1
2 2
3 3 3
2 2 2 2
1 1 1 1 1

N/2

5-3
5-4.

If (N/2) → print i

else print n-i

p=0
j=0
} start

Arrays & Strings

* Array - Same data type - contiguous memory location
* most common array - string - terminated by null character.

sizeof (array) → no. of elements × size of element

eg: 5 × 4 = 20
(5 elements, type: int)

* Contiguous memory locations in index

* Array - itself an array - not pointer - atleast can't do all things that could be done with an usual pointer.

Pointer to an array

```
int *p;
int sample[10];
p = sample
```

same as

p = & sample[0]

* name of array sample - indicates base address
* Note: Array is an array - !

Passing to functions

* only able to pass a pointer to an array to a function.

" No need to create anything new - access from memory instead! "

① func1 (int * x) → pointer

② void func1 (int x[10]) → sized array

③ void func2 (int x[]) → unsized array.

Receive a pointer to a 1-d array: - 3 ways!

① pointer
② sized array
③ unsized array.

] each tells an integer pointer going to be received.

func1(int array[]) → Some length - But array going to be received.

Note

func1 (int a[10])

↳ * Not important - C doesn't do bound checking!

Note: C only care about int - an int pointer is going to be given!

Strings

* most common — 1d array — character string (null terminated)
* null terminated string - only defined string eg C.

C++ → Supports oop approach

Note: Char array length → 1 char always long than the longest string it will hold.

eg: To hold 10 char string.

char str[11]; → Room for Null (compiler adds null char - default)

---

* String constant → list of char enclosed in double quotes!

strcpy (S_1, S_2) → Copies S_2 to S_1

strcat (S_1, S_2) → Concatenates S_2 at the end of S_1

strlen (S_1) → length of S_1

strcmp (S_1, S_2) → 0 if same, < 0 if $S_1 < S_2$
> 0 if $S_1 > S_2$

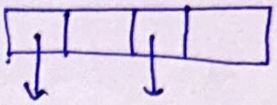
<String.h>

* strchr(str1, 'h'); → returns the pointer to the first occurance of 'h'

* (int) addr - (int) strchr(str1, 'h') → 0

∴

* strstr(str1, "hello") → returns pointer to first occurance of str1.



difference b/w these addresses is index.

2-d arrays

* Array of 1-d arrays.

Size = size of 1st index (x) size of 2nd index (x) size of base index

1	2	3	4
5	6	7	8
9	10	11	12

* 3 × 4 × 4 = 48 bytes allocated

* Teacher has 3 classes, each class has 30 students!

* Roll number of each student!

Array of strings

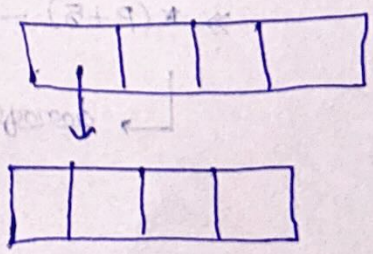
1) str [ ] [ ]

↓ no. of strings

↳ size of each string.

* gets (str_array [2]) → get string & store in 2nd index

gets(&str_array [2] [0]) → Same : why? :

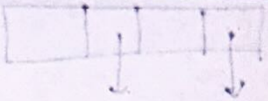


1 has the array pointer to another string array!

get strings - until a next line is given as i/p

while (*Str1 [i] != '\n')

multidimensional array



int m [4] [3] [6] [5]; → example.

"Pointers & arrays are closely related"

Indexing pointers

* Array name without index → pointer to 1st element in the array!

» Char P [10]

P and &P[0] → Same!

address of 1st element same as the address of the array.

Note:

* Array without index generates a pointer.
* pointer can be indexed as if it were declared to be an array!

» int *P, i[10];

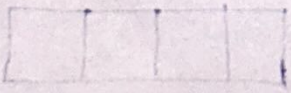
* P now holds address of i

» P = i;

» P[5] → 6 th element of i

» *(P+5) → 6 th element of i

↳ dereference!



2-d

a same as &a[0][0]

a[0][4] → *(a+4) (or) *((int *) a + 4)

a[1][2] → *((int*) a + 12) → Note a → 10 by 10.

∴ +12

10 9 8 7 6 5

fmin = 10	9 < fmin	8 < fmin	7 < fmin	6 < fmin	5 < fmin
smin = 9	fmin = 9	fmin = 8	fmin = 7	fmin = 6	fmin = 5
	smin = 10	smin = 9	smin = 8	smin = 7	smin = 6

Some case

54, 56, 78, 23, 34

fmin = 54	54 < 78	54 > 23	56 > 23
smin = 56	No change	fmin = 23	No change
		smin = 54	

Algorithm failed! → need extra.

else if (smin > array[i])

smin = array[i]

---

1 2 3 4 5 6 7 8 9 10

fmin = 1	fmin < 2	tmin = 2	fmin < 2
smin = 2	fmin < 2 → yes	smin = 2	fmin < 3
tmin = 3			tmin < 3

∴ fmin = 1
smin = 2
tmin = 2

not working!

---

1 2 3 4 5 6 7 8 9 10

fmin = 1	fmin < 2
smin = 2	smin < 2 && fmin > 2
tmin = 3	tmin < 2 && fmin > 2
	↳ No

What's wrong

1, 1, 1, 2, 3, 4

fmin = 1
smin = 2
tmin = 3
OK!

