Bell Laboratories
1972	C	Dennis Ritchie at AT & T's Bell Laboratory

The development of C was a cause of evolution of programming languages like Algol 60, CPL (Combined Programming Langauge), BCPL (Basic Combined Programming Language) and B.

Algol-60 : (1963) :

ALGOL is an acronym for Algorithmic Language. It was the first structured Pointer procedural programming language, developed in the late 1950s and once widely used in Europe. But it was too abstract and too general structured langauage.

CPL : (1963) :

CPL is an acronym for Combined Programming Language.It was developed at Cambridge University.

BCPL : (1967) :

BCPL is an acronym for Basic Combined Programming Language. It was developed by Martin Richards at Cambridge University in 1967. BCPL was not so powerful. So, it was failed.

B : (1970) :

B language was developed by Ken Thompson at AT & T Bell Laboratories in 1970. It was machine dependent. So, it leads to specific problems.

C : (1972) :

'C' Programming Langauage was developed by Dennis Ritchie at AT & T Bell Laboratories in 1972. This is general purpose, compiled, structured programming langauage. Dennis Ritchie studied the BCPL, then improved and named it as 'C' which is the second letter of BCPL .

Structure of C Program:

The Basic Structure of C Program is as follows:

```
Document Section

Links Section(File)

Definition Section

Global Variables Declaration Section

void main( ){

variable declaration section

function declaration section

executable statements

}
```

```
Function definition 1

Function definition 2

------------------------
------------------------
------------------------
Function definition n
```

Where,

Document Section : It consists of set of comment lines which include name of a program, author name, creation date and other information.

Links Section (File) : It is used to link the required system libraries or header files to excute a program.

Definition Section : It is used to define or set values to variables.

Global variable declaration Section : It is used to declare global or public variable.

void main() : Used to start of actual C program. It includes two parts as declaration part and executable part.

Variable declaration section : Used to declare private variable.

Function declaration section : Used to declare functions of program from which we get required output. Then, executable statements are placed for execution.
Function definition section : Used to define functions which are to be called from main().

Character Set:

A character refers to the digit, alphabet or special symbol used to data represetation.

1. Alphabets : A-Z, a-z
2. Digits : 0-9
3. Special Characters : ~ ! @ # $ % ^ & * ( ) _ + { } [ ] - < > , . / ? \ | : ; " '
4. White Spaces : Horizontal tab, Carriage return, New line, form feed

Identifier :

Identifier is the name of a variable that is made up from combination of alphabets, digits and underscore.

Variable :

It is a data name which is used to store data and may change during program execution. It is opposite to constant. Variable name is a name given to memory cells location of a computer where data is stored.

Rules for varibales:

1. First character should be letter or alphabet.
2. Keywords are not allowed to use as a variable name.
3. White space is not allowed.
4. C is case sensitive i.e. UPPER and lower case are significant.
5. Only underscore, special symbol is allowed between two characters.
6. The length of indentifier may be upto 31 characters but only the first 8 characters are significant by compiler.
7. (Note: Some compilers allow variable names whose length may be upto 247 characters.
   But, it is recommended to use maximum 31 characters in variable name. Large variable name leads to occur errors.)

Keywords :

- Keywords are the system defined identifiers.
- All keywords have fixed meanings that do not change.
- White spaces are not allowed in keywords.

- Keyword may not be used as an indentifier.
- It is strongly recommended that keywords should be in lower case letters.
- There are totally 32(Thirty Two) keywords used in a C programming.

There are totally 32(Thirty Two) keywords used in a C programming.

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

Enumeration: **E**numeration allows you to define a list of aliases which represent integer numbers. Key word is enum.
Example:

```
#include <stdio.h>
enum boolean {FALSE=0, TRUE };
enum months  {Jan=5, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec};

void main() {
  enum months month;
  enum boolean mybool;
  printf("Month %d\n", month=Aug);
  printf("Bool %d\n", mybool=TRUE);
}
```

Escape Sequence Characters (Backslash Character Constants) in C:
C supports some special escape sequence characters that are used to do special tasks.These are also called as 'Backslash characters'.
Some of the escape sequence characters are as follow:

Character Constant	Meaning

\n	New line (Line break)
\b	Backspace
\t	Horizontal Tab
\f	Form feed
\a	Alert (alerts a bell)
\r	Carriage Return
\v	Vertical Tab
\?	Question Mark
\'	Single Quote
\"	Double Quote
\\	Backslash
\0	Null
%%	% character

Constants in C :

A constant is an entity that doesn't change during the execution of a program.
Followings are the different types of constants :

Real Constant :

1. It must have at least one digit.
2. It must have a decimal point which may be positive or negative.
3. Use of blank space and comma is not allowed between real constants.
4. Example:
   +194.143, -416.41

Integer Constant :

1. It must have at least one digit.
2. It should not contain a decimal place.
3. It can be positive or negative.
4. Use of blank space and comma is not allowed between real constants.
5. Example:

1990, 194, -394

Character Constant :

1. It is a single alphabet or a digit or a special symbol enclosed in a single quote.
2. Maximum length of a character constant is 1.
3. Example:
   'T', '9', '$'

String Constant :

1. It is collection of characters enclosed in double quotes.
2. It may contain letters, digits, special characters and blank space.
3. Example:
   "Technowell Web Solutions, Sangli"

Data Types in C :

"Data type can be defined as the type of data of variable or constant store."
When we use a variable in a program then we have to mention the type of data. This can be handled using data type in C.
Followings are the most commonly used data types in C.

Keyword	Formate Specifier	Size	Data Range
char	%c	1 byte	-128 to +127
unsigned char	<-- -->	8 bytes	0 t0 255
int	%d	2 bytes	-32768 to +32767
long int	%ld	4 bytes	-231 to +231
unsigned int	%u	2 bytes	0 to 65535
float	%f	4 bytes	-3.4e38 to +3.4e38
double	%lf	8 bytes	-1.7e38 to +1.7e38
long double	%Lf	12-16 bytes	-3.4e38 to +3.4e38

Qualifier :

When qualifier is applied to the data type then it changes its size or its size.
Size qualifiers : **short, long**
Sign qualifiers : **signed, unsigned**

Enum Data Type :

This is an user defined data type having finite set of enumeration constants. The keyword 'enum' is used to create enumerated data type.

Syntax:

enum [data_type] {const1, const2, ..., const n};

Example:

enum mca(software, web, seo);

Typedef :

It is used to create new data type. But it is commonly used to change existing data type with another name.

Syntax:

typedef [data_type] synonym; OR typedef [data_type] new_data_type;

Example:

typedef int integer;
integer rno;

Operators in C :

"Operator is a symbol that is used to perform mathematical operations."
When we use a variable in a program then we have to mention the type of data. This can be handled using data type in C. Followings are the most commonly used data types in C.

Operator Name	Operators
Assignment	=
Arithmetic	+, -, *, /, %
Logical	&&, ||, !
Relational	<, >, <=, >=, ==, !=
Shorthand or Compact	+=, -=, *=, /=, %=
Unary	++, --
Conditional	()?:
Bitwise	&, |, ^, <<, >>, ~

1.Assignement Operator: It is used to assign a value to variable.

Example:

```
#include <stdio.h>
#include <conio.h>
void main()
{
	int a,b;
	a = 53;
	printf("Value of A:%d\n",a);		// 53
	b = a;					// Interchange of value using assignment
	printf("Value of B:%d\n",b);		// 53
}
```

2.**Arithemetic Operator:**

It is also called as 'Binary operators'. It is used to perform arithmetical operations. These operators operate on two operands.

