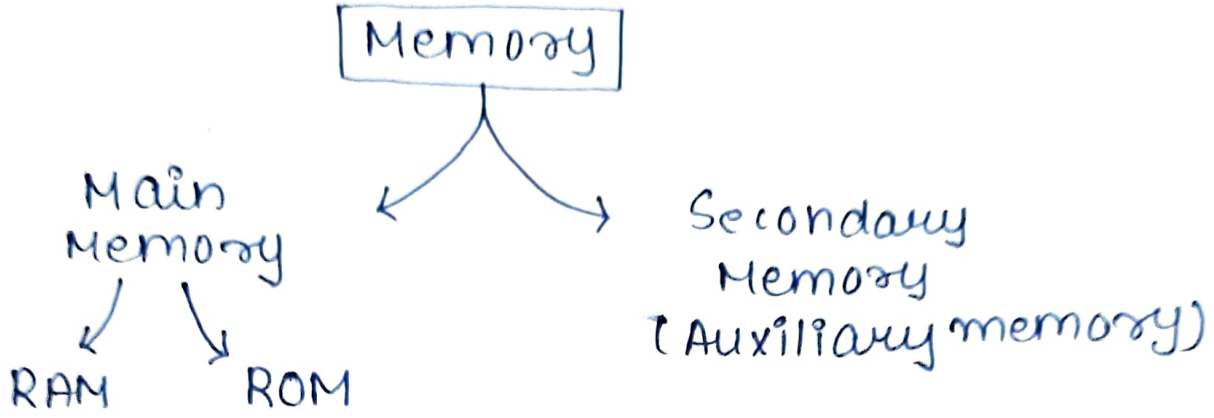
CH:1 Introduction to Computing

① Computer = An electronic device which may be used to perform various computations involving arithmetic and logical operations.



→ secondary memory is slower compared to main or primary memory.

② Software = set of instructions and data given to the computer.

↳ system software (A)
↳ application software (B)

(A) act as an interface between the user and bare hardware.

(B) meant to perform a specific activity and aid users in various ways.

③

Compiler	Interpreter
→ scans whole program	→ scans line by line.
→ after compilation source code is not required.	→ For every execution, run source is required.
→ fast execution	→ slow execution.
→ object code is generated when code is free from errors.	→ NO object code file is generated.

④ Algorithm = finite set of unambiguous instructions.

⑤ pseudo code = means to represent an algorithm in coded form.

⑥ flowchart = similar to algorithm which helps us graphically visualize the flow of control with sequence of statements.

⑦ ALU = actual computation happens here.
CU = coordinates all the activities.

- 3 main programming constructs
  - → imperative (simple)
  - → conditional (>, <, =)
  - → iterative (repeat)

⑧ low level → mnemonic ↔ middle level → Binary
- assembly language
- assembler

high level → operator
- eng language
- compiler / interpreter

→ RAM = volatile memory
ROM = non volatile memory

CH:2 Overview of C

① main - only one
- execution starts from here.

② /* ... */ = comments

③ # = preprocessing operator used to access the functions stored in the library.

↳ create → compile → link → execute.

④ DOS = DISK operating system.

⑤ variable = container used to store value.

⑥ 5 fundamental data types.

→ int → float → void
→ char → double

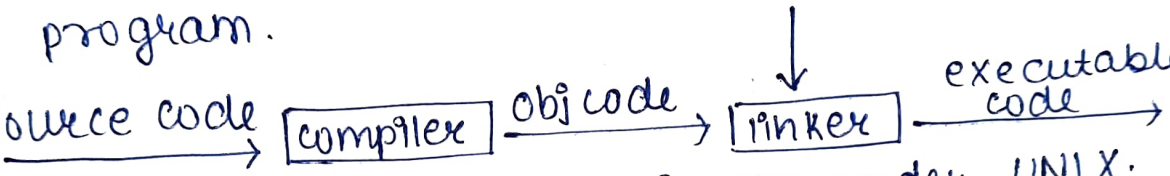
• Keyword void means that the function does not return any info to the OS.

⑦ Keywords cannot be used as variable names.

⑧ Basic structure of C programs.

