* strdup("Adam"); Stores Adam in String Space – returns pointer

get words: char array[][];

scanf("%s", array[i]); → OK

But

char *array[]

I need to give pointer - the address of String - not String itself

char *temp, *array[n];

Idea loop:

scanf("%s", temp);

array[i] = strdup(temp);

---

* void ** → pointing void * (pointer)!

* void * → no data type info!

Programming Paradigms

* C, - Procedural - Concurrent programming - two function seemingly run - simultaneously (Networking).
* MIPS - Assembly language
* ATM machines - 2 people accessing some ATM - only one able to do!

'Concurrent programming' - same source (only one at a time)

Lisp, Scheme - representative of functional programming paradigm.

Scheme & Function languages:

* 1 Seat - can't allocate to many people - concurrent prog.
* Rely on the return value of a function to move forward.
* 'No side effects' - Pass by reference : org. changed!
* Scheme : no side effects : make partial result + ... + → Final.

Small results → Largeer partial → Final. (o/p)

Scheme - easy, pearl - modern obj lang of pearl: python! (web prog)

Lecture-2

C/C++

* Bool
* Char (256 chars)
* Short
* Int
* long
* float
* double

Scalar

Arbitrarily precise num

1byte | 1byte | 2bytes | 4bytes | 8bytes | 4bytes | 8bytes.

---

binary digit → 1 bit → 0 / 1 (Bool) — True / False

$2^8 = 256$

'A' = 65

8bits: 1byte:

64 + 1

$2^6 + 2^0$

0 1 0 0 0 0 0 1

2^7 2^6 2^0 = 65.

Shorts

0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 1

2^{16} possibilities

2^9 $2^2 + 2^1 + 2^0 = 7$

512 + 7 = 519

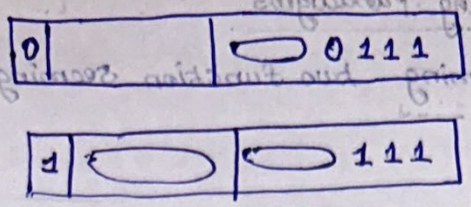
0 ——— ———

$2^{15} + 2^{14} + \ldots\ldots + 2^0$

$2^{15} - 1$

why not $2^{16} - 1$

'For sign bit'

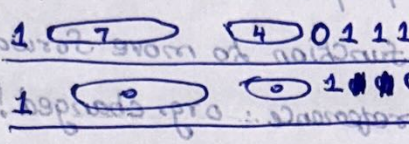


Why?: Addition, Subtraction - need to be followed.

0 1 1 1
0 0 0 1
1 0 0 0

7+1 = 8

For simplicity: I need this format for (-)ve numbers



Engineer: 7 + (-7) → 16 zeros.

0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0
Invert

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

1

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Overflow? Ignore

my Engineering:

Invert all, add 1 → representation of (-)ve num

('Same addition rules')

Now: All negative numbers (16 bits) have 16th bit 1

while +ve numbers up to $2^{15}-1$ have zero

Allow $2^{15}+\dots$ → collision [-ve / +ve]

So Boolean: $2^{15}-1$

char ch = 'A';

short s = ch; → No cast [implicit]

(65) - ASCII

-2^{15} to 2^{15}

short to char (loss)

S | 0 | 1 000011 — Post extra room

ch | 01000011

Short $S = 2^{10} + 2^3 + 2^0$;

Int $I = S$;

| 100 | 00001001 |

'more space' - copy

(no problem)

Int → 2^{-31} to 2^{31}

Int $I = 2^{23} + 2^{21} + 2^{14} + 7$;

| Some 1 | Some 1 | — | — |

Ignored!

Short $S = -1$;

Int $I = S$;

(-)ve ← | 11... | 1.....1 |

| 11... | 11... | 11... | 11... |

sign bit (1) - so replicate although

'sign expand'

'symmetric' : -2^{31} to 2^{31}

Floats - 4 bytes.

* Integer with decimal points

$\pm -2^9, \ldots\ldots, 2^0, 2^{-1}$

Sacrifice : 2^{30}.

Assume: $2^{-1}, 2^{-2}$ (sacrifice: $2^{31}, 2^{30}$)

| 10100100 | 0 | 0 | 0 | 0000 1111 |

2^{-1}

7.5

(or more)

| 10100100 | 0 | 2^0 2^{-1} 2^{-2} ..... 2^{-9} |

$\pi = 3.14\ldots\ldots$

'use 9 bits for decimal points'

only precision: Not possible with low memory!

32-bit - IEEE standard

| 1 bit | 8 bits | 23 bits |

sign | magnitude (exp) | xxxx

$(-1)^s$ 0 | $1.xxxxxx\ldots * 2^{exp-127}$

$2^8 = 2^{56}$

7.0×2^0

3.5×2^1

1.75×2^2

Int to float & viceversa

★ Protocol: float: take: Convert: we get original

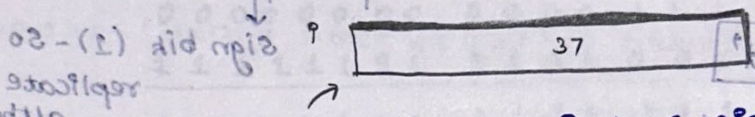
```
int i=5;
float f=i;
cout << f << endl;

int i=37;
float f=*(float *)&i;
```

$5.0 \rightarrow 1.25 \times 2^2$

exp = 2, .xx = .25

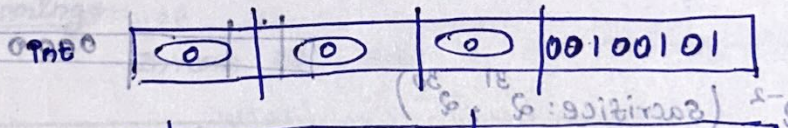
Returns float pointer – dereference!

