file: c.
```
# include <stdio.h>
# include "fun.c"
extern int a = 4;
int main ()
{
printf ("%d", a);
fun ();
}
```

* Arrays:

→ Syntax:

Datatype arrayname[size];
(n)

1st Index [0]
last Index [n-1]

03/02/2022

double cgpa[5];

0	1	2	3	4

occupies 8 x 5 = 40 bytes

cgpa[0] : (n-1) cgpa[3]

* Array Initialization:

* Declare an array to store the roll numbers of 5 students who were absent.

int rollno[5] = {11, 5, 66, 10, 3};

11	5	66	10	3

rollno[0] rollno[1] ... rollno[5]

int rollno[5] = {11, 5, 66, 10};

11	5	66	10	0

→ float marks[3] = {26.5};

must.

26.5	0	0

→ int rno[] = {11, 5, 66, 10};

0	1	2	3
11	5	6	10

OUTPUT:

66

printf("%d", rno[2]);

* printf("%d %d %d %d", rno[0], rno[1], rno[2], rno[3]);

~~for(rno[0];~~ int size;

* for(size=0; size<=size-1; size++)
{
printf("%d", rno[size]);
}

* Assign values to array elements:

int rno[5];

0	1	2	3	4
5	4	26	55	15

rno[0]=5;
rno[1]=4;
rno[2]=30 - rno[1];
rno[3]=55; ↔ (or) rno[3] = rno[0] * rno[1]
rno[4]=rno[0]+10;

Swap the 2nd & 4th elements in the array of roll numbers.

int temp;
temp = rno[1];
rno[1] = rno[3];
rno[3] = t;

* Read the values of array elements.

```
int rno[5];
int i;
printf(" Enter 5 integers");
for(i=0; i<=4; i++)
  {
   scanf("%d", &rno[i]);
  }
```

(or)

```
for(i=0; i<5; i++)
  {
  printf(" Enter the %dth element", i+1);
  scanf("%d", &rno[i]);
  }
```

* Write the statement to copy one array into another.

```
int i;
int rno[5];
int abs[5];
for(i=0; i<5; i++)
 {
 printf("Enter the %dth element", i+1);
 scanf("%d, &rno[i]);
 }
```

```
for(i=0; i<5; i++)
 {
  abs[i] = rno[i];
 }
```

* Write a program to store square of 1st 5 natural numbers into an array & print them. Print their sum.

```
#include <stdio.h>
int main()
 { int i, sum=0;
   int square[5];
  for(i=1; i<=5; i++)
   {  square[i-1] = i*i;
      printf("%d", &square[i-1]);
      sum += square[i-1];
   }
  printf("%d", sum);
  return 0;
 }
```

* * Write a program to read internal marks of 6 subjects & print them.

```
# include <stdio.h>
int main
{ int i;
  int marks [6];
  for(i=0; i<6; i++)
    {
      printf(" Enter the subject marks");
      scanf("%d", &marks[i]);
    }
  for(i=0; i<6; i++)
    { printf("%d", marks[i]);
    } return 0; }
```

* Pass an array to a function :

```
# include <stdio.h>
void display(int rno[], (int r[], int s)
int main()
{
 int rno[5] = {61,62,56,57,5};
  display(rno); //display(rno,10) //display(rno[3])
  return 0;
}
Void display (int r[]) //(int r[], int s)//
{int i;
 for(i=0; i<5; i++)
 {
 printf("%d", r[i]);
 }
 }
```

```
# include <stdio.h>
void display(int r);
int main()
{  int i;
 int rno[5] = {61,62,56,57,5};
 for (i=0; i<5; i++)
   {
     display(rno[i]);
   }
   return 0;
 }
void display(int r)
 {
   printf("%d", &r);
 }
```

```
for(i=0; i<5; i++)
{
  if(abs[i]==55)
  {
    printf("55 is absent");
    break;
  }
}
if(i==5)
  printf("present");
```

05/02/2022

SORTING

BUBBLE SORT:

1) 34 20 3 14

20 34 3 14

20 3 34 14

20 3 14 34 } 1st iteration

3 20 14 34

3 14 20 34

3 14 20 34

3 14 20 34 } 2nd iteration

3 14 20 34

3 14 20 34

3 14 20 34

3 14 20 34 } 3rd iteration

for n numbers: there are (n-1) iterations.