↳ Documentation section
↳ link section (# include)
↳ Definition section
↳ Global Declaration section
↳ main() func section
↳ subprogram section

⑨ Source code is the file containing C source program.



⑩ linking is done automatically under UNIX.

⑪ program = Sequence of instructions written to perform a specific task in the computer.

CH:3 Constants, Var, Data Types

① Trigraph characters = sequences to provide a way to enter certain characters that are not available on some keyboards. (??=, ??-)

→ Every C word is classified as either a key word or an identifier.

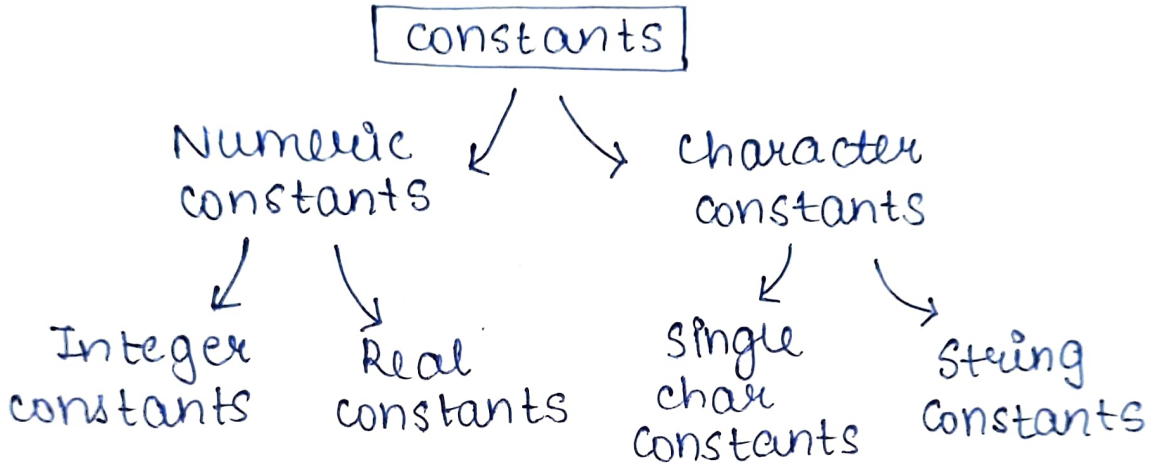
② Keyword = Those words in C which have fixed meanings and these meanings can not be changed.

③ Identifiers = refer to names of variables, func^n and arrays. User defined, 1st letter character.

→ There are total 32 Keywords in ANSI C.

(*) Rules for Identifiers:
- ↳ first char must be an alphabet / _
- ↳ Must consist only letters, digits, _
- ↳ ONLY 31 characters are significant.
- ↳ Cannot use a Keyword.
- ↳ Must not contain white space.

(4) constants = fixed values that do not change during execution.



(5) Datatypes : Primary, Derived, User-defined.
→ int, char, float, double, void

- Short int, int, long int
- signed (-128 to 127), unsigned (0 to 255), void / generic
- float, double, long double

(6) variable → global / external variable
→ local variable.

(7) Storage class = provides info abt variable location
It decides the portion of the program within which the variables are recognized.
- ↳ auto
- ↳ static
- ↳ extern
- ↳ register

- location
- visibility

CH:4 Operators and Expressions

→ operators are used in a program to manipulate data and variables.

① Arithmetic (+, -, *, /, %)
② Relational (<, <=, >, >=, ==, !=)
③ Logical (&&, ||, ! = AND, OR, NOT)
④ Assignment (=)
⑤ Increment / Decrement (++, --)
⑥ Conditional (? : = ternary operator)
⑦ Bitwise (&, |, ^, <<, >>)
⑧ Special (sizeof, comma, pointer, member selⁿ)

→ Type conversion ↳ Implicit
↳ Explicit

- when compiler itself makes any changes in the code, it is known as implicit type conv.
- when user does it manually it is known as explicit type conversion.

↳ Header files are of two types
① one which comes with the compiler.
② user defined
↳ 2 ways to include < >, " ".

→ n++ = postfix = value changes in next step
++n = prefix = value changes in same step.

