'Hello world' - Brian Kernighan, - Canadian Computer Scientist (Bell Labs) - creators of C - Dennis Ritchie, Ken Thomson.

The only way to learn a new prog. language is to write programs with it: The first program is the same for all languages: display the two words.

'compile'
'load' - memory
'Run'
'Find o/p'

'Helloworld'

Software: Tells hardware what to do?

* Algorithm: (recipe) : sequence of instruction (code)

printf ("Hello world")
↓
No semicolon

3 <-- error: expected ;
↳ column
↳ line

\n → newline

'point' quotes'

printf ("Have fun with \"this\" course!");

Looping errors

Comment: //
multiple comment: /* */

* Pre processor directive
* main function
* variable declaration
* Executive statements
* return statements

) Basic

Variable names

* Don't use special chars, accented, No spaces!
* can use _ (underScore) - Snake case.

%.1f → (1.95) → 1.950000 (double)

double variable

Specify how many decimals

%.2f → 2 decimal places.

%.2f → 5/2 (error)

↓

Ans: int

↳ 'Required: float (double)'

%d → 2.0/5.0 (error)

↳ Req: integer - Have - float

---

int/int = int

int/double = double

double/int = double

double/double = int

---

166 % 20 → 6 dollars need) int

166 / 20 → 8 (20 dollar) bills.

Conversion

* (double) variable

```
int main()
{
 int a = 2;
 double b;
 a = (double) 12;
 printf ("%.2f", a); → warning (lf but i/p is int)
```

O/p : 0.000000

```
b = (double) 12;
printf ("%.2f", b); → 12.000000
printf ("%d", b); → Throws random value! (or) 0
```

---

decimal to integer

```
int one = (int) done;
int two = (int) dtwo;
```

Static condition

if (1), if (200), if (0)

Dynamic condition

if (a == b), if (a > b)

Double → Including decimal places must be accurate

double a = 5.0000000000000001 } Beyond precision level (equal)
b = 5.0000000000000000

'up to some digits: precision level'

---

char word [5 1]; %s → Format specifier,
(string)

↓

updatable

word [1] = 'a'

---

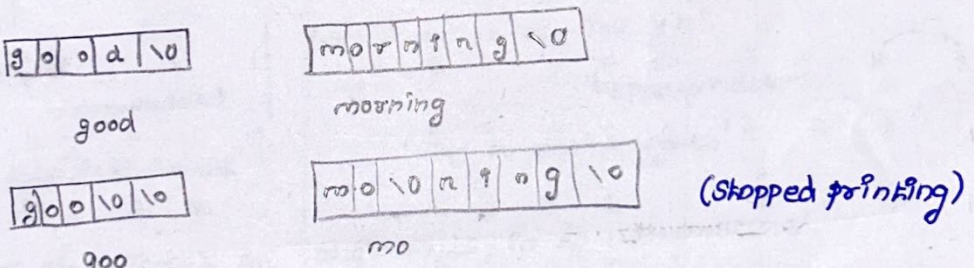
word1 [5] → Good
word2 [8] → morning

word1[3] = '\0'
word2 [2] = '\0'

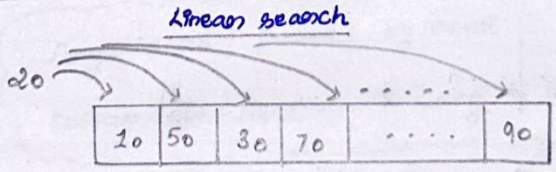
printf ("%s %s", word1, word2); → Goo mo

∴ \0 → end character. 'print up to \0'

Initial

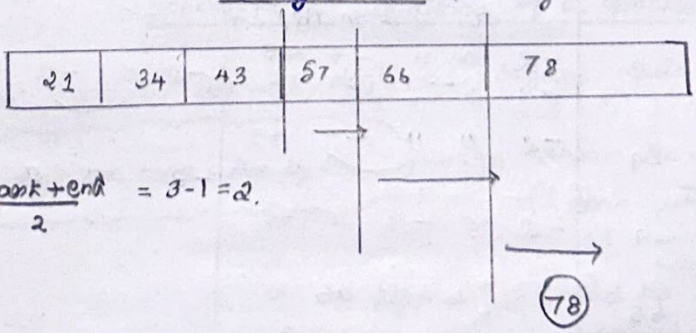


Linear search



Binary search (Sorting must)

78



Split (Find middle): $\frac{\text{start} + \text{end}}{2}$ = 3-1 = 2.

