C-Programming

If/else → chap ③ → ③
Logical op → chap ④ → ③
Loops → chap ⑤ ⑥ → ⑤
Switch → chap ⑦ → ⑪
Functions → chap ⑧ → ⑩
Pointers → ⑥

CHAPTER-1

constants (also called LITERALS)

- primary constants
  - Integer const.
    - * Should not have decimal point
    - * No commas / blanks
  - Real const
    - * Should have a decimal point
    - * Also called Floating point const.
    - * Floating form → 3.4
    - * Exponential form
      3.42 e^{-4}
      (3.42 → mantisa, e^{-4} → exponent)
  - Character const.
    - * It is a ~~single~~ SINGLE ALPHABET
    - * Single digit or single special symbol
    - * Enclosed b/w single inverted commas
- Secondary constants

Variables (also called IDENTIFIERS)

* First character must be an alphabet or underscore ('_')

Comments

* Enclosed b/w ~~//~~ ~~/*~~ /* */ or //.

- **main ()**
  - * It is a function
  - * Statement which belongs to main () is enclosed b/w { }
  - * It always returns an integer value which is why there is int before main ()
- **printf ()**
  - * TO USE PRINTF() WE REQUIRE A HEADER FILE
    # include <stdio.h>

* Syntax → printf (" format string ", variables);

- scanf()

* In order to enter input through keyboard, we use scanf().
* syntax → scanf (" format string", & variable);
* & → ampersand → Address of operator → The memory location at which value given to a variable is stored.

- Format specifiers

int → %d
float → %f
char → %c

* Syntax of program

```
#include <stdio.h>  ⟶ Header file
int main(){         ⟶ Function
int , , ;
float , , ;   } declaration
char , , ;
. . .
. . .         } Body
. . .
return 0;
}
```

* ~~Format specifiers~~

(%5d, x) → print x after 5 spaces from right

→ x

(%0

* Format specifiers

- ("%5d", x); → print x such that x is at 5th place from right

→ ␣␣␣␣x
1 2 3 4 5

("%5d", -7) → ␣␣␣-7

("%5d", +7) → ␣␣␣+7
1 2 3 4 5

only for small no's

* ("%-5d", x) → [illegible]

→ x␣␣␣␣
1 2 3 4 5

("%-5", -x) → -x␣␣␣
1 2 3 4 5

* ("%.xf", e) → prints e such that there is x digits after decimal point

("%.3f", e) → 2.721

("%.3f", 1.23456) → 1.234

CHAPTER-2

* Variables and constants together are called → OPERANDS
* C-follows only one variable on LHS of =

  z = k * l ✓

  k * l = z ✗
* % → modular division

  x%y → print remainder of x÷y

  10%5 → 0
* Sign of remainder is same as sign of numerator

  -10%3 → -1

  10%-3 → 1
* pow( ) → power of a number

  → Works only on real numbers

  → For this to be used, we need #include <math.h>

  pow(3.0, 2.0) → $3.0^{2.0}$
* abs( ), sqrt( ), pow( ), sin( ), cos( ), tan → #include <math.h>

Int float

integer / integer → integer	int / float → float	float / int → float	float / float → float
5/2 → 2	5./2.0 → 2.5	5.0/2 → 2.5	5.0/2.0 → 2.5

* If LHS ≠ RHS, the value of expression in RHS is promoted or demoted to type of variable in LHS

Eg int i;
float f;
i = 3.5; → LHS → int, RHS → float ⇒ i → printed as 3
f = 30; → LHS → float, RHS → int ⇒ f → printed as 30.0

2) float a,b,c;
int s;

s = a*b*c*/100 * 4-3

Here, RHS → float
LHS → int
RHS → All would be promoted to float & the final one would be demoted to int and stored in s.

k → int a → float

Instruction	Result	Instruction	Result
k = 2/9	0	a = 2/9	0.0
k = 2.0/9	0	a = 2.0/9	0.2222
k = 2/9.0	0	a = 2/9.0	0.2222
k = 9/2	4	a = 9/2	4.0
k = 9.0/2	4	a = 9.0/2	4.5
k = 9/2.0	4	a = 9/2.0	4.5

Hierarchy

Priority → *, /, % > +, - > =

*, /, % : Left to right
=: Right to left

* Format specifiers → %d, %f, %c
Escape sequence → \n, \t etc

CHAPTER 3 - If, else statement

- **if-else statement**

```
if (condition)
    statement 1;
else
    statement 2;
```

- **if-else if -statement**

```
if ( )
    printf ( );
else if ( )
    printf ( );
else if ( )
    printf ( );
```

$x == y \rightarrow x = y$
$x != y \rightarrow x \neq y$
$x < y$, $x > y$
$x <= y$, $x >= y$

common mistake → if ( ) ;

CHAPTER 4 - LOGICAL OPERATORS

* && → and
  || → or
  ! → not

* **Conditional operator**

statement ? ① : ②

If statement is true, ① is stored
else ② is stored

CHAPTER 5 — WHILE LOOP

```
i=1; → Initialising the loop counter
* While (condition is true) → Testing of loop counter
{
    code
}
increment; → incrementation of loop counter
```

CHECKS CONDITION & EXECUTES CODE

* i=1

- printf("%d", i++); → prints 1
  printf("%d", i); → prints 2

  (i incremented)

  Here i is printed first and then it is incremented
  PRE-INCRIMENTATION OPERATOR
  which is why (1) is printed first, (1) is incremented and stored to i & in 2nd statement (2) is printed

- print f("%d, ++i); → prints 2
  printf("%d, i); → prints 2

  → First incremented & then printed
  POST-INCRIMENTATION OPERATOR

* i++ → i=i+1
  i-- → i=i-1
  i+=10 → i=i+10
  i*=10 → i=i*10

* lvalue required error → Means something is wrong on LHS of =

* +=, -=, *=, /=, %= → compound operators

CHAPTER-6 → FOR & DO-WHILE LOOPS

* for (initialising the loop counter ; test counter ; incrementation)
  {
  code
  }

Break statement

* When break is encountered in a loop, control automatically passes to the first statement after the loop. That is, the control exits the loop.

Continue Statement

* When continue is encountered in a loop, control automatically passes to beginning of the loop.

* do-while loop

First executes code and then checks condition

```
do
{
  code;
} while ( condition );
```

; → imp, not there in other loops

* do-while loop executes its statement at least once, even if condition fails for the first time.

* What can be done in one loop can always be done ~~in~~ using other two

* for(; ;) → is an infinite loop
  while ( ) → error
  (nothing filled → no condition)

CHAPTER 7: SWITCH-CASE

* switch (integer expression)
{
  case constant 1:
    do this; ①
  case constant 2:
    do this; ②
  case constant 3:
    do this ③
  default:
    do this;
}

* Integer or a character constant — FLOAT NOT ALLOWED
* Constant used in each case must be different from those used in other cases.
* Any C-statement
* Default statement is optional

* In this program ~~this~~ this program checks if (integer constant) == constant 1, if true, ① is executed and all the other statements including default is executed/printed.

* In order to print only one statement, we should introduce a break statement after each c-statement.

* Switch case is just a replacement of long else-if ladder

Eg →

```
#include <stdio.h>
int main()
{ int x;   printf("enter value of x");
           scanf("%d",&x);
  if (x==0)
     printf("x is 1\n");
else if (x==2)
     printf("x is 2\n");
else if (x==3)
     printf("x is 3\n");
else
   printf("x is not 1,2,or 3");
return 0;
}
```

```
#include <stdio.h>
int main(){
  switch(x)                    checks if x=1
  {case 1:
     printf("x is 1\n");
     break;  ──→ suppose x==1 is satisfied & there is no break all expression bellows this will get printed until we reach the next break
  case 2:
     printf("x is 2\n");
     break;
  case 3:
     printf("x is 3\n");
     break;
  default:
     printf("x is not 1,2 or 3");
  }                  ──→ If none of the cases are satisfied, default gets executed
  return 0;
}
```

* Control cannot be used in switch.

* exit() ⟶ It is a library which helps to exit the program
  ⟶ #include <stdlib.h> is necessary

Syntax:-

```
#include <stdio.h>
void message();          ---------> Function prototype declaration
int main() {
printf("Anything");
message();  ---------> Function call
return 0;
}
void message()  ---------> Function defination
{
  printf("What is to be executed on ur message() function");
}
```

* Here main() ⟶ Calling function
  message() → called function.

* Passing values b/w Functions

```
#include<stdio.h>
int calsum(int x, int y, int z);   // we are using int ⇒ calsum() returns an intiger value
int main() {
int a,b,c,sum;
printf("enter three numbers");
scanf("%d %d %d", &a, &b, &c);
sum = calsum(a,b,c);   -> we pass this value(a,b,c) to calsum() ie a→int x, b→int y, c→int z
printf("%d", sum);
return 0;
}
int calsum(int x, int y, int z).
{int d; d=x+y+z; return(d); }
```

CALLED ACTUAL ARGUEMENTS: a b c

> x,y,z are actually called PARAMETERS, & the value of x(a here), y(b here), z(c here) are called FORMAL ARGUEMENTS

x,y,z → FORMAL ARGUEMENTS

d is returned back to main() and is

- Order of passing arguements

* In called function, the arguements are passed from right to left.

```
void ___ ()   --> called function
{ int a=1;
  printf (" %d %d %d \n", a, ++a, a++ );
}
```

* Expected out put → 1, 2, 2 ✗
* Since arguements are passed from right to left, a=1 is first fed to a++, ++a & then A

a	++a	a++
3	3	1 ← a=1

∴ Out put → 3, 3, 1

* DEFAULT RETURN TYPE OF ANY function is INT.

int sum(int a, int b) —same as→ int sum (a, b)

* A c-program must contain a main() function.
  If no of function = 1 —must be→ main()
  > 1 —one of it must be→ main()

* Actual arguements → can be ~~int~~ const/ variable (expression
  formal arguements → must be variable

* return (s) → returns both control & value (of s)
  return; → returns only control.

return can only return only one value (variable)

return (a, b) ← 2 variables ✗ INVALID

can be ~~done~~ tackled by using pointers

Chap 9 - POINTERS

* We called functions and passed variable/expressions from main() to a function → "call by value" function

* If we pass location number/address → "call by reference"
* If we pass both address & varibble → "Mixed call". function
*

```
#include <stdio.h>
int main(){
int i=72;
printf("value of i is %d ",i);
printf("value of i is %d",*(&i));
printf("address of i is %u",&i);
return 0;
}
```

& → can be used only with variable

* → can be used with constant/variable/expression

*(&i) → called as "value at address" or indirect operation.

