25/09/23

Unit - 1st & 2nd

Fundamentals of C

History of C -

Year	language	Developed by
1960	ALGOL	International Group
1967	BCPL	Martin Richards
1970	B	Ken Thompson
1972	Traditional C	Dennis Ritchie
1978	K&RC	Kernighan and Ritchie
1989	ANSI C	ANSI Committee
1990	ANSI/ISOC	ISO Committee
1999	C99	Standardization Committee

ALGOL - Algorithmic language
BCPL - Basic combined programming language
ANSI - American National Standard Institute

- C is a structured, high-level, machine independent language.
- The root of all modern language is ALGOL. It was first computer language to use a block structure.
  (sequence of declarations, definitions and statements enclosed within curly braces {})
- BCPL was developed primarily for writing system software.
- B was created by using many features of BCPL and used to create early versions of UNIX OS.

- C was evolved from ALGOL, BCPL and B by Dennis Ritchie at the Bell Laboratories in 1972.
  ↓
  of AT&T (American Telephone and Telegraph)
- UNIX was written in C language thats why known as Mother of all programming languages. (also many languages uses the concepts of C and it is most popular programming language)
- C is highly portable language.

Basic Structure of C program -

Documentation section → (set of comment lines)

Link Section → (Provides instructions to the compiler to link functions from system library)

Definition Section → (Defines all symbolic constants)

Global Declaration section → (Global variables are declared outside of all functions)

main() Function section → (Every C program must have one main function)

{

Declaration Part
Executable Part

] → (All statement in both parts end with a semicolon(;))

}

Subprogram Section → (Contain all user defined functions that are called in main function)

Function 1
Function 2
-
-
Function n

(user-defined functions)

Note - • All sections, except main function section may be absent when they are not required.
• main() marks the beginning of program so it is required to have only one main() function in every program.

- Closing brace '}' marks the end of the program.
- printf() is executable statement.
- /* ..... */ → Multiline comment
- // → Single line comment.

Example-

```
//Name of Program          → Documentation section
#include<stdio.h>
#include<conio.h>           → Linking Section
#define max 100             → Definition section
void add();
int x=100;                  → Global declaration section
int main()                  → main() function section
{
int a=100;                  → Declaration part
printf("Hello main");
return 0;                   → Executable part
}
void add()
{
printf("Hello add");        → Subprogram section
}                             (Function definition)
```

Note- • The keyword void means that the function does not return any information to the operating system.
• The keyword int means that the function returns an integer value to the OS.

The main function-

It is a part of every C program.

- main()
- int main()
- void main()
- main(void)
- void main(void)
- int main(void)

Preprocessor Compiler Directive -

They are used to store library files like <stdio.h>, <conio.h> etc. Special commands that are processed by the preprocessor before the actual compilation of code begins. They start with '#' symbol and typically used to include header files.

Examples -

#include → Some functions are stored in c Library. Library functions are grouped category wise and stored in different files known as header files. If you want to access the functions stored in library, it is necessary to tell the compiler about the files to be accessed.

#include <filename>

Examples - • <stdio.h> → Built-in header file (printf(), scanf() etc. are pre-defined in this file)
(standard input & output) Keywords

• <conio.h> → Header file (clrscr(), getch() etc. are defined in this file)
(console input & output)
{use to clear screen/ previous output}
{used to hold screen of compile}

#define → It is not a statement so not should be end with a semicolon. used to define symbolic constant / create macros. Symbolic constants are written in uppercase.

Example -

- #define PI 3.1416
- #define PRINCIPAL 5000.00
- #define MAX

printf() - All output to screen is achieved using readymade Library functions.
printf("<format string>", <list of variables>);
<format string> contain %f for float, %d for int, %c for char.

scanf() - It receives input from the keyboard.
(referencing operator) ← Ampersand(&) before the variables in the scanf() function is must. & is an 'Address of' operator. It gives the location no. used by the variable in memory.