---

when to stop: element found / element not found

Bubble Sort

5 1 4 2 8 → (1 5 4 2 8)

(1 5 4 2 8) → (1 4 5 2 8)

(1 4 5 2 8) → (1 4 2 5 8)

(1 4 2 5 8) → (1 4 2 5 8)

(1 4 2 5 8) → (1, 2, 4, 5, 8)

(1, 2, 4, 5, 8) → (1, 2, 4, 5, 8)

(1, 2, 4, 5, 8) → (1, 2, 4, 5, 8)

(1 2 4 5 8) → (1 2 4 5 8)

(1 2 4 5 8) → (1 2 4 5 8)

(1 2 4 5 8) → (1 2 4 5 8)

(1 2 4 5 8) → (1 2 4 5 8)

---

5 1 4 2 8

1 5 4 2 8

1 4 5 2 8

1 4 2 5 8

1 4 2 5 8

---

1 4 2 5 8

1 4 2 5 8

1 2 4 5 8

1 2 4 5 8

1 2 4 5 8

---

1 2 4 5 8

1 2 4 5 8

1 2 4 5 8

1 2 4 5 8

1 2 4 5 8

---

on

-2 45 0 11 -9

-2 45 0 11 -9

-2 0 45 11 -9

-2 0 11 45 -9

-2 0 11 -9 45

-2 0 11 -9 45

-2 0 11 -9 45

-2 0 11 -9 45

-2 0 -9 11 45

'No need as we know we placed the largest at the end'

-2 0 -9 11 45

-2 0 -9 11 45

-2 -9 0 11 45

-2 -9 0 11 45

-9 -2 0 11 45

Steps: n-1 times

Step: 0	Step: 1	Step: 2	Step: 3
0, 1	0, 1	0, 1	0, 1.
1, 2	1, 2	1, 2	
2, 3	2, 3		
3, 4			
4-0	4-1=3	4-2=2	

n=5

< n-1

< 4 = 3.

Step: 1

0, 4 (i=0; i < n-1-i; i++)

↓

i < 4-0

i < 4

0, 1
1, 2
2, 3
3, 4

4 times

---

function – Argument & return. (Sum of two numbers)

sum(a, b)
↳ Arguments

'No type definitions'

int sum(int x, int y)
↳ parameters.

'Need to define types'

'we are giving (copy) the arguments in to parameters'.

Arguments: passed by value

parameter: Receive a copy of argument's value.

Prototype & definition of a function

'Function must be defined / declared' before main function

'compiler goes through the code - line by line' → see: where is definition - (I don't saw it)

↓

Need to acknowledge I have defined that!

Acknowledge function: int sum(int, int); → function prototype

function definition:

```
int sum (int a, int b)
{
  return (a+b);
}.
```

Decompose

① Prototype

② Define function!

```
int Sum(int a, int b)
{
 return 'a'  ——————→ 97 (∵ Integer return)
}
```

'Converted to ASCII then returned'

---

* 1 meter = 3.28083 feet
* 1 gram = 0.0022205 pounds
* T(Fah) = 32 + 1.8(T °C)

```
4
10 m
1245.243 g
37.2 c
23 g
```

o/p

```
32.808000 ft.
2.745761 LPS
98.960000 f
0.050715 lbs
```

Conversion_mgc

---

a=1

$b = \underset{3}{\overset{2}{++a}} + \underset{3}{\overset{3}{++a}}$

1)

$= 3 + 3 = 6$

2 → Assigned

2)

```
b = ++a + a++ ;
    3 ←——— (3) → Now
```

$= 3 + 2 = 5$

a=1

3) $b = (\underset{3}{\overset{2}{++a}} + \underset{(3)}{\overset{2}{a++}}) + ++a$

Compiler: Solve (2 by by)

$= (3 + 2) + ++a$

$= 5 + ++a$

$= 5 + 4 = 9$

4) $b = (a++ + ++a) + a++ ;$

$= (1 + 3) + a++$

$= 4 + 3$ Now (a=4)

$= 7$

∴ printf("%d, %d, %d", ++a, a, a++)

Stack formation: LIFO ← (Left to right)

∴ (++a, a, a++)
(2) (2) 1
3 2

↳ During printing.

---

r value: (right value)

Left value (l value): An object that has an identifiable location in memory. (having an address).

* capability to hold data.

∴ (a+b) → only returns doesn't hold (R value)

* Don't use ++ before/after (R value).

Fibonacci number

0, 1, 1, 2, 3, 5, 8, 11, 20, 34, 55, ...

$$\begin{cases} 0, & n=0 \\ 1, & n=1 \\ (n-1)+(n-2), & \text{elsewhere} \end{cases} \quad n \geq 0$$

fibo(3) = fibo(2) + fibo(1)

fibo(4) = fibo(3) + fibo(2)
= (fibo(2) + fibo(1)) + (fibo(1))

Sum of digits

47253 = 4+7+2+5+3 → until 0
= 21

(47253 % 10) + fun(47253/10)
(4725 % 10) + fun(4725/10)
(472 % 10) + fun(472/10)
(47 % 10) + fun(47/10)
(4 % 10) + fun(4/10)
0 → stop

3 + 5 + 2 + 7 + 4 + 0 (stop)
21 18 13 11

Return.