Example:

```
#include <stdio.h>
void main()
{
	int a,b,c,d,e,f,g;
	clrscr();
	printf("Enter First Number :");		// 5
	scanf("%d",&a);
	printf("Enter Second Number :");	// 2
	scanf("%d",&b);
	c = a + b;
	printf("Addition is : %d\n",c);		// 7
```

```
    d = a - b;
    printf("Subtraction is : %d\n",d);          // 3
    e = a * b;
    printf("Multiplication is : %d\n",e);       // 10
    f = a / b;
    printf("Division is : %d\n",f);             // 2
    g = a % b;
    printf("Modulus is : %d\n",g);              // 1
}
```

3.**Logical Operator :**

Sometimes, we have to check more than one condition at a time then it is operator which is primarily used to check more than two conditions. This operator returns 1 if condition is true otherwise 0. (In C, Every non-zero value is treated as TRUE and zero as FALSE while in conditions)

Example:

```
#include <stdio.h>
void main()
{
    int no1=2, no2=5;
        system(“clear”);
    printf("\n %d",(no1 && no2));          // returns 1
    printf("\n%d",(no1 || no2));           // returns 1
}
```

4.**Relational Operators:**

It is also used to check conditions. These operators return 1 if condition is true otherwise 0.

Example:

```
#include <stdio.h>
void main()
{
    int a=6, b=2;
```

```
    system(“clear”);
    printf("A<=B : %d\n",(a<=b));          // 0 - False
    printf("A>B : %d\n",(a>b));            // 1 - True
    printf("A!=B : %d\n",(a!=b));          // 1 - True
}
```

5.**Shorthand or compact Operators:**

It is used to perform mathematical operations at which the result or output can affect on operands.

Example:

```
#include <stdio.h>
void main()
{
    int a,b;
    a = 18;
    b = 4;
    printf("Value of A : %d\n",a);              // 18
    printf("Using of B : %d\n",b);              // 4
    b += a ;                                    // b = b + a
    printf("Using += (i.e b=b+a): %d",b);       // 22

    // Change the operator as -=, *=, /=, %=
}
```

6.**Unary Operators:**

It operates on a single operand. Therefore, this operator is called as 'unary operator.' It is used to increase or decrease the value of variable by 1.

Example:

```
#include <stdio.h>
void main()
{
```

```
    int a=4, b;
    printf("Value of A : %d\n",a);          // 4
    a++;                                    // Post increment of a
    printf("Value of A : %d\n",a);          // 5
    ++a;                                    // Pre increment of a
    printf("Value of A : %d\n",a);          // 6
    b=--a;                                  //Pre decrement of a
    printf("Value of A : %d\n",a);          // 5
    printf("Value of B : %d\n",b);          // 5
    b=a++;
    printf("Value of A : %d\n",a);          // 6
    printf("Value of B : %d\n",b);          // 5
    b++;
    printf("Value of B : %d\n",b);          // 6
}
```

Note:

Pre increment: First increments the value by 1 and then assigned that value for that operator.
Ex notation: ++a;
Post increment: First assigns the value for that operand and then increments the value by 1.
Ex notation: a++;
Pre decrement: First decremts the value by 1 and then assigned that value for that operand.
Ex notation: --a;
Post decrement: First assigns the value for that operand and then decrements the value by 1.
Ex notation: a--;

7.Terinary Operator(Conditional Operator):

Conditional operator is also called as 'ternary operator.' It is widely used to execute condition in true part or in false part. It operates on three operands. The logical or relational operator can be used to check conditions.

Example:

```
#include<stdio.h>
main()
{
        int a=5,b=3;
        system("clear");
        (a>b)?printf("Yes\n"):printf("No\n");                    \\Yes
}
```

8.Bitwise Operators:

Bitwise operators are used for manipulation of data at a bit level. They can be directly applied to char, short int and long.
We have some bitwise operators:

1. Bitwise complement Operator
2. Bitwise OR operator
3. Bitwise XOR operator
4. Bitwise AND operator
5. Bitwise left shift operator
6. Bitwise right shift operator

1.Bitwise complement operator:

This operator gives the complement form of a given number.This operator symble is ~(pronounced as tild).Bitwise complement operator is nothing but 1's complement or unary bitwise operator.Complement form of a positive number can be obtained by changing 0's as 1's and vice-versa.

Example:

x=10 ==> ~x=~(00001010)
==> ~x=11110101

This is equivalent decimal of -11.

Note:

1.If given number(Ex:x) is a +ve,then its complement is -(x+1).
2.If given number(Ex:x) is a -ve,then its complement is -(x+1).

2.Bitwise OR operator:

Bitwise OR operator or Bitwise inclusive OR operator performs OR operation on the bits of the numbers.This symble is | (pronounced as pipe).We know that the truth table for OR gate,the result is false when the both statements are false otherwise true.

Ex: 9 | 12.....means 9 Bitwise OR 12

9..... 0 0 0 0 1 0 0 1
12..... 0 0 0 0 1 1 0 0
---
9 | 12..... 0 0 0 0 1 1 0 1

which is equivalent to 13 of decimal.

3.**Bitwise XOR operator:**

Bitwise XOR operator or Bitwise exclusive OR operator performs XOR operation on the bits of the numbers.This symble is ^ (pronounced as cap).We know that the truth table for XOR gate,the result is true when the both statements are false or both statements are true otherwise false.

Ex: 9^12.....means 9 Bitwise XOR 12

9..... 0 0 0 0 1 0 0 1
12..... 0 0 0 0 1 1 0 0

9^12..... 0 0 0 0 0 1 0 1

which is equivalent to 5 of decimal.

4.**Bitwise AND operator:**

Bitwise AND operator performs AND operation on the bits of the numbers.This symble is &(pronounced as ampersand).We

know that the truth table for AND gate,the result is true when the both statements are true otherwise false.
Ex: 9&12.....means 9 Bitwise AND 12

9..... 0 0 0 0 1 0 0 1
12..... 0 0 0 0 1 1 0 0

9&12..... 0 0 0 0 1 0 0 0

which is equivalent to 8 of decimal.

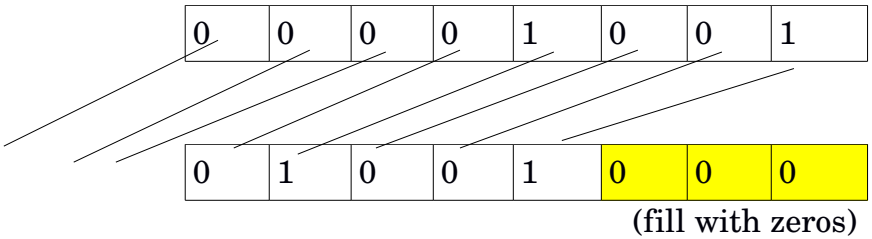
5.Bitwise left shift operator:

Bitwise left shift operator performs left shift operation on the bits of the given number.This symble is <<(pronounced as double less than).

Ex: x<<n.....means to shift n positions of bits of x towards left.

If x=9.....

x<<3.....



which is equal to 72.
Note: you can simply calculate the answer like **x*pow(2,n).**
here is 9<<3.....9*pow(2,3)=9*8=72.

6.Bitwise right shift operator:

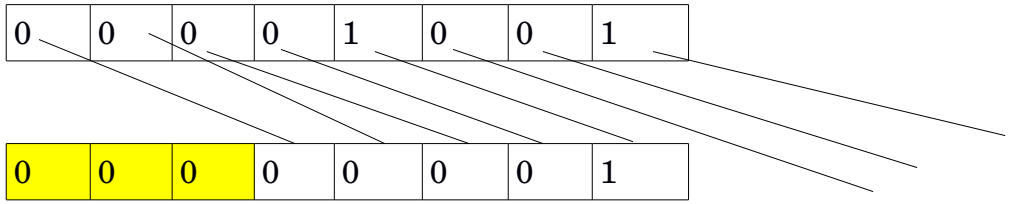
Bitwise right shift operator performs right shift operation on the bits of the given number.This symble is >>(pronounced as

double greater than).