%u → format specifier to print a unsigned number (∵ sign of address of anything is unknown)

* Pointer → variable that stores address of another variable

j → pointer variable

address of i → 87994 → &i

address of j → 87999 → &j

```
int i=72;
int *j ;
```

→ declaration of pointer variable

```
float *j ;
j=&i;
```

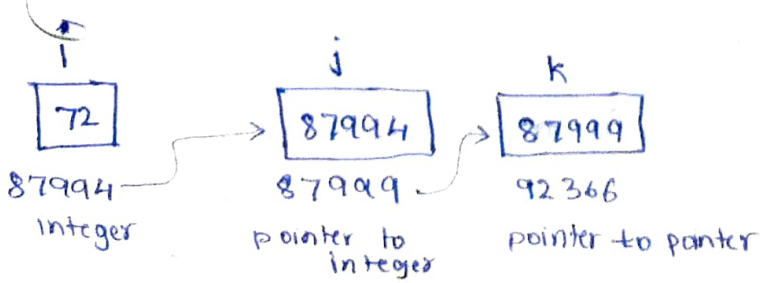
→ This does not mean that j is a float

→ This means that j stores address of a ~~int~~ float.

```
int **k ;
k=&j;
```

→ stores the address of a pointer variable

→ i.e stores address of variable (say j) which ~~address~~ itself stores address of another variable.



* In call by value, the value of actual arguements is passed to formal arguements. changing the values of formal arguements does not alter the output.

```
#include <stdio.h>
int sum(int a, int b); printf("enter a & b"); scanf("%d %d", &a, &b);
int main() {
printf("The value of a and b is %d & %d", a, b);
printf("The sum of a and b is %d", sum(a,b));
return 0;
}
int sum (int a, int b)
{
    int c;
    c = a + b;
    b = 3434;
    a = 2322;
    return c;
}
```

changing value of a & b here wont affect the actual value of a & b, it would remain same as entered by user as value of a & b is passed from main() to sum()

* Swapping values of two variables ★VVIP

```
#include <stdio.h>
void swap(int *a, int *b);
void wrong_swap(int a, int b);
int main() {
int a = 3, b = 4;
```

call by value:

```
printf("The values of a & b before swap is %d, %d", a, b);
wrong_swap(a,b);
printf("The value of a & b after swap is %d & %d", a, b);
```

call by reference:

```
printf("The value of a & b before swap is %d, %d", a, b);
swap(&a, &b);
printf("The value of a & b after swap is %d, %d", a, b);
```

```
return 0;
}
```

```
void wrong_swap (a, b) {
    int t;
    t = a;
    a = b;
    b = t;
}
```

call by value doesnot swap the value as wkt changing values of formal arguements doesnot alter actual arguements

*a = 3	*b = 4	t .	
*a = 3	*b = 4	t = 3	t = *a
*a = 4	*b = 4	t = 3	*a = *b
*a = 4	*b = 3	t = 3	*b = t

In these statements values in RHS is being assigned to LHS [This is true in every case, I have written this only for reference since this is required in this]

```
void swap(*a, *b) {
    int t;
    t = *a;      -> t = value at address of a
    *a = *b;     -> value at address of a = value of address of b
    *b = t;      -> value at address of b = t
}
```

output

```
The value of a & b before swap is 3 & 4
The value of a & b after swap is 3 & 4
```
} WRONG - SWAP, Call by Value

```
The value of a & b before swap is 3 & 4
The value of a & b after swap is 4 & 3
```
} Call by reference

* Returning two or more variables → In order to do this we pass address of variable.
  - Calculating sum and average of two numbers

#include

PTO

Note: Here we are declaring sum & avg. as pointer.

```c
#include<stdio.h>
void sum_and_avg(int a, int b, float *sum, float *avg);
int main(){
int a, b;
float sum, avg;
printf("Enter two numbers");
scanf("%d %d", &a, &b);
sum_and_avg(a, b, &sum, &avg);
printf("The sum of a & b is %f", sum);
printf("The average of a & b is %f", avg);
return 0;
}
void sum_and_avg(int a, int b, float *sum, float *avg)
{
 *sum = a+b;
 *avg = (a+b)/2;
}
```

We are declaring sum & avg here only coz it will be altered in function and stored and then printed.. i.e we are sending address of variable sum and average so that the function stores the values of sum and avg in it & we print it later

We use pointers

1) To change value of actual arguement

2) To return more than one variable

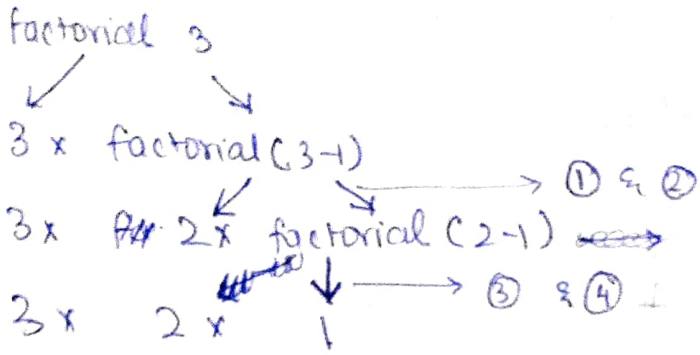
Chapter 10 → Recursions

* Recursions ⟶ Alternatives for loops
* A function is recursive if a statement within the body of the function calls the same function. called CIRCULAR DEFINATION
* Whenever we make a function call, the parameters and the return address get stored at a place called stack. The stack unwounds when control returns from the called function. Thus, during every recursive function call, we are working with a fresh ste of parameters.
* While writing recursive functions, it must have an if statement.
* Any function including main() can becomes recursive function
* To find factorial

```
#include <stdio.h>
int factorial(int);
int main(){
int x;
printf("enter a number");
scanf("%d",&x);
factorial(x);
printf("The factorial of %d = %d", x, factorial(x));
return 0;
}
int factorial(int x){
    if (x==0 || x==1)  ⟶ Called as base condition
        return 1;
    else
{ int f = x * factorial(x-1); ⟶ Calling itself
  return f;
}
}
```

Condition/ statement which doesnt call the function (itself)

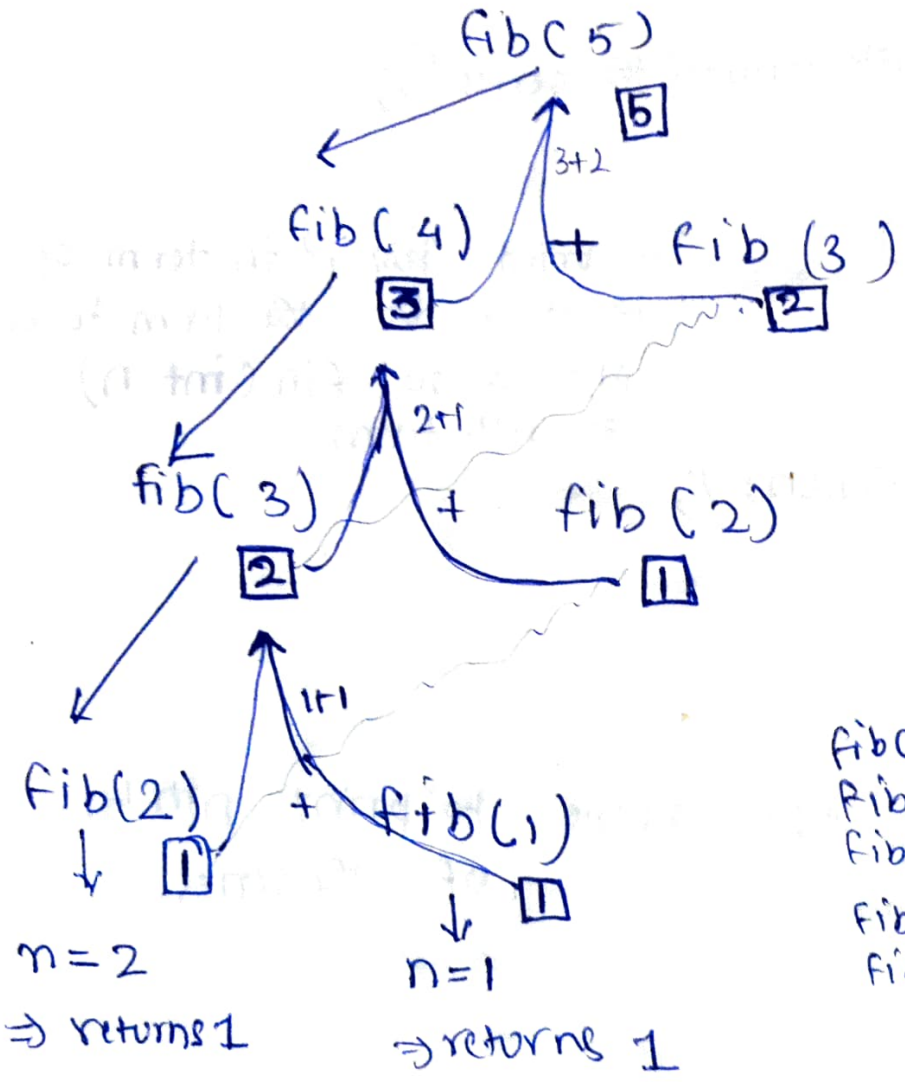
x = 3



* Printing Fiboracci series → 1, 1, 2, 3, 5, 8, 13, 21, ...

fib(n) = fib(n-1) + fib(n-2)

```
# include <stdio.h>
int fib(int n);
int main()
{
   int n, i;
   printf("Enter the nth term of the series");
   scanf("%d", &n);
   for (i=1; i<=n; i++)  ---> To print 'i' th term of series
   {                          ie it feds the term to be
                              printed to fin(int n) till
     fib(n);                  the nth term
     printf("%d /t", fib(n));
   }
   return 0;
}
int fib(int n)  ---> This helps to print nth term
{                    of the series.
   if (n==1 || n==2)
     return 1;
   else
     return(fib(n-1) + fib(n-2));
}
```



fib(5) 5
Fib(4) 3
fib(3) 2
fib(2) 1
fib(1) 1

CHAPTER II: DataTypes

[1 byte = 8 bits]

[| | | | | | | |]

1 bit → 1 byte

Primary Datatypes

- int
  - short (2 byte int)
    - signed
    - unsigned
  - int (4 byte int)
    - signed
    - unsigned
  - long (4 byte int)
    - signed
    - unsigned
- float
  - float
  - double
  - long double
- char
  - signed
  - unsigned

short ↔

- Integer

int

- Short (2 byte int)
  - ↓
  - * atleast 2 bytes big
  - * Never bigger than int
  - 2 bytes in VS code (32-bit)
  - 2 bytes in Turbo C (16-bit)
- int (4 byte int)
  - ↓
  - * Never bigger than long
  - 4 bytes → VS code
  - 2 bytes → Turbo C
- long (4 byte int)
  - * atleast 4 bytes big
  - 4 bytes → VS code
  - 4 bytes → Turbo C

* Range → 2 byte → -32768 to +32767

4 byte → -2147483648 to +2147483647

(%od) (%ld) → long signed int

* signed int → left most bit stores sign → 0 if +ve, 1 if -ve
* unsigned int → all bits stores numbers → range → 0 to 4294967295 (double of i)
  - (%ou) (%lu)
    - ↓ for long unsigned int
  - as left most bit is not used to store sign of no & is free to store value of no
  - for 4 byte
  - 0 to 65535 for 2 byte

* while declaration, int & signed is default

```
short a;            → short signed int a
short unsigned b;   → short unsigned int a
int i;              long x;
unsigned j;         long unsigned y;
```

• Character

Char (occupy 1 byte)

signed (%c)
* Range → -128 to +127
* declaration → char x;

unsigned (%c)
* range 0 to 255
* declaration → unsigned char y;

* signed → default

```
# include (stdio.h)
int main()
{
  char ch = 128;
  char dh = -132;
  printf ("%d, %d", ch, dh);
  return 0;
}
```

→ max range of char is 127, so 128 is 1 more than 127. So 1st value from -ve side i.e -128 is assigned to ch.