2) 34 40 62 13 49 43 23 11 92 30

1)

34 40 62 13 49 43 23 11 92 30

34 40 62 13 49 43 23 11 92 30

34 40 62 13 49 43 23 11 92 30

34 40 13 62 49 43 23 11 92 30

34 40 13 49 62 43 23 11 92 30

34 40 13 49 43 62 23 11 92 30

34 40 13 49 43 23 62 11 92 30

34 40 13 49 43 23 11 62 92 30

34 40 13 49 43 23 11 62 30 92

2)

34 40 13 49 43 23 11 62 30 92

34 13 40 49 43 23 11 62 30 92

34 13 40 43 49 23 11 62 30 92

34 13 40 43 23 49 11 62 30 92

34 13 40 43 23 11 49 62 30 92

34 13 40 43 23 11 49 62 30 92

34 13 40 43 23 11 49 30 62 92

③ 34 13 40 43 23 11 49 30 62 92.

13 34 40 43 23 11 49 30 62 92

13 34 40 23 43 11 49 30 62 92

13 34 40 23 11 43 49 30 62 92

13 34 40 23 11 43 30 49 62 92

④ 13 34 23 40 11 43 30 49 62 92

13 34 23 11 40 43 30 49 62 92

13 34 23 11 40 30 43 49 62 92

⑤ 13 23 34 11 40 30 43 49 62 92

13 23 11 34 40 30 43 49 62 92

13 23 11 34 30 40 43 49 62 92

⑥ 13 11 23 34 30 40 43 49 62 92

13 11 23 30 34 40 43 49 62 92

⑦ 11 13 23 30 34 40 43 49 62 92.

```
* for(i=0 ; i<n-1 ; i++)
  {
  for(j=0; j<n-1; j++)
    { if (a[j] > a[j+1])
        t = a[j];
        a[j] = a[j+1];
    }} a[j+1] = t;
```

int t;i;j

SELECTION SORT:

1) 18 16 22 12 10.

2) 10 16 22 12 18

3) 10 12 22 16 18

4) 10 12 16 22 18

5) 10 12 16 18 22.

```
for(i=0; i<n-1; i++)
  {  min I = i;
  for (j=i+1; j<n; j++).
    {if(a[j] < a[min I])
        min I = j;
    }
  t = a[min I];
  a[min I] = a[i];
    a[i] = t;
  }
```

* MULTIDIMENSIONAL ARRAYS: [2D] 08/02/2022

datatype arrayname[rows][columns];

int marks[3][5]

	0	1	2	3	4
0	[0][0]	[0][1]	[0][2]	[0][3]	[0][4]
1	[1][0]	[1][1]	[1][2]	[1][3]	[1][4]
2	[2][0]	[2][1]	[2][2]	[2][3]	[2][4]

int a[2][3] = { { 1, 2, 3 }, { 4, 5, 6 } };

(or) int a[2][3] = {1, 2, 3, 4, 5, 6};

* ASSIGNING:

a[0][0] = 10;
a[0][1] = 15;
a[0][2] = 20;
a[1][0] = 11;
a[1][1] = 20;
a[1][2] = 40;

	0	1	2
0	10	15	20
1	11	20	40

if a[0][2] = a[1][1] * a[1][0]
= 800
a[1][0] = a[0][0] + 5
= 15.

```
* for(i=0; i<x; i++)
  {
    for(j=0; j<c; j++)
    { a[i][j] = i*j; scanf("%d", &a[i][j]);
    } }
```

```
for(i=0; i<2; i++)
{
 for(j=0; j<3; j++)
 {
  printf("%d\t", a[i][j]);
 }
 printf("\n");
}
```

```
* int a[2][3], i, j;
  int b[2][3];
  int s[2][3];
  for(i=0; i<2; i++)
  {
    for(j=0; j<3; j++)
    {
      s[i][j] = a[i][j] + b[i][j];
    }
  }
```

1	1	1
2	2	3

2	1	1
5	5	5.

* datatype arrayname[ ][ ][ ][ ] ... ;

int marks[13][65][5];

```
* for(i=0 ; i<13; i++)
  {
    for(j=0; j<65;j++)
     {
      for(k=0; k<5; k++)
      {
      scanf("%d", &marks[i][j][k];
      }
     }
  }
```

* int a[5];
* int a[5][3];

function declaration : void fun(int a[][3]);
function call: fun(a);
function definition : void fun(int a[][3])
int a[5] [3];
func declaration : void fun(int a[ ]);
func call : fun(a[0]);
func defn : void fun(int a[ ])

not mandatory

↓ mandatory (column name)

10/02/2022

The efficiency of the bubble sort is $O(n^2)$.
Sequential sort is $O(n)$.
Binary search is $O(\log n)$

arr[0]	arr[1]	arr[2]	arr[3]
2000	2004	2008	2013

address of i^{th} element = $i \times 4 + 2000$
↓ index

address of i^{th} element = base address + index × size of element

* 11/02/2022

```
→ scanf("%d", &n);
  {
    int a[n];
    :
  }
```

```
void fun(int s, int x[*])

void fun(int s, int x[3])
 {
 }.
```