Von Neumann Architecture

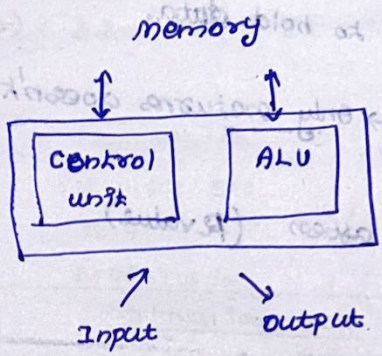
"Inside of a Computer"

* Still - modern Computers are using this architecture.

* John William mauchly & John Eckert used this already in their work

→ BIG ENIAC Computer.

Abstract model: → Splits 4 distinct parts.



eg: Read the 1st value from memory
2nd value from memory
Add 2 values
Store it (Back at the memory)
Return it.

---

memory → Program
→ Data

memory → RAM
→ ROM

Now: (modern): Computer's memory Communicate directly with I/P & O/P

"No need of CPU"

---

memory representation

* OS (role): manage memory & addressing.

How Computer knows where it saved.

memory: hardware (involved in storing info for use)

↙ RAM — (Temporary) easy, quick to access. → Execute programs

↘ ROM — Non volatile, lasting. (cheap)

Execution of programs → uses RAM (values are stored - read/written from RAM)

RAM - sequence of binary memory cells - populated with 0's or 1's.

each cell : bit

Word: fundamental unit of data (group of bits)

can be moved b/w RAM & Computer processor.

size of word: bit / byte

1 byte = 8 bits.

---

* 'Modern computers uses 64 bits for a word' (to form)

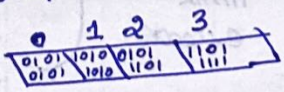
why? (32, 64, 16, ..) not higher?

Result of the hardware architecture evolved over time

* why groups: This allows to address each word. (memory address)

* Computer address: whole number that describes the location of the word in the Computer memory.

eg: 8 bits →



each address: for 8 bits. (word)

---

* In c programming: we can get the memory addresses during the execution of a program.

'obtain the address of the variable'

Use of memory address allows low-level access to the Computer - by allowing direct address to the Computer's memory.

↓

∴ Allows speed & optimization of memory.

'extremely powerful: application developers'

* 'low level access': Through syntax - access memory directly using pointers

Size of datatypes

» sizeof

char → c = \0 ——————→ 1 byte

int → i = 0 ——————→ 4 bytes

double → d = 0 ——————→ 8 bytes

char list[3] = {\0, \0, \0} ——→ 3 bytes (3x1)

double list[3] = {0.0, 0.0, 0.0} ——→ 24 bytes. (3x8)

int list[3] ——→ 12 bytes (3x4)

'may vary with computers: but almost same'

Largest Integer

(Finite amount of space)

$2147483647 = 2^{31} - 1$

'Large numbers - be careful'

```
int num = 2147483645;
for (i=0; i<8; i++){
    printf("%d \n", num);
    num = num+1;
}
```

2147483645

2147483646

2147483647

-2147483648

-2147483647

-2147483646

-2147483645

-2147483644

as limit exceeds.

Rounding numbers & circumvent errors

```
double num=0.25
```

%.40lf

↓

0.2500000.....00

(40 dig)

num = 0.3

%.5lf → fine 0.30000

%.10lf → fine 0.3000000000

%.20lf → 0.299.... 8890 (why?)

%.40lf → 0.299.... 88977...484--37

* Stored with some accuracy (not accurate)

Float

* 0.30000001192092 8... 000000 (Rounded with something control).

* 0.25.....000

---

why?

In computer - Everything is binary → 0's & 1's

∴ 2^3 power - correctly done (Accurate)

0.3 → Not powers of 2 (no accuracy decimal - but not random).

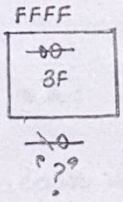
Power of 2 → Accurate

Others - make it power of 2 (no decimal - not 100% accurate).