↳ modulo operator only with integer.

CH:5 Managing IO operations

① getchar = read one char fm std input
② putchar = read one char fm std input
③ control string = combⁿ of format specifⁿ, escape sequences and characters that are to be printed on the screen.

%c = single character
%d = decimal character
%e, %f, %g = floating pt. value.
%h = short integer
%i = decimal/hexadecimal/octal integer
%o = octal integer

CH:6 Decision Making And Branching

↳ If Statement
↳ Switch Statement
↳ Conditional Statement
↳ goto Statement

① If Statement

(A) simple if

```
if (test exp)
 {
   Stat-block;
 }
statement-x;
```

(B) if else statement

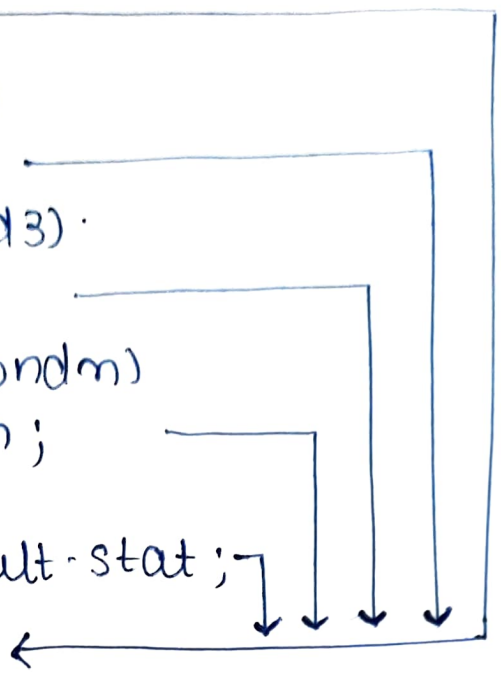
```
if (test exp)
   { True block stat(s)
   }
else
   {
   } False block stat(s)
Statement -x
```

(C) Nesting if else statements

```
if (test cond-1)
{ if (test cond-2);
      { stat-1;
      }
   else
      {stat-2;
      }
  }
else
   { stat-3;
   }
statement-x
```

(1) else if ladder

```
if (cond 1)
   stat - 1;
else if (cond 2)
      stat - 2 ;
   else if (cond 3)
      stat - 3
      else if (condm)
        stat - n ;
          else
           default - stat ;
statement - x
```



- else is always paired with most recent unpaired if.

(2) Switch statement

→ tests the value of given variable against a list of case values and when match is found, stats associated with that block are executed.

→ It is a multiway decision making stat.

```
switch (exp)
{ case value1 :
            blck 1
            break;
  case value2 :
            blck 2
            break;
  ....
  default :
          default - blck
           break;
}
statement - x ;
```

③ Conditional operator (? :)

conditional exp ? exp1 : exp2

↳ code becomes more concise.
↳ poor readability.
↳ better to use if statements when more than a single nesting of conditional operator is required.

④ goto statement

↳ when control is to be transferred.

(A) Forward jump (B) Backward jump

```
goto label;
   ——
   ——
   ——
label:
Statement;
```

```
label:
Statement;
   ——
   ——
   ——
goto label;
```

↳ often used at the end of program to direct the control to go to the input stat.
↳ used for unconditional branching.

CH:7 Decision Making and Looping

LOOPS

counter controlled loop
→ we exactly know how many times loop works.

sentinel (indefinite repitition) controlled loop
→ we do not know how many times loop works.

↳ entry controlled = while, for loop
↳ exit controlled = do while loop

① while loop

↳ entry controlled / sentinel based

```
while (test cond)
{
  body of the loop
}
```

② do while loop

↳ exit controlled

↳ body of loop is always executed atleast once.