'Evaluates location of i'



I want i to consider as float pointer
(pointer to consider as pointing to float).

evaluate as float (but actually int). [Ridiculously small number]



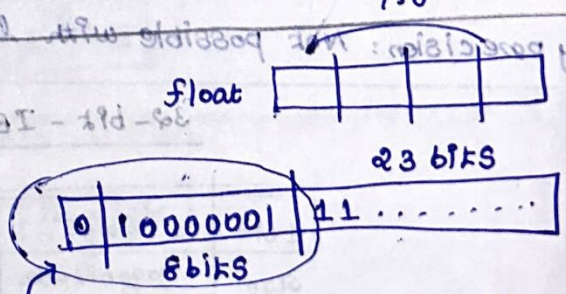
00100101 $\times 2^{-23}$

0. _____

extremely small number.

float f = 7.0;

short s = *(short *)&f;



1.11×2^2 → exp − 127 = 2

exp = 129

float

short: see two bytes

0 1000000 1110

'Large value'

For a moment consider as short – 2 bytes – replicate!

Lecture-3

double – 64 bit rep

1	11 bits	52 bits

double d = 3.1416;

char ch = *(char *) &d; ⟶ 1st byte (replicate)

↳ deal with address (don't look at what's d)

'Some negative values' ∴ Conversion: 3

short s = 45;

double d = *(double *) &s;

4	5

'Takes 8 byte'

what's next 6 bytes: Anything (residue)

6 bytes

4	5	.

Memory crash! ⟵ Then double → converts 64 bit rep → normal

Big endian format: , **little endian**

4	5	...

⟶ double decode ⟶ —.— format.

Big endian

00...	00000 1

LSB Stores MSB

(1) Big endian

Linux – little endian

0 ...	0 ... 0

2^0 2^7 2^8 2^{15}

(256) little endian

'Reverse order'

Points LSB: Consider Big endian

struct fraction {

int num; ⟶ 8 bytes / more [To indicate how much more memory]

int dnum;

};

'Say: Stacked'

7	pi.dnum
22	pi.num

Base address.

fraction pi;

pi. num = 22;

pi. dnum = 7;

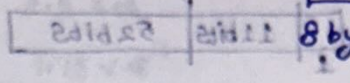
&(pi.dnum) ⟶ (int *)

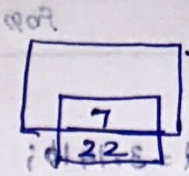
(fraction *) &(pi.dnum)

'Stops thinking as int *, fraction *'

Assume: (fraction *) – 8 bytes.

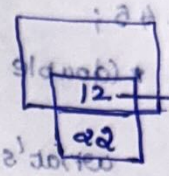
((fraction *) & (pf.dnum))





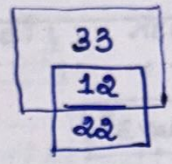
((fraction *) & (pf.dnum)) → num = 126. [changing num]

* Travel to that struct (address) – dereference → struct element
* struct element – access dnum



(fraction *) & (pf.dum)) → dnum = 33;

changing num
Changes dnum



① Forgets who owns that space – hypnotised (Assumes) fraction base address is dnum (usual)

② change num (base) : now (base : dnum) – Changes dnum

③ Change dnum (4 bytes above) : changes some other bytes above dnum – not owned by fraction – struct!

---

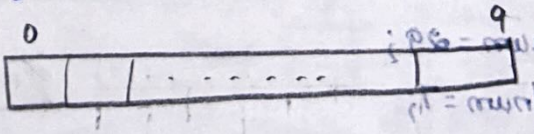
int array [10];

((fraction *) & (pf. dnum)) [0]. dnum;

* 1st index means address eg num (default)
* dereference &num address to get num value.

---

int array [10]; [40 bytes memory – each block : 4 bytes]



array [0] = 44;

array = &array [0]

array [10] = 1; → No bound checking in C, C++ (on raw arrays)!

array[25] = 25;
array[-4] = 77;] memory ka side effect

* array → anonymously base address (0th index) of array.
* array[10] → must be in accordance to documentation -

Consistent. 'Good GDB' - tells error! - crash - index exceeded

* array + K = array[K]
= &array[K]] pointer arithmetic (boundaries)

Problem: memory: left & right → of array → owned by others.

(Base address: int)

*array = array[0], *(array + K) = array[K]

→ *(array - 4) = 77; int (base) — 16 bytes ago!

int arr[5];

arr[3] = 128;

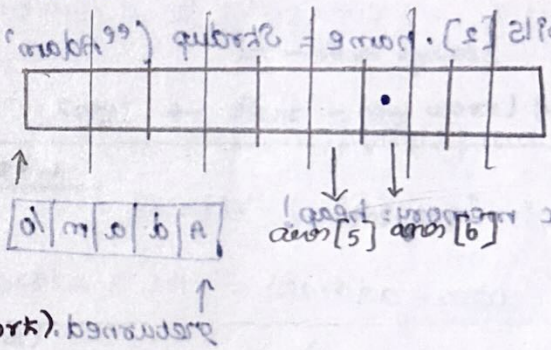
(Short *) arr[6] = 2

Cout << arr[3] << endl

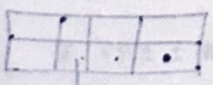
error printing - '128'

→ value at 3rd index (short).