C Tokens - The smallest lexical or individual units are known as C tokens.

- Keywords (float, while)
- Identifiers (main, b_salary)
- Constants (-15.5, 100)
- Strings ("ABC", "123")
- Operators (+, -, *, /)
- Special symbols ([], {})

Keywords - These are the reserved word having fixed meanings and these meanings can't be changed. These are served as basic building blocks for program statement. There are 32 keywords available in C.

Identifiers - It refers to the names of variables, functions and arrays. ~~It is the name of a memory location which stores some data.~~

Constants - Fixed values that do not change during the execution of a program.

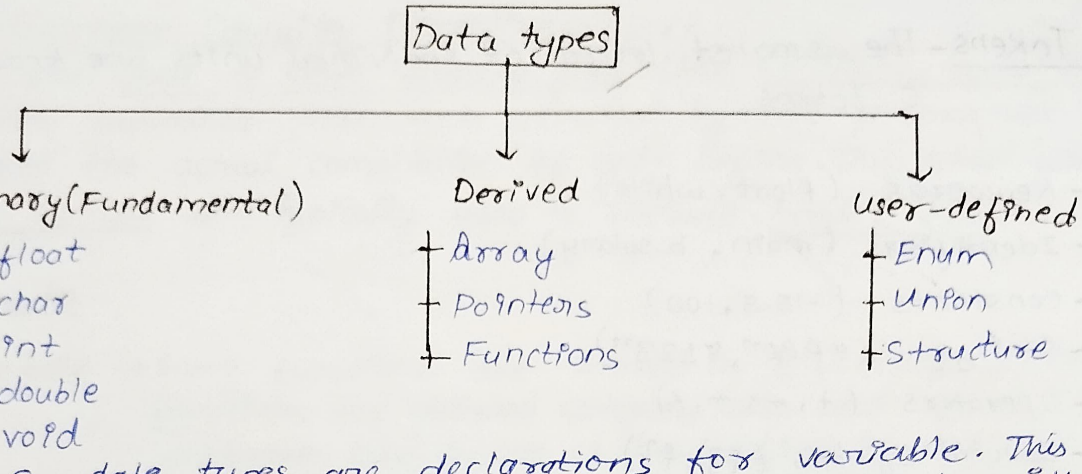
Variables - A data name that may be used to store a data value. The name of a memory location which stores some data.

~~**Data types**~~

Comments - Lines that are not part of program. They are not executed and ignored by the compiler.

Compilation - A computer program that translates C code into machine code.

Hello.c ⟶ C compiler ⟶ a.exe (windows)
a.out (linux and mac)



In C, data types are declarations for variable. This determines the type and size of data associated with variables.

Operators - A symbol that perform some operations or manipulation.

1) Arithmetic operators -

+ , - , * , / , %

2) Relational operators -

= , != , > , < , <= , >=

3) Logical operators - Always gives a value of 1 or 0 as output.

&& → Logical AND
|| → Logical OR
! → Logical NOT

4) Assignment operators -

= , += , -= , *= , /= , %=

~~5) Logical operators.~~

5) Increment and decrement operators -

++ and --

Pre
- ++m → (m+1) {First increment then assign} (increase, then use)
- m++ → (m+1) {First assign then increment} (use, then increase)
- --m → (m-1) {First decrement then assign} (decrease, then use)

Post
- m--- → (m-1) {First assign the decrement} (use, then decrease)

↓

(First print then decrement)

6) Conditional operator – (Ternary operator)

exp1 ? exp2 : exp3

The operator ? : works as follows : exp1 is evaluated first. If it is non-zero (true), then the exp2 is evaluated and becomes the value of the expression. If exp1 is false, exp3 is evaluated and its value becomes the value of the expression.