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

③ for loop

↳ entry controlled / counter based

↳ more concise loop structure

```
for (initialize; test condn; incr/decr)
{
  body of the loop
}
```

eg:

```
n=1
do
{
  ——
  ——
n=n+1;
}
while(n<=10);
```

(do)

```
n=1;
while(n<=10)
{
  ——
  ——
n=n+1;
}
```

(while)

```
for(n=1;n<=10;
    ++n)
{
  ——
  ——
  ——
}
```

(for)

④ Jumping in loops

↳ break (only 1)

↳ switch

↳ goto (label)

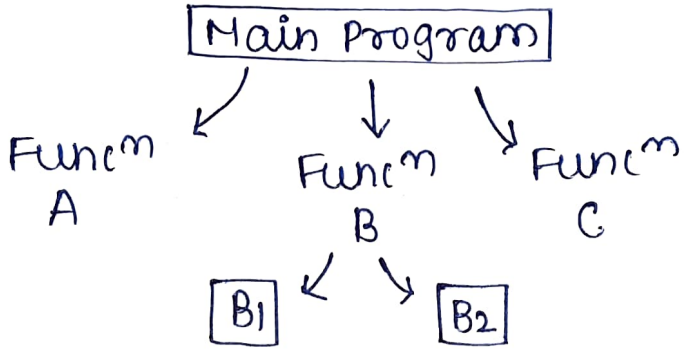
⑤ Structured programming
- ↳ sequence (straight line)
- ↳ selection (branching)
- ↳ Repitition (looping)

⑥ Break = Terminates the loop and takes the program control to the statement immediately following the loop.

⑦ Continue = Skips the remaining part of the loop and takes the program control to the next loop iteration.

CH:10 User Defined Func^n

• In C, subprograms are referred to as func^n.



① Elements of user defined functions
- ↳ Func^n definition (A)
- ↳ Func^n call (B)
- ↳ Func^n declaration. (C) / prototype

(A) Func^n definition / func^n implementation
- → func name
- → func type
- → list of parameters
- → local variable declar^n
- → func statements
- → return statements

(B) Func call = can be called simply using the func name followed by a list of actual parameters if any, enclosed in parantheses.

↳ It is a postfix expression.

(C) Func declaration

→ Func type, return type
→ Func name
→ parameter list
→ Terminating semicolon.

② Difference Between (A) and (C).

Definition	Declaration
• variable/func can only be defined once.	• variable/func can be declared any no. of times.
• Memory will be allocated.	• Memory will not be allocatid.
eg: int f(int a) { return a; }	eg: int f(int);
→ system allocates memory by seeing above func defin.	→ just for informing the compiler that a func named f with ret type and arg as int will be used in func.

③ Category of functions:

① no arg no return value.
② with arg no return value.
③ with arg one return value.
④ no arg but return value.
⑤ return multiple values.

④ Recursion: When a function calls itself, recursion occurs.
↳ works on stack which follows LIFO.
↳ must have a base condition.

⑤ local variables (Automatic)
global variables (External)
static variables
register variables.

⑥ storage classes = auto, static, extern, register.

⑦ arguments = set of values that are passed to a function to enable the function to perform a desired task.

⑧ Modular programming = A SD approach that organizes a large program into small, independent program segments called modules.

→ A return statement is required if the return type is anything other than void.

↳ It is efficient to write code using recursion.

(A) direct recursion

```
func1()
{
  func1()
}
```

⊙ when time and space complexity are concerned, iteration is preferred over recursion.

(B) Indirect recursion

```
func2()
{ func1()
}
func1()
{ func2()
}
```

⊙ Know recursion trees well.

CH:12 Pointers

① Pointer = It is a derived data type in C.
↳ It is a variable that holds address of another variable of same datatype.

→ pointer variable's size is 2 byte for any datatype.
→ allows us to occupy memory runtime.
→ pointers can access value outside of function body.
→ can help us easily implement data structure program.

② we need 2 unary operators
& = address
* = value

→ <datatype> * <variable name>
eg: int * p

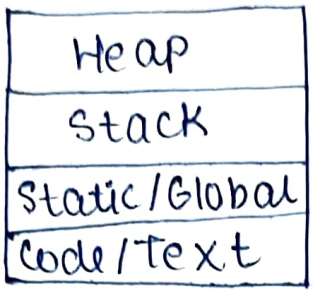
Types = void, null, pointer to pointer, dangling, function.