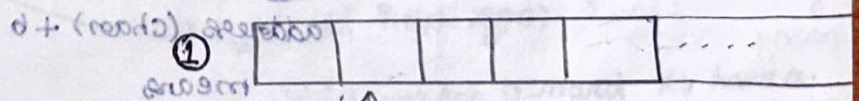
Consider a short.

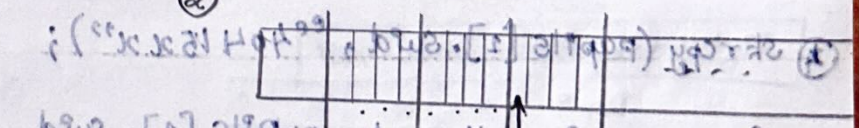


((short *)((char *)(&arr[1])) + 8)[3] = 100;

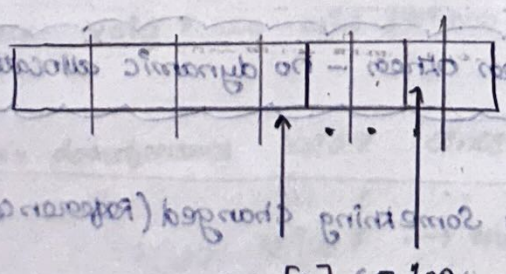


(short *) 2 bytes



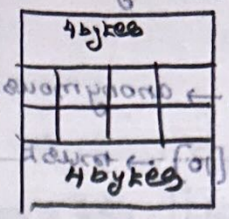


③



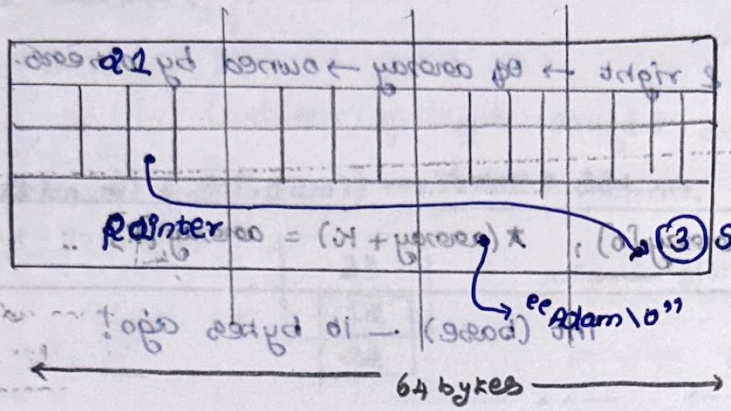
Ans[3] = 100

```
struct student {
    char * name;
    char surd[8];
    int numunits;
};
```



≥ 16 bytes

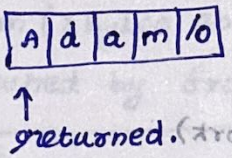
`student pupils[4];` array of struct.



`pupils[0].numunits = 21;`

`pupils[2].name = strdup("Adam");` → dynamically allocates enough memory - writes - returns address.

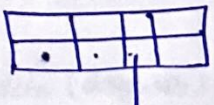
dynamic memory: heap!



③ `pupils[3].name = pupils[0].surd + 6;`

address (char) + 6 means

Source → destination



④ `strcpy(pupils[1].surd, "404 15 x x");`

Just copies that to `pupils[1].surd`

Copies: one char after other - no dynamic allocation

---

* 'Strcheck' - Result: Something changed (Reference)!
* make sure! (good address).

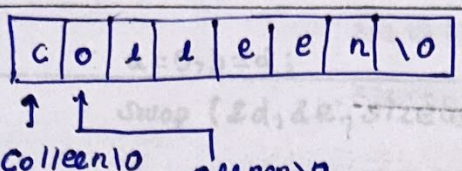
strcpy (pupils [3].name, "123456");

name → pointer [points to data: Somewhere written]

suid → array [actually have data there]

* "Dynamically allocated" ← free (need to)

⑤ pupils [7].suid [11] = 'A'; → no! (bound exceeded) – modern Compilers!



%s : from base address to \0

generic business

```
void swap (int * ap , int * bp)
{
  int temp;
  *temp = *a;
  * a  = *b;
  * b  = temp;
}
```

Passing address (Any change: reflect)

int → four bytes!

Copy! ← float → may work! (Same 4 bytes)

Lecture-4

Generic swap: 'Do bit copy'

```
void swap (void * vp1, void *vp2)
{
  void temp = * vp1;
  * vp1 = *vp2;
  *vp2 = *temp;
}
```

void * → no type info

→ wrong (need info: type)

void → not allowed to have a void variable

Void – must have some type

Note: void * → void pointer (also 4 bytes) – but no type info

Can't dereference void* (since : dereference int : get 4 bytes)

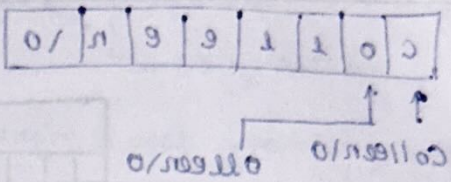
void * → No type info

(Can't dereference)!