---

char c = '?';

Start: 65520 (memory address)



→ replaced

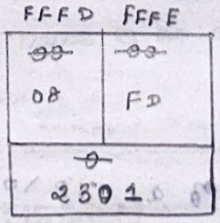
int iInt[3] = {1,2,3}

↓

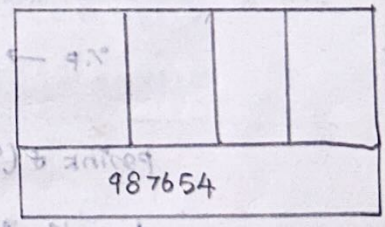
12 boxes

(4 each)

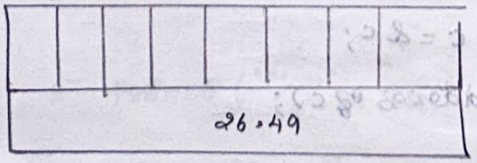
short s = 2301;



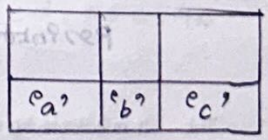
int i = 987654



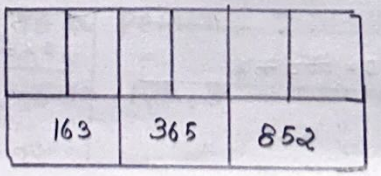
double d = 26.49 (8 bytes)



char c[5] = {'a', 'b', 'c'};



short sshort[3] = {163, 365, 852}



double dDouble[2] =

{86.263, 265.274}

↓

16 boxes

(8 each)

```
#include <stdio.h>
int main() {
    int a = 10, b = 11;
    {
        int c = 12;
        int d = 13;
        int e = a + c;
        c = b + d;
        printf("c: %d \n", d);
        printf("e: %d \n", e);
    }
int f = 14;
int g = 15;
printf("f: %d \n", f);
printf("g: %d", g);
}
```

→ Variables inside the block not accessible outside this block.
But a, b → accessible here but not f & g. (since not yet declared)

'variables inside a function: not accessible by other functions outside'

* 'Pass the values' — for other function to access

Pointers.

a = 42; d = 58.394;

addresses: int * address of A = & a;

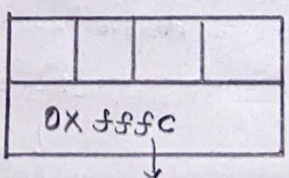
↓

pointer to an integer. (At this address there is an integer)

* %p → pointer.

```
printf("address of a: %p \n", address of A);
double * address of D = &d;
printf("%p \n", address of D);
char * address of c = &c;
printf("%p \n" address of c);
```

int * address of a = & a;



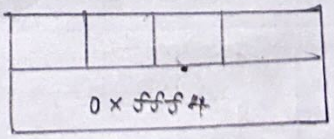
↓

Hexadecimal (Address)

O/P

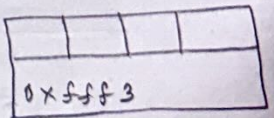
fffc

double * address of d = & d



O/p : ffff4.

char * address of c = &c



O/p : fff3

32 bit : 4 bytes for address (pointers)

64 bit : 8 bytes for address (pointers)

Pointers : Points to an address which holds an Integer value.

↓

point to another variable (address of A) points to another variable.

Dereferencing

* get the value of the variable - over pointer is pointing

(* address : Access the variable.)

Manipulate values

a=12, b = 15.43

*ptr = 1234; ⟶ a = 1234

*dptr = *dptr + 1; ⟶ b = 15.43

"Through dereferencing we are changing the values".

Pointer variable

int *a; ⟶ *a (dereferencing)

a → access address

*a → value of (associated) with that address.

---

* Array is a pointer - name of an array is the address of the first element in this array.

arr [3] = {15, 16, 17}

* Printf ("%d", arr) ⟶ -1604955172 (address)

* Printf ("%p", arr) ⟶ 0x7ffcd080cb4c

* printf ("%d", *arr) ⟶ 1

"Array is a pointer - which holds the address of the array's values 1, 2, 3" - or atleast address of the 1st element.

* printf ("%d", *arr);
↳ value of address {1, 2, 3}

Ans: 1

* printf ("%d", arr);

Ans: 6422292 (memory address - int)

* printf ("%p", arr);

Ans: 006FFF14 (Hexadecimal)

* printf ("%p", *arr);
↳ value = 1 (converting to hexadecimal)

Ans: 00000001
(8 digits) - 64 bits

---

arr[3] = {12, 13, 14}

∴ int *ptr = arr;

*ptr = 2; → first memory changed!

arr[3] = {2, 13, 14}

(ptr + 1) → address of second element of array.

*(ptr + 1) = 3;

*(ptr + 2) = 4;

arr[3] = {2, 3, 4}

Reverse array

i/p → 1 2 3 | 4 5 6

o/p → 6 5 4 3 2 1

$\frac{6}{2} = 3 - 1 = 2$

$\frac{1}{2} = 0$

$\frac{6-1}{2} = 2$

1 2 3 | 4 5

$\frac{5-1}{2} = 2$

1 2 | 3 4

$\frac{4-1}{2} = \frac{3}{2} = 1$

Using pointers to functions

Arrays & pointers

* int array[] = {6, 2, -4, 8, 5, 1};
  ↓
  ffe8, ..., fffc

* printf ("%d", array); ↴

Some compilers

error: format specifies int, but argument int *

* 'Array itself a pointer to an integer'

* The variable arr (array) holds itself the address eg where array starts.

printf ("%p", array); → ffe8 (address eg 0th index)

∴ array equals to &array[0]

int * ptr = array

∴ array – itself pointer

Correct

int * ptr = arr

∵ ptr – pointer variable, arr holds address eg arr[0];

Here * doesn't mean pointer dereferencing.