Ex: x>>n.....means to shift n positions of bits of x towards right.

If x=9.....

x>>3.....



(fill with zeros)

which is equal to 1.

Note: you can simply calculate the answer like **x/pow(2,n). (i.e.,quotient)**
here is 9<<3.....9/pow(2,3)=9/8=1.

Operators Precedence and Associativity :

In C, each and every operator has a spcial precedence which is associated with it. There are various levels of precedence. This precedence is especially used to determine to evaluation of expression which has more than one operator in it. The operators which has higher precedence are executed first and vice-versa. Operators which has same precedence level are evaluated from left to right. It is dependant on it's level. This feature is well known as 'Associativity of an operator.'

Associativity	Operators	Description
Left To Right	( )	Function
	[ ]	Array
	-->	Pointer to member
	.(dot)	Structure member
Right To Left	-	Unary Minus
	+	Unary Plus
	++ / --	Increment/Decrement
	~	One's Complement
	&	Address of
	(type)	Type casting
	sizeof	size (in bytes)
	!	Logical Not
	*	Poiner Reference
Left To Right	*	Multiplication
	/	Division
	%	Modulus
Left To Right	+	Addition
	-	Subtraction
Left To Right	<<	Left Shift
	>>	Right Shift
Left To Right	<	Less than
	<=	Less than or equal
	>	Greater
	>=	Greater than or equal
Left To Right	==	Equality
	!=	Not equal
Left To Right	&	Bitwise AND
Left To Right	^	Bitwise XOR
Left To Right	|	Bitwise OR
Left To Right	&&	Logical AND
Left To Right	||	Logical OR
Left To Right	?:	Conditional(Terinary Operator)
Right To Left	=,*=,+=	Assignment
Left To Right	,(comma)	comma

Decision Making Statements / Conditional Statements :

C program executes program sequentially. Sometimes, a program requires checking of certain conditions in program execution. C provides various key condition statements to check condition and execute statements according conditional criteria.
These statements are called as "Decision Making Statements" or "Conditional Statements."
Followings are the different conditional statements used in C.

1. If Statement
2. If-Else Statement
3. Nested If-Else Statement

4. Switch Case

If Statement :

This is a conditional statement used in C to check condition or to control the flow of execution of statements. This is also called as 'decision making statement or control statement.' The execution of a whole program is done in one direction only.

Syntax:

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

In above syntax, the condition is checked first. If it is true, then the program control flow goes inside the braces and executes the block of statements associated with it. If it returns false, then program skips the braces. If there are more than 1 (one) statements in if statement then use { } braces else it is not necessary to use.

Program:

```
#include <stdio.h>
#include <conio.h>
void main()
{
int a;
a=5;
if(a>4)
printf("\nValue of A is greater than 4 !");
if(a==4)
printf("\n Value of A is 4 !");
}
```

Output :

Value of A is greater than 4 !

If-Else Statement :

This is also one of the most useful conditional statement used in C to check conditions.

Syntax:

```
if(condition)
{
true statements;
}
else
{
false statements;
}
```

In above syntax, the condition is checked first. If it is true, then the program control flow goes inside the braces and executes the block of statements associated with it. If it returns false, then it executes the else part of a program.

Program:

```
#include <stdio.h>
#include <conio.h>
void main()
{
int no;
printf("\n Enter Number :");
scanf("%d",&no);
if(no%2==0)
printf("Number is even !\n");
else
printf("Number is odd !\n");
}
```

Output :

Enter Number : 11
Number is odd !

Nested If-Else Statement :

It is a conditional statement which is used when we want to check more than 1 conditions at a time in a same program. The

conditions are executed from top to bottom hecking each condition whether it meets the conditional criteria or not. If it found the condition is true then it executes the block of associated statements of true part else it goes to next condition to execute.

Syntax:

```
if(condition)
{
if(condition)
{
statements;
}
else
{
statements;
}
}
else
{
statements;
}
```

In above syntax, the condition is checked first. If it is true, then the program control flow goes inside the braces and again checks the next condition. If it is true then it executes the block of statements associated with it else executes else part.

Program:

```
#include <stdio.h>
#include <conio.h>
void main()
{
int no;
printf("\n Enter Number :");
scanf("%d",&no);
if(no>0)
{
```

```
printf("\n\n Number is greater than 0 !");
}
else
{
if(no==0)
{
printf("\n\n It is 0 !");
}
else
{
printf("Number is less than 0 !");
}
}
}
```

Output :

Enter Number : 0
It is 0 !

Switch case Statement :

This is a multiple or multiway brancing decision making statement.
When we use nested if-else statement to check more than 1 conditions then the complexity of a program increases in case of a lot of conditions. Thus, the program is difficult to read and maintain. So to overcome this problem, C provides 'switch case'.
Switch case checks the value of a expression against a case values, if condition matches the case values then the control is transferred to that point.

Syntax:

```
switch(expression)
{
case expr1:
statements;
break;
case expr2:
```

```
statements;
break;
' ' ' ' ' ' ' ' ' ' ' ' ' ' ' '
' ' ' ' ' ' ' ' ' ' ' ' ' ' ' '
case exprn:
statements;
break;
default:
statements;
}
```

In above syntax, switch, case, break are keywords.
expr1, expr2 are known as 'case labels.'
Statements inside case expression need not to be closed in braces.
Break statement causes an exit from switch statement.
Default case is optional case. When neither any match found, it executes.

Program:

```
#include <stdio.h>
#include <conio.h>
void main()
{
int no;
printf("\n Enter any number from 1 to 3 :");
scanf("%d",&no);
switch(no)
{
case 1:
printf("\n It is 1 !");
break;
case 2:
printf("\n It is 2 !");
break;
```

```
case 3:
printf("\n It is 3 !");
break;
default:
printf("\n Invalid number !");
}
}
```

Output 1 :

Enter any number from 1 to 3 : 3
It is 3 !

Output 2 :

Enter any number from 1 to 3 : 5
Invalid number !

Rules for Declaration of switch case:

- The case label should be integer or character constant.
- Each compound statement of a switch case should contain break statement to exit from case.
- Case labels must end with (:) colon.

Advantages of switch case:

- Easy to use.
- Easy to find out errors.
- Debugging is made easy in switch case.
- Complexity of a program is minimized.

Looping Statements / Iterative Statements :

'A loop' is a part of code of a program which is executed repeatedly.
A loop is used using condition. The repetition is done until condition becomes true.
A loop declaration and execution can be done in following ways.

- Check condition to start a loop
- Initialize loop with declaring a variable.
- Executing statements inside loop.
- Increment or decrement of value of a variable.

Types of looping statements:

Basically, the types of looping statements depends on the condition checking mode. Condition checking can be made in two ways as : Before loop and after loop. So, there are 2(two) types of looping statements.

1. Entry controlled loop
2. Exit controlled loop

1. Entry controlled loop :

In such type of loop, the test condition is checked first before the loop is executed.
Some common examples of this looping statements are :

- while loop
- for loop

2. Exit controlled loop :

In such type of loop, the loop is executed first. Then condition is checked after block of statements are executed. The loop executed atleat one time compulsarily.
Some common example of this looping statement is :

- do-while loop

While loop :

This is an entry controlled looping statement. It is used to repeat a block of statements until condition becomes true.