Try new!

```
void swap (void *vp1, void *vp2, int elemSize)
{
    char buffer [elemSize];
    memcpy (buffer, vp1, size);
    memcpy (vp1, vp2, size);
    memcpy (vp2, buffer, size);
}
```

size = elemSize



I need to copy (temp) - third variable to swap

int → int temp

type → not known (Nothing as void temp)

Solution - array of characters

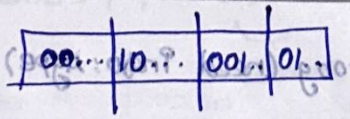
'1 2 3 4' → everything as a number '0/1'

char [size];

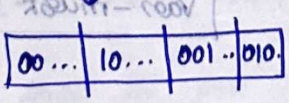
say int [4] ⇒ 4 bytes

addr - address, vp1 → address

Copy bytes/bits [So no change!]



memcpy (destination, source); → Takes size of element.



Considers [So int]

Now all perfect!

void *memory (void *to, const void *from, size_t numBytes);

strcpy → only to char

Energy: Copy & allocate!

Problem: many mistakes!

"Can't be copied"

entire-4!

dereferent data type: int, short

"Not the one we want"

sometimes!

2 bytes alone copied

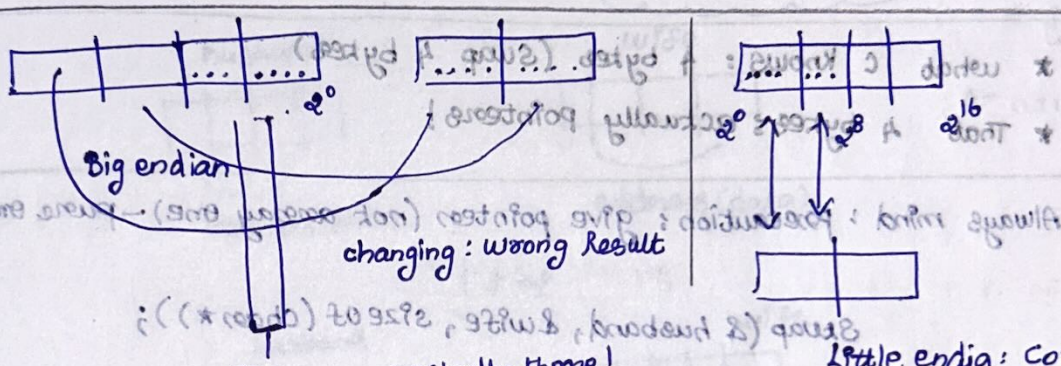
"Big endian"

a=5, b=2;

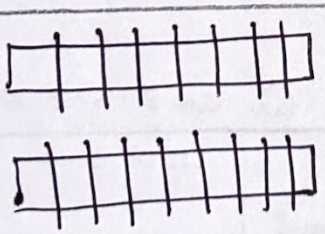
swap (&d, &e, sizeof(int));

"Everything legal in C – also work in C++"

* Some compilers let → dereference void * (Avoid it – leave it)



But: Be standard: don't use dereferent ones to swap!



double (given sizeof(int))

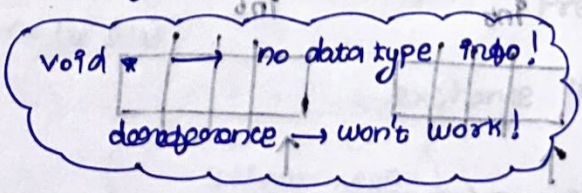
only 4 bytes – swapped: other 4 remains

"Disasters"

Already pointers

* Const → when sharing going on – I want it not to getting altered.

*



char * husband = strdup ("Fred");

"Fred \0"

char * wife = strdup ("Wilma");

(returns pointer to F)

swap(husband, wife, sizeof(char*));

F	r	e	d	\0

w	i	l	m	a	\0

char * means 4 bytes

memcpy( ) → does copies pointers, swaps

end: husband has wilma's address

wife has fred's address.

---

① husband wife → husband wife

"fred\0" "wilma\0"

"fred\0" "wilma\0"

---

* what C knows: 4 bytes (swap 4 bytes)
* That 4 bytes: actually pointers!

Always mind: precaution: give pointers (not array one) — pure one

swap(&husband, &wife, sizeof(char*));

'more liked'

&husband

&husband

↳ gives address

eg that array pointer

husband (pointer)

'more safe'

Actually VS code · swap(husband, wife, sizeof(char*)); works fine!

---

maybe: problem:

int int

pointer pointer

memcpy( ) → 3 times → swaps 4 bytes to 4 bytes.

husband → Fred\0

wife → w i l m a \0

4 bytes! first 4 bytes swapped! [remaining not]

Swap (&husband, &wife, sizeof(char*));

husband → Fred\0

Passing address of husband and wife!

wife → wife\0

pointers address

husband

address (data)

wife

address (data)

Swap

husband → wife's address

wife → husband's address

husband's address → "Fred\0"

wife's address → "wilma\0"

"one-more layer" — saves data - does job

Swap (husband, wife, sizeof(char*))

No addresses

wilma\0
Fred\0
→ 6 bytes

Say:

wilma\0
Fre\0

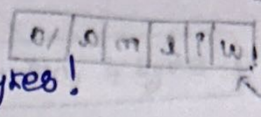
exchange 4 bytes

Fre\0 a\0 → Fre (alone read)

wilm → Non-Stop (read until \0)