∴ ptr (pointer variable) is in the address

(or)

int *ptr = &arr[0]

∴ ptr → holds address.

Wrong

int * ptr = arr[0]

int * int

Errors.

Array & pointers

ptr + 1 ⟶ &arr[1]

∴ we already assigned ptr = arr. (address).

*ptr = 10;
*(ptr+1) = 5;
*(ptr+2) = -1;
) {10, 5, -1, 8, 5, 1};

*array = 3

∴ array ⟶ address of array[0]

∴ *array = 3 (dereferencing)!

---

*(array + 1) = 10;
*(array+2) = 99;
) ⟶ {3, 10, 99, 8, 5, 1}

ptr++; ⟶ {3, 7, 99, 8, 5, 1}

*ptr = 7; ⟶ ptr = &arr[1] (now)

---

ptr += 3; ⟶ arr[3] (0+3=3)

*ptr = 8; ⟶ {3, 7, 99, 8, 5, 1}

'Directly access memory address'

'Pointer arithmetic'

&arr[0] ⟶ arr
&arr[1] ⟶ arr + 1
&arr[2] ⟶ arr + 2
&arr[3] ⟶ arr + 3

∴ scanf("%d", arr+i);

pointer_array_get_array_pointers.c

{1, 2, 3, 4, 5}

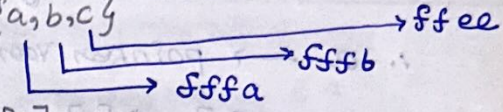
arr[0] = *arr
arr[1] = *(arr + 1)
arr[4] = *(arr + 4)

short a[3] = {234, 655, 843};

short b[2] = {12, 62};

short c[4] = {3456, 3467, 23, 276}

short *arrays[3] = {a, b, c}



∴ a → &a[0], b → &b[0], c → &c[0]

∴ array of pointer variables

*arrays[0] ⟶ *(&a[0]) ⇒ 234

∴ *arrays[0] = 5; ⟶ {5, 655, 843}

*(arrays[0] + 1) = 2; ⟶ {5, 2, 843}.

∴ (array[0] + 1 = ffffa) ⟶ next address memory

(or)

*arrays[0][1] ⟶ same as *(arrays[0] + 1)

c = {6, 7, 8} ⟶ Printf(c) ⟶ 6

∴ First value alone saved, c → not an array.

a[3] = {1, 2, 3}, b[] = {4, 5, 6}, c[] = {7, 8, 9}

*d = {a, b, c}

*d[0] = value of (&a)

= value of (&a[0])

= 1

*(d[0] + 1) = *(&a[1])

= 2

changing

*d[0] = 1	*d[1] = -4	d[2][0] = -7
*(d[0]+1) = -2	*(d[1]+1) = -5	d[2][1] = -8
*(d[0]+2) = -3	*(d[1]+2) = -6	d[2][2] = -9

∴ d[2] → &c[0]

d[2][2] → c[2] (value)

↓ Dereferencing.

∴ a → pointer variable

a[0], a[1], a[2] → integer values.

Likewise

d[2] is the address (pointer variable) of c

d[2][2] = c[2], value of that index of c

```
int a[] = {1,2,3,4}
printf("%d", a);      → 642342 (address)
printf("%p", a);      → 0061FF10
printf("%d", a[0]);   → 1.
printf("%d", *a);     → 1.
```

---

1) a[][] = {"Hari", "Ha"}

2) a[] = {a, b, c}

Different.

arrays

1) Array of (multi) chars.

2) Array of addresses (∴ a, b, c pointers)

"Array itself a pointer"

Array of Arrays

```
a[3] = {1245, 1924, 2345};
b[2] = {24, 256};
short *k[2] = {a, b};
```

↓

∴ It points to the address of a, b, c

∴ pointer array

Passing an array.

printf ("%d", *c) ⟶ 6422292 ⇒ &a

∴ dereferencing

printf ("%d", c) ⟶ 6422268 [address of first element of B]

printf ("%d", *(c+1)); ⟶ 6422280 ⇒ &b

∴ we need to un box twice

c ⟶ 6422288

*c ⟶ 6422292 (&a)

*(c+1) ⟶ 6422280 (&b)

*(c[0]) ⟶ 1	(*a = a[0])
*(c[0]+1) ⟶ 2	(*(a+1) = a[1])
*(c[0]+2) ⟶ 3	(*(a+2) = a[2])
*(c[1]) ⟶ 4	(*(b)) = b[0]
*(c[1]+1) ⟶ 5	(*(b+1)) = b[1]
*(c[1]+2) ⟶ 6	(*(b+2)) = b[2]

without index

* dereference twice.