Syntax:

```
while(condition)
{
statements;
increment/decrement;
}
```

In above syntax, the condition is checked first. If it is true, then the program control flow goes inside the loop and executes the block of statements associated with it. At the end of loop increment or decrement is done to change in variable value. This process continues until test condition satisfies.

Program:

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

```
void main()
{
int a;
a=1;
while(a<=5)
{
printf("\n Hi");
a+=1;          // i.e. a = a + 1
}
}
```

Output :

Hi
Hi
Hi
Hi
Hi

For loop :

This is an entry controlled looping statement.
In this loop structure, more than one variable can be initilized. One of the most important feature of this loop is that the three actions can be taken at a time like variable initilisation, condition checking and increment/decrement. The for loop can be more concise and flexible than that of while and do-while loops.

Syntax:

```
for(initialisation; test-condition; incre/decre)
{
statements;
}
```

In above syntax, the given three expressions are seperated by ';' (Semicolon)

Features :

- More concise

- Easy to use
- Highly flexible
- More than one variable can be initilized.
- More than one increments can be applied.
- More than two conditions can be used.

Program:

```
#include <stdio.h>
#include <conio.h>
void main()
{
        int a;
        for(i=0; i<5; i++)
        {
                printf("\nHi");
        }
}
```

Output :

Hi
Hi
Hi
Hi
Hi

Do-While loop :

This is an exit controlled looping statement.
Sometimes, there is need to execute a block of statements first then to check condition. At that time such type of a loop is used. In this, block of statements are executed first and then condition is checked.

Syntax:

```
do
{
```

statements;
(increment/decrement);
}while(condition);
In above syntax, the first the block of statements are executed. At the end of loop, while statement is executed. If the resultant condition is true then program control goes to evaluate the body of a loop once again. This process continues till condition becomes true. When it becomes false, then the loop terminates.

Note: The while statement should be terminated with ; (semicolon).

Program:

```
#include <stdio.h>
#include <conio.h>
void main()
{
int a;
a=1;
do
{
printf("\nHi"); // 5 times
a+=1;
// i.e. a = a + 1
}while(a<=5);
a=6;
do
{
printf("\nBye");              // 1 time
a+=1;
// i.e. a = a + 1
}while(a<=5);
}
```

Output:

Hi
Hi
Hi
Hi
Hi
Bye

Break Statement :

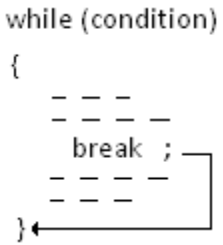
It is used to exit from a loop or a switch, control passing to the first statement beyond the loop or a switch.

With loops, break can be used to force an early exit from the loop, or to implement a loop with a test to exit in the middle of the loop body. A break within a loop should always be protected within an if statement which provides the test to control the exit condition.The statements after break statement are skipped.

Syntax:

break;

Figure:



Program:

```
#include <stdio.h>
#include <conio.h>
void main()
{
int i;
for(i=1; ; i++)
{
```

```
if(i>5)
break;
printf("%d\t",i);
}
}
```

Output:

1 2 3 4 5

Continue Statement :

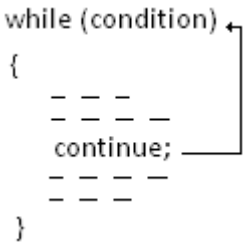
Sometimes, it is required to skip a part of a body of loop under specific conditions. So, C supports 'continue' statement to overcome this anomaly.

The working structure of 'continue' is similar as that of that break statement but difference is that it cannot terminate the loop. It causes the loop to be continued with next iteration after skipping statements in between. Continue statement simply skipps statements and continues next iteration.

Syntax :

continue;

Figure:



Program:

```
#include <stdio.h>
void main()
{
int i;
for(i=1; i<=10; i++)
{
if(i==6)
continue;
```

```
printf("\n %d",i);
}
}
```

Output :

1
2
3
4
5
7
8
9
10

Goto Statement :

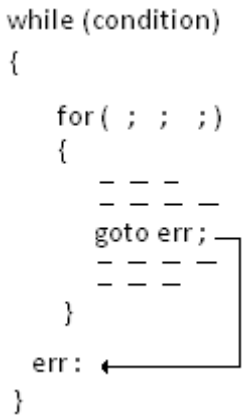
It is a well known as 'jumping statement.' It is primarily used to transfer the control of execution to any place in a program. It is useful to provide branching within a loop.

When the loops are deeply nested at that if an error occurs then it is difficult to get exited from such loops. Simple break statement cannot work here properly. In this situations, goto statement is used.

Syntax :

goto [expr];

Figure :



Program:

```
#include <stdio.h>
#include <conio.h>
void main()
{
int i=1, j;
while(i<=3)
{
for(j=1; j<=3; j++)
{
printf(" * ");
if(j==2)
goto stop;
}
i = i + 1;
}
stop:
printf("\n Exited !");
}
```

Output :

```
* *
Exited
```

Functions:

1. Functions
2. Types of Functions:
    - Built in functions
    - User defined functions

3. Types of function definition or Types of function declaration
4. Categories of functions
5. Parameter passing mechanism
6. Advantages
7. Recursion

Functions in C :

The function is a self contained block of statements which performs a coherent task of a same kind.
C program does not execute the functions directly. It is required to invoke or call that functions. When a function is called in a program then program control goes to the function body. Then, it executes the statements which are involved in a function body. Therefore, it is possible to call fuction whenever we want to process that functions statements.

Types of functions :

There are 2(two) types of functions as:
1. Built in Functions
2. User Defined Functions

1.Built in Functions :

These functions are also called as 'library functions'. These functions are provided by system. These functions are stored in library files.

Examples:

scanf()
printf()
strcpy ()
tolower ()
toupper()
strcmp ()
strlen ()
strcat ()

2.User Defined Functions :

The functions which are created by user for program are known as 'User defined

functions'.

Syntax:

```
void main()
{

<return_type><function_name>([<argu_list>]);              // Function prototype

<function_name>([<arguments>]);                           // Function Call
}

<return_type> <function_name>([<argu_list>])              // Function definition
{
<function_body>;
}
```

Example Program:

```
#include <stdio.h>
#include <conio.h>
void add()
{
int a, b, c;
printf("\n Enter Any 2 Numbers : ");
scanf("%d %d",&a,&b);
c = a + b;
printf("\n Addition is : %d",c);
}
void main()
{
void add();
```

```
add();
}
```
Output :

```
Enter Any 2 Numbers : 23 6
Addition is : 29
```

Advantages :

- It is easy to use.
- Debugging is more suitable for programs.
- It reduces the size of a program.
- It is easy to understand the actual logic of a program.
- Highly suited in case of large programs.
- By using functions in a program, it is possible to construct modular and structured programs.
- Reusability

Types of function definition or Types of function declaration:

There are different methods to define functions. The following two ways are used in defining functions:

1. Non-ANSI style
2. ANSI style

Non-ANSI style:

The general form of a Non-ANSI function definition is as follows.

```
type name_of_function (parameter_list)
parameter definition;
{
variable declaration;                    //with in the function
statement1;
statement2;
...
...
```

```
...
return(value_computed);          //returns(sends) the output result to calling function
}
```

where,

type.....datatype of return value
name_of_function.....user defined function
parameter_list.....parameters received from calling function
parameterdefinition.....type declaration of parameters of parameter_list

Note: All the statements that are placed between the left brace and the corresponding right brace would constitute the body of a function.