→ max range of char is -128, so -132 is 4 less than -132, so 4th value from +ve side side i.e 124 is assigned to dh

OUTPUT → -128, 124
(ch, dh)

• float

float

float (4 bytes)
* Range → -3.4×10^{38} to $+3.4 \times 10^{38}$
(%f)

double (8 bytes)
* Range → -1.7×10^{308} to $+1.7 \times 10^{308}$
(%lf)

long double (10 bytes)
* Range → -1.7×10^{4932} to $+1.7 \times 10^{4932}$
(%Lf)

* A small integer say 23 is small enough to be int, but if we want it to be treated as long, a suffix L or l is added → 23L
* 3.14 is by default double, if we want it to be treated as float an 'f' suffix is used, 3.14f

---

* Range can be calculated by knowing the storage of prototype

range → 2^{bits} → eg → for a short int → 2 bytes → 16 bits = $2^{16} = 65536$

$\frac{65536}{2}$

= 32768

∴ Range → −32768 to 32767
since it has 0 to

unsigned → 0 to 65535

- similarly → 4bytes → 4×8 bits → 2^{32} → 4294967296 → ÷2 → 2147483648
(long int, int)

unsigned
0 to 4294967295

signed
−2147483648 to
+2147483647

- Storage classes ⇒ location of storage, Default value

Storage class → scope of variable
→ life

Automatic | Register | Static | External

Automatic ↓
Default storage class.

	Storage class	Storage	Default value	Scope	Life
1)	Automatic auto int;	RAM	Garbage value	Block	Till control remains inside block
2)	Register register int;	CPU registers	Garbage value	Block	Till control remains inside block
3)	Static static int;	RAM	Zero	Block	Persists b/w different function calls.
4)	External extern int;	RAM	Zero	Global	As long as programs execution doesnt end

CHAPTER 12 – C Preprocessor

A c-program → is a source code

- **Macro expansion**

```
# include <stdio.h>
# define PI 3.1428
int main()
{
    float r = 6.25, area;
    area = PI * r * r;
    printf("Area = %of \n", area);
    return 0;
}
```

PI → Macro template
3.1428 → Macro expansion → capital letters

* During processing, every macro template is replaced by macro expansion

- **Macro with Arguments**

```
# include <stdio.h>
# define AREA(x) (3.14 * x * x)
int main()
{
    float r1 = 6.25, r2 = 2.5, a;
    a = Area(r1);
    printf("Area of circle = %of \n", a);
    a = Area(r2);
    printf("Area of circle = %of \n", a);
    return 0;
}
```

x → argument

* Donot give space b/w macro template and argument

* → Should be closed by brackets.

a = Area(r1); → a = 3.14 * r1 * r1

a = Area(r2); → a = 3.14 * r2 * r2

expansion:

* Macros can split into multiple lines using \ at end of

```
# define LOOP for(;;) \
    printf("\n"); \
    :
```

* Macro expansion is just similar to functions.
  - → Use functions when the formulas and values are large
  - → Use macros when the formulas and values are small, because using macros would increase the program size.
  - → Function are slower than macros

- **FILE inclusion**

#include <File name> → The command would be searched in only standard list of directories

#include " File name" → The command would be searched in current dictionary as well as standard list of directories

- **Conditional compilation**

#if → if
#else → else
#ifdef → if defined
#ifndef → if not defined
#endif → used to end if
#elif → elseif

```
1   #include<stdio.h>
2   #define A 10
3   #define B 30
4   #define C 20
5
6   int main()
7   {
8       int result;
9       #if (A > B)
10          result = A+B
11          printf("sum = %d", result)
12      #elif (A == B)
13          result = A-B
14          printf("diff = %d", result);
15      #else
16          result = A * B;
17          printf("mul = %d", result);
18      #endif
19      #ifdef C
20          printf("Defined");
21      #ifndef X
22          printf("Not defined;)
23      #endif
        return 0;
        }
```

false ← #if (A > B)
false ← #elif (A == B)
true ← #else
true ← #ifdef C
true ← #ifndef X

OUTPUT

mul = 300
Defined
Not defined

C is defined
X is not defined

Use

* There are 23 lines in this program, using #if, #else.... will reduce the no of lines which would be compiled.
* Here lines 8, 9, 11 & 12 will not be compiled because the condition is false.
* if normal if, else statements are used, whole program is compiled

- Miscellenious directives

a) #undef → undefine

Suppose you have a program #define P, and now you want to change the program and remove all P.

#undef P → it will make all statements with ifdef P false

b) # pragma

1) pragma startup [redacted] → This would place the output of function 1 above main().

pragma exit [redacted] → This could place the o/p of function 2 below main().

NOTE: function 1 & function 2 should not pass or recieve any value

```
#include<stdio.h>
void fun1();
 void fun2(); void fun(3)
# pragma startup fun2
# pragma startup fun1
# pragma exit fun3
int main()
{ printf("inside main\n");
 return 0;
}
void fun1()
{ printf("inside fun1\n");
 }
void fun2()
{ printf("inside fun2\n");
 }
void fun3()
{ printf("inside fun3\n");
```

We want 2 fun to be executed before main → progmas should be defined in reverse order ∴ executed at last.

OUTPUT

```
Inside fun1
Inside fun2
Inside main
Inside fun3
```

2) pragma warn
↳ suppresses warnings (not errors)

pragma warn - rvl (return value)
pragma warn - par (parameter not used)
pragma warn - rch (unreachable code)

* This is useful when while compiling big programs, you would like to remove all errors and then turn your attention to warnings. At that time you can suppress warnings with # pragma warn.

CHAPTER 13- ARRAYS

* Array of characters called → Strings
Array of ints / floats → simply Array

* int marks [n]; → called subscripted variable
↳ Called as subscripts
↳ No of elements to be stored in variable (marks)
n → starts with 0.

marks [2] → not 2nd element, it is the 3rd element.

```
* int i, sum=0, avg, marks [10];
  for ( i=0; i<=9; i++ ) // stores marks of student 1, 2, ..... 10
  {
     printf (" The marks of Student %d: \t", i+1);   0th element → i=0
     scanf (" %d", & marks [i] );                    i+1
  }
  for ( i=0; i<=9; i++ )
  {   sum += marks [i];
  }
  avg = sum/10;
  printf ("average = %d /n", avg);
```

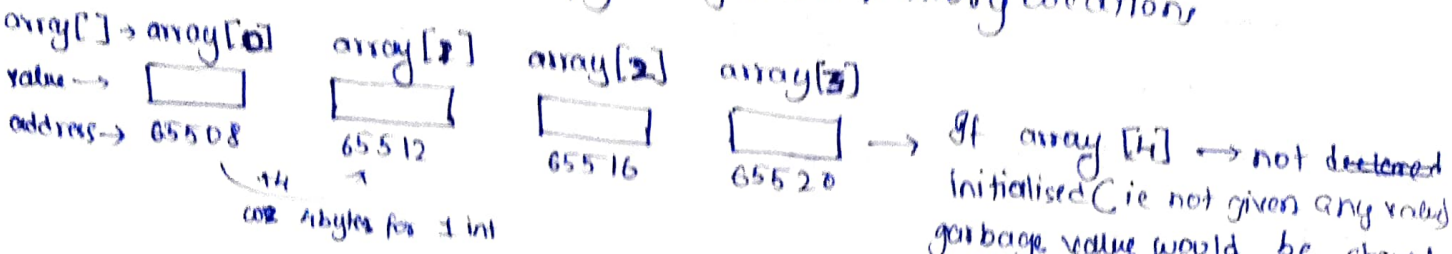
• Array initialisation

```
int num [6] = { 1, 2, 3, 4, 5, 6 }   → Arrays can be initialised while declaring them
int num [ ] = { 1, 2, 3, 4, 5, 6 }   → Both same
```

- Storage

int array[4]; → 4 elements
→ 4 bytes × 4 = 16 bytes

* The values would occupy adjacent memory locations



→ If array[4] → not declared initialised (ie not given any value) garbage value would be stored
→ Storage class is auto by default

- Passing array elements to functions

It can be passed by "call by reference" or "call by value".

```
#include <stdio.h>
void fun1(int )
void fun2( int *)
int main()
{   int array[4] = {1,2,3,4}; i
    for (i=0; i<=3; i++)
    {   fun1(marks[i]);
    }
    for (i=0; i<=3; i++)
    {   fun2 (&marks[i]);
    }
}
void fun1 (int x)
{   printf(" %d", x);
}
void fun2 (int *y)
{   printf (" %d", *y);
}
```

value of array is passed

address of each variable is passed and stored in y

→ value at address of y (which is value at address of array, which is array itself) is printed

- Pointers and Arrays

* Incrementing the pointer variable would add n to the address stored

int it. where → n → no of bytes of prototype of variable
→ 4 if int
→ 4 if float
→ 1 if character

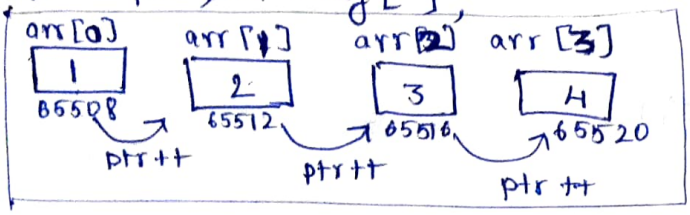
```
int i = 10; * ptr = &i;
printf (" The value of ptr is %u \n", ptr );
ptr ++ ;
printf (" The value of ptr is %u \n", ptr);
```

output

The value of ptr is 65508
The value of ptr is 65512 } +4 as int type which is 4 bytes

use of this

```
int arr [4] = { 1, 2, 3, 4 }
int * ptr = & array [ ];
```



printing -> ("%d", * ptr);
This would change from 1 → 2, 2 → 3, 3 → 4 every time ptr is incrimented.

* As a pointer can be incrimented, it can be decremented too.