7) Special operators –

- Comma operator (,)
- Sizeof operator
  ↳ It returns the no. of bytes the operand occupies.
  Ex → m = sizeof (sum), n = sizeof (int)

Space occupied by data types –

Char → 1 byte (8 bits)
int → 2 bytes (16 bits)
float → 4 bytes (32 bits)

8) Bitwise operators – It is used for manipulation of data at bit level. Not applied to float or double. Return output as integer value.

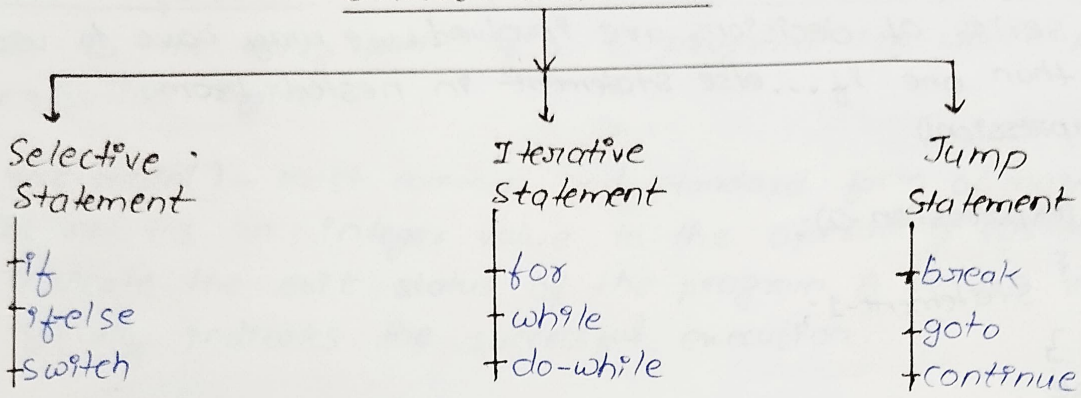
& → bitwise AND
| → bitwise OR
^ → bitwise exclusive OR
<< → shift left
>> → shift right

Precedence of operators –

Precedence is used to determine how an expression involving more than one operator is evaluated.

Operator	Description	Associativity
( ) [ ]	Functional call Array element Reference	Left to Right
+ - ++ -- ! ~ * & sizeof (type)	Unary plus Unary minus Increment Decrement Logical Not ones complement Pointers reference Address size of an object Type cast (conversion)	Right to left
^	Exponent	right to left
* / %	Multiplication Division modulus	Left to Right
+ -	Addition Subtraction	Left to Right
<< >>	Left shift Right shift	Left to Right
< <= > >=	Less than less than or equal to greater than greater than or equal to	Left to Right
== !=	~~Assi~~ Equality Inequality	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 expression	Right to left
= *= /= %= += -= &= ^= |= <<= >>=	Assignment operators	Right to left
,	Comma operator	Left to Right

Control Statement



Decision making with if statement -

- **Simple if statement -**

```
if (expression)
{
   statement-block;
}
statement-x;
```

- statement-block may be a single statement or a group of statements.
- If only single statement is there then no need to put curly braces, if in case more than one statement (compound statement) is given then there is necessity of putting curly braces {}.

- **if-else statement -**

It is an extension of simple if statement.

```
if (expression)
   {
      True-block statement(s)
   }
else
   {
      False-block statement(s)
   }
statement-x;
```

- **Nested of if...else statements -**

When series of decisions are involved, we may have to use more than one if...else statement in nested from.

```
if (expression1)
  {
    if (expression-2);
      { statement-1;
      }
    else
      { statement-2;
      }
else
  {
    statement-3;
  }
   statement-x;
```

- **else-if ladder -**

```
if(condition 1)
     statement-1;
else if (condition 2)
        statement-2;
   else if (condition 3)
          statement-3;
        else if (conditionn)
               statement-n;
             else
               default-statement;
statement-x;
```

De Morgan's Rule -

x becomes !x
!x becomes x
&& becomes ||
|| becomes &&

Examples: !(x&&y||!z) becomes !x||y&&z
!(x<=0||!condition) becomes !x>0&&condition.

main() function –

It is the entry point of a C program. Its where program execution begins.