→ pointer is tight bound which means to store address of int variable, the type of pointer must also be int.
→ pointer variable always stores base value.

```
void increment (int n) ──→ local variable
{                      ──→ called func
  n = n+1;             ──→ formal args
}
int main ()            ──→ calling func
{                      ──→ local variable
  int n = 10;          ──→ actual args.
  increment (n);
  print ("N = %d \n", n);
  return 0;
}
```

Basic Terminology

(*) Applications Memory



↳ generic pointer = special type of pointer that can be pointed at objects of any data type.

(2) * refers to dereferencing which means you try to modify or access the value.

int
↳ int *
↳ int **
↳ int ***

(3) Call by value = Here the actual parameters will be copied to formal parameters and these two different parameters store values in different locations.

(4) Call / pass by Reference = Here both actual and formal parameters refers to same memory location. ∴ Any changes made to the formal parameters will get reflected to the actual parameters.

→ Memory is here always represented in Hexadecimal.

10 - A	12 - C	14 - E
11 - B	13 - D	15 - F

Arrays

(1) Array = set of elements having same name and same datatype

↳ Syntax: datatype varname [size]

eg: int A [10]

↳ Base address

⊙ Base address = Address fm where computer starts to store values.

```
main()
{ int A[1000] double (&A[0])
  double (A) ← same
}          ↳ Base address of array A
```

→ A[i] = *(A+i)
&A[i] = A+i

↳ Array always starts from zero as we need to access the value stored at base address.

⊛ size of array = $\frac{\text{size of [A]}}{\text{size of (A[0])}}$

① Code to find Sum of Elements

```
SOE (int *p, int size){
   int ans=0
for (int i=0, i<size, i++)
 { ans+=p[i];}
return ans;}
```

② Make double of an array
› Inside main function, create one array [1,2,3,4,5]. Make double [2,4,6,8,10]

```
void makedouble (int *p, int size)
 { for (int i=0, i<size, i++)
    { *(p+i) = *(p+1) *2;
                or
      p[i] = p[i] *2 }}
```

→ Size of any pointer variable varies depending on the compiler.

⊙ A string is a sequence of characters that is treated as a single data item.
↳ C does not support string as a data type.
↳ Every string has a null character = \0. The null character denotes the end of the string.

⊙ If a string has 10 characters then the size of character array will always be = total no. of characters + 1.
ie. In our case it will be 11 bytes.

→ string length = length of string without null character.
→ null character occupies 1 byte.

(*) difference between array and a pointer
· Base add of array cannot be changed whereas the value of pointer can be changed.

① char C[] = "Hello"
→ compiler implicitly includes the null character.

② index by index
→ Here, the user needs to include the null character explicitly.

C[0] = H ... C[5] = 0
C[1] = E ... C[6] = \0.

→ do include string.h (header file)
strlen(C);
C → character Array
strlen → Return unsigned value.

Now, let us see a C program to print the entire array.

① Code to print entire char array.

```
void print (char *c)
  { int i = 0;
    while (c[i] != '\0')
    { printf ("%c", c[i]);        // *(c+i)
      i++;
    }
    printf ("\n");
  }
```

indent

→ # = preprocessing operator. ie. here we try to execute

⊛ Macros : Macros are relatively very small functions in terms of execution. They are generic. In other words, they are simple functions without datatype.

→ generally they are represented in capital letters.

```
# define MAX(x,y)((x)>(y)?(x):(y))
```

① Code to find Max of an Array

↳ normal method

```
int findMax (int *p, int size){
    int max= *p;
    for int (i=1 , i<size , i++) {
        max= MAX (max, *(p+i));
    }
    retuen max; }
```

② now let us see how to do this using recursion.