1) Addition of a number to pointer

```
ptr = ptr + 4;
```

ptr (65508) → arr[0]
ptr + 4 (65520) → arr[3]
→ Jumps 4 places ahead

2) Subtraction of a number from pointer

ptr -= 4 → Jumps 4 places back

3) subtraction of one pointer from another

say * x = & arr [1]
* y = & arr [3]

x - y → will give the no of blocks from arr [1] & arr [3]

x - y = 2 → arr[1] & arr[3] are 2 blocks apart
arr [1] + 2 = arr [5]

```
ptr = & arr [0];
ptr = arr;
```
Equivalent

4) comparision of pointers (only == & !=) (not <, >)
* only when both pointers points to some array

- Passing arrays to functions

```
#include<stdio.h>
void passingArray(int *, int)
void passingarray(int [], int )
int main()
{
    int num[ ] = {10,20,30,40,50};
    passingArray(&num[0], 5);
    passingarray(&num[0], 5);
} return 0;

void passingArray(int *j, int n)
{   int i;
    for(i=0; i<=n; i++)
    {  printf("element = %d \n", *j);
       j++;
    }
}
void passingarray(int j[], int n)
{ int i;
  for(i=0; i<=n; i++)
  {  printf("element = %d \n", j[i]);
  }
}
```

→ Passing array

- num[i] same as → *(num+i) = *(i+num) = i[num]

(num+i)
↓ gives
address of *(num+i)

- Flexible arrays

* If we want to create an array ~~with~~ whose no of elements (size) is unknown ~~and~~ i.e we would like to input the

of array as input while executing the program, we use malloc() function which requires #include<stdlib.h> memory allocation

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int size, *ptr;
    printf("Enter the required size of array\n");
    scanf("%d", &size);
    ptr = (int *) malloc (size * sizeof(int));
    printf("Enter the %d elements \n", size);
    for (int i=0; i<size ; i++)
    {
        scanf("%d", &ptr[i]); // stores the element one by one
    }
    for (int i=0; i<size; i++)
    {
        printf("element = %d \n", ptr[i]);
    }
    return 0;
}
```

Here, we input no of bytes in malloc() it returns base address to ptr of type void *. Which is why it is being typecasted to integer pointer (int *)

Typecasting

syntax

variable

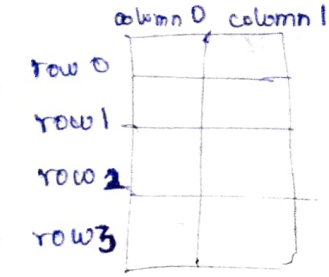
* sorting → Arranging array elements in ascending/descending order
Bubble sort → compares adjascent elements repeatedly
Selection sort → compare 0th element with all others, 1st with others etc.

CHAPTER 14: Multidimensional arrays

• 2-D arrays

array [4] [2]

- rows → No of times input would be given → optional
- Columns → No of elements in each input → compulsory to mention



2d array is two different 1-D array printed one beside other

2D array of 4 1-D array which has two integers

Eg: Roll no & marks

Stud [4] [2]

- 4 times input → i.e 4 students
- roll no & marks

	rollno	marks
Stud 0	1	56
Stud 1	2	86
Stud 2	3	90
Stud 3	4	100

• Initialising 2-D array

```
int stud [4][2] = { {1,56}, {2,86}, {3,90}, {4,100} };
int stud [4][2] = { 1,56, 2,86, 3,90, 4,100};
int stud [ ][2] = { 1,56, 2,86, 3,90, 4,100};
```

• Memory

s[0][0]	s[0][1]	s[1][0]	s[1][1]	s[2][0]	s[2][1]	s[3][0]	s[3][1]
1	56	2	86	3	90	4	100
65500	65504	65508	65512	65516	65520	65524	65528

If we want to print a random variable say, marks of student 3

```
printf("%d\n", stud[2][1]);
```

• pointers and 2-D arrays

```
for (i=0; i<4; i++)
    printf("Address of %od th element of 1-D array = %u \n", i, s[i]);
```

prints address of roll no's i.e 1st column of 2D array

output

Address of 0th element of 1D array = 65500
Address of 1st element of 1D array = 65508
" " 2nd " " " " = 65516
" " 3rd " " " " = 65524

To prove that 2D array is made up of 2-1D array

* The address of 1st 1D array → s[i] i<=0<3
So address of 2nd 1D array → (s[i]+1)
Value of 2nd 1D array elements → *(s[i]+1)
↓ same as
((s+i)+1)

The address of 1st 1D array elements → s[i]
Value of 1st 1D array elements → *(s[i])
The address of 2nd 1D array elements → (s[i]+1)
↓ same as
(*(s+i)+1)
The value of 2nd 1D array elements → *(s[i]+1)
↓
((s+i)+1)

((s+i)+1) → prints address of some garbage value

mat[i][j] → *(*(mat+i)+j)

- Pointer to array

```
int s[4][2]={1,56,2,98,3,90,4,60};
int (*p)[2];   → pointer to array of two integers
int i,j,*pint;
for(i=0;i<4;i++)
{  p=&s[i];   → address of array stored in p
   pint=(int*)p;   → assigned to integer pointer pint
```

parenthesis impt coz without it, it would become array of pointers

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

prints value of elements of 1st column when j=0 & 2nd column when j=1

USED TO PASS 2D array to functions

Passing 2D-array to function

```
int main()
{ int a[3][4] = { 1,2,3,4,
                  5,6,7,8,
                  9,10,11,12 };
  display(a, 3, 4);
}
```

base address

```
void display( int q[][4] ; int row, int column)
{   int i,j;
    for(i=0; i<row; j++)
    {   for(j=0; j<column; j++)
        { printf("%d", q[i][j];
        } printf("\n");
    }
}
```

same as (*q)[4]

pointer to array of 4 integers

- Array of pointers

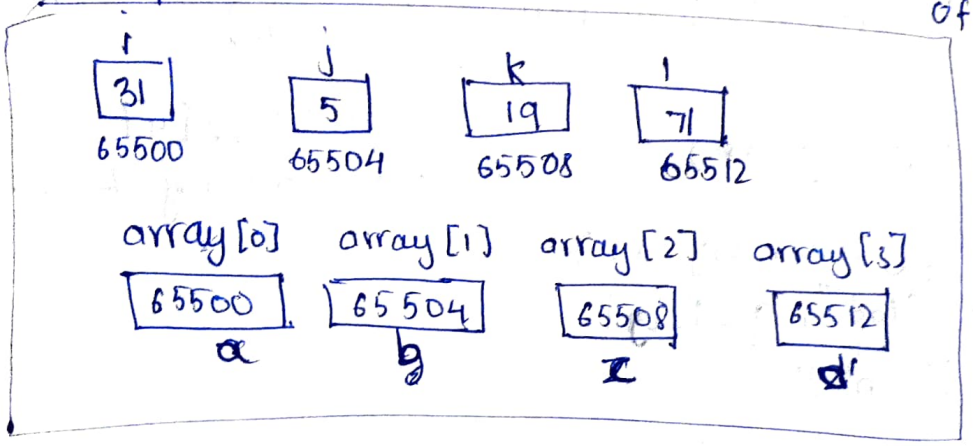
The elements of this array are address of other variables / expressions

```
int *array[4];  ⟶ Array of a pointer
int i=31, j=5, k=19, l=71, x;
arr[0] = &i;
arr[1] = &j;
arr[2] = &k;
arr[3] = &l;
for(x=0; x<4; x++)
{
    printf("%d", *(array[x]));
}
```

→ prints value at address of array[x] i.e value of i, j, k, l

o/p

31
5
19
71



---

* a[i][j] = *(a[i]+j) = *(*(a+i)+j)

CHAPTER - 15 - STRINGS

* Array of characters ⟶ strings → 1-D array terminated by null ('\0')

```
char name[] = {'a','b','c','d','e','\0'}
```

↓ null character

* Storage → Same as arrays i.e contigious memory locations with last memory location stored with \0.
* Initialisation →

```
char name[] = {'a','b','c','d','e','\0'}
char name[] = {"abcde"}
```

ASCII value of null character → 0

"abcde" → without null
"abc def" → Not valid due to space → \0 is automatically inserted

```
char name[] = "ABCDE";
for(i=0; name[i]!='\0'; i++)
{
    printf("%d", name[i]);
}
```

```
char name[] = "ABCDE";
char *ptr;
ptr = name  → assigning address of name[] to pointer
for(; *ptr != '\0'; ptr++)
{ printf("%c", *ptr);
}
```

output → ABCDE

* name[i] = *(name+i) = *(i+name) = i[name]

```
char name[25];
printf("Enter elements \n");
scanf("%s", name);   → %s → format specifier for string
printf("%s", name);
```

scanf( ) → "Debashis Roy" → Not valid due to space

```
char *name[25];
printf("enter your name");   → Debashish Roy   (you can enter any number of names)
gets(name);   → replacement for scanf()
puts("Hello!");
puts(name);   → replacement for printf()
```

OUTPUT → Debashish Roy
Hello!

puts() → can display only one string at a time
→ places the cursor on next line

gets() → receives only one string at a time, but it can receive multi-word string

We can do this using scanf()

```
scanf("%[^\n]s", name);
```
→ It will keep receiving characters into it until \n is encountered

* gets() → function used to store multi-word strings
puts() → function used to print string (print anything)

• pointers and strings

- pointer & string

```
char   string [ ] = "SHIVKARAMAN"; -> printf("%s", string);
char   *string = "SHIVKARAMAN"; -> printf("%s", string);
char   string [20]= "SHIVKARAMAN"; -> printf("%s", string);
```

1) Once a string is defined using char string[ ] or char string[20] it cannot be initialised to something else

2) A string defined using pointers i.e *string = " "; can be re initialised.

3) If we want to change/alter the characters (character by character), it is possible in only string[ ] & string[20] and not *string.

- Standard Libraries

1) strlen() [len → length]

* Counts the number of characters of string excluding null character
* Function is declared under <string.h> header file.

```
#include<stdio.h>
#include<string.h>
int main() {
  char string[ ] = "STRING";
  int len1, len2;
  len1 = strlen (string);        -> passing base address
  len2 = strlen ("Humpty Dumpty");
  printf(" length of string 1= %d \n", len1);   -> New string declared in strlen() itself
  printf(" length of string 2 = %d \n", len2);
}
```

O/P
length of string 1 = 6
length of string 2 = 13 (space also counts)

2) strcpy() [cpy → copy]

* Copies elements of string into another
* <string.h> header file req

```
char source[ ] = "STRING"; target[20];
strcpy (target, source);
```

-> Base address of string in which

```
char source[] = "copy me"; target[20];
strcpy(target, source);
```

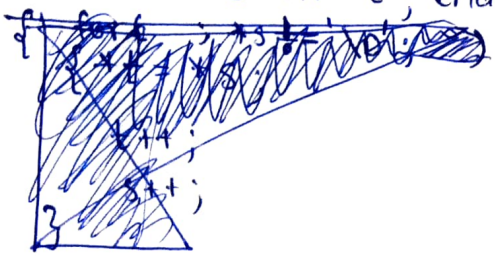
target → Base address of string in which elements are to be copied

source → Base address of original string

How can we do a similar thing using our own function ?