- **int main()** – most common and standard form of main function. It returns an integer value to the operating system to indicate the exit status of the program. A return value of 0 typically indicates the successful execution.

- **void main()** – Non-standard form of main function. It does not return any value to operating system.

Errors in C – Errors are wrong code, which is not supported to syntax.

- **Linker error:-** It is related to pre-defined functions (like scanf(), printf() etc.)

- **Runtime errors:-** Errors that occurs during runtime and does not produce any output i.e. output not possible. Screen tends to blink but does not give any output.

- **Logical errors:-** It will show success, compile & also give output but not desired output.

Iterative Statement -

- **The while statement** - The while is an entry-controlled loop statement.

```
while(condition)
{
   body of the loop
}
```

- **The do Statement** (do-while) - Makes a test of condition before the loop is executed. Even if the condition becomes false, the body of loop is always executed at once.

```
do
 {
   body of the loop
 }
while(condition);
```

- **The For statement** - for(initialization ; condition ; updation)

```
for(initialization ; test-condition ; increment)
{
 body of the loop
}
```

Switch statement - C has a built-in multiway decision statement known as switch. The switch statement tests the value of a given variable against a list of case values and when a match is found, a block of statements associated with that case is executed.

```
switch(variable/expression)
{
 case <value 1>:
        statement 1;
        break;
 case <value 2>:
        Statement 2;
        break;
 case <value 3>:
        statement 3;
        break;
 case<value 4>:
        statement 4;
        break;
 .....
 .....
 .....
 default;
        default-block;
        break;        ——————> (not necessary to use)
}
```

Jump Statements -

Break statement -

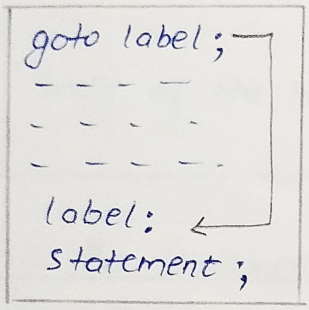
It is used inside loop and switch statement sometimes it becomes necessary to come out of the loop even before the loop condition becomes false, In such situations break statement is used to terminate the loop.

Continue statement - (skipping the part of loop)

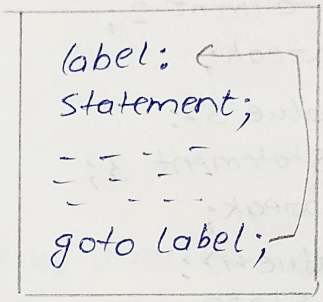
It is used when we want to go to the next iteration of the loop after skipping some of statement of loop. It transfer the execution of loop to the next statement.

Goto statement -

The goto requires a label in order to identify the place where the branch is to be made. Another use of the goto statement is to transfer the control out of a loop when certain peculiar conditions are encountered.



Forward Jump



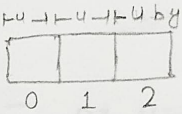
Backward jump

Arrays

Arrays are the collection of similar data types stored at contiguous memory locations.
(adjacent/neighbouring/continuous) {linear storage}

Syntax:

```
int marks[3];
char name[10];
float price[2];
```



→ Total 12 bytes reserved in memory by marks array.

{in general size of (int) = 4 byte
but in turbo C it is 2 byte}

Initialization of Array-

```
int marks[] = {97, 98, 99}
int marks[3] = {97, 98, 99}
```

Array memory reserved : memory reserved by single data type (like int, float, char) * no. of elements stored

Array - Collection of similar type of data items and each data items is called an element of the array.

→ When an array is declared the compiler allocates space in the memory sufficient to hold all the elements of the array.