∴ c has the address of its first element.

*c ⟶ 6422292

**c ⟶ 1

1. dereference array c ⟶ a (pointer)
2. dereference a ⟶ 1.

*((*c) + 1) ⟶ deref c, then deref &c[1]

*((*c) + 2) ⟶ 3

*((*c) + 1) ⟶ 2

(((c+1)) ⟶ 4

(((c+1)) + 1) ⟶ 5

(((c+1)) + 2) ⟶ 6

c is a pointer (address)

1. dereferencing c gives its element

$*c \rightarrow$ 0th element

$*(c+1) \rightarrow$ 1st element

∴ c's element itself an array (array : pointer)

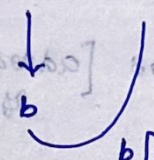
∴ $**c \rightarrow \&a$ (dereference)

$*((*(c)+1)) \rightarrow \&a[0]$

$*((*(c))+2) \rightarrow \&a[2]$

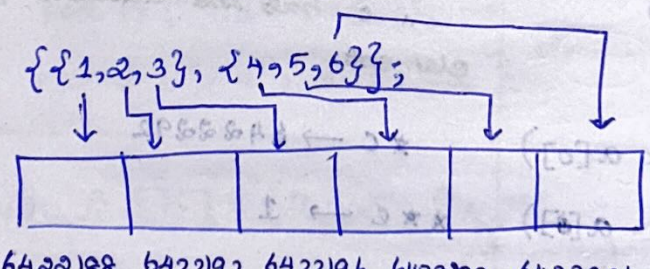
$*((*c)+2) \rightarrow \&a[3]$

$*((*(c+1))+2) \rightarrow \&b[3]$



---

* getting an multidimensional array.
* pointing a multidimensional array
* setting all the elements to zero.



'sequence'

int arr [a] [b]

~~pointer~~ array (arr, a, b)

↳ pointer.

∴ scanf ("%d", arr+0);

arr += 1;

① arr [0] [0]

② arr [0] [1]

③ arr [1] [0]

④ arr [1] [1]

∴ C → Intelligent enough.

∴ arr + = 1 → It assigns the int number in the next four bytes of memory.

'By address we are getting multidimensional array'

Return array

get_multidimensional_array.c

return_array.c

```
#include <stdio.h>
int *get_array()
{
  int i;
  static int *arr[6];
  for (i=0; i<6; i++)
  scanf ("%d", arr + i);
  return arr;
}
int main()
{
int i;
int *a;
a = get_array();
for (i=0; i<6; i++)
printf ("%d", *(a+i));
return 0;
}
```

→ Local variable can't be returned so the keyword static.

Return (array) → pointer, copy to pointer in main then print

Print arrays by using pointers → Best.

* 'Array of pointers' → Each word - stored in a null terminated array of characters.

* Then array that holds the pointers to all of these. (strings)

'Array of Arrays'

* word - null terminated array of characters.

```
char a[4];
char b[5];
char c[9];
char *words[3] = {a, b, c}
```

∴ a, b, c → arrays (pointers).

*words[3] = {0xffffc, 0xffffb, 0xffffed}
a b c

Access the a,b,c strings

%s → words[0] They
%s → words[1] great
%s → words[2] outdoors.

---

* Sort characters
* get array string
* print array string.

} See hub.

* Reverse string

Dynamic memory allocation

(At run time)

* Happen in new memory location – (Heap)
* Sofar – automatically saved in – Stack (come & go) – persist only during their own block / function time (Function-done – they too)

Dynamically allocated

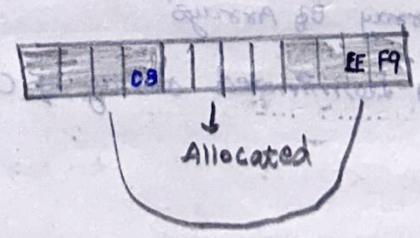
* persist – explicitly managed by user from heap
* malloc – memory allocation

↓

Library #include <stdlib.h>

---

* malloc(4); → 4 bytes



Something happens (depends architecture)

* Left → 4 more bytes (number 8 stored in the left block) last one.