```
#include<stdio.h>
void function(char *target ; char *s);
int main()
{ char source[] = "copy Me"; target[20];
  function(target, source)
  printf("source string = %s", source);
  printf("copied string = %s", target);
  return 0;
}
```

```
void function(char *t, char *s)
{ for( ; s != '\0'; (t++) & (s++))
  { *t = *s;
  }
  *t = '\0';
}
```



→ last element has to be null as it is not copied in above loop

3) strcat()

* Appends characters of one string at end of other string.
* <string.h>

```
char source[] = "Folks", target[30] = "Hello";
strcat(target, source);   → This copies source to end of target string.
printf("source = %s\n", source);
printf("target = %s\n", target);
```

o/p

source = Folks

target = Hello Folks

4) strcmp( ) (cmp → compare)

* Compares two strings
* <string.h>
* Two strings are compared character-by-character until there is a mismatch or till end of string.
* If two strings are identicle, strcmp() returns 0. If not, it returns the difference b/w the ASCII value of first mis-match.

```
char string1[ ] = " Jerry", string2[ ] = Ferry;
int i;
i = strcmp(string1, string2);
printf("%d", i);
```

output → 4 → difference b/w ASCII values of F & J since it is the first mismatch

```
i = strcmp(string1, Jerry boy);
```

output → -32 → The null character at end of Jerry (string 1) doesnot match with blank (space) in Jerry boy. The value returned (-32) is difference b/w ASCII value of null and blank

using user-defined function

```
void function(char * string1[], char * string2[ ], int x);
int main() {
    char string1[ ] = "Jerry", string2[ ] = "Ferry";
    int i;
    i = function(string1, string2, strlen(string1));
    printf("%d", i);
}

void function(char * string1[], char * string2[ ], int x)
{
    int i;
    for(i=0; i<x; i++)
    { if (string1[i] == string2[i])
        continue;
      else
        printf("%d", string1[i] - string2[i]);
    }
```

* null character ⟶ string terminator
* in [redacted] scanf("%[^\n]s",)
^ means from beginning, \n means up to end
* 3 = integer | 3.0 = double | '3' = character | "3" = string

Important library functions

strlen(string); ⟶ length of string
strcpy(target, source); ⟶ copies source to target
strcat(target, source); ⟶ copies source at end of target
strcmp(string1, string2); ⟶ compares two strings
strupr(string); ⟶ converts string to uppercase
strlwr(string); ⟶ converts string to lowercase

CHAPTER 16: Handling Multiple Strings

- 2-D strings

Initialization:

6 names (rows), each name can have max 20 characters (columns)

- char string[6][20] = { "akshay", "parag", "raman", "srinivas", "gopal", "rajesh" }

- char string[6][20];
  printf("Enter the string \n");
  scanf("%s", &string[i][0]);

& important

Note, this is zero, coz this represents the base address of all 1D arrays

Storage

	20 columns →									
65500	a	k	s	h	a	y	\0			
65520	p	a	r	a	g	\0				
65540	r	a	m	a	n	\0				
65560	s	r	i	n	i	v	a	s	\0	
65580	g	o	p	a	l	\0				
65600	r	a	j	e	s	h	\0			

Each 1D string's is reserved 20 blocks even though the no of characters is less than 20. Wasting rest.
This can be avoided using ARRAY OF POINTERS TO STRINGS

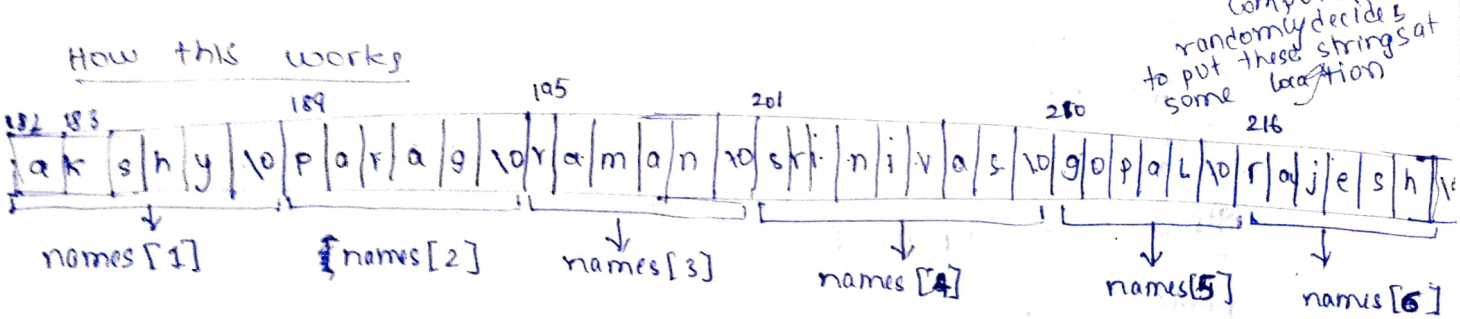
* Array of pointers to strings → Better alternative for 2D string

* It is simply array of pointers

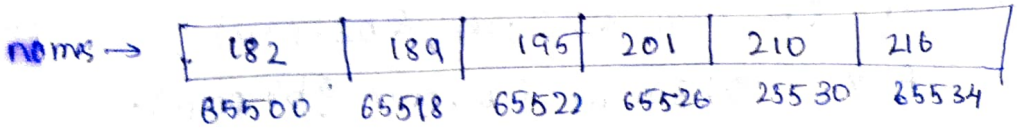
char *names[] = { "akshay", "parag", "raman", "srinivas", "gopal", "rajesh" };

printf("%s \n", names[i]);
└ i => prints akshay

How this works



Now the base address of each string is stored in pointer names[]



Draw back

1) We can initialize the string while declaring only. We cannot use scanf( ) and take input separately from user.

Solution to this drawback → Using an actual string, and malloc() function and copying actual string to Array of pointers

** VIP

This is of 1-D array using malloc() and Array of pointers to string

```
char *name[6], n[50], *p;
int len, i;
printf("Enter the string \n");
scanf("%s", n);
len = strlen(n);
p = (char *) malloc(len+1);   → (len+1) coz to include '\0'
strcpy(p, n);
name[0] = p;
printf("%s", name[i]);
```

CHAPTER 18 - STRUCTURES

* Array & strings → Similar data type
Structure → Dissimilar data types → User defined data type.

Syntax

```
int main()
{
    struct book ──→ Data type defination ──→ structure decleration
    { char name;
      float price;
      int pages;
    } ;
      ↑ important
    struct book b1, b2, b3;
                ↑ variables
    ↑ datatype
    printf("enter name, price and no of pages of three books");
    scanf("%c %f %d", &b1.name, &b1.price, &b1.pages);
    scanf("%c %f %d", &b2.name, &b2.price, &b2.page);
    scanf("%c %f %d", &b3.name, &b3.price, &b3.page);
    printf(
    printf(
    printf(

    return 0;
}
```

- Array of structures

* If no of books increases to say 10, we should create an array

```
int main()
{ struct book{
    char name;
    float price;
    int pages;
    };

    struct book b[10];
    int i, n;
    printf("enter the name, price & pages of books \n");
```

```
for(i=0; i<10; i++)
{
    scanf("%c%f %d", &b[i].name, &b[i].price, &b[i].pages);
    while((ch = getchar()) != '\n')
    ;
}
```

→ Some use as `fflush(stdin)`

→ Without this, when we press enter after entering every input, '\n' is stored as next input.

```
void linkfloat()
{
    float a=0, *b;
    b=&a;
    a=*b;
}
```

→ Just use this function in code

→ This forces linking of floating-point emulator into an application

- Structure declaration

* It doesn't reserve any memory, it just defines it.

```
struct book{
char name;
float price;
int pages;
};
struct book b1, b2, b3;
```

same ⇌

```
struct {
char name;
float price;
int pages;
} b1, b2, b3;
```

```
struct book{
char name[10];
float price;
int pages;
};
struct book b1;
strcpy(b1.name, "Harry");
scanf("%c %f %d", b1.price, b1.pages);
```

⇌ same

```
struct book{
char name[10];
float price;
int pages;
};
struct book b1;
b1 = {"Harry", 130.0, 1200};
```

Note that if we want to assign a string to name[] we should use strcpy()

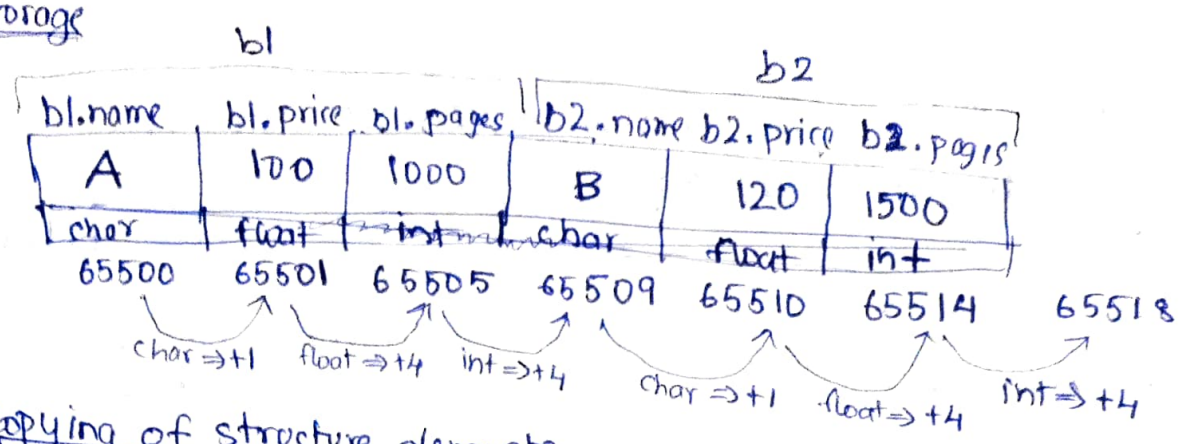
`b1.name = "Harry"` → WON'T WORK

because whst strings cannot beassigned like this

`b1={0}`

All elements of b1 are initialised to 0

- Storage



- copying of structure elements

[1] Copying element by element

```
struct employee
{ char name[20];
  int age;
  float salary;
};
struct employee e1, e2;
e1 = {"Harry", 25, 50,000};
strcpy(e2.name, e1.name);
e2.age = e1.age;
e2.salary = e1.salary;
```

[2] copying all at once

```
struct employee
{ char name[20];
  int age;
  float salary;
};
struct employee e1, e2;
e1 = {"Harry", 25, 50000};
e2 = e1;
```

- Nested structures

```
struct address
{
   char phone[15];
   char city[25];
   int pin;
}
struct employee
{
   char name[20];
   struct address a;
}
struct employee e = {"Harry", "9740333938", "Mysore", "57000"};
printf("\n name : %s \t phone : \t %s
```

```
printf("name : %s  phone : %s
        city : %s  pin : %d", e.name, e.a.phone, e.a.city, e.a.pin);
```

- **Passing structure elements**

```
struct emp
{ char name[20];
  char phone[10];
  int salary;
};
void mixed_call(char *, char *, int)
void call_by_value(struct emp e)
void call_by_ref(struct emp *p)
int main()
{ struct emp e1={"Harry", "9740333938", 50000};
  mixed_call(e.name, e.phone, e.salary);
  call_by_value(e1);
  call_by_ref(&e1);
}
```

This is an integer

These are base address of strings name and phone

```
void mixed_call(char *a, char *b, int c)
{
  printf("%s %s %d", a, b, c);
}
```

stores base address of name

value of salary

base address of phone