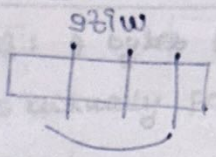
Recommended: Swap(&husband, &wife, sizeof(char*));

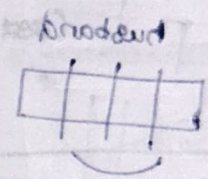
char *
char **
char ***
] All pointers: 4 bytes!



lsearch

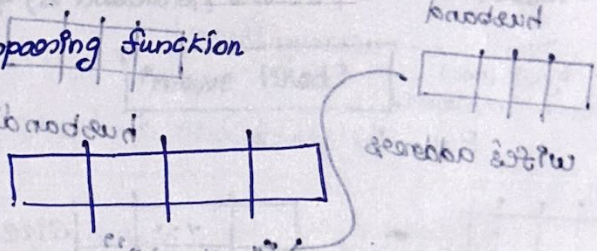
```
int lsearch(int key, int array[], int size)
{
    for (int i=0; i<size; i++)
    {
        if (array[i] == key)
            return i;
    }
    return -0;
}
```





generic lsearch

* Have a seperate Comparing function



* Tell: elemSize → how far can I move address each loop

match: return value of i

lsearch

```
void *lsearch(void *key, void *array[], int n, int elemSize,
              int (*cmpfn)(void*, void*));
```

Just write code inside lsearch

```
void *lsearch (void *key, void *base, int size, int elemSize)
{
    for (int i=0; i<size; i++)
    {
        void *elemAddr = (char*) base + i * elemSize;
        if ( memcmp(elemAddr, key, elemSize) == 0)
            return elemAddr;
    }
    return NULL;
}
```

→ void * (can't return int)

---

my function: takes: int/char/float/double

```
* intCmp ( void *ptr1, void *ptr2)
```

→ something like this

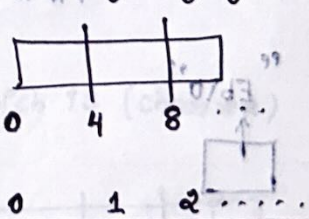
```
void *lsearch (void * key, void *array, int n, int elemSize,
               int (*cmpfun) (void *, void *),
               "own - own Comparison function" )
```

Lecture – 5

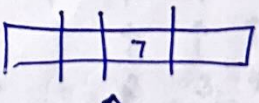
```
void *lsearch (void *key, void *base, int n, int elemSize, int
               (*cmpfn) (void *, void *))
{
    for (int i=0; i<n; i++) {
        void * elemAddr = (char *) base + i * elemSize;
        if (cmpfn (key, elemAddr) == 0)
            return elemAddr;
    }
    return NULL;
}
```

"handle byte"

"byte by byte"




```
int array [] = {4, 2, 3, 7, 11, 6};
int size = 6;
int key = 7;
lsearch
```



returns this address

(int *)

this case

```
int * found = (int *) lsearch (&number, array, &b, sizeof(int), intCmp);

if (found == NULL)
    printf ("Not found");
else
    printf ("Found");
```

Cast to type of function: (*cmpfn)

'match prototype exactly'

Really:

search: not done with float!

dereference!

*ptr1 - *ptr2

do with int, char. → won't work!

only for int

for float: use memcmp

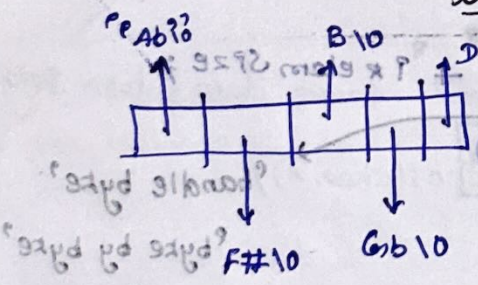
why
char *
int * → take 4 bytes!
extra 3

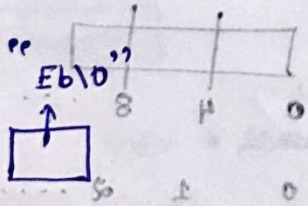
```
int intCmp (void *elem1, void *elem2)
{
    int *ptr1 = elem1;
    int *ptr2 = elem2;
    return (*ptr1 - *ptr2)
}
```

0 → match
-1) smaller / greater

'what's C is happening' ! → Not the best!

lsearch array of C strings

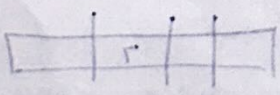




char * *

dereference array

dereference element (array) - pointer



giving void * lsearch → used void *

Compare strings!

* our Comparison function: accept : Char * , Char *

compare two strings!

pointers

In array : 2 hops] from actual characters!
In key : 1 hop.

lsearch knows : Four bytes

* Char * notes [] = { "Ab", "F#", "B", "Gb", "D" };

Normal (not in heap).

* char * favouriteNote = "Eb"

* char ** found = lsearch (& favouriteNote, notes, 5, sizeof(char *), StrCmp);

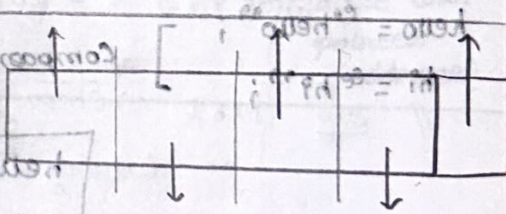
(actual width)

why char **

true data type of key (Same as) → all the pointers computed manually.