* 8 → reserving space (4 → malloc + 4 - extra at right)

* Right → EEF9 → how much remaining space in heap (Hexadecimal)

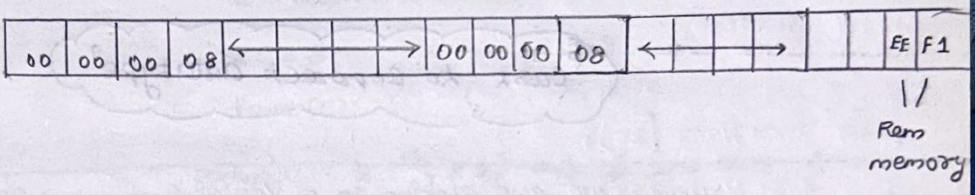
EEF9 → 61177.

↓

why? : subsequent calls eg malloc knows - how much space is left.

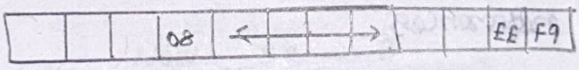
Another malloc!

```
malloc(4);
malloc(4);
```



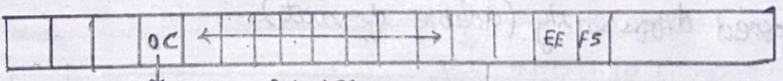
In that architecture course: malloc allocates in multiples eg 4.

If `malloc(1);`



(Still 4 bits)

malloc(5) → 8 bytes



12

8 + (4) → space remaining.

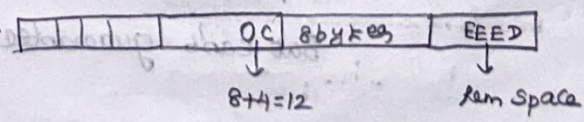
(Nearest multiple eg 4) = 8

Size of function to store a variable (Just right amount of space)

`malloc(sizeof(int));` → 4 bytes

`malloc(sizeof(double));` → 8 bytes

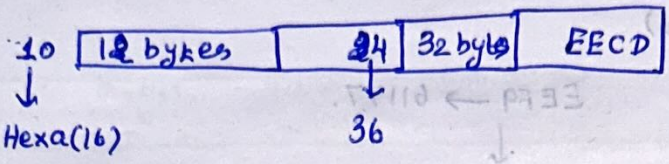
↓



more variables

malloc (3*sizeof(int));

malloc (4*sizeof(double));



Pointers - storage where space is reserved.

int *intptr;

double *doubleptr;

malloc (size of (int)); ⟶ intptr = malloc (size of (int));

doubleptr = malloc (size of (double));

"Cast to correct datatype"

Now → we are storing in a variable ⟶ stored in stack

"Not heap"

"heap - has values"

Stack - has references.

we can dereference - gets values

double - stored differently (double format).

Release memory!

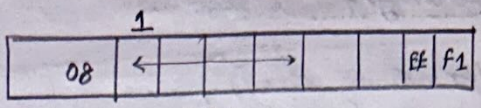
"we need to delete" → won't destroyed automatically!

free (a);

"can I still access the freed location"?

yes!

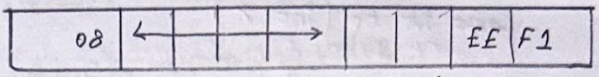
But can't guarantee - value.




```
int *a;
a = malloc (sizeof(int));
*a = 1;
free(a);
printf("%d", *a); → 1 (Even after free - we can
```

still access - why? - ∴ a has still has the reference to that address'

Twist!



→ Since a freed - ready to occupy

```
int *c;
c = malloc (sizeof (int));
*c = 3;
free(c);
```

→ 'same memory of a'

printf("%d", *a) → 3 → Not 1

∴ a's memory location - freed - now allocated to c.

The value of c is changed to '3'.

Now *a → 3

'No longer guarantee the value of a'

↓

Not reliable (eroradicate) behaviours

Above program 'Architecture dependant'

'Memory management'.

'dilemma_of_malloc_free.c'

Array

```
array = malloc (5*sizeof (int));
```

↓

whose stacks!

101 (say!)

101 to 105

```
array[0] = 1
array[1] = 2
```

(or) *array = 1

*(array+1) = 2.

free (array);

24 bytes (5×4 + 4 (remaining size))

−1 = FFFF FF (Hexadec rep - 2's complement)

good manners:

use (int *) malloc (sum * sizeof (int));

↓

∴ malloc returns generic pointer!