```
void call_by_value(struct emp e)
{
  printf("%s %s %d", e.name, e.phone, e.salary);
}
```

data type (structure) of variable e

```
void call_by_ref(struct emp *p)
{
  printf("%s %s %s", p->name, p->phone, p->salary);
}
```

Datatype

pointer of type struct emp

(or) `(*p).name`

(or) `(*p).phone`

(or) `(*p).salary`

- typedef keyword

Used to create a nickname like thing mostly used in structures

```
int main()
{
  struct books
  {
    char name[15];
    int price;
  };
  struct books b1,b2;
  .
  .
  .
}
```

```
int main()
{
  typedef struct books
  { char name[15];
    int price;
  } new;
  new b1,b2;
  .
  .
  .
}
```

→ we are creating an alternate of struct books

→ Here we are using new just like we use int, float or char

CHAPTER-Console I/O

*Console → keyboard and screen

Console I/O functions

Formatted

We format how output should look,
* Place no screen at which o/p it to be printed
* How many places would be present b/w 2 values
* How much places after a decimal point etc.

	I/P	O/P
char, int, float, String	scanf()	printf()

Unformatted

char	getchar()	putchar()
int		
float		
string	gets()	puts()

format specifiers

Data type		Format specifier	Storage
int	short signed	%d or %i	2 bytes
	short unsigned	%hu	2 bytes
	long signed	%ld	4 bytes
	long unsigned	%lu	4 bytes
	unsigned hexadecimal	%x	
	unsigned octal	%o	
float	float	%f	4 bytes
	double	%lf	8 bytes
	long double	%Lf	10 bytes
char	signed char	%c	1 byte
	unsigned char	%c	1 byte
string		%s	

RIGHT ALLIGNED

%5d → 63

%-5d → 63

LEFT ALLIGNED

%5d → -63

%-5d → -63

3.142857

%.2f → 3.14

%.3f → 3.142

%5.2f → 3.14 RIGHT ALLIGNED

%-5.2f → 3.14 LEFT ALLIGNED

String

%5s → RIGHT ALLIGNED

%-5s → LEFT ALLIGNED

%20.10s → Allot 20 columns & print first 10 characters of string (Left Alligned)

Escape sequences

\n → newline
\t → Tab
\b → backspace → moves one space back
\\ → backslash
\' → single quote
\f → form feed
\" → double quote
\a → alert → Alerts user by surrounding speaker of computer
\r → carriage return → Takes cursor to beginning of line in which it is currently

printf(" Lets go\n"); → Lets go
printf(" \" Lets go \n"); → "Lets go
printf(" \" Lets go \" \n"); → "Lets go"

same for \'

- **ignoring characters**

%o*c → ignores '.' or '/' or '-' → Means if '.' or '-' or '/' is entered, it will be ignored

date → dd/mm/yyyy → input

scanf(" %od %o*c %od %o*c %d", &d, &m, &y);

- **Mismatch**

```
char z
char i=25;
printf("%c %d %f", z, z, z);
printf(" %c %d %f", i, i, i);
```

output

%c	%od	%of
z	122 (ASCII value)	Garbage value
i	25	Garbage value

When we use %d against a character, it either prints its ASCII value or value assigned to it

printf() identifies the mismatch and performs specified conversions and prints output

- **sprintf() & sscanf()**

sprintf() → Writes output to an array of characters (string)

```
int i = 10
char ch = 'A';
float a = 3.14;
char string[20];
sprintf(str, "%d %f %c", i, a, ch);
printf("%s", string);
```

→ stores values of i, a, ch and string

→ prints string on screen

sscanf() → Counterpart of sprintf

→ It reads characters from string & stores it in variables.

```
char string[] = "shivkaraman 18";
char name[20];
int age;
sscanf(string, "%s %d", name, &age);
printf("%s \n %d", name, age);
```

O/P

name: shivkaraman
age : 18

- Unformatted console I/O
  - getchar() & putchar()

getchar() → Takes a character input

putchar() → prints character

```
char ch = getchar();
putchar(ch);
```

↓

You should hit the enter key after entering the input for function to digest what you have typed

getch() → No enter required
→ requires <conio.h>

- gets() & puts()

gets() → for multi-word string
→ Terminates when it encounters '\n' (enter key is hit)

puts() → prints string
→ print only one string at a time

```
char string[10];
gets(string);
puts(string);
```

* If input string exceeds the bounds of string, it is dangerous

fgets(base address of string, string size, stdin);

↓

takes input only till (size-1) (cos last is for '\0')

```
char string[10];
fgets(string, 10, stdin);
puts(string);
```

I/P → Shivkaraman is a cs student
1 2 3 4 5 6 7 8 9 10 11

O/P → shivkaram
1 2 3 4 5 6 7 8 9

CHAPTER 19 - File I/O

- FILE pointer

* FILE → structure (a standard structure)
→ Needed for communication b/w file and program

- Opening file

```
FILE *ptr;
ptr = fopen("file name", "mode");
```
→ Opens a file

Opening a file in read mode

1) It searches file to be opened in the disk
2) loads file from disk → buffer
3) It sets up a char pointer which points to first char of file
4) It sets up a FILE structure & returns base address

Reading files

```
int main()
{
    FILE *ptr;
    char ch;
    ptr = fopen("file name.extension", "r");
    while (ch != EOF)
    {
        ch = fgetc(ptr);
        printf("%c", ch);
    }
    printf("\n");
    fclose(ptr);
    return 0;
}
```

- `"r"` → opens file in read mode
- `EOF` → End of file
- `while (ch != EOF)` → print matter in opened file
- `ch = fgetc(ptr);` → assigns matter in file character by character
- `fclose(ptr);` → closes function

- ## Writing to file (write mode)

```
FILE *ptr;
char ch = 'A';
ptr = fopen("file name.ext", "w");
while (
{
    fputc(ch, ptr);
}
fclose(ptr);
```

- `"w"` (write mode) → opens the file (if exists) or create one of given name
- `fputc(ch, ptr);` → over writes (means replaces the matter already present) the content

- ## Closing function

`fclose(ptr);` → Not filename, FILE pointer

- fscanf, fprintf & fgetc & fputc

```
fscanf (ptr, "%c", &ch);   } Read mode
fgetc (ptr);

fprintf (ptr, "%d", num);  } Write mode
fputc (ch, ptr);
```

- Types of functions

1) Text files (.txt, .c)
2) Binary files (.jpg, .dat)

- Opening modes

"r" → read mode → opens file (doesnot create) & sets up a ptr that points to first char
→ Returns NULL if no file exists
→ Reading from file

"w" → Write mode
→ Overwrites the file
→ NULL if unable to open
→ Writing to file

"a" → Creates file or opens file if file already exists and returns the address of last character (Null if unable to o

"r+" →

"a" → opens / creats file, sets up a ptr which points to last char
→ Appends i.e whatever we want is added after the matter already present in file (i.e adding new contents at end of file)

"r+" → read mode + Writing new contents + modify existing content
→ points to first char.

"w+" → Write mode + modifying existing contents (contents are overwritten)

"a+" → Reading + apprnding + cannot modify existing content
→ points to first char

"rb"
"wb"
"ab"
} for binary files

BY SHIVKARAMAN

Counting no of char, spaces, tabs ----

```
FILE *ptr;
ptr=fopen("          ","r");
int nob=0,not=0,noc=0,nol=0; char ch;
While(ch!= EOF)
{
   ch=fgetc(ptr);
   noc++;
   if(ch==' ');
      nob++;
   if(ch=='\t');
      not++;
   if(ch=='\n');
      nol++
}

printf("lines: %d . chars: %d tabs: %d spaces: %d",nol, noc, not, nob);
```

- **File-copy program**

* copying contents of file into another file character by character

```
{ FILE *fs, *ft; char ch;
fs=fopen("source.c","r"); → opening file (source) in read mode
if(fs==NULL)
{
   printf("File cannot be opened\n");
   return 0;
}

ft=fopen("target.c","w"); → opening target file into which contents are to be copied in write mode
if(ft==NULL)
{
   printf("File cannot be opened\n");
   return 0;
}
```

```
while(ch != EOF)
{
  ch=fgetc(fs);  ⟶ storing contents of source file into ch character by character
  fputc(ch,ft);  ⟶ Writing to target file char by char
}

fclose(fs);
fclose(ft);
return 0;
}
```

- **String (Line) I/O in Files**

fgets (base add of string, SIZE, FILE pointer); ⟶ read mode
fputs (base add of string, FILE POINTER);

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
int main()
{
  FILE *ptr;   char s[100];
  ptr=fopen("file.c","w");
  if(ptr==NULL)
  { printf("cannot open file");
    return 0;
  }
  printf("Enter content \n");
  while( strlen(gets(s))>0)
  {
    fputs(s,ptr);
    fputs("\n",ptr);
  }
  fclose(ptr);

printf("READ MODE \n");
```

Here in this program, instead of reading/writing char by char, we read/write line by line with help of strings

WRITE MODE

```
ptr= fopen("file.c", "r");
if (ptr==NULL)
{   printf("cannot open file\n");
    return 0;
}
```

→ 0 - 99 → 100 elements

```
while (fgets(s, 99, ptr) != NULL)
        printf("%s", s);
fclose(ptr);
return 0;
}
```

- Reading in Binary

File opened in text mode → fscanf, fprintf
File opened in Binary mode → fread, fwrite

```
fwrite(address of data to be written, size of data, no of such type of data, file ptr);
```

```
struct threeNum                    WRITING
{
   int n1, n2, n3;
};
int n;
struct threeNum num;
FILE *ptr;
ptr=fopen("filename.dat", "wb");        → write in binary
for (n=1; n<5; n++)
{  num.n1 = n;
   num.n2 = 5*n;
```

```
    num.n3 = 5*n+1;
    fwrite (&num, sizeof (struct threeNum), 1, ptr);
  }
fclose(ptr);
return 0;
```

Reading

```
fread (address, size, number, ptr);
```

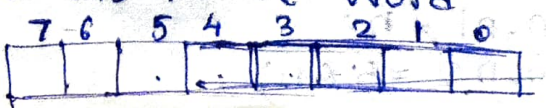
```
struct threeNum
{ int n1, n2, n3;
};
int n;
struct threeNum num;
FILE *ptr;
ptr=fopen ("file.dat", "rb");        → Read in binary
for (n=1; n<5; n++)
{ fread (&num, sizeof (struct threeNum), 1, ptr);
  printf ("n1.: %d  n2: %d  n3: %d", num.n1, num.n2, num.n3);
}
fclose(ptr);
return 0;
```

Chapter 21 : Bits

* C-lang understands → Hexa decimal, octa decimal and decimal numbering
  But Hardware → only BINARY

* 4 bits → nibble
  8 bits → byte
  16 bits → word

32 bits → double-word



7	6	5	4	3	2	1	0
1	0	1	1	0	1	1	0

$= 0 \times 2^0 + 1 \times 2^1 + 1 \times 2^2 + 0 \times 2^3 + 1 \times 2^4 + 1 \times 2^5 + 0 \times 2^6 + 1 \times 2^7$