Example: To calculate sum of two integers by using functions concept(Non-ANSI style).

```
#include<stdio.h>
main(){
int a,b,sum;
scanf("%d%d",&a,&b);
sum=add(a,b);                    //passing parameters to calling function (add)
printf("sum=%d",sum);
}
int add(x,y)                     //called function
int x,y;
{
int k;
k=x+y;
return k;
}
```

ANSI style:

The general form of an ANSI function definition is as follows.

```
type name_of_function (parameter definition)
{
```

```
    variable declaration;                   //with in the function
    statement1;
    statement2;
    ...
    ...
    ...
    return(value_computed);         //returns(sends) the output result to calling function
    }
```

where,

type.....datatype of return value
name_of_function.....user defined function
parameterdefinition.....type declaration of parameters of parameter_list

Example: To calculate sum of two integers by using functions concept(ANSI style).

```
    #include<stdio.h>
    main(){
    int a,b,sum;
    scanf("%d%d",&a,&b);
    sum=add(a,b);                   //passing parameters to calling function (add)
    printf("sum=%d",sum);
    }
    int add(int x,int y)            //called function
    {
    int k;
    k=x+y;
    return k;
    }
```

Note1:

In the Non-ANSI style, only a list of parameters was given and all these parameters were separated by a comma. But, in the ANSI style the parameter list is replaced by the parameter definition.

Note2:

Most of today's compilers use the ANSI style. Therefore, it is necessary for the programmers to write the programs using the ANSI style of function definition.

Categories of function:

We have some other types of functions where the arguments and the return value may be present or absent. Such functions can be categorized into:

1. No arguments and no return value
2. Arguments but no return value
3. No arguments but return value
4. Arguments and return value

1.Functions with No Arguments and No Return values :

Here, the called function does not receive any data from the calling function. And, it does not return any value back to the calling function. Hence there is no data transfer between the calling function and the called function.

Example:

```
//Write a C Program to illustrate the func&on with no arguments and no return value
#include<stdio.h>
main()
{
        read_value();                           //calling function
}
read_value()                                    // No return value and is a called function
{
        char name[10];
        printf("Enter your name:\n");
        scanf("%s",name);
        printf("your name is %s\n",name);
}
```

2. **Functions with Arguments but No Return Value :**

Here, the called function receives the data from the calling function. The arguments and parameters should match in number, data type and order. But, the called function does not return any value back to the calling function.Instead, it prints the data in its scope only. It is a one-way data communication between the calling function and the called function.

Example:

```
#include<stdio.h>
main(){
int a,b,sum;
scanf("%d%d",&a,&b);
add(a,b);
}
int add(int x,int y)                    //called function
{
int k;
k=x+y;
printf("sum=%d",k);
}
```

3.Funcitons with No Arguments and Return Values :

Here, the called function does not receive any data from the calling function. It manages with its local data to perform the specified taks. However, after the specified processing the called function returns the computed data to the calling function. It is also a one-way data communication between the calling function and the called function.

Example:

```
#include<stdio.h>
main(){
int sum;
sum=add();
printf("sum=%d",sum);
}
```

```
int add()                    //called function
{
int a,b,k;
scanf("%d%d",&a,&b);
k=a+b;
return k;
}
```

4.Arguments and return value:

Here, the called function receives data from the calling function.After the specified processing the called function returns the computed data to the calling function. It is a two-way data communication between the calling function and the called function.

Example:

```
#include<stdio.h>
main(){
int a,b,sum;
scanf("%d%d",&a,&b);
sum=add(a,b);
printf("sum=%d",sum);
}
int add(int x,int y)
{
int k;
k=x+y;
return k;
}
```

Parameter passing mechanism:

The process of transmtting the values from one function to other is known as parameter passing. There are two methods of parameter passing

1. Call by value
2. Call by reference

1.**Call by value:**

when the values of the arguments are passed from a calling function to a called function the values are copied into the called function.If any changes are made to the values in the called function, there will not be any change in the orginal values within the calling function.

Example:

```
#include<stdio.h>
#include<math.h>
main()
{
int a,b,x;
int hi(int x,int y);
a=10;b=20;
printf("a=%d\tb=%d\n",a,b);
x=hi(a,b);
printf("a=%d\tb=%d\n",a,b);
printf("x=%d\n",x);
}
int hi(int x,int y)
{
int sum;
sum=x+y;
x=pow(x,2);
y=sqrt(x);
printf("x=%d\ty=%d\n",x,y);
return sum;
}
```

Output:

a=10 b=20
x=100 y=10
a=10 b=20
x=30

Note: There is no change in the values of a and b before and after the function execution .

2. Call-by-Reference :

In this method, the actual values are not passed, instead of their values addresses are passed to a calling function. Here, no values are copied as the memory locations themselves are referenced. If any modification is made to the values in the called function, then the orginal values will get changed within the calling function. Passing of addresses requires the knowledge of pointers.

Note: When the pointers are passed as arguments, the following two points must be considered.

1.In the calling function, the function is invoked with the function name and addresses of actual parameters enclosed within the parentheses.

i.e., function_name(&var1, &var2, ..., &varn);

where, var1, var2, ... , varn are actual parameters(arguments)

2.In the parameter list of the called function each and every formal parameter (pointers) must be preceded by an indirection operator (*).

i.e. data_type function_name(*var1, *var2, . . . *varn) ;

Example:

```
#include<stdio.h>
main()
{
int a,b;
int hi(int *x,int *y);                                          //function prototype
a=10;b=20;
printf("--------Before calling function----------\n\ta=%d\tb=%d\n",a,b);   //Before calling function
hi(&a,&b);                                                      //function call
printf("--------After calling function----------\n\ta=%d\tb=%d\n",a,b);    //after calling function
}
int hi(int *x,int *y)
```

```
{
int temp;
temp=*x;
*x=*y;
*y=temp;
}
```

Output:

```
--------Before calling function----------
        a=10   b=20
--------After calling function----------
        a=20   b=10
```

Recursion:

When a function of body calls the same function then it is called as 'recursive function.'

Example:

```
Recursion()
        {
                printf("Congratulations\n");
                Recursion();
        }
```

The two basic phases of a recursive process:

- winding(recursive or tail part)
- unwinding(basic or terminating part).

In the winding phase, each recursive call perpetuates the recursion by making anadditional recursive call itself. The winding phase terminates when one of the calls reaches a terminating condition.

A terminating condition defines the state at which a recursive function should return instead of making another recursive call . Every recursive function must have at least one terminating condition otherwise, the winding phase never terminates. Once the winding phase is complete, the process enters the unwinding phase, in which previous instances of the function are revisited in reverse order. This phase continues until the original call returns, at which point the recursive process is complete.

Note: In the recursion process we use the stack principle(Last-In-First-Out) to obtain the result.

Types of Recursion:

1. Basic Recursion
2. Tail Recursion

Example for Basic Recursion:

```
#include<stdio.h>
main(){
int a,f;
printf("Enter a no: ");
scanf("%d",&a);
f=fact(a);
printf("%d\n",f);
}
fact(int x){
        if(x==0)
                return 1;
        else
                return x*fact(x-1);

}
```

Tail Recursion:

A recursive function is said to be tail recursive if all recursive calls within it are tail recursive. A recursive call is tail recursive when it is the last statement that will be executed within the body of a function and its return value is not a part of an expression. Tail-recursive functions are characterized as having nothing to do during the unwinding phase.

Example for Tail Recursion:

```
#include<stdio.h>
main(){
int a,f,b=1;
printf("Enter a no: ");
scanf("%d",&a);
```

```
f=fact(a,b);
printf("%d\n",f);
}
fact(int x,int y){
        if(x==0)
                return y;
        else
                return fact(x-1,x*y);

}
```

Features :

1. There should be at least one if statement used to terminate recursion.
2. It does not contain any looping statements.

Advantages :

1. It is easy to use.
2. It represents compact programming strctures.

Disadvantages :

1. It is slower than that of looping statements because each time function is called.

Note : It can be applied to calculate factorial of a number, fibonancci series,GCD of integers.