→ The size of array should be known at compile time. Hence, we can't use the variable for specifying the size of array in declaration, the symbolic constant can be used to specify the size of array.

Arrays
- Single dimensional (Also called Vector)
- Multidimensional (Two dimensional called Matrix)

Format of Array-

Data type	Name of array[size];

- int a[5];

```
#define SIZE 5 → (valid)
main
{
  int roll[SIZE]
}
```

int a=5, roll[5] → {Not valid}

Location/Address of K^{th} element in array-

$LOC(A[K]) = B.A + w(K - \text{lowerbound index})$

$= B.A. + wK$

∴ $LOC(A[K])$ → location of Kth index of array

B.A. → Base Address

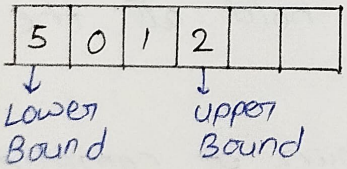
w → capacity (size) of each cell (in bytes)

K → No. of index ~~Upperbound index~~

Number of elements in each array-

No. of elements = Upperbound − Lowerbound + 1

= UB − LB + 1



or A[5:50]

Lower Bound — Upper Bound

Multidimensional Array –

	0	1	2
0	0,0	0,1	0,2
1	1,0	1,1	1,2
2	2,0	2,1	2,2

- Row Major Representation

5	6	7	8	9	1	2	3	4

- Column Major Representation

5	8	2	6	9	3	7	1	4

$$\begin{bmatrix} 5 & 6 & 7 \\ 8 & 9 & 1 \\ 2 & 3 & 4 \end{bmatrix}$$

RMO (Row Major Order) – ∴ n = no. of ~~Column~~ Row

$$LOC(a[i][j]) = B.A. + w[n(i - \text{lowerbound of Row index}) + (j - \text{lowerbound of column index})]$$

i means 1 to i → U.B, L.B
j means 1 to j → U.B, L.B

CMO (Column Major order) – ∴ m = no. of ~~column~~ Row

$$LOC(a[i][j]) = B.A + w[m(j - \text{lowerbound of column index}) + (i - \text{lowerbound of row index})]$$

Example-1) let the base address (B.A) of the first element of the array is 400 and each element of array occupies 2 bytes in the memory then calculate the address of the 4th element in the array a[10].

Sol^n – Given, BA = 400, K = 4, w = 2 bytes

$$LOC(A[4]) = 400 + 2 \times 4$$
$$= 400 + 8$$
$$= 408$$

L.B ← [1 or 18, 0 to 18] { Depends upon you what you use as L.B (0 or 1) }

Ex-2) Consider the linear array A[5:50], B[-5,10], C[18].

(i) Find no. of elements in each array = U.B ~~-~~ L.B + 1

Sol^n – No. of elements in A[5:50] = 50 − 5 + 1 ⇒ 46

No. of elements in B[-5,10] = 10 − (−5) + 1 ⇒ 16

No. of elements in C[18] = 18 − 0 + 1 ⇒ 19

(ii) Suppose BA of array = 300, w = 4byte. Find the address of A[15], A[40], A[55].

Soln • LOC(A[15]) = BA + w(K - lowerbound)
= 300 + 4(15 - 5)
= 300 + 4x10
= 300 + 40
= 340

• LOC(A[40]) = 300 + 4(40-5)
= 300 + 4(35)
= 300 + 140
⇒ 440

• LOC(A[55]) → Not possible

A[1:20, 1:50]

Q- Each element of an array (data)[20][50], requires 4 bytes of storage the BA of the data is 2000. Determine the location of data[10][10] when the array is stored as row major and column major.