$= 2 + 4 + 16 + 32 + 128$

$= 182$

- Hexa decimal → [redacted]

Built using combination of 0-9 and A-F. A-F rep → 10-15

Decimal	Hex	Binary	Deci	Hex	Binary	Hex	Binary
0	0	0000	6	6	0110	B	1011
1	1	0001	7	7	0111	C	1100
2	2	0010	8	8	1000	D	1101
3	3	0011	9	9	1001	E	1110
4	4	0100	10	A	1010	F	1111
5	5	0101					

Eg

1100	0101	0011	1010
12	5	3	10
C			A

C53A → Hexadecimal

1) We break a binary number in multiples of 4

- Bit operation 0 x C53A → Actual rep of Hexa decimal

1) set bit 3 to 0 → BIT MANIPULATION (write operation)
2) Check whether bit 6 is 1 or 0 → CREAN OPERATION)

operator	Meaning	
~ (Bitwise NOT)	1's compliment	→ Can operate on int & char and not float
>>	Right shift	
<<	Left shift	
&	Bitwise AND	
|	Bitwise OR	
^	Bitwise XOR	

Function → Showbits ( ) → Shows binary equivalent of integer

1 1's compliment → called as Bitwise NOT operator

operator → ~

```
unsigned char c, cp;
c = 32;
cp = ~c;  → cp → stores 1's comp of
printf("~c = %d\n", cp);   → Here, decimal value of 1's comp of 32 is printed
printf("~c = %ax\n", cp);  → Hexadecimal value of 1's comp of 32 is printed in small case alphabets
printf("~c = %X\n", cp);   → in capital words
printf("~c = %#X\n", cp);  → 0X # is included in Hexa decimal
```

output

```
~ch = 223
~ch = df
~ch = DF
~ch = 0XDF
```

2 Right shift & Left shift

Right shift → >> Left shift → <<

- if ch = 1 1 0 1 0 1 1 1

ch>>1 → shifts all bits 1 place towards right

0 1 1 0 1 0 1 1

same for left shift

ch>>3 → shifts all bits 3 places towards right

0 0 0 1 1 0 1 0

Printing a number in binary

n = 225 →

128	64	32	16	8	4	2	1
1	1	1	0	0	0	0	1
7	6	5	4	3	2	1	0

```
int i;
unsigned char k, andmask;
for(i=7; i>=0; i--)  → 8 bits 0→7
{
  andmask = 1<<i;  → 7 6 5 4 3 2 1 0 : 1 0 0 0 0 0 0 0 when i=7
  k = n & andmask;  → Bitwise and
  if(k==0)
    printf("0");
  else
    printf("1");
}
```

andmask → 10000000
n → 11100001
10000000
k ≠ 0 so prints 1

Shortcut

Left shift → $64 << 2 \rightarrow 64 \times 2^2 = 256$

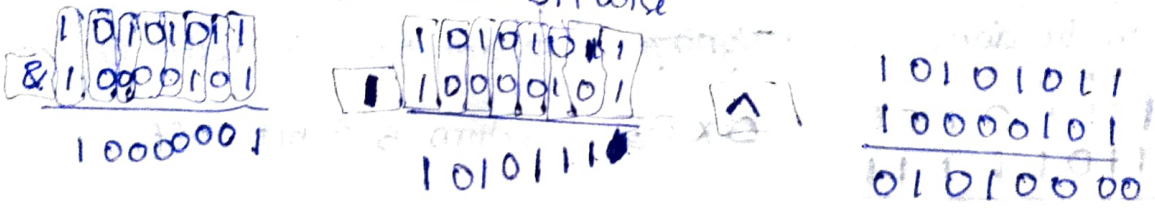
$x << y \rightarrow x \times 2^y$

Right shift → $x >> y \rightarrow \frac{x}{2^y}$

If one of the number whose bits are to be shifted is negative the left most bit (sign bit) would be 1. If we right shift the number by any number, 1 would be preserved at left most bit. i.e -ve sign always remains

- **Bitwise AND, OR & XOR operators**

Operations are performed bitwise



- **Use of Left assignment (<<)**

Used to set a particular bit to 1

Eg: As we did in showbits( ) function

Eg → to insert / set 3rd bit to 1

unsigned char a = 1; → 0 0 0 0 0 0 0 1 (7 6 5 4 3 2 1 0)

a = a << 3; → 0 0 0 0 1 0 0 0 (7 6 5 4 3 2 1 0)

- **Use of &**

1) To check whether a particular bit of an operand is ON(1) or OFF(0)

2) To turn OFF(0) a particular bit

1) if we want to check bit no 5 is ON or OFF in 1 0 1 1 0 1 0 1 (7 6 5 4 3 2 1 0)

unsigned char ch=1, andmask;

andmask = ch << 5;

```
if (andmask & num) == 0)
    printf("The 5th bit (or) bit no 5 is OFF");
else
    printf("The " " " " is ON");
```

1011 0101 → input
00100000 → andmask
00100000 → input & andmask ≠ 0

If we want to assign 1011 0101 as input we use hexa-decimal

B 5

0xB5

Ex instead of assigining 1 to an unsigned char & then left shifting to desired position, we can directly use hexa decimal value

7 6 5 4 3 2 1 0
0 0 1 0 0 0 0 0

2 0

0x20

Also with hexa decimal, we can check if two or more bits are ON or OFF

2) Turning of a particular bit OFF

This can be done by ANDing bit with 0

7 6 5 4 3 2 1 0
1011 0101 → 0xB5 → turn 5th bit off
andmask 1101 1111
ANDing 1001 0101 → 5th turned OFF

- Use of | (or)

Used to turn a bit ON

1011 0101 → turn on 6th bit
ormask 0100 0000
ORing 1111 0101 → 6th bit turned ON

- Use of ^ (XOR)

Used to toggle (change) a bit ON or OFF. A number XORed twice gives original number

Anything ANDed with 0 → 0
Anything ORed with 1 → 1
1 XORed with 1 → 0
0 XORed with 0 → 0

- Compound operators

a = a << 1 → a <<= 1
a = a >> 1 → a >>= 1;
a = a | 0x2A → a |= 0x2A
a = a & 0x4A → a &= 0x4A
a = a ^ 0x2A → a ^= 0x2A

#define _BV(x) (1<<x) (BV stands for Bit Value)

unsigned char c=225; i;

i = c & _BV(3); → same as

x = 1
x <<= 3
i = c & x

CHAPTER – 22 – Miscellenious features

- Unions

* In structure, each member has its own storage location, where as members of union uses a single shared memory location
* The single shared memory location is equal to the size of its largest data member

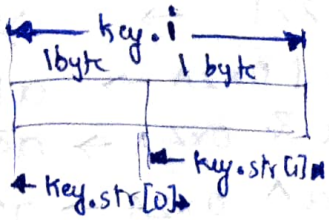
eg struct employee → Tot 50 bytes	union employee → Tot 40 bytes

{ int array[10]; //40 bytes
char string[10]; //10 bytes
};
	{ int array[10]; → 40 bytes (shares memory)
	char string[10]; (shares memory)
	};

- *Union cannot handle all members at once since it is shared memory

Eg

```
union u
{
  short int i; → 2 bytes
  char str[2]; → 2 bytes
};
int main() {
union u key;
 key.i = __ ;
 key.str[0] = _;
  key.str[1] = _ ;
}
```



- Enums → Enumerated data type

* The compiler treats the enums as integers. Each value on the list corresponds to an integer starting with 0.

Eg
```
enum mar_status
{ single, married, divorced, widow
};  0       1        2        3

struct employee
    {
      char name[20];
      int age;
      enum mar_status ms;
    };
struct employee e;
strcpy (e.name, "Harry");
e.age = 23;
```

e.ms = single; → 0

printf("Name : %s \n", e.name);
printf("Age : %d \n", e.age);
printf("Mar_status : %d \n", e.ms);

Name : Harry
Age : 23
Mar status : 0

enumerated data types cannot be used in input using scanf() & printf() i.e input through keyboard

* The integer value of enums can be user defined also.

```
enum mar_status
{ single =100, married =200, divorced =300
};
```

- **Pointers to function in c**

* Mentioning name of array, we get its base address, in the same way mentioning name of function gets its base address.

```
#include<stdio.h>
void display():
{ printf("\n Hi \n");
}
int main()
{
    void (*ptr)(); → This declaration states that it refers nothing & returns nothing
    ptr = display; → Assigning base address of function
    printf("Address of function : %u \n", ptr);
    (*ptr)(); → function call using ptr (or ptr())
    display(); → Normal function call
    return 0;
}
```

syntax of pointer to function

```
return-type (* ptr)(arguements)
```

int, void, ... — actual arguements that we want to pass

eg: `int (*ptr)(int, int);`

- Gets int from function (return-type)
- passes two arguements

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

* If we remove perenthesis from *ptr

`int * ptr (int, int)`

Here, ptr(int, int) is a function taking 2 integer and returns address of an integer variable (int *);

Here int * → Becomes return-type & ptr → function call

- **Function with variable arguements**

stdarg.h
- va_list → standard datatype
- va_start → va_start(ptr, x); → This makes ptr point to first variable in list
- va_arg → ~~xxxxxxx~~

`max = va_arg(ptr, int);` → Will assign integer being pointed by ptr to max and ptr would point to next variable in list

```
#include <stdio.h>
#include <stdarg.h>
int findmax(int, ...);
int main()
{
    max = findmax(5, 23, 15, 1, 92, 50);
    printf("max = %d \n", max);
    return 0;
}
```

→ The no of arguements after 1st arguement is unknown

```
int findmax (int tot_no, ....)
{
    int max, i, num;
    va_list ptr;
    va_start(ptr, tot_no);
    max = va_arg(ptr, int);
    for(i=0; i<tot_no; i++)
    {
        num = va_arg(ptr, int);
        if (num > max)
            max = num;
    }
    return max;
}
```

↑ 5 → This is the total no of variable arguments

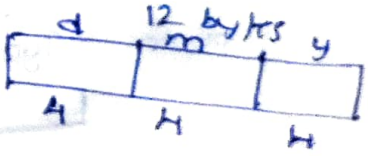
→ pointer declaration

→ ptr points to first variable argument in the list, i.e 23 in this case

→ the integer being pointed by ptr (23) is assigned to max and ptr points to next integer (15)

- **Bit fields**

```
struct date
{   unsigned int d;   → 4 bytes
    unsigned int m;   → 4 bytes   } 12 bytes
    unsigned int y;   → 4 bytes
};
int main()
{
    struct date d1 = { 21, 10, 2002 };
}
```