Array:

Basically array is a static datastructure.It is a collection of homogenous data stored under unique name. The values in an array is called as 'elements of an array.' These elements are accessed by numbers called as 'subscripts or index numbers.' Arrays may be of any variable type. Array is also called as 'subscripted variable.'Arrays of any type can be formed in C.

Syntax:

```
datatype Array_name[dimension or length];
```

Arrays can be created from any of the C data types int, float, char.In C, arrays starts at position 0. The elements of the array occupy adjacent locations in memory.

Unsized array initialisation:

If unsized arrays are declared, the C compiler automatically creates an array big enough to hold all the initializers. This is called an unsized array.

Example declaration/initializations are as follows:

```
char e1[] = "read error\n";
char e2[] = "write error\n";

int sgrs[][2] =
{
  1,1,
  2,4,
  3,9
  4,16,
};
```

Array Initialization:

Arrays may be initialized at the time of declaration.

The following example initializes a ten-element integer array:

```
 int i[10] = { 1,2,3,4,5,6,7,8,9,10 };
```

another Example:

```
char str[9] = "I like C"; (or)

char str[9] = { 'I',' ','l','i','k','e',' ','C','\0' };
```

When the string constant method is used, the compiler automatically supplies the null terminator.

Multi-dimensional arrays are initialized in the same way as single-dimension arrays, e.g.:

```
 int sgrs[6][2] =

{
1,1,
2,4,
3,9,
4,16,
5,25,
6,36
};
```

Types of an Array :

1. Single Dimensional Array or 1D array
2. Multiple Dimensional Array (which may 2D,3D,.....nD,etc. Array)

Single dimensional array:

The array which is used to represent and store data in a linear form is called as “single or one dimensional array.”

Syntax:

```
<data-type> <array_name> [size];
```

Example:

```
int a[3] = {2, 3, 5};
char ch[20] = "TechnoExam" ;
float stax[3] = {5003.23, 1940.32, 123.20} ;
```

Total size(in bytes):

total size = length of array * size of data type;

Example:

In above example, 'a' is an array of type integer which has storage size of 3 elements. The total size would be 3 * 4 = 12 bytes

Memory Allocation :

Memory allocation for one dimensional array is given below

Example Program: Calulate sum of the given integers.

```
#include<stdio.h>
main()
{
        int i,n;
        printf("Enter the length of an array:");
        scanf("%d",&n);
        int array[n],k;
        printf("Enter array elements...\n");
        for(i=0;i<n;i++)
        {
                scanf("%d",&k);
                array[i]=k;                     //Reading array elements
        }
        int sum=0;
        for(i=0;i<n;i++)
        {
                sum=sum+array[i];               //Adding array elements
        }
        printf("sum=%d\n",sum);
```

}

Features :

- Array size should be positive number only.
- String array always terminates with null character ('\0').
- Array elements are countered from 0 to n-1.
- Useful for multiple reading of elements (numbers).

Disadvantages :

- There is no easy method to initialize large number of array elements.
- It is difficult to initialize selected elements.

Two Dimensional Array :

The array which is used to represent and store data in a tabular form is called as 'two dimensional array.' Such type of array specially used to represent data in a matrix form.

Syntax:

<data-type> <array_name> [no.rows][no.columns];

Total size(in bytes):

Total size = no.rows*no.columns * size of data type;

Example:

int a[3][4];

In above example, a is an array of type integer which has storage size of 3 * 4 matrix. The total size would be 3 * 4* 4 = 48 bytes.

It is also called as 'multidimensional array.'

	column0(j=0)	column1(j=1)	column2(j=2)	column3(j=3)
row0(i=0)	A[0][0]	A[0][1]	A[0][2]	A[0][3]
row1(i=1)	A[1][0]	A[1][1]	A[1][2]	A[1][3]
row2(i=2)	A[2][0]	A[2][1]	A[2][2]	A[2][3]

Note: Where A[i][j].....Represents the element which is in the position i and j.

Example: Read and display a 3*3 matrix

Program:

```
#include<stdio.h>
main(){
        system("clear");
        int i,j;
        int matrix[3][3];
        printf(".....Enter the array elements.....\n");
        for(i=0;i<3;i++)
        {
                for(j=0;j<3;j++)
                {
                        scanf("%d",&matrix[i][j]);
                }
        }
```

```
        printf("........The given matrix is.........:\n");
        for(i=0;i<3;i++)
        {
                for(j=0;j<3;j++)
                {
                        printf("%8d",matrix[i][j]);
                }
                printf("\n");
        }
}
```

Iutput:

.....Enter the array elements.....

4

9

2

5

2

9

4

2

8

Output:

.....The given matrix is.....:

4 9 2

5 2 9

4 2 8

Limitations of two dimensional array :

- We cannot delete any element from an array.
- If we dont know that how many elements have to be stored in a memory in advance, then there will be memory wastage if large array size is specified.

Note: The dimensions of a array is depends on the number of pair of brackets which are representing in the array declaration.

Ex:if we have 4 pair of brackets in the array declaration i.e.,the 4D array.

Structure :

A structure is a collection of one or more variables, possibly of different data types, grouped together under a single name for convenient handling.Keyword 'struct' is used to declare structure.

The variables which are declared inside the structure are called as 'members of structure'.

Syntax:

```
struct structure_nm
{
      <data-type> element 1;
      <data-type> element 2;
      - - - - - - - - - - -
      - - - - - - - - - - -
      <data-type> element n;
}struct_var;
```

Example :

```
struct emp_info
{
        char emp_id[10];
        char nm[100];
        float sal;
}emp;
```

Note :

1. Structure is always terminated with semicolon (;).asa

2. Structure name as emp_info can be later used to declare structure variables of its type in a program.

Instances of Structure :

Instances of structure can be created in two ways as,

Instance 1:

```
struct emp_info
{
        char emp_id[10];
        char nm[100];
        float sal;
}emp;
```

Instance 2:

```
struct emp_info
{
        char emp_id[10];
        char nm[100];
        float sal;
```

```
};
struct emp_info emp;
```

Here, emp_info is a simple structure which consists of stucture members as Employee ID(emp_id), Employee Name(nm), Employee Salary(sal).

Aceessing Structure Members :

Structure members can be accessed using member operator '.' . It is also called as 'dot operator' or 'period operator'.

Syntax:

```
structure_var.member;
```

Program:

```
#include <stdio.h>

struct stud_info

{

    char nm[100];

    char id[100];

}info;

void main()

{
```

```
        system("clear");
          printf("Input:\n");
        printf("\n Enter Student Name : ");
        gets(info.nm);
        printf("\n Enter Id : ");
        gets(info.id);
          printf("Output:\n");
        printf(" Student Name : %s\n",info.nm);
        printf("Student Id: %s\n",info.id);
}
```

Input:

Enter Student Name : XXX

Enter Id : R121000

Output:

Student Name : XXX

Student Id : R121000

Structure With Array :

We can create structures with array for ease of operations in case of getting multiple same fields.

Program :

```
#include <stdio.h>

struct emp_info
{
	int emp_id;
	char nm[50];
}emp[2];

void main()
{
	int i;
	for(i=0;i<2;i++)
	{
		printf("\nEnter Employee ID: ");
		scanf("%d",&emp[i].emp_id);
		printf("\n Employee Name : ");
		scanf("%s",emp[i].nm);
	}
	for(i=0;i<2;i++)
	{
	  printf("\n\t Employee ID : %d",emp[i].emp_id);
	  printf("\n\t Employee Name : %s",emp[i].nm);
	}
}
```

Array in Structures :

Sometimes, it is necessary to use structure members with array.