Soln. Index of lowerbound [1][1]

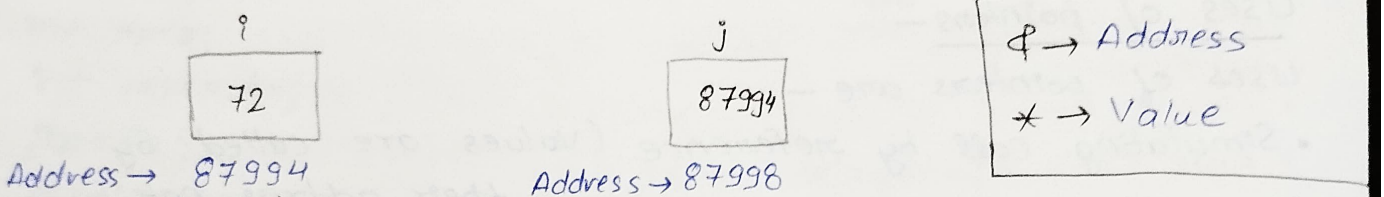
BA = 2000, i = 10, j = 10, m = 20, n = 50, w = 4

RMO ⇒ LOC(data[10][10]) = 2000 + 4[50(10-1) + (10-1)]
= 2000 + 4[50x9 + 9]
⇒ 2000 + 4(450 + 9)
⇒ 2000 + (4 x 459)
⇒ 2000 + 1836
⇒ 3836

CMO ⇒ LOC(data[10][10]) = 2000 + 4[20(10-1) + (10-1)]
= 2000 + 4[20x9 + 9]
⇒ 2000 + 4x139
⇒ 2000 + 756
⇒ 2756

Pointers (dereferencing operator)

A pointer is a variable which stores the address of another variable that stores some value.



It points to a memory location where the first byte is stored.
↳ Base address

- j is a pointer
- j points to i

The address of (&) operator is used to obtain the address/location of a given variable. Eg- &i = 87994 and &j = 87998

Format specifier for printing pointer address is '%u' or '%p'.

Value at Address (*) operator/dereferencing operator –

It is used to obtain the value present at a given memory address. It is denoted by (*).

Eg- *(&i) = 72
*(&j) = 87994

```
//Program
void main
{
int *p, a=5;
p=&a;
printf("Value of a : %d", a);
printf("Value of a : %d", *p);
printf("Address of a : %u", p);
getch();
}
```

type of value that pointer will point to

```
int x=5;
int *ptr;
ptr=&x;
```

```
int x=5,
*ptr=&x;
```

Both are same

Pointer is derived data type in C. It is built from one of the fundamental data type in C. Pointers contains memory addresses as their values

Uses of pointers –

Uses of pointers are –

- Simulating call by reference (values are called by their address not by their variable name)
- Returning more than one value from a function.
- Accessing dynamically allocated memories.
- Implementing data structure like linked list, tree and graphs.
- Improving efficiency.

Address (&) operator –

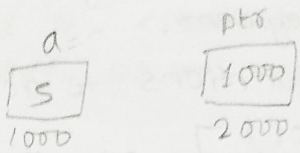
C provides an address operator(&) which returns the address of variable when placed before it.

Double Pointer –

A pointer-to-pointer (double pointer) is used to store the address of another pointer.

*ptr = a (we can say that)