'symmetry'

* Length of each string may be different!



*

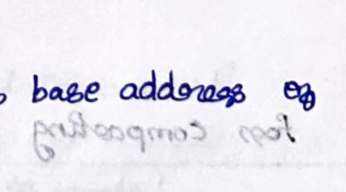
array → base address of 0th index (pointer)

*array → dereference − get in to pointer − holds base address of 'A'

Say: passing (lsearch)

* for lsearch: (void *) → array which is (char **)

* base address : pointers! (elemAddr)



passing to a function: StrCmp

(x4)

pointers

Need to write in a way

* For having string (0) : (*array, key)

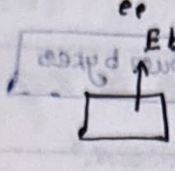
Compare : string 2 and key (*(array[1]), key)

Dereference : ((char**), (char*))

Why so much pain?

* do the same to key whatever doing to array (char**)
* make it char**

how:



&key char**

1st dereference : get pointer
next : get address of E

"Now scores leveled"

In bsearch: I am sending!

(char**, char**) to check

* C doesn't know; it knows 4 bytes (not even pointer)
* In StrCmp - Compare (char**, char**) !

```
hello = "hello";
hi = "hi";
```
] Compare (hello, hi);

hello - base address of h
hi - base address of h

for comparing arrays: Send string's base address (string)

*(char**)

```
int StrCmp (void *ptr1, void *ptr2)
{
    char *s1 = *(char**) ptr1;
    char *s2 = *(char**) ptr2;

    return (strcmp (s1, s2));
}
```

→ inbuilt

char * (strcmp i/p)

Problem

```
char *key1, *key2;
scanf("%s", key1);
```

What happens

```
char array[10];
scanf("%s", array);
```

hello → | h | e | l | l | o | \0 | | | | |

array

What can I do?

Save in some array[n]

↓ strdup / just assign!

```
scanf("%s", key1);
```

eg: `scanf("%s", array);`

`key1 = array`

(or)

`key1 = strdup(array);`

pointer

Uninitialized (no address to save)!

---

say `int a=10, b;`

`char *key1;`

a [10] b [?]

[?] key (uninitialized)

Initialize with pointer address!

Pointer arithmetic

`*(char *) + (char *)`

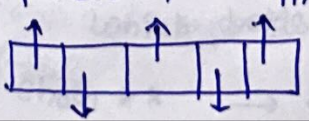
"Same types can be added & subtracted"

Job: lsearch: Search in array eg words!

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/*
int strCmp(void *ptr1, void *ptr2)
{
```

Actually receiving `(char **)` "Hi\0"

say



* dereference array to get to the pointers!

* dereference pointer to get to the (base address) - char*

2 hops to get to data - char**

Key: actually char*, passing &Key → char**

dereference to get to the pointer (&Key → *(&Key)

Key

('2 hops') (data ← *Key ← Key)

strcmp (char* a, char* b);
inbuilt

*/ */

int strcmp (void * Ptr1, void *Ptr2)

{

char *Pd1 = *(char**) Ptr1;
char *Pd2 = *(char**) Ptr2;

dereference one time to get (char*)

// Say: Ptr1 → char **

Casting → (char*) Ptr1 → doesn't get the job

Why? we need to enter/exit layers of dereference

return (strcmp(Pd1, Pd2));

0 → same
-ve → Pd2 > Pd1 (data)
+ve → Pd1 > Pd2 (data

}

Now: Research (same thing) — Universal!

void * bsearch (void * array, int size, int elemSize, void *Key,
(int) (*cmpfn) (void *, void *));

```
int main() {
    int n, i;
    printf("Enter the number of words: ");
    scanf("%d", &n);
    char *array[n], *key, temp[10];

    for(i=0; i<n; i++)
    {
      printf("Enter word %d: ", i+1);
      scanf("%s", temp);
      array[i] = strdup(temp);
    }

    printf("Enter key: ");
    scanf("%s", temp);
    key = strdup(temp);          (or) directly  key = temp

    char **found = (char**) lsearch(array, &n, sizeof(char *),
                                    &key, strcmp);

    if(found)
      printf("Found at %d.\n", (int)(found - array) + 1);
    else
      printf("Not found.\n");