Here, in this program, each of d, m & y would occupy 4 bytes i.e 32 bytes each. But

date (max) → 31 → binary 11111 → 5 bits
month → 12 → 1100 → 4 bits
year → max unknown

so we are wasting 21 bits and 28 bits of memory in d & m

So we use bit fields

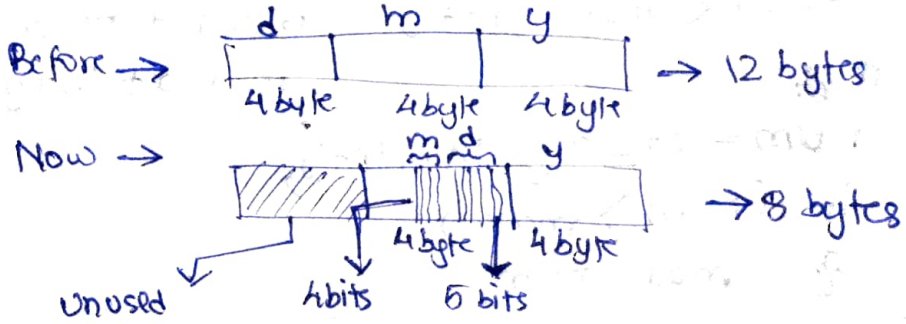
```
struct date
{
    unsigned int d :5 ;
    unsigned int m :4 ;
    unsigned int y ;
};
```

→ means use only 5 bits

→ Use only 4 bits

```
struct date d1 = { 21,10,2002 };
```

~~Here~~ Now



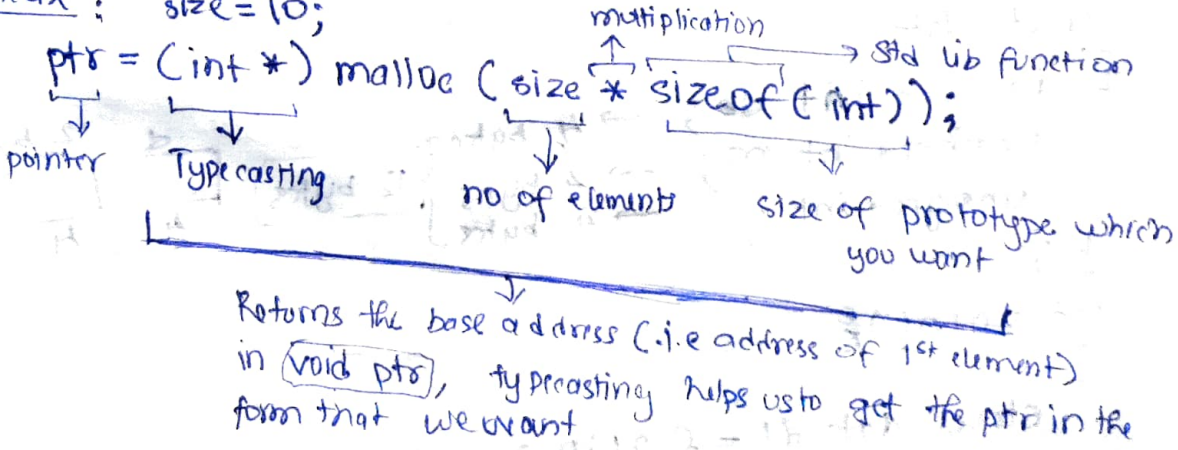
- **Dynamic Memory Allocation**

1 malloc()

* malloc() → memory allocation

Syntax : size = 10;

```
ptr = (int *) malloc (size * sizeof(int));
```



2 calloc()

* calloc() → contigious memory allocation
* It initialises each memory block with default value 0.

syntax

```
ptr = (int *) calloc (size, sizeof(int));
```

[3] free()

* free() → frees the allocated memory i.e deallocation
* The memory allocated using calloc/malloc is not deallocated automatically

Syntax

```
free(ptr);
```

[4] realloc()

* realloc() → memory reallocation

Syntax

```
ptr = realloc(ptr, newsize * sizeof(int));
```

Special functions

- rand() → Generates a random no
  rand()%100 → Generates a number b/w 0 & 100
  rand()%100+1 → Generates a number b/w 1 & 100
- srand(time(0)) → This helps rand() generate different no's each time the code runs
- fscanf() → storing contents from file → fgetc()
  fprintf() → Writing to a file → fputc()
- fscanf(ptr, "%c", &ch); ch=fgetc(ptr);
  fprintf(ptr, "%d", num); fputc(ch, ptr);
- Writes to a string → sprintf()

  Reads char from string and stores it into variables → sscanf()

  sscanf(string, "%s,%d", name, &x); → A string

  sprintf(string, "%d %f %c", a, b, c);

* fflush(stdin);

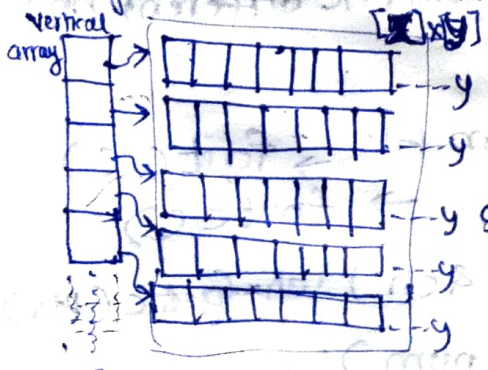
```
while((ch=getchar())!='\n')
    ;
```
ch → int

1-D array dynamically

```
int main()
{
    int *array;
    int size;
    printf("Enter the size of array\n");
    scanf("%d", &size);
    array=(int*)malloc(size*sizeof(int));
    for(int i=0; i<size; i++)
    {
        printf("Enter the %d th element\n", i+1);
        scanf("%d", &array[i]);
    }
```

2D-array dynamically

Say we want to create array [x][y]



Vertical array → It stores pointers that points to ~~one~~ corresponding 1D array of 2D array

Step 1: Creating Vertical array containing pointers that points to 1D array

```
int **array;
array=(int**)malloc(x*sizeof(int*));
```
x → no of rows

int* → pointer, coz we are storing pointers

base address of vertical pointer is returned

Step 2:

```
for(int i=0; i<x; i++)
{
    array[i]=(int*)malloc(y*sizeof(int));
}
```
base address of each horizontal array is returned

to access the element of Vertical array one by one, we run the loop

```
#include <stdio.h>
#include <stdlib.h>
int main()
{ int row, column, **array;
  printf(" Enter size of array in row x column");
  scanf ("%d, %d", &row, &column);
**array = (int **) malloc (row * size of (int*));
 for(int i=0 ; i<row; i++)
     array[i] =(int *) malloc( column * size of (int));
 for(int i=0 ; i<row; i++)
 {
    for (int j=0; j<column ; j++)
    { printf(" Enter elements \n");
      scanf("%d", &array[i][j] );
    }
 }
 return 0;
}
```

- <windows.h> functions

Sleep(delay); → delays the program execution by 'delay' no. of milli seconds, if delay = 1000, the program executes after (usually loop)

- <Windows.h> functions
  - system("cls"); → clears screen
  - Steep (delay);

Outputs are printed in intervals of "delay" ms.

```
while ( )
{
 printf("x");
 Sleep(delay);
}
```

if delay = 1000, Runtime of loop is 1000 milli seconds, i.e x is printed after each loop run

Important points

- **Functions**
  - * Variable/values passed from main → Actual Arguements
    - These variables are stored in function → Parameters
    - and value of parameters is Formal arguements Pg 5
  - * The arguments are passed from R-L in called function

- **Pointers**

  Type of pointer

  1) Wild Pointer : A pointer which is not ~~the~~ initialised while declaring

  int * ptr ; → ptr = [ ] → Garbage value, add → 64450

  2) NULL POINTER :- A pointer initialised to NULL or 0.

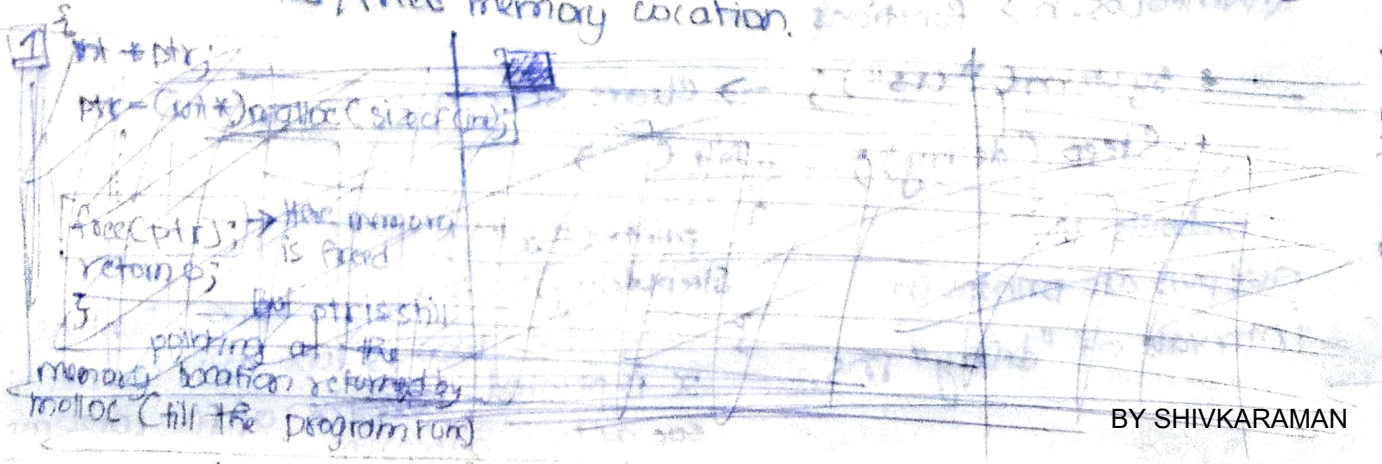
  int *ptr = NULL;
  int *ptr = 0;

  ptr = [0] 64465

  3) Void Pointer : A pointer that points to any type of data i.e char/int/float. If we assign address of a int, it becomes integer pointer, if we assign address of char it becomes char pointer and so on. It can point to any.
  - * Void ptr cannot be dereferenced i.e Value at operator cannot be used directly. However, void pointer can be typecasted and then be deferenced.
  - * Pointer arithmetic is not possible (typecasting makes it possible)

  ^ void * ptr;

  4) Dangling pointer : A dangling ptr is a ptr that points to a deleted/freed memory location.



1) Dynamic memory allocation

```
int main(){
int *ptr = (int *)malloc( sizeof(int));
  .
  .
free(ptr);
ptr = NULL;
return 0;
}
```

→ Here, the memory is cleared, but ptr is still pointing to the address returned by malloc which is been free

→ Now ptr is not dangling as it is not pointing to non-existing memory.

2) On function

```
int* fun()
 { int num = 10;
   return &num;
 }
int main(){
 int *ptr;
  ptr= fun();
 printf("%d", *ptr);
 return 0;
 }
```

→ This is a local variable, and this vanishes when the function call is over. So if num doesnt exist &num also doesnot exist

↓ So

→ This becomes the dangling pointer

→ Shows error SEGMENTATION FAULT

3) Variable out of scope

```
int main(){
 int *ptr;
 { int x;
   ptr= &x;
 }
 printf("%d", *ptr);
 return 0;
 }
```

→ x is a local variable and exists only till the braces close... &x also doesnot exist.

→ Garbage value