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a=5;
int *ptr;
int **ptrptr;
ptr=&a;
ptrptr=&ptr;
printf("Address of a = %p\n",&a);
printf("Address of a = %p\n", ptr);
printf("Address of value *PTR = %d\n", *ptr);
printf("Address of PTR = %p\n", &ptr);
printf("Address of PTRPTR = %p\n", ptrptr);
printf("Value at PTRPTR = %d\n", **ptrptr);
printf("Address of PTRPTR = %p\n", &ptrptr);
getch();
}
```



value of ptr

void pointer ~~/null pointer~~ –

A pointer that does not have specific data type. i.e. it points to data of any type. It hold address of any type and can be typecasted to any type.

- Void pointers can't be ~~defer~~ dereferenced directly.
- So first void pointers are typecasted into any data type and then ~~def~~ dereferenced.

• printf("%d", *(int *)ptr);

Null pointer –

Pointer thas has no valid address and is allocated to zero or NULL. It does not point to anything.

Dynamic memory Allocation -

In DMA, memory is allocated at runtime or at time of execution from the heap segment.

Four functions to active this task are -

- malloc()] → Allocation Purpose
- calloc()]
- Realloc() → modification Purpose
- Free() → Deallocation Purpose

Heap is the segment of memory where dynamic memory allocation takes place. An area where memory is allocated or deallocated without any order or randomly.
(Heap → Available memory space)

- In static memory allocation, memory is allocated during compile time. The memory allocated is fixed and cannot be increased or decreased during runtime.

Problems faced in SMA -

- Size of memory is fixed as compile time.
- If value stored by user in array at runtime is less than the size specified then there will be wastage of memory and in case more than the size specified then the program may crash.

Note -

- Pointers play an important role in dynamic memory allocation.
- Allocated memory can only be accessed through pointers.

malloc() - (memory allocation)

- malloc is a built-in function declared in <stdlib.h> header file. It reserves block of memory with given amount of bytes. Takes only single argument.
- Return value is a void pointer to allocated space.
- Therefore void pointer needs to be casted to appropriate time as per requirement.
- If the space is insufficient, allocation of memory got failed then returns a null pointer.

Syntax : (int *)malloc(size * sizeof(int));
ptr=(int *)malloc(10 * sizeof(int));

calloc() - (contiguous Allocation)

- It is used to allocate multiple blocks of memory.
- It accept two arguments (n, sizeof(int)).
- Memory allocated by calloc is initialized to zero but in case of malloc is initialized with some garbage value.

Syntax : (int *)calloc(10, sizeof(int));
ptr = (int *)(alloc(10, sizeof(int));

Note: Both malloc and calloc return NULL when sufficient memory is not available in the heap.

realloc() - (reallocation)

It is used to change size of memory block without losing the old data.

Syntax - ptr = (int *)realloc(ptr, 9 * sizeof(int));

previously allocated memory

New size

free() -

This function is used to release the dynamically allocated memory in heap.

Syntax - free(ptr);

The memory allocated in heap will not be released automatically after using the memory. The space remains there and can't be used.
It is the programmer's responsibility to release the memory.

String

String constant (String literal) -

String constant is a sequence of character enclosed by double quotes (" "). "String constant itself becomes a pointer to the first character in array. Each character occupies 1 byte and a computer automatically insert null character (\0) at the end."

- char str[ ] = "Varanasi"; or char str[9] = "Varanasi";

String Variable -

String variable is a one dimensional array of ASCII characters terminated by null character. To create string variable we need to declare a character array with size sufficient to hold all characters. plus null character.

- char str[ ] = {'V', 'a', 'r', 'a', 'n', 'a', 's', 'i', '\0'}; or
  char str[9] = {'V', 'a', 'r', 'a', 'n', 'a', 's', 'i', '\0'};

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
char str[]= "Varanasi";
int i=0;
clrscr();
while(str[i]!='\0').
{
printf("Char = %c\n", str[i]);
printf("Address = %p\n", &str[i]);
i++;
}
getch();
}
```

String Functions- (Important)

strlen()- (string length)

This function returns the length of the string i.e. no. of characters in string excluding null characters.

type strlen(char const * string)

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
char str[10];
clrscr();
printf("Enter the string : ");
gets(str);
printf("Length of string %d", strlen(str));
getch();
}
```

strcmp()- (string compare)

This function is used for lexico graph comparison of two string, if the two string match strcmp() return value '0' otherwise it returns non-zero values. It compares the string character by character and comparison stops when either end of string reached.

strcmp(s_1, s_2) return values -

- <0 when $s_1 < s_2$ (return negative value)
- $=0$ when $s_1 = s_2$ (return value zero)
- >0 when $s_1 > s_2$ (return positive value)

```
{
char str1[10], str2[10];
printf("Enter the string:");
gets(str1);
gets(str2);
if(strcmp(str1,str2)==0)
printf("String are same");
else
printf("String are not same");
}
```

Strcpy()- (String copy)

This function used to copy one string to another string including null character. The function returns a pointer to destination.