* Pointer : variable
→ But stores address of another variable.

```
int *p;
int a = 35;
   p = &a;
printf("%d", a);   35
printf("%p", p);   4026   } → address of 'a'
printf("%p",&a);   4026   }
printf("%d", *p);  35.  → value at the address stored in p.
```

* Pre-processor directives:

1) Macro substitution #define CUBE(x) (x*x*x)
2) File Inclusion # include
3) Conditional compilation

12/02/2022

POINTERS: "%p" → display the address

An address expression, uses '&' to get the address of a variable.

A variable's address is the first byte of space occupied by that variable.

```
int a;
int *p;
a = -123;
p = &a
```

p is a pointer to the variable a

A pointer points to no variable contains the special null-pointer constant "NULL"

```
printf("%d %d", a, *p);
```

→ dereferencing expression

```
int x;
int *p = NULL;
int *q;
```

*&x = x
↓
opp. to each

*p * *q
↓ indirection operator; * ↓ multiplication operator

* SYNTAX:

data declaration:
datatype variable;

* pointer declaration:

datatype * pointervariable;

Declaring Pointer variables:

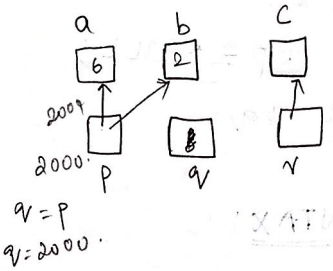
```
char   *p;
int    *n;
float  *x;
```

```
{
 int a;
 int *p;
 a=25;
   p=&a;
  printf("%d %p\n",a,&a);
  printf(" %p %d %d\n", p, *p, a);
 return 0;
}.
```

```
int a,b,c;
int *p;*q*r;

a=6
b=2
p=&b;
q=p;
r=&c;

p=&a;
*q=8;
```




```
*r=*p;

*r=a+*q+*&c;
printf("%d %d %d\n", a,b,c);
printf(" %d %d %d\n",*p,*q,*r);
```

Adding pointers:

```
int a,b,r;
int *pa=&a;
int *pb=&b;
int *pr=&r;
printf(" enter the first no: ");
 scanf(" %d ", pa);
printf(" enter the second no: ");
 scanf(" %d", pb);
  *pr=*pa+*pb;
  printf("\n %d + %d is %d", *pa,*pb,*p
```

→ A single pointer can point towards many variables

→ A single variable can have many pointers pointing towards it.

```
int *p;
int *q;
int *t;

t=p;
p=q;
q=t;
```

* POINTER TO POINTER:

```
int a;
int *p;
int **q;

a = 58;
p = &a;
q = &p;
printf("%3d", a);
printf("%3d", *p);
printf("%3d", **q);
```


```
58
287650
234560
```

OUTPUT:

```
58
58
58
```

* COMPATIBILITY:

integer variable's address can be stored in integer pointer only.

b = *p

⇒ datatype of variable should be same as the datatype of the pointer.

⇒ A pointer (pointer to pointer) variable has size = 4 bytes (irrespective of datatype).

* For hexadecimal datatype ("%0xd").

14/02/2022

p = &a } X. Not possible.
&a = 2000 }

printf("%d", a); } Prints some
printf("%p", &a); } garbage value.

int *p = NULL; → initialization

NULL POINTER: A pointer which points to a null variable.

VOID POINTER:

```
void *v;
int *p;
char *c;
```

p = c; X

```
v = p;
c = v;
```

⇒ c also points to the address of variable which is pointed by p.

* We cannot dereference using a void pointer.

⇒ p = (int *)c

```
double d = 2.5;
int a;   a = (int)d;
```

* POINTER APPLICATIONS:

1) Pointers & arrays:

→ array name 'a' is a pointer constant.

int a[5];

Value at a & address of first element is always same.

```
int a[5] = {10,20,30,40,50};
int *p;
p = &a[0];
printf("%d", a[0]);
        *p
```

- p = a ⇒ 'p' is pointing towards the base address of the array.

- p++ ⇒ Increments by the size of the type of variable it is assigned.

Ex:

```
⇒ int *p;
  p = &a[0];
printf("%d", *p);
  p++;
printf("%d", *p);
  p+=3;
printf("%d", *p);
```

OUTPUT:

10

20

40

- p = &a[1];

```
p--;
printf("%d", *p);
```


p+1; p-1; p++; p--; p1-p2; p1 == NULL

p1 > p2;

```
p = a;
for(i=0; i<5; i++)
 { printf("%d", *(p+i));
 }
```

⇒ p++; ⇒ Ex: for int = increments by 4
double = increments by 8.
char = increments by 1.