But!

1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

fmin = 1
smin = 1
tmin = 2

1 2 3 4 5 6

fmin = 1 | fmin > 4
smin = 2 | fmin > 4
tmin = 3 | tmin > 4

6 5 4 3 2 1

fmin = 6
smin = 5
tmin = 4

fmin > 3
fmin = 3
smin = 6
tmin = 5

fmin > 2
fmin = 2
smin = 3
tmin = 6

fmin > 1
fmin = 1
smin = 2
tmin = 3

−8 −3 −10 −32 −1

fmin = −8
smin = −3
tmin = −10

fmin > −32
fmin = −32
smin = −8
tmin = −3

fmin > −1
smin > −1
tmin > −1

problem:
fmin = −10
tmin = −8
?

−8 −3 −10 −32 −1

fmin = −8
smin = −3
tmin = −10

fmin > −3
smin > −3
tmin > −3

fmin > −10
fmin = −10
smin = −8
tmin = −3

fmin > −32
fmin = −32
smin = −10
tmin = −8

fmin > −1
smin > −1
tmin > −1.

```
if (array[i] < fmin)          ⟶ i from 1 to n-1
    tmin = smin
    smin = fmin
    fmin = array[i]

else if (array[i] < smin)
    tmin = smin;
    smin = array[i];

else if (array[i] < tmin)
    tmin = array[i];
```

Insert an element

10 20 40 50 60 → 30 (at index 2)

10 20 30 40 50 60

temp = 40
array[2] = 30
element = 40

10 20 30 50 60

temp = 50
array[3] = 40
element = 50

10 20 30 40 60

temp = 60
array[4] = 50
element = 60

10 20 30 40 50

array[5] = 60

10 20 30 40 50 60

Case: 1 – empty array

10	10	10	10

→ 0th index 1

for loop fails ✓

array[0] = element

2 → 3 0 → 1 1 → 2 3 → 4

0 2 → 1 0 0 3 → 1 2 0 0 3 → 1 2 0 4 0 3

index = 10
count = 10

count = 11

index = 12
count = 7

count = 13

10 20 30 40 50 → 10 30 40 50

delete.

Case: 1

*Can't delete from empty

10
delete index 0
count 0
↓
empty

2-d array as pointer → 1st element address.

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

0000	0004	0008	0012
0016	0020	0024	0028
0032	0036	0040	0044

Pointer: pure address → we need to dereck

① (0000 + 4 * 0) + 0 , (0000 + 0) + 1

(0000 + 0) + 2

② (0000 + 4 × 1)

↓

int pointer

Problem: * (((int *) array + 4 *) + 3)) ;

Problem.	60	64	68
	70	74	78
	80	84	88

→ should be as

60	64	68
72	76	80
84	88	92

This way: (* + 3) → Not working!

0 + (0+0) * 4 | 0 + (1 + 0) * 4 | 0 + (2+0) * 4 = 8

0 + (1) * 4 | 0 + (2 + 1) * 4 | 0 + (2+1) * 4 = 12

0 + (2) * 4 | 0 + (1 + 2) * 4 | 0 + (2+2) * 4 = 16

Problem!

0	4	8
12	16	20
24	28	32

(0+0)+0

→ (0+0)+4

(0+0)+8

(0+4)+0 = 4 → need: 12

(I need = 12)

Solution: Start from 12, 24 } In 2, 3 row!

~~0 + (2 * (1+1) * 2)) = 8~~

(2 * 2 * 4) = 12

(or) Simple

P = P + 3 * 4 → Incrementing by 4 → Simple Solution!

↓

character pointer (int *) → P+1.

0000 + (0x4)

0000 + (1x4) = 0004 → Int correct.

0 4 8 12
16 20 24 28

Array Initialization

» coorray [ ] = {1, 2, 3};

» choor array_name [size] = "string"; → choracteor array!

eg: choor str[9] = "I like C"; Same as

choor str[9] = {'I', ' ', 'l', 'i', 'k', 'e', ' ', 'c', '\0'};

2-d array

* subaggregate grouping - braces eg for each dimension

Int soors [3][2] = {1, 2, 3, 4, 5, 6};

Int soors [3][2] = {{1, 2}, {3, 4}, {5, 6}}; → No need for enough initializers

eg:

Int soors [4][2] = {{1, 1}, {2, 2}, {3, 3}}

last one set to zero : by default.

Unsized array initialization

» choor c1[] = "Read error \n"; → %s → string

"Must - specify - left most dimension"

* we can build tables of varying length
* Compiler - automatically allocates enough storage.

Int soors [ ][2] = {{1, 1}, {2, 2}, ... , {10, 10}};

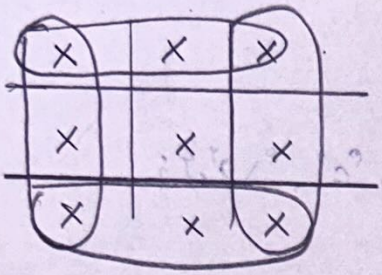
Advantage: we can lengthen / shorten array - without changing array dimension!

Variable length array

* C99 - array dimension need not be constant - may be any valid expression.
(Runtime) → Variable - length array.

> "Only - local arrays can be of variable length"

Create a variable length array!



① 3 rows
② 3 Columns
③ 2 diagonals.