```
strcpy(str1, str2)
```

destination ↑ str1

source ↑ str2

```
{
char str1[20], str2[20];
printf("Enter the string:");
gets(str2);
strcpy(str1, str2);
printf("string str1:%s %s\n", str1,str2);
strcpy(str1, "SMS");
strcpy(str2, "Varanasi");
printf("String str1:%s %s", str1, str2);
}
```

Strcat() – (String concatenation)

This function is use to append copy of a string at the end of other string, null character of string1 is removed & string2 is added at the end of string1.

```
{
char str1[20], str2[20];
printf("Enter the string :");
gets(str1);
gets(str2);
strcat(str1, str2);
printf("string str1 %s", str1);
}
```

Note –

(does not track spaces)

scanf() cannot input multi-word strings with spaces. So, gets() and puts() is used.

gets() = input a string (even multiword including space)

puts() = Output/ a string
print

~~For accessing parameters use (. or →)~~
i.e.

Structure -

Format of structure -

```
Struct name
{
datatype member1;
datatype member2;
datatype member3;
};
```

```
Struct student          ← Tagname
{
int rollno;
char name[20];          ← Data members
float marks;
}s1,s2,s3;
```

- For accessing parameters use (. or →) i.e. name.member1 = name→member1
  - . → Member access parameter

```
//Program for structure
#include<stdio.h>
#include<conio.h>
#include<string.h>
Struct emp
{
int id;  → 2 (for turbo c)
char name[20];
float bill;  → 4
};  → Declaration is not necessary here
e1;
void main()
{
Struct emp e1={1,"Sam",100.50};  } Declaration is necessary
Struct emp e2,e3;
clrscr();
e2.id=2;
Strcpy(e2.name, "Abhi");
e2.bill=20.10;
printf("Enter detail of e3 id, name, bill");
scanf("%d %s %f", &e3.id, e3.name, &e3.bill);
printf("Employee detail : %d %s %f\n", e1.id, e1.name, e1.bill);
printf("Employee detail : %d %s %f\n", e2.id, e2.name, e2.bill);
printf("Employee detail : %d %s %f\n", e3.id, e3.name, e3.bill);
printf("Size of structure : %d", sizeof(e1));
getch();
}
```

→ 26 (2+20+4)

Structure-

Structure is user define data type the purpose of Structure is to store related fields of different data types. Structure is capable to store heterogeneous data, the data of different type can be grouped together under single name using structure. The data elements of structure are refered to as members.

Union-

Union is defined as ~~derived~~ user-defined datatype like structure and it can also contain member of different datatype, the syntax used for declaration of a union type is 'union' keyword instead of 'struct'.

The main difference between union and structure is in the way memory is allocated for the members, in the structure each member has its own memory location. where as members of union share same memory location. When a variable of type union is declared compiler allocates sufficient memory to hold the largest member in the union, since all member share same memory location, we can use only one member at a time. Thus, union is used for saving memory.

Format-

```
union union_name
{
 datatype member1;
 datatype member2;
 datatype member3;
};
```

~~1 Program~~

// Program for union

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
union book
{
int bn;  → 2
char name[10]; → 10
float price; → 4
}ub;
void main()
{
union book ub;
clrscr();
ub.bn=1;
printf("Book no. : %d\n", ub.bn);
strcpy(ub.name, "Prog in C");
//ub.name = "Prog in C";
printf("Book name : %s\n", ub.name);
ub.price = 100.5;
printf("Book price : %3.2f\n", ub.price);
printf("Size of union : %d", sizeof(ub));
                              → 10
getch();
}
```