(2) Code to find Max of an array using Recursion.

```
# include<stdio.h>
int findMax(int *, int);
int main void{
   int A[] = {20,1,2,30,15,5,20,25};
   int size = size of(A)/size of(A[0]);
   int ans = find Max(A, size);
   printf("Max Value = "%d" ", ans);
   return 0;}
int find Max (int *p, int size){
    int max = *p;
    for(int i=0 , i< size , i++){
     max = Max(max, p(i));
    }
 } return max;
```

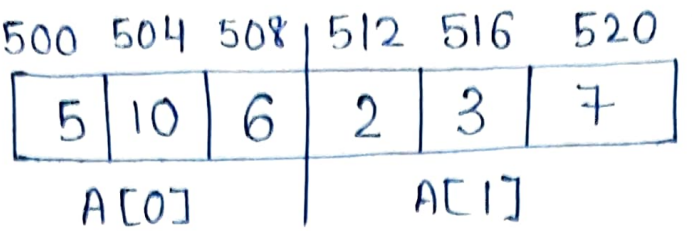
(3) Code to find Max of an Array using MACROS.

```
int findMax(int *p , int size)
{
   return (size==1)? *p :
     Max(*p , findMax((p+1), size-1));
```

(*) Multi dimensional Array.

- An array with more than two dimensions is known as multi dimensional array. Each element is defined by two subscripts. the row index and the column index.

(A)



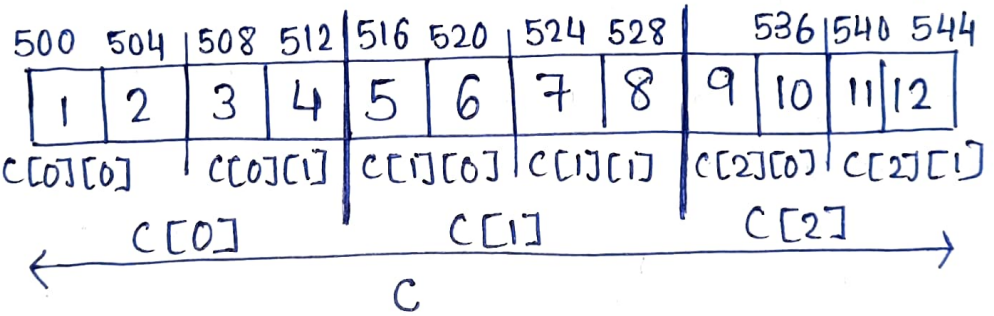
① Print A or &A[0] // 500
② Print *A or A[0] // 500
③ Print &A[0][0] // 500
④ Print A+1 or &A[1] // 512
→ 500 + size of 1D array of 3 integers
→ 500 + 12 = 512
⑤ Print *A+1 or A[1] or &A[1][0] //512
⑥ Print *(A+1)+2 // 520

→ A[i] is same as *(A+i)
→ A[i][j] = *(*(A+i)+j) = *(A[i]+j)

✱ declaration:

1D: void F1(int *p) or void F1(int P[])
2D: void F1(int (*p)[3]) or void F1(int p[][3])

(B)



① Print c // 500
② Print *c or c[0] or c[0][0] // 500
③ Print *(c[0][1]+1) or c[0][1][1] // 4
* = dereference and get value.
④ Print *(c[1]+1) // 524

- you cannot perform dereferencing using generic pointer.

Step 4 Calculate V_{DS}

$$V_{DS} - V_{DD} + I_D \cdot R_S + I_D R_D = 0$$

$$\therefore V_{DS} = V_{DD} - I_D (R_D + R_S)$$

* App^n Memory:

Heap
Stack
global
code

← every func call → stack frame
↓
compile time memory allocation

→ instruction

Heap = pool of memory
Stack memory is very limited

malloc } Memory allocation f^n
calloc }

→ dynamic memory allocation and deallocation

malloc (bytes)
malloc (4) → fm heap mem we demand 4 bytes)
↳ return base address.

```
int *p;
type casting
p = (int *) malloc (sizeof (int));
```

```
*p = 20;
free (p); // deallocn
```

(*) Basic Operations:

(1) Traverse = print all array elements one by one.

(2) Insertion = Add elements at the given index.

(3) deletion = deletes an element at given index.

(4) Reversing
- ↳ print in reverse order
- ↳ Reversing the elements in an array
- ↳ Reversing only first half of elements.

(5) Update = updates an element at given index.

(6) search = searches an element using given index or by the value.

(7) Sorting = arranging in an order.

```
// enter array elements
for(i=0;i<n;i++)
    scanf("%d",&a[i]);
// printing array elements
for(i=0;i<n;i++)
    printf("\n%d",a[i]);
```