Program :

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

struct result
{
	int rno, mrks[5];
	char nm;
}res;

void main()
{
	int i,total=0;
	printf("\n\t Enter Roll Number : ");
	scanf("%d",&res.rno);
	printf("\n\t Enter Marks of 3 Subjects : ");
	for(i=0;i<3;i++)
	{
		scanf("%d",&res.mrks[i]);
		total = total + res.mrks[i];
	}
	printf("\nRoll Number : %d",res.rno);
	printf("\nObtained Marks are :");
	for(i=0;i<3;i++)
	{
		printf(" %d",res.mrks[i]);
	}
```

```
    printf("\nTotal is : %d",total);
}
```

Structures within Structures (Nested Structures) :

Structures can be used as structures within structures. It is also called as 'nesting of structures'.The member operator . is used to access members of structures that are themselves members of a larger structure. No parentheses are needed to force a special order of evaluation; a member operator expression is simply evaluated from left to right.

Syntax:

```
struct structure_nm
{
    <data-type> element 1;
    <data-type> element 2;
    - - - - - - - - - - -
    - - - - - - - - - - -
    <data-type> element n;

    struct structure_nm
    {
        <data-type> element 1;
        <data-type> element 2;
        - - - - - - - - - - -
        - - - - - - - - - - -
        <data-type> element n;
    }inner_struct_var;
}outer_struct_var;
```

Example :

```
struct stud_Res
{
```

```
    int rno;
    char nm[50];
    char std[10];

    struct stud_subj
    {
        char subjnm[30];
        int marks;
    }subj;
}result;
```

In above example, the structure stud_Res consists of stud_subj which itself is a structure with two members. Structure stud_Res is called as 'outer structure' while stud_subj is called as 'inner structure.' The members which are inside the inner structure can be accessed as follow :

```
result.subj.subjnm
result.subj.marks
```

Program :

```
#include <stdio.h>
#include <conio.h>
struct stud_Res
{
    int rno;
    char std[10];
    struct stud_Marks
    {
        char subj_nm[30];
        int subj_mark;
    }marks;
}result;
```

```
void main()
{
        printf("\nEnter Roll Number : ");
        scanf("%d",&result.rno);
        printf("\nEnter Standard : ");
        scanf("%s",result.std);
        printf("\n Enter Subject Codeor name : ");
        scanf("%s",result.marks.subj_nm);
        printf("\nEnter Marks : ");
        scanf("%d",&result.marks.subj_mark);
        printf("\n\nRoll Number : %d",result.rno);
        printf("\nStandard : %s",result.std);
        printf("\nSubject Code : %s",result.marks.subj_nm);
        printf("\nMarks : %d",result.marks.subj_mark);
}
```

Additional Features of Structures :

1) The values of a structure variable can be assigned to another structure variable of the same type using the assignment operator. It is not necessary to copy the structure elements piece-meal. Obviously, programmers prefer assignment to piece-meal copying**.**

2) One structure can be nested within another structure. Using this facility complex data types can be created.

3) Like an ordinary variable, a structure variable can also be passed to a function. We may either pass individual structure elements or the entire structure variable at one go.

4) The way we can have a pointer pointing to an int, or a pointer pointing to a char, similarly we can have a pointer pointing to

a struct. Such pointers are known as ‘structure pointers’.

Summary of structure:

(a) A structure is usually used when we wish to store dissimilar data together.

(b) Structure elements can be accessed through a structure variable using a dot (.) operator.

(c) Structure elements can be accessed through a pointer to a structure using the arrow (->) operator.

(d) All elements of one structure variable can be assigned to another structure variable using the assignment (=) operator.

(e) It is possible to pass a structure variable to a function either by value or by address.

(f) It is possible to create an array of structures.

Storage Class :

'Storage' refers to the scope of a variable and memory allocated by compiler to store that variable. Scope of a variable is the boundary within which a variable can be used. Storage class defines the scope and lifetime of a variable.

From the point view of C compiler, a variable name identifies physical location from a computer where varaible is stored. There are two memory locations in a computer system where variables are stored as : Memory and CPU Registers.

Functions of storage class :

- To detemine the location of a variable where it is stored ?
- Set initial value of a variable or if not specified then setting it to default value.
- Defining scope of a variable.
- To determine the life of a variable.

Types of Storage Classes :

Storage classes are categorised in 4 (four) types as:

1. Automatic Storage Class
2. Register Storage Class
3. Static Storage Class
4. External Storage Class

Automatic Storage Class:

- Keyword : auto
- Storage Location : Main memory
- Initial Value : Garbage Value
- Life : Control remains in a block where it is defined.
- Scope : Local to the block in which variable is declared.

Syntax :

```
auto [data_type] [variable_name];
```

Example :

```
auto int a;
```

Program :

```
#include <stdio.h>
#include <conio.h>
void main()
{
    auto int i=10;
    {
        auto int i=20;
```

```
            printf("\n\t %d",i);
        }
        printf("\n\t %d",i);
}
```

Output :

20

10

Register Storage Class :

- Keyword : register
- Storage Location : CPU Register
- Initial Value : Garbage
- Life : Local to the block in which variable is declared.
- Scope : Local to the block.

Syntax :

register [data_type] [variable_name];

Example :

register int a;

When the calculations are done in CPU, then the value of variables are transferred from main memory to CPU. Calculations are done and the final result is sent back to main memory. This leads to slowing down of processes.

Register variables occur in CPU and value of that register variable is stored in a register within that CPU. Thus, it increases the resultant speed of operations. There is no waste of time, getting variables from memory and sending it to back again.

It is not applicable for arrays, structures or pointers.

It cannot not used with static or external storage class.

Unary and address of (&) cannot be used with these variables as explicitly or implicitly.

Program :

```
#include <stdio.h>
#include <conio.h>
void main()
{
	register int i=10;
	{
		register int i=20;
		printf("\n\t %d",i);
	}
	printf("\n\t %d",i);
}
```

Output :

20
10

Static Storage Class :

- Keyword : static
- Storage Location : Main memory
- Initial Value : Zero and can be initialize once only.
- Life : depends on function calls and the whole application or program.
- Scope : Local to the block.

Syntax :

static [data_type] [variable_name];

Example :

static int a;

There are two types of static variables as :

a) Local Static Variable
b) Global Static Variable

Static storage class can be used only if we want the value of a variable to persist between different function calls.

Program :

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

void main()
{
	int i;
	void incre(void);
	for (i=0; i<3; i++)
	incre();
}

void incre(void)
{
	int avar=1;
	static int svar=1;
	avar++;
```

```
        svar++;
        printf("\n\n Automatic variable value : %d",avar);
        printf("\t Static variable value : %d",svar);
}
```

Output :

Automatic variable value : 2 Static variable value : 2

Automatic variable value : 2 Static variable value : 3

Automatic variable value : 2 Static variable value : 4

External Storage Class :

- Keyword : extern
- Storage Location : Main memory
- Initial Value : Zero
- Life : Until the program ends.
- Scope : Global to the program.

Syntax :

```
extern [data_type] [variable_name];
```

Example :

```
extern int a;
```

The variable access time is very fast as compared to other storage classes. But few registers are available for user programs.

The variables of this class can be referred to as 'global or external variables.' They are declared outside the functions and can be invoked at anywhere in a program.