    return 0;
}
```

∴ Can't directly copy a string in a pointer
↓ Instead give - initialize with string's address

why char ** found ?

array[] = {1, 2, 3, 4, 5};

array → int *

charu[] = {'a', 'b', 'c'};

charu → char *

∴ charu → pointer holding address (1 hop-data)

lsearch returning elemAddr → array[] = { ... };

array → has address of 1st pointer which holds address of a char

∴ char ** → 2 hops to reach data.

That's why char **

Note: (char *) , (char **) , (char ***) → same 4 bytes

But pointer arithmetic: not possible with dereferent!

Str Cmp

void * → but giving char **

So strcmp (char * ___ , char * ___);

Just do 1 hop (dereference) then send data!

why

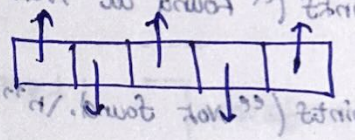
* (char * *) → instead of (char *)

char * * → pointer pointing pointer having (pointing) to a char.

char * → makes C thinks pointer - pointing char.

strcmp (char * , char *);

giving casted char *



Search for data (char) → but actually a pointer

Just casted: (let think char *)

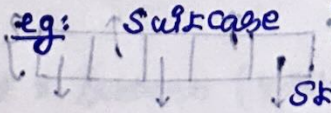
Not dereferenced

* (char * *) → dereference and make it char *

(char *) → Tell C to assume as char * → No work done (not dereference one level of memory)

'Technically wrong'

eg: Suitcase inside suitcase



strcmp → accept inner suitcase

① (char **) → dereference → (char *)

mean: I opened outer one gave inner suitcase to strcmp

No problem!

② (char *) → a suitcases → but I pretend – It's the innermost one → At some point

strcmp – opens that suitcase (actually not innermost) to get to data – but sees another suitcase.

cheated! error

→ void * → No data info → can't be dereferenced!

→ void ** → It knows pointer pointing to void *

→ char * → array of characters pointing.

1 hop – get char data.

→ char ** → 2 hops to get char data

pointer pointing to a pointer which points to char data

---

bsearch

Forget &key (& in ptr) → key → In strcmp → I am dereferencing

get to the char

key → "hello"

(h)

again strcmp (char * —, char * —);

what I'm doing: strcmp (char * —, char);

(Assume char as my key address)

error! Disaster!

why: &key [instead of using like] → key (char *)

```
int StrCmp (void *Ptr1, void *Ptr2)
                  └→ array (char **)
{
    int *Pd1 = *(char **) Ptr1;

return (Strcmp (Pd1, (char*) Ptr2));
}
```

→ This kind I can use bsearch(..., key, ...)

Reason: Common: Easy to understand!

So whatever I'm doing to array, do it to key!

So - less number of problems to tackle - when it's uniform!

---

* many comparison functions → deals with same data type
* Be a Roman in Rome → give what they want!
* Use char **, even key is char * by &key!

---

binary search!

bsearch()
bsearch()

```
void *bsearch (void *key, void *base, int n, int elemSize,
               (int) (*cmpfn) (void *, void *));
```

Binary search - done in sorted!

→ use address carefully!
→ In oop → function inside classes → methods! [look similar]
→ methods: have the address of the relevant object lying around as this invisible parameter okay! via the invisible parameter]

In Function: No behind the scenes stuff → No oop.

this used in Java - In receiving object → keyword → if name of method variables & global class variables [properties] - clash.

---

int (*cmpfn) (void *, void *)

↳ previous version * (must) to know pointer to function.

No problem if old way; Now: no need!

---

generic algorithm to generic DS

* Do it in C way!
* Ignore templates, vectors etc..

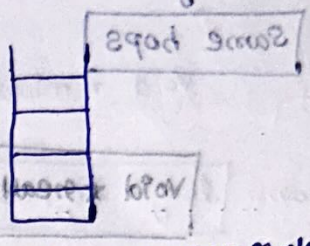
Int specific - Stack — no void * business yet!

* behaviour .h, .cc scheme (C++)
* .c (C)

No const, class, public/private

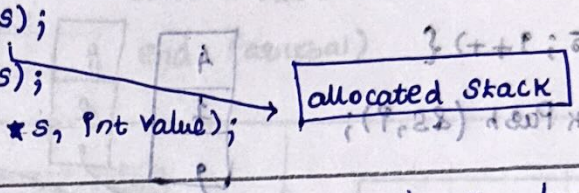
typedef required in C — not in C++

```
typedef struct {
    int *elems;
    int logicalLen;
    int allocLength;
} stack;
```



stack: number of elements (length).

→ void StackNew (stack *s);
→ void StackDispose (stack *s);
→ void StackPush (stack *s, int value);

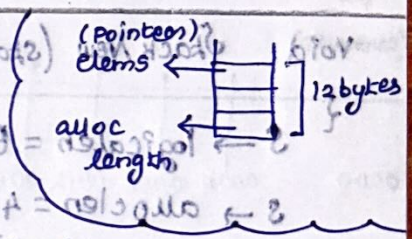


* Dynamically allocated - Note: allocLength - how much allocated!
* (*elems) - how much I'm using!

→ void StackPop (stack *s);

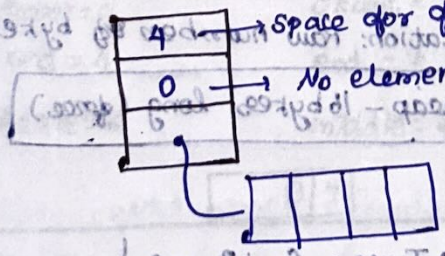
---

* stack s; → s → allocated with 12 bytes



* StackNew (&s);

Now - initialized:



Array of integers (4)

Purpose: StackNew → Initialize

we are doing dynamically!

---

* Use of logicalLen → use to place next element

(Address)

* Initially - 4 - when 5th element! - panic

* reallocate with double the size - continue!

Generic

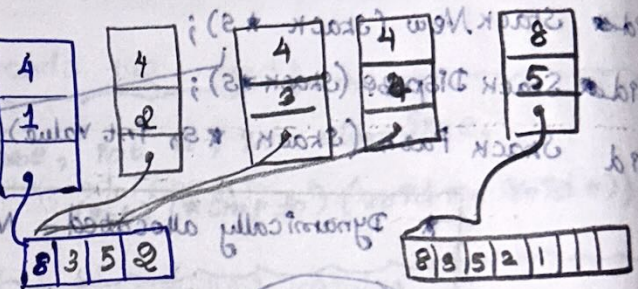
* manually do - Some functions! - always void *

* manually Compute insertion index to push next Callers

* Same hops