↓

cast it to (int *)

Structures

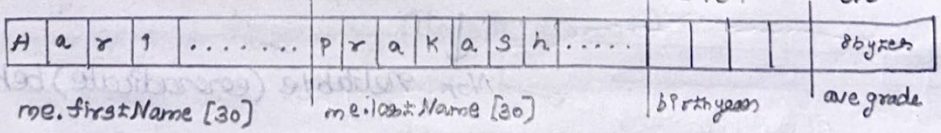
```
struct student
{
  char firstName [30];
  char lastName [30];
  int birthyear;
  double aveGrade;
};
```

→ struct student me = {"Hani", "Prakash", 2008, 9.4};

Access

me.firstName

me.lastName



"contagious memory location"

double → %lf

modify

```
struct student learnmem;
printf ("Name: ");
scanf ("%s", learnmem.firstName);
```

Pass structures to functions

```
void printStudent (struct student stud) {
    :
}
```

'By value'

'Nothing changed inside the function won't reflect in main'

Pass by reference

```
readStudent (&me);   → passing pointer
printStudent (me);   → passing value.
```

```
void readStudent (struct Student *studptr) {
scan ("%s", (*studptr).firstName);
```

↳ 'Dereference go to that variable'

∴ (%s, name) → we give variables directly for strings

Student.firstName → not &Student.firstName

'Dereference' → .

%02d

↳ 2 digits – 0 before

1 → 01

Access using arrow operators

```
scanf ("%d", &(*studptr).birthYear);
```

First dereference – then address → else error.

(Store at that variables address)

order of precedence → (. first)

---

* → pointer to a structure doesn't have any components.

only structure has.

↓ → ↓

Pointer Component of the Structure

'De referencing happens automatically

---

* → has higher order of Precedence than &
* No need for parenthesis.

---

eg: StudPtr → firstName

↓ ↓

Pointer (me.firstName)

Get size of a structure in memory!

* Atleast added up together - of all data sizes.
* Some compilers - padding - adding spaces in between elements of the structure. (So may be more - memory reading efficient)

'Never be smaller - for sure'

Sizeof (struct Student) ⟶ 22

(or)

Sizeof (me.firstName); ⟶	5
;	5
;	4
Sizeof (me.aveGrade);	8
	22

---

we can use %d too!

Correct way: to print size : %zu

z → length modifier

u → unsigned.

"%zu"

Codeblocks ⟶ 24

(zero padding!)

space!

return Struct

struct Student function (struct Student Stu)

{

}

Structures from user i/p

x, y → coordinates

Pass by value: copy (Nothing will be updated)

Pass by reference: Direct address

&ptr → x = &(*ptr).x

Array of Structures

3 points – No need to declare 3 times

↳ Time & Space waste.
(Lot of code)

&stud → address of the struct stud.

Declare

Struct Point pentagon[5]; → Declare

Assign

pentagon[4].y = 4.1;

(pentagon+4) → y = 4.1;

↓

Address – need to dereference

Allocate memory for structures

User - determine number of points

Struct Point *polygon (Pointer ∴ we don't know no. of points)

(Struct Point *) malloc (sizeof (struct point) * num);
↳ no. of points

∴ polygon = (Struct Point *) malloc (num * sizeof (struct Point));

Linked list

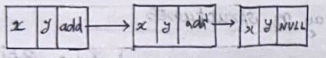
eg:

'Not contiguous' - Anywhere in the memory (next element)

* ∴ The 1st one has next element's address.

∴ next element → Struct

'type eg pointer' → Struct point *next;



* move to next element → via previous element (stored address)

```
Struct Point {
    int x;
    int y;
    Struct Point *next;
};

Struct Point Pt1 = {1,2, NULL};
Struct Point Pt2 = {3,4, NULL};
Struct Point Pt3 = {5,6, NULL};
Struct Point *Skank, *ptr;
Skank = &Pt1;
Pt1.next = &Pt2;
Pt2.next = &Pt3;
```

Malloc → Contiguous

We don't need that

* Skank → & Pt1
* follow (next) → go to Pt2, Pt3

```
Skank = &Pt1.
Pt1.add = &Pt2
Pt2.add = &Pt3
```

```
ptr = Skank
```

Both pointers.

'Just print'

```
while (ptr != NULL)
{
    printf ("%d, %d\n", ptr->x, ptr->y);
                        ↓
                      pointer
}
```

Enhance (prev. program)

Linux

WebLinux

History of Linux & Command line:

window, macos, Ios, Unix, Linux etc.

* Linux - open source - Linux kernel (created: Linus Torvalds) - developed by thousand of programmers

OS - Hardware & Apps (Intermediate)

Provides services to apps

Services:

1) File management (layout) - Harddrive - manage local free structure
2) Memory management
3) Management of I/P & O/P S
4) Manage of running apps

Operating System

1940's (mid) - first computers - Vaccum tubes - evacuated glass Containers - Control - Electric Current → (AS on/off Switches) - Large

* Programming - done manually - by moving around tubes
* Limited → I/P & O/P
* Programmer - also in charge of the Operating the machine via direct interaction → 'Heavy lifting'!

Designer - builder too!

Programmer as well as operator!

'Invention of Transistor' → Beginning of O.S.