Program :

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

extern int i=10;
void main()
{
	int i=20;
	void show(void);
	printf("\n\t %d",i);
	show();
}
void show(void)
{
	printf("\n\t %d",i);
}
```

Output :

```
	20

	10
```

String :

A string is a collection of characters or a character array is also called as String.Strings are always enlosed in double quotes as "string_constant".

Strings are used in string handling operations such as,

- Counting the length of a string.
- Comparing two strings.
- Copying one string to another.
- Converting lower case string to upper case.
- Converting upper case string to lower case.
- Joining two strings.
- Reversing string.

Declaration Syntax:

```
char string_nm[size];
```

Example:

```
char name[50];
```

String Structure :

When compiler assigns string to character array then it automatically supplies **null character ('\0')** at the end of string. Thus, size of string = original length of string + 1.

```
char name[7];
name = "Orange"
```

O	R	A	N	G	E	\0'

Read Strings :

To read a string, we can use scanf() or gets() function with format specifier %s.

```
char name[50];
scanf("%s",name);
```

The above format allows to accept only string which does not have any blank space, tab, new line, form feed, carriage return.

Write Strings :

To write a string, we can use printf() or puts() function with format specifier %s.

```
char name[50];
scanf("%s",name);
printf("%s",name);
```

Example:

```
main( )
{
char name[ ] = "Klinsman" ;
char *ptr ;
ptr = name ;                          /* store base address of string */
while ( *ptr != '\0' )
{
printf ( "%c", *ptr ) ;
ptr++ ;
}
}
```

Note:

name[i] , *(name+i), *(i+name), i[name].....indicates same results

Note:

we cannot assign a string to another, whereas, we can assign a char pointer to another char pointer .

string.h header file :

'string.h' is a header file which includes the declarations, functions, constants of string handling utilities. These string functions are widely used today by many programmers to deal with string operations.

Some of the standard member functions of string.h header files are,

Function Name	Description
strlen	-Returns the length of a string.
strlwr	-Returns upper case letter to lower case.
strupr	-Returns lower case letter to upper case.
strcat	-Concatenates two string.
strcmp	-Compares two strings.
strrev	-Returns length of a string.
strcpy	-Copies a string from source to destination.

Example Program for string copy by using pointers:

```
main( )

{

char source[ ] = "AndhraPradesh" ;

char target[20] ;

xstrcpy ( target, source ) ;

printf ( "\nsource string = %s", source ) ;

printf ( "\ntarget string = %s", target ) ;

}
```

```
xstrcpy ( char *t, char *s )

{

while ( *s != '\0' )

{

*t = *s ;

s++ ;

t++ ;

}

*t = '\0' ;

}
```

Pointer :

Pointer is a variable which holds the memory address of another variable. Pointers are represented by '*'. It is a derive data type in C. Pointer returns the value of stored address.

Note:

C provides two operators &(address operator and indicates address of) and *(indirection operator and indicates address at) which allow pointers to be used in many versatile ways.

Syntax:

```
<data_type> *pointer_name;
```

In above syntax,

* indicates ..... variable pointer_name is a pointer variable.
pointer_name requires memory location
pointer_name points to a variable of type data type.

How to Use ?

```
int *tot;
```

Features of Pointer :

- Pointer variable should have prefix '*'.
- Combination of data types is not allowed.
- Pointers are more effective and useful in handling arrays.
- It can also be used to return multiple values from a funtion using function arguments.
- It supports dynamic memory management.
- It reduces complexity and length of a program.
- It helps to improve execution speed that results in reducing program execution time.

Example program:

```
#include<stdio.h>
main( )
{
int i = 3 ;
printf ( "\nAddress of i = %u", &i ) ;
```

```
printf ( "\nValue of i = %d", i ) ;
printf ( "\nValue of i = %d", *( &i ) ) ;
}
```

The output of the above program would be:

Address of i = some address

Value of i = 3

Value of i = 3

Note:

*(&i) =i(Because * and & are complements)

Note:

int i.....i is an ordinary int

int *j.....j is a pointer to an int or a integer pointer

int **k.....k is a pointer to a pointer to an int or k is a pointer to an integer pointer

float ***p..... l is a pointer to a pointer to a float pointer

Pointers to a function:

call by reference:

Function call through the sending the addresses of the arguments .The addresses of actual arguments in the calling function are copied into formal arguments of the called function. This means that using these addresses we would have an access to the actual arguments and hence we would be able to manipulate them. The following program illustrates this fact.

```
#include<stdio.h>
main( )
{
int a = 10, b = 20 ;
```

```
swapr ( &a, &b ) ;
printf ( "\na = %d b = %d", a, b ) ;
}
swapr( int *x, int *y )
{
int t ;
t = *x ;
*x = *y ;
*y = t ;
}
```

The output of the above program would be:

a = 20 b = 10

Note: This program manages to exchange the values of a and b using their addresses stored in x and y.

Another Example:

```
#include<stdio.h>
main( )
{
int radius ;
float area, perimeter ;
printf ( "\nEnter radius of a circle " ) ;
scanf ( "%d", &radius ) ;
areaperi ( radius, &area, &perimeter ) ;
```

```
printf ( "Area = %f", area ) ;
printf ( "\nPerimeter = %f", perimeter ) ;
}
areaperi ( int r, float *a, float *p )
{
*a = 3.14 * r * r ;
*p = 2 * 3.14 * r ;
}
```

Pointer to an Array :

```
#include<stdio.h>
main( ) {
int s[5][2] = {
{ 1234, 56 },
{ 1212, 33 },
{ 1434, 80 },
{ 1312, 78 }
};
int ( *p )[2] ;
int i, j, *pint ;
for ( i = 0 ; i <= 3 ; i++ ) {
p = &s[i] ;
pint = p ;
```

```
printf ( "\n" ) ;
for ( j = 0 ; j <= 1 ; j++ )
printf ( "%d ", *( pint + j ) ) ;
}
}
```

And here is the output...

1234

1212

1434

1312

56

33

80

78

Note:

int *(a+1)=a[1]

int *(*(b+3)+4)=b[3][4]

Array of Pointers to Strings :

A pointer variable always contains an address. Therefore, if we construct an array of pointers it would contain a number of addresses .

Example1:

```
main( )
{
```

```
char names[ ][10] = {
"akshay",
"parag",
"raman",
"srinivas",
"gopal",
"rajesh"
};
int i ;
char t ;
printf ( "\nOriginal: %s %s", &names[2][0], &names[3][0] ) ;
for ( i = 0 ; i <= 9 ; i++ )
{
t = names[2][i] ;
names[2][i] = names[3][i] ;
names[3][i] = t ;
}
printf ( "\nNew: %s %s", &names[2][0], &names[3][0] ) ;
}
```

And here is the output...

```
Original: raman srinivas
New: srinivas raman
```

Example2:

If the number of exchanges can be reduced by using an array of pointers to strings. Here is the program...

```
main( )
{
char *names[ ] = {
"akshay",
"parag",
"raman",
"srinivas",
"gopal",
"rajesh"
};
char *temp ;
printf ( "Original: %s %s", names[2], names[3] ) ;
temp = names[2] ;
names[2] = names[3] ;
names[3] = temp ;
printf ( "\nNew: %s %s", names[2], names[3] ) ;
}
```

And here is the output...

```
Original: raman srinivas
New: srinivas raman
```

Pointers used in structures:

The way we can have a pointer pointing to an int, or a pointer pointing to a char, similarly we can have a pointer pointing to a struct. Such pointers are known as 'structure pointers'.

Let us look at a program that demonstrates the usage of a structure pointer.

```
main( )
{
struct book
{
char name[25] ;
char author[25] ;
int callno ;
};
struct book b1 = { "Let us C", "YPK", 101 } ;
struct book *ptr ;
ptr = &b1 ;
printf ( "\n%s %s %d", b1.name, b1.author, b1.callno ) ;
printf ( "\n%s %s %d", ptr->name, ptr->author, ptr->callno ) ;
}
```