```
void *realloc (void *ptr, size_t size);
```

```
for (i=0; i<5; i++) {
    Stack Push (&s, i);
}
Stack Dispose (&s);
```

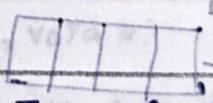


"Double the size"

```
void StackNew (stack *s)
{
    s->logicalen = 0;
    s->alloclen = 4;
    s->elems = (int*) malloc (4 * sizeof (int));
}
```

dynamic allocation: Raw number of bytes

blob in heap - 16 bytes long (free)



```
assert (s->elems != NULL);
```

→ True: Continues!

False: ends program

'Debug' - precaution!

Better - use - now → after some time program crashes

∴ Atleast: we know what's wrong!

It won't dereference NULL pointers!

Like: I'm absent! Your code broke here! - Sweet of you!

{10 9 8 7 6 5 4 3 2 1}

pointer arithmetic:

Normal bsearch

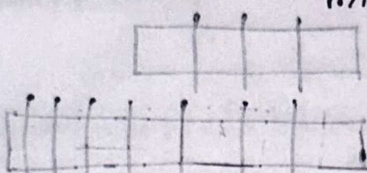
- start
- end
- middle

when (start == end) → stop!

middle > key

start, end = middle

middle < key



start = middle + 1

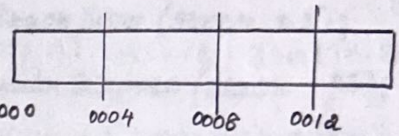
end (as usual)

0|1|2|3|4

$\frac{0+4}{2} = 2$

0|1|2|3|4|5

$\frac{0+5}{2} = 2$



$\frac{0012 - 0000}{4} = \frac{12}{4} = 3+1$

= 4

(elements)

① 0|1|2|3|4

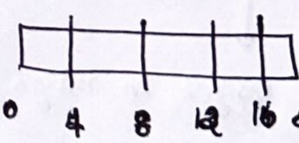
5 key

start = 0
end = 4
middle = 2
→ start = 3
end = 4
middle = 3
→ start = 4
end = 4
middle = 4

4

start = 0
end = 4
middle = 2
→ start = 3
end = 4
middle = 3

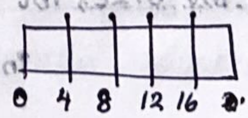
3|4



$\frac{16-0}{2} = 8$

0+8 ✓

Idea:



$\frac{\frac{16-0}{4 \to size}}{2} = \frac{4}{2} = 2 \text{(mid)}$

array + 2 x elemsize → address of mid!

Silly mistake!

(int) (*cmpfn)(void*, void*) → casting what's mean?

I need to indicate return type (not casting)

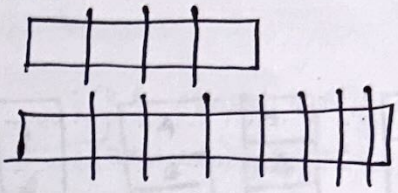
correct: int (*cmpfn)(void*, void*)

Pointer math

Errors: (All pointers)

```
void * start = (void *) array;
void * end = (void *) ((char *) array + (size - 1) * elemSize);
while (end >= start)
{
    void * middle = (void *) (((long) start + (long) end) / 2);
```

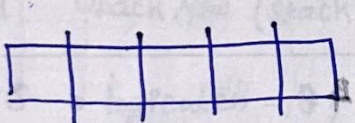
Error: long: 8bytes, Pointer: 4bytes!



"Division!"

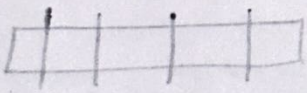
Int → For Int: Core dumped

If I need to do it array pointer way!



0008 → middle.

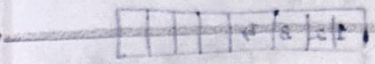
(array (start) + end)/2 → Not working!



"Core dumped"

```
void bsearch (void *array, int size, int elemSize, void *key,
              int (*cmpfn)(void *, void *))
{ int start = 0, end = size - 1;
  while (end >= start)
  {
    int middle = ((end + start) / 2);
    int r = cmpfn ((char *) array + middle * elemSize), key);
    if (r == 0) {
```

```
        return ((void *) (array + middle * elemSize));
    }
    else if (r > 0)
    {
        end = middle;
    }
    else
    {
        start = middle + 1;
    }
}
return NULL;
}
```



Stack.h

```
typedef struct {
    int *elems;
    int logLength;
    int allocLength;
} Stack;
```

Lecture: 6

```
void StackNew(Stack *s);
void StackDispose(Stack *s);
void StackPush(Stack *s, int value);
int StackPop(Stack *s);
```

Stack.c

```
void StackNew(Stack *s)
{
    s->logLength = 0; // Stack: empty
    s->allocLength = 4; // Initially: 4
    s->elems = malloc(4 * sizeof(int));
    assert(s->elems != NULL);
}
```

→ Good habit!

```
void StackDispose(Stack *s)
{
    free(s->elems);
}
```

→ delete in C++

free(s) → wrong! s → Not allocated (dynamically)
local - (not sure!)