* a ⇒ gives the first element of the array

*(a+1) ⇒ gives the second element.

a++ ⇒ not possible.

* To access the element at i^{th} index.

⇒ we can

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

*Write a program to store 5 subject marks into an array & pass it to a function which increments all the subject marks by 2.

```
#include <stdio.h>
 void increment(int a[]); (int *a)
int main()
{ int a[5];
   int i;
  for(i=0; i<5; i++)
                    scanf("%d ", (a+i));
   {
     scanf("%d", &a[i]);
   }.
  increment (a);
          for(i=0; i<5; i++)
           { printf("%d", a[i]); }.
            return 0; } printf("%d",*(a+i))

void increment (int a[ ]);
 { int i;
   for(i=0; i<5; i++)
   {
     printf("%d", a[i]+2);
   }        *(a+i)+= 2;
}.
```

*By default; all the arrays are passed by "CALL BY REFERENCE".

*Write a program to store 10 characters in a character sort the elements using the pointer in descending order.

```
#include<stdio.h>
 void sort (char a[]);
 int main()
 { char a[10];
    int i;
   for(i=0; i<10; i++)
    {
     scanf("%c", &a[i]);
    }
   sort (a);
    for(i=0; i<10; i++)
      {
        printf("%d", &a[i]);
      }
    return 0;
   }
void sort(char a[])
  { int t,i,j;
     for(i=0; i<10; i++)
```

```
{ for(j=0;j<10;j++)
    { if a[j] < a[j+1]
      {
        t = a[j];
        a[j] = a[j+1];
        a[j+1] = t;
      }
    }
}.
}.
```

* x = y+2
  - x → lvalue
  - y+2 → rvalue

lvalue: value would be modified

15/02/2022

* POINTER TO POINTER:

→ It is a variable which stores pointer to another variable.

→ Only an address can be stored in a pointer variable.

* COMPATIBILITY:

→ Do not store the address of data variable of one type into a pointer variable of another type.

→ The size of all the pointers is same i.e. 4 bytes.

→ The dereference type is a type of variable that the pointer is dereferencing.

* VOID POINTER:

→ A void pointer is a special type of pointer; it can point to any datatype.

→ It is ~~can~~ compatible for assignment purpose only with all other types.

* L-value and R-value:

→ An expression is L-value/R-value.

→ An L-value expression must be used whenever it is receiving value.

→ An R-value expression can be used to supply a value i.e. for assigning / receiving.

→ R-value expression should only be on right side.

• L-value can be written on any side.

R-value	L-value
a+2	x[1]
x[1]+10+b	

* POINTER APPLICATIONS:

2 Basic uses of pointers are:

→ use of ~~arrays~~ pointers in implementing arrays.

→ In dynamic memory allocation

* POINTER ARITHMATIC & ARRAYS:

address = pointer + (offset * size of element)

The following operations are valid:

→ p++, p--, p+1, p-1

→ p1-2 ⇒ should give only a valid address.

→ Comparisons ⇒ provided both pointers are pointing towards a single array.

* p1+p2; p1-p2 → These operations are valid; provided it gives a valid address.

* DYNAMIC MEMORY ALLOCATION:

→ The space for all the local declarations as well as addresses (pointer) is allocated in stack area.

→ It allocates space while compiling only.

→ All the variables (run-time variable) for which the memory is allocated during execution, are allocated in heap area.

→ Dynamic Memory allocation is process of allocating the memory to the run-time variables during the execution of program.

* For dynamic memory allocation:

For dynamic memory allocation.

- malloc() } allocation
- calloc() } allocation
- realloc()
- free()

present in <stdlib.h> / <alloc.h>

These are used for dynamic memory management.

→ All these functions will return only void pointers.

```
* # include <stdio.h>
  # include <stdlib.h>
  int main()
  { int *p;
    int n;
    printf("How many integers ?");
    scanf("%d", &n);
    p=(int*) malloc (n*sizeof(int));
    ≡
    free(p); }
```

values initialized with garbage value.

→ An allocated space should be deallocated by the user only.

In stack: allocation & deallocation of memory is automatic.

In heap: allocation & deallocation should be done by the user only..

p = (int *) calloc(n, sizeof(int))

values are initialized with zero.

⇒ specifically used for arrays.

p = (int *) realloc (p, newsize);

free(p);

freeing of the allocated spaces helps to allocate again in large applications.