Barbaros Preveze | b.preveze@cankaya.edu.tr | Room: 106 | 1

__Program.__ A set of instructions with aim to solve a problem

__Instructions:__ Computer language has only 2 alphabets

English (A-Z)
Computer (0-1)

```
0011010110   a      add a and b and
1001101101   b      store to c
----------
Add          <- c
```

C is a high level language with low level features because there's no access to all HW.

$\xrightarrow{\text{input}}$ [Processing] $\xrightarrow{\text{output}}$

__variables:__ Temporary locations in memory of computer to store data

```
type name;          type -> makes the necessary memory
int myintvalue;             allocation
                    name -> to have access to variable
```

integer no → 18 (2 bytes)
real number → 14.27 (4 bytes)
char string → "ABCD" → 5 bytes (Depending on the size of string)

| A | B | C | D | \0 |

↳ Null shows the end of array.

long int → 4 bytes
float → 4 bytes   double more larger location

* C is key sensitive $a \neq A$
* Keywords (for, while, return...) can not be used as variable name
* Numbers can be used in variable names

General Form of a C Program

```
#include <headerfile.h>
int i; // Global variables
main()
{
int j; // variable declarations
printf("hello world"); // Some commands here
f1() // calling some functions
}
```

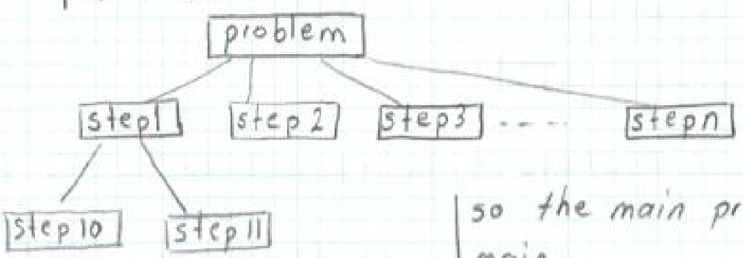
②

```
f1()
{
  → variable declaration

}
```

FUNCTION

An independent piece of a program which can do a predefined work.



so the main program is like

```
main
{
  do step1
  do step2
    :
  do stepn
  done
}
```

* Every program is collections of some functions and they at least have the function main()

* variables declared in functions are limeted to their own function

* A function can call other functions and its also posible that main can call itself.

There are some predefined library functions such as sin(x), pow(x, y) ...

we can assign a constant value $a = 5$
a variable name $a = b$
or an expression $a = b+5$ to any variable.

$i = j = k = 11$ in this case $i = 11$
$j = 11$
$k = 11$

a=f1() => a takes the value returning from function f1()

Basic I/o functions

scanf("format string", & var1, & var2, ...);

read 2 variables one after the other.

%d for int variables
%f for float variables
%c for char variables

Ex: scanf ("%d %f %c, &i , &f1, &ch);
printf (" i= %d , f1= %f , ch= %c ", i, f1, ch)

2 ↵ // user enters
3 ↵ // "
a ↵ // "

i=2 , f1= 3.00 ,ch= a → printf prints output

in printf we can also use some control characters

\n new line
\t tab (8 blanks)
\\ back slash itself

printf ("This is a \ name") → This is a
ame

printf ("This is a \\ name") → This is a \ name

Expressions

1) Mathematical Expressions
2) Logical Expressions
3) Combination of Expressions

General Form of Expressions

operand operator operand (binary operators)
3 * 5

operator operand (unary operator)
- i

operators

-	subtraction	
+	addition	
*	multiplication	
/	division	
%	remainder	
++	increment	
--	decrement	

Ex/

17.2 - 11.3
1 - 3
1 - 13.2 → one of them is int and one is double output will be in double format.
i - J
i - 13.1

Ex 2) 7/14 = 0 → output is in int format
7/14.0 = 0,5 → " " " floating format

Ex 3) 3 % 5 = 3 (3 mod 5 = 3)
14 % 3 = 2

Ex 4) ++ var <=> var = var+1 <=> var++

++var increment variable
return new value
var++ increment variable
return old variable

i=5
J=++i
→ i=6 J=6

i=5
J=i++
→ i=6 J=5

priority of operations is;

- \- + higher priority
- * / %
- \+ - lower priority

Ex 5) a + b * c - d * e
(a+b) * (c - d) * e → using pharanteses I can change the order

Ex 6) $\underbrace{((a+b) * (c-d))}_{T}$ / (c * b)

with (c*b)

result = T

with c*b (without pharant.)

result = T * b

Mathematical Library functions

\# include <math.h>

sqrt(x) $\rightarrow$ x is double computes square root of x ($\sqrt{x}$)

pow(x, y) $\rightarrow$ x^y

Ex: pow(3, 2) = $3^2 = 9$

pow(16, 0.5) = $16^{\frac{1}{2}} = 4$

pow(17.1, 1/3.0) = $\sqrt[3]{17.1}$

pow(17.1, 1/3) = $\sqrt[0]{17.1} = 1$

$\downarrow$

0

ceil(x) x is double, smallest int greater than x

floor(x) " " " , largest " less " "

sin(x)

cos(x)

tan(x)

log(x) $\rightarrow$ log of x in base e

log10(x) $\rightarrow$ log of x in base 10

exp(x) $\rightarrow$ e^x

Expressions:

logical Expressions: A combination of conditional statements, it's either true or false

operand operator operand (operations in conditions)

<		
>	ex:/ 12 < 15 $\rightarrow$	True
<=	i < 10 $\rightarrow$	Results depend on i and J
=>	i <= J $\rightarrow$	
!=		
==	5 > 15 $\rightarrow$	False

priority of conditional statements

< > higher priority

<= >= $\updownarrow$

== != lower priority

operator

&& and → True if both operands are True

|| or → " " one of the operands is True

! not → True if the operand is False

TRUE and FALSE

No boolean types in C

To show true and false we use integers

$\downarrow$ true → 1, false → 0

0 → False (and in logical expressions all non zero

1 → True values are True)

a = 15

a > 10 → 1

a && J < 10

15 && 1

T and T = T = 1

↳ (a value is True if it not "0")

combining logical and mathematical expressions

1) use a mathematical exp. in a logical exp.

2) use a logical exp in a mathematical exp.

$\underbrace{a > b \;\&\&\; J + k}_{\text{logical expression}}$ → a = 1, b = -2, J = 3, k = 4

1 > -2 && 3 + 4

1 && 7

T && T = T

ex/ m = 4

k = 1

l = 2

i = k + l + (m == 1)

i = 2 + 1 + 0 = 3

$\downarrow$ False (m ≠ 1)

ex/ if m is 0 increment it...

m = m + (m == 0)

operator

&& and → True if both operands are True

|| or → " " one of the operands is True

! not → True if the operand is False

TRUE and FALSE

No boolean types in C

To show true and false we use integers

$$\underset{1}{\text{true}} \qquad \underset{0}{\text{false}}$$

0 → False (and in logical expressions all non zero

1 → True values are True)

$a = 15$

$\underbrace{a > 10}$

$1 \leftarrow$

$a \ \&\& \ \underline{J < 10}$

$15 \ \&\& \ 1$

T and T = T = 1

↳ (a value is True if it not "0")

combining logical and mathematical expressions

1) use a mathematical exp. in a logical exp.

2) use a logical exp in a mathematical exp.

$\underbrace{a > b \ \&\& \ J + k}_{\text{logical expression}} \longrightarrow$

$a = 1$
$b = -2$
$J = 3$
$k = 4$

$1 > -2 \ \&\& \ 3 + 4$

$1 \ \&\& \ 7$

$T \ \&\& \ T = T$

ex/ $m = 4$

$k = 1$

$l = 2$

$i = k + l + (m == 1)$

$i = 2 + 1 + \underset{\text{False } (m \neq 1)}{0} = 3$

ex/ if m is 0 increment it...

$m = m + (m == 0)$

ex/ a>=0 && b>=0
!(a<0) && !(b<0)

ex/ a = -2
!(-2<0)
F

ex/ a=0
!a<0
1<0
False

(a=0) => (!a=1)

<u>Control Statements</u>: The change sequence of execution of instructions

if - else
switch-case
?

P1:
I1
I2
I3
I4
I5

<u>if-else</u>:

if (condition) → logical or mathematical expressions
[if - block] → a group of instructions
else
[else block]

```
if (a>0)
a=1
printf ("AB");
else
a=2
```

this will have some errors because we haven't use brackets even there are more than 1 statements in if block

error1: it will print "AB" in all cases

error2: else mismatch (no if found)

```
main()
{
int i;
:
}
```

int i; → we can not use this variable in other functions because it's a local variable

```
f()
{
int J;
=
}
```

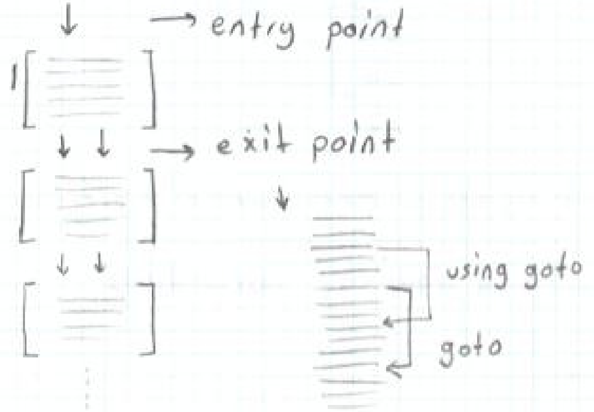
int J; → local variable

```
main()
{
int i;
i=5
if(i<5)
{
int j
J=15
i=11
}
}
```

→ we can declare variables after opening brackets but can not be used outside the blocks

structured programming

P1:




```
if (cond1)
   if (cond 2)
      if (cond 3)
          I1
      else
          I2
   else
      I3
else
   I4
```

I1 is executed if all cond1 cond2 and cond3 are True.

I2 is executed if cond1 and cond2 are True and cond3 is false.

ex/ Read a number if it's even print even else print odd

```
scanf("%d", &i)
if (i%2==0)
   printf("EVEN\n")
else
   printf("ODD\n")
```

switch_case

```
switch ( integer value / var)
{
  case value1:
  statement1;
  breaak;         → if we dont use break it continives
  case value 2:     through the other statements
  statement 2
  break;

  default:
  default statements;
}
```

ex/

```
char ch;
scanf ("%c", &ch)
switch (ch)
{
  case 'A':
  printf ("A is typed /n");
  break;
  case 'B':
  printf ("B is typed /n");
  defaut:
  printf (" A or B is not typed /n");
}
```

* An important Note:

```
char ch='A';
printf ("%c", ch)  → prints 'A'
but
printf ("%d", ch)  → prints int equavalant of A (65)
                     because %d is written inste-
                     ad of %c
```

```
switch (ch)
{
  case 'A':
  case 'B':
   printf ("A or B is typed");
   break
    ;
}
```

we can "or" them but we can not "and" them

example/

```
if (ch==val1)
    statement1;
else if (ch==val2)
    statement2;
else
    default statemets;
```

→ if we would use else instead of else if statement2 would be default statement.

→ if ch not equal to val1 and not equal to val2 default statements will execute.

<u>Usage:</u> (cond) ? <True case>: <False case>

~~ex~~ calculate absolute value of j and assign it to variable i

$i = (J < 0) \; ? \underbrace{-J}_{\text{if Yes}} : \underbrace{J}_{\text{if No}}$

is $(J<0)$

<u>Nested ? operations</u>

$$(\text{cond1}) \; ? \; \underbrace{< \text{True case}}_{\text{cond1} = T} : \underbrace{(\text{cond2}) \; ? \; \overbrace{< \text{True Case} >}^{\text{cond1}=F,\ \text{cond2}=T} : \overbrace{< \text{False case} >}^{\text{cond1}=F,\ \text{cond2}=F}}_{\text{cond1}=F}$$

ex/ calculate signum of i and multiply with b

$$\left.\begin{array}{lll} \text{if} & i<0 & i=-1 \\ \text{if} & i>0 & i=1 \\ \text{if} & i=0 & i=0 \end{array}\right\} \quad ((i>0)\;?\;1 : (i<0)\;?\;-1 : 0) * b$$

loops:

we have 3 different forms in looping and these are for, do-while, while

for loop

for (initialisation part, condition part, increment/decr.)
  Body of the loop

initialisation: Executed once when we first came to for loop
condition: in each iteration it's evaluated before execution of body loop
Inc / Dec: in each iteration it's evaluated **after** execution of loop Body.

```
for ( I ; C ; inc/dec)
    I
    C → True
    B
    Inc/Dec
    C → True
    B
    Inc/Dec
    C → False
      exit the loop
```

example/ write a program in only 2 lines that reads inputs from the keyboard until a "0" is entered and adds it to sum.

```
for (scanf ("%d", &a); a!=0; scanf ("%d", &a))
sum += a;
```

ex:/ for (i=0 ; i<10 ; i++)
; // Null Loop

} it creates delay in the program

ex:/ for (; ;) → infinite loop
body → Repeats the body forever

while condition

```
while (condition)
   Body;
```

```
while (i<10)
do-sth;
```

ex/
```
sum=0;
scanf("%d , &a);
while (a!=0)
{
  sum = sum+a;
  scanf("%d", &a)
}
```

```
for (scanf("%d", &a); a!=0; scanf("%d , &a))
```

do-while loop

```
do
 {
  Body of loop
 } while (condition);
```

→ We have to use the brackets even we write a single statement.

in do-while the statement executes at least once because the condition is checked after executing the statement here.

ex/
```
sum=10
do
 {
  scanf("%d", &a)
  sum=sum+a;
 } while (a!=10);
```

here sum = 10 + a

```
sum=10
while (a!=10);
{
scanf("%d", &a)
sum=sum+a;
}
```

here sum=10

Data Structure: A set of memory locations (variables) with a common name

A	B	C	D

data1

index: 0 1 2 3 4 5

data2

Arrays: A set of variables of same type with a common name and indexes to access each part

```
char array1[10]; → global array
{
char array2[10]; → local array
 :
}
```

① Decleration: type name[size];

ex/ int A[0]; float B[4]; (it's a static decleration arrays are static variables)

P1
↓
compile → static
↓
execution → Dynamic

For example memory allocation for arrays is done during compiling the program so it's static. But scanf is done during execution and it's dynamic.

we can not write int A[n] becouse the program must know the n value and make the memory allocation according to this value.

② Access to elements of arrays

use indexes from 0 to size-1

Ex:/ int A[4]   A[0]=0
A[1], A[2], A[3]

Ex:/ int A[100] → 0 - 99

```
for(i=0; i<100; i++)   } Read values for all
 scanf("%d", &A[i]    } elements of array A
```

③ initialization:

a) in decleration Ex:/ int A[4] = {1, 2, 3, 5}
A[0]=1, A[1]=2, A[2]=3, A[3]=5

Ex:/ int B[ ] = {2, 3, 5, 7}
↳ the size will be four in this case

Ex:/ char C[ ] = {'A', 'B', 'C', 'D'};
C[0] C[1] C[2] C[3]

int A[2] = {1, 2, ~~3~~}
2 elements so it will not work.

b) Global arrays : global variable save initialized to zero

c) Using loops

```
int A[100]
for (i=0; i<100; i++)
 A[i]=0;
```

Problem: write a program to $\overbrace{\text{read 100 int number}}^{I}$ into an array then $\underbrace{\text{sort them}}_{II}$, and $\underbrace{\text{print the sorted form.}}_{III}$

I) Read values
II) Sort them
  a) find largest value → assume 1st as largest
     compare it with others
  b) bring it to begining
  c) repead a, b for remaining elements
III) Print outputs.

```
#include <stdio.h>
main()
{
 int data[100];
 int k, J, temp;
I {for (i=0; i<100; i++)
  {scanf("%d", &data[i])
```

↓

```
for (i=0, i<100; i++)
 {
 j= i;
 for (k=i+1; k<100; k++)
   if data [k]> data [j]
   j=k
   temp= data[j]
   data[j] = data[i]
   data [i] = temp
 }
 for(i=0; i<100; i++)
 printf ("%d, & data[i]);
 return 0;
 }
```

II.a — the `for (k=i+1 ...)` block

II.b — the `temp` swap block

III — the final `for`/`printf`/`return` block

STRINGS (Char arrays)

String: a set of characters   ex/ "ABCDE"

in c there's no string type we use character arrays for strings

decleration examples:

char ch[20];   \0 ≡ NULL character and it's ascii code is zero ("0")

ch[0] = 'A'
ch[1] = 'B'
ch[2] = 'C'
ch[3] = 'D'
ch[4] = '\0'

if we want to create an array with size n we must create it with size of n+1 (1 for the null character.)

'A' → a character $\boxed{A}$
"A" → a string with NULL character $\boxed{A|\backslash 0}$

char c[ ] = "ABCDE";   is similar to creating an array with 6 characters. But since it has 5 characters in, strlen[c] = 5

String library functions

1) strlen (char c[ ])

strcpy (char D[ ], const char S[ ])

strcpy (c, "Name") → copies string Name to c and puts a null char to the end.

c[0] = 'N'
c(1) = 'a'
c[2] = 'm'
c[3] = 'e'
c[4] = '\0'

~~c[ ] = Name~~ can not be used we have to use strcpy for this.

strcmp (S1, S2)   (−) if S1 < S2   'a' > 'A'
 0 S1 = S2
 (+) if S1 > S2

Burada dönen deger farklı olan ilk karakterlerin integer karsılıkları arasındaki farktır.

```
ex/ char S1[ ] = "ABCD"
    char S2[ ] = "CDE"
    int r
    r = strcmp (S1, S2);
    r = -2
```

__important Note:__

```
int A[4];
A[6] = 1;
```
it will overwrite 1 onto something else and c will not prompt us so we must be carefull on this point.

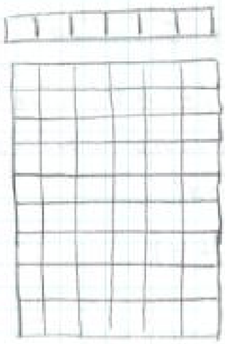
__strcat (S1, S2)__

```
Ex/ char S1[7] = "ABC"
         S2[ ] = "CDE"
         int r;
         strcat (S1, S2)
```
⟶ $\overbrace{A \mid B \mid C \mid C \mid D \mid E \mid}^{S1}$

we wrote 7 for size of S1 otherwise it doesn't fit to S1 together with S2.

* An array which elements itself is also an array

```
char temp[6];      // aranacak değer
char data[9][6];
```



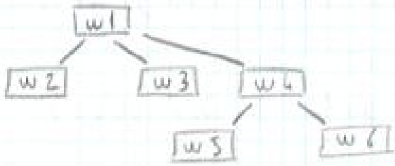
array of arrays

```
for (i=0; i<9; i++)
  scanf("%s", data[i]);
scanf("%s", temp)
```

```
for(i=0;i<9 && strcmp(data[i],temp); i++)
     ; //do nothing
```

//when data[i] equals to temp, strcmp will give "0" and it will mean false and break the for loop the value stored in "i" is index of the data that we are looking for. if i=9 then it will mean the "for loop" is not broken until the end of the array, and It means the data is not found.

Functions: Function is a part of a program independent from other parts, may have local variables it can be used more than once.




```
w1()
 {
  w2()
  w3()
  w5()
  w6()
 }
```

Decleration of Functions:

ret-type name (parameter list)
(int, double)

```
#include <...>
decloration of functions
main()
{

}
functions
```

Example:

```
int f1 (int i, int j, float k);
```

if we declare it as; int f1 (int, int, float)
the first and second variables must be int and third must be float.

```
int f1 (int i, int j)
{
  if (i<j)
  return (i-j);
  else
  return (i+j);
}
```

```
void f2 (int i, int j)
{
  - - -


   return;
}
```

if it's a void function no value retuns from the function so we write only return to the end of function.

ex/
```
main()
{
main()
:
```
infinite

Example: write a function to find largest value in an array and return its index (largest value)

```
#include <stdio.h>
int findmax (int data[], int size)
main()
{
int array[10], max;
for(i=0; i<10; i++)
scanf("%d", & array[i]);
max= findmax (array, 10);
printf("The maximum value is %d \n", max);
return;
}
int findmax (int data[], int size)
{
  int i, j, max;
```

```
max=data[0];
i=0
for(j=1; j<size; j++)
  if(data[j] > max)
    max = data[j]
    return max;
}
```

A function can call other functions and it also can call itself. if a function calls itself it's called recursive function.

Recursion:

Recursive functions are easier to implement but they're slower.

in recursive functions second call of the function must be simpler than the first call (for example $n-1 < n$)

And there should be a simple case where computation is easy and function should check this case at the begening.

Example:

$$n! = n * \underbrace{(n-1)!}$$

$$\downarrow$$

$$(n-1) \cdot \underbrace{(n-2)!}$$

$$\downarrow$$

$$\cdots$$

$$\downarrow$$

$$2 \cdot \underbrace{1!}_{\text{simple case}}$$

Design a Recursive Function

1) Define the problem using the recorsive method (define the simpler step which can be used to solve the difficult one
2) Define the simple case

3) implement function defined in step1 by checking simple case at the begining otherwise function will be called infinitely.

Example: Find sum of n int values by using a recursive function.

1) Sum of n int values = sum of (n-1) + lastone
2) if n=1, sum is that value → simple case
3) int sum(int data[], int size)

```
{
  if(size==1)
  return data[0];
  else
  return sum(data, size-1) + data[size-1];
}
```

Example: Find the factorial value of a given int value

1) fact_of(n) is n * (fact-of(n-1))
2) fact of 1 is 1 → simple case
3) int fact-of(int n)

```
{
 if(n==1)
 return 1;
 else
 return n * fact-of(n-1);
}
```

STRUCTS: A group of elements of different types. '.' is used to access to these elements.

Ex/a) Create a record for a student that has elements;

name, ID, CGPA, credits, address, Tel

```
struct student
{
 char name[30];
 long ID;
 float CGPA;
 int credits;
 char address[100];
 char Tel[12];
};

struct student S1;
```

b) read details of a student into this record

```
scanf("%s %d, %f %d %s %s", S1.name, S1.ID
S1.CGPA, S1.credits, S1.address, s1.Tel);
```

c) printout the details that you read in previous step

```
printf("Name is % \n  s1.ID is %d \n  s1.CGPA  is
%f \n", s1.name, S1.ID, S1.CGPA, S1.credits, S1.address
.S1.Tel);
```

d) Use the same record definition to store n students in details (n=10)

```
type name[size]; ⟶ array declaration format
struct student S2[10];
for (i=0; i<10; i++)
  scanf ("%s %ld %f %d %s %s, s2[i].name
&s2[i].ID, &s2[i].CGPA, &s2[i].credits, s2[i].ad
ress, s2[i].Tel);
```

& sign is put if the variable is not an array.

e) sort the list according to name list

```
for (i=0; i<10; i++)
  for (j=i+1; j<10; j++)
    if (strcmp( s2[i].name, s2[j].name >0)
```

```
{
  temp = s2[i];
  s2[i] = s2[j];   → we can use assignment ("=") for
  s2[j] = temp;        struct elements
}
```

POINTERS

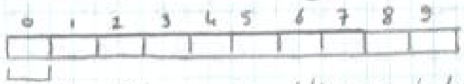
Address: an integer value which shows the location of a byte in the memory. Byte is the smallest addressable unit in the memory.

pointer is a variable to store the address of a location in the memory.

source program → $\boxed{\text{C Compiler}}$ → executable program

This means C compiler needs the addresses

when we create an array;

int array[10] = 

2 bytes (because it's an integer array and size of integers is 2 bytes)

if a pointer p is pointing to an address of int value p++ will increment p value by 2 if it's pointing to an address of float value p++ will increment p value by 4 an 1 for char.

Syntax of decleration of pointer is;

type *name;

Example:/

int *c; the pointer named c is pointing to an address which has an int value.

operator used in pointer variables

1) get address of a location (&)

d= &ch // d can hold an address because it's a pointer.

```
int *c
float *d
float ch;
```

2) access to a location at the given address (*)

*d=18 means put 18 to the location that d is pointing to

printf("%f", *d); → print the value which is stored at the address that d is pointing to

<u>Example:</u>

```
int i;
int *J;
J=&i; //address of i is stored in pointer J
*J=17; //17 is assigned to the address that J is keeping (it means i=17)
```

& (address of)
* (at location)

Ex/
```
int *p
*p=5 // assign 5 where p is pointing to

p=&m
m=*p // assign to m what is in address of p
```

mathematical operations:

++ goes next location
-- goes previous location

we declare the type for pointer so it knows how many bytes should it go.

Arrays & Pointers:

Example: int A[10];
A[2] = 17

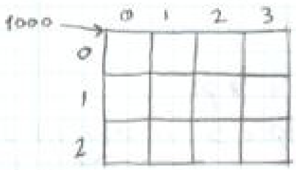
Here to find A[2] C automatically uses starting address of A and number of elements + size of each element. A name of array without [ ] is a pointer to address of the beginning of the array.

```
A[2] = (A+2)          for(i=0; i<10; i++)
A[i] = A+i               *(A+i) = 0;
A[4] = 5
*(A+4) = 5
```

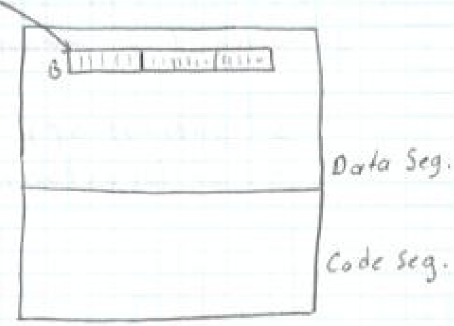
→ "A" shows A[0], +4 will increment the address value that A is pointing to 4 times (each 2 bytes because int values are 2 bytes) and by using * it assigns 5 to the address that is being pointed to at the moment and that address is address of A[4]

2D Arrays

int B[3][4]



assume address of here is 1000



B[0][0] at 1000
B[0][1] at 1002
⋮
B[0][3] at 1006
B[1][0] at 1008
⋮

so if B (B[0]) is at 1000 B[1][2] is at 1000 + 1 * 4 + 2

in row 1 ← ↳ number of elements in the first row

This method is called "Row Major method".

$$\boxed{B[i][j] = B + i * columns + j}$$

Arrays and Pointers

passing parameters to functions 1) call by value
2) call by referance

1) call by value.

```
void f(int k)                        main()
{     → create k                     {
        initialize it                  int i
   k=1                                 i=5
                                       f(i);
      → delete k                       printf("%d", i); → 5
}                                    }
```

2) call by reference

```
void swap (int *i, int *j)
{
  int temp
  temp = *i
  *i = *j;
  *j = temp;
}
```

→ since here is void if we don't use * swap won't be done.

```
main()
{
  int m,n;
  m=10;
  n=5;
  swap(&m, &n);
  printf("%d %d", m,n); → 5,10
```

int *A[10]; → 10 pointers pointing to 10 integers

Name of a function is also a pointer to that function

function_return_type (*pointer type) (list of parameters)

Example:

```
int (*p) (int i, int j)
int (*c) (char *i)
```

function name is also a pointer

in arrays of pointers

syntax:

```
type *arr-name[size]
Ex/ int *A[10];
```

```
struct t
{
 int i
 float j;
};
struct t *p;
struct t s;
p= &s
(*p).i=1;      → (p->i) is same as *P.i
(*p).j = 2,7;
```

Access to field of a struct variable using pointers

1) using * and . operators
2) using → operator

```
p → i = 2
pointer → field_name
p->i
(&s) ->j;
```

Example:

```
struct t
{
  int i
  float J
  char c[10];
};
 struct t A[10]
 struct t *b
 b= & A[2]
 b -> i = 4;
 strcpy ( b -> c, "init");
 b -> J = 2,7;
 b++;  //now b will point to next struct element in
 Array A

 b= & A[0]
 for (k=0; k<10; k++)
 {
  b -> i = k;
  strcpy (b -> c, "init");
  b -> J = 0;
  b++;
 }
```

variables:

variables can be declared dynamically or statically in static decloration array size is fixed and allocated during compiling.
in dynamic decloration memory allocation is done by malloc and it can be changed by realloc later. in dynamic the size of the array can be wanted from the user and the allocation can be done according to that number.

Example:/

```
char *c;
c =(char *) malloc(4);

struct t *p;
p = (struct t*) malloc(sizeof(struct t));
p ->i =4   // or(*p.i =4)
```

using dynamic allocation we can create an array without defining its size

```
int A[10]; → static and we define the size of array
```

<u>malloc</u>

```
int *B;
int c;
scanf("%d", &c);
B= (int *) malloc (c * sizeof(int)); // b is an array
B[0]=1;                                 of int with
B[2] = B[1] = 2;                        size of c
   :
free (B); // lets the pointer B free
```

<u>realloc</u>

we may allocate larger memory for the memory that we allocated with malloc, by using realloc

```
A= (int *)malloc ( sizeof(int) * 10)
. - -
- - -
```

if the memory is not enough we may allocate a larger area for A by

```
A=realloc(A, 15 * sizeof(int));
```

realloc

1) allocates a larger location
2) copies contents of previous location to new one
3) free previous location

Example:

```
int *A
A=(int *) malloc (sizeof (int) *10);
if (A==NULL)
{
  printf ("memory allocation error");
  return;
}
```

Example 2

```
c = (char *) 2400
*c = 'A';  →  c[0] = 'A' now
```

FILES

A group of data items stored in a peripheral devices like harddisc or floppy disc

- open or create a file
- read or write a file
- movement in a file (I/O operation from a file)

File types in C

① Text files
- ① starting data is done using characters
- ② End of line CR + LF
- ③ End of file ctrl + Z

②- Binary files
- ① Data is stored using binary form
- ② only one character for end of line without interpreting it when reading (like "\n")
- ③ No EOF

257 = 0000000100000001 (256 ↑, 1 ↑)

For example 257 is stored in binary form to binary file. like this.

if we want to use Files at first we have to define a pointer pointing to structure File and this is done by

→ f is a pointer to structure "File"

FILE *f;

Then we use this structure for all operations will be done on this file

opening a file:

f=fopen("c:\\dl\\Text.c", "r+")

r
w
w+
a
a+

✓ - Yes
X - No

mode	cursor_place	previous_contents	Reads	Writes
w	Begining of file	deleted	X	✓
r	Begining of file	not deleted	✓	X
a	End of file	not deleted	X	✓
w+	Begining of file	deleted	✓	✓
r+	Begining of file	not deleted	✓	✓
a+	End of file	not deleted	✓	✓

if we want to use this file as binary file we only use modes like;

wb, rb, ab, w+b, r+b, a+b

we only write b (binary) to the end of modes

I/o functions:

1) getc ex/ char ch;
ch= getc(f);
gets a char from file that is pointed by f and assigns it to ch

2) putc ex/ char ch;
putc(f, ch)
writes ch to the file

3- fscanf (file-pointer, format string, variables)

Ex: fscanf (f, "%c", &ch)

Has the same logic as scanf. The only difference is we denote the file pointer (f)

Ex2: fscanf (f, "%d %s %f", &i, s, &j)

4- fprintf (file pointer, format string, var);

Ex: fprintf (f, "%d", i); // used in text files

Again has the same logic as printf. The only difference is we denote the file pointer. (f)

5- fread (address of variable, size, repeat, file pointer)

it reads the content where the cursor is cursor is currently on from the file and assigns it to the variable that we have given the address of.

Ex: fread (&i, sizeof(int), 1, f)

Ex: fread (array, sizeof (struct s), 10, f)

I assume that size of array is 10 and I'm reading all items by repeading read 10 times

a struct array and since its an array I don't need to denote "&" sign because it's already a pointer to array[0].

6- fwrite (address of variable, size, repeat, file pointer)

Has same logic as fwrite in usage. But this time it writes the contents located at the address of variable that we gave to function, to the file (it writes it to the place where the cursor is at the moment)

Ex: fwrite (array, sizeof(struct s), 10, f)

writes the contents of array from 0 to 9 (because repeat is given as 10) to the file pointed by file pointer "f"

Changing the pos indicator:

we use fseek for changing the pos indicator

fseek(file-pointer, length, starting point)

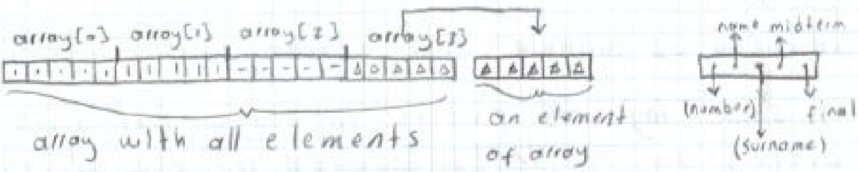
"f" is used in our notes — length will be gone after starting point

starting point:
- SEEK-SET → begin from the begining
- SEEK-END → begin from the end
- SEEK-CUR → begin from the current location of pos-indicator.

ex/ fseek(f, 2 * sizeof(struct s), SEEK-SET)


```
struct student
{
  int number
  char name[10];
  char surname[10]
  int midterm;
  int final;
} array[4];
```

↳ variable decleration



Now if changed one of the items of array[2] (this may be name, surname etc...) we have to send the pos-indicator to the begining of array[2] record in the file in order the replace all items of array[2] with the changed one which is still only in the struct array.

in order to send the pos indicator to the begining of the record of array[2] in the file, we write the command;

```
fseek(f, 2*sizeof(struct s), SEEK_SET)
```

Now it understands that the pos_indicator in the file which is pointed by f will go to starting point of the file and goes forward for 2 times size of (struct s) and it comes to the begining of array[2] because array[0] and array[1] are passed by going 2 times sizeof (struct s) then when we use

```
fwrite(array[2], sizeof(struct s), 1, f)
```

the contents in array[2] will be overwriten onto the array[2] located in the file.

closing the file:

fclose(file.pointer)   Ex:/ fclose(f);

An Example Program

Assume you have a file at C:\Text.c and its a binary file (stored in binary form). There are 10 struct records writen into the file before the structure of struct is

```
struct student
{
  char name[10];
  char surname[10];
  int score;
} array[10];
```

Now I want you to write a program that reads the items from the file and calculate the average of their scores. And print all students scores and the difference of the score from the average.

<u>answer:</u>

```
struct s Buf[10];
float ave = 0;
int i;
file *f;
if ((f=fopen("Text.dat", r+b)) == NULL)
{
 printf("error");
 return;
}
 fread(buf, sizeof(struct s), 10, f);
 for(i=0; i<10; i++)
 ave += buf[i].score
 ave /= 10;
 for (i=0 ; i<10; i++)
 printf ("%s has score %f which is different from
average by %f \n", buf[i].name, buf[i].score,
ave - buf[i].score);
 fclose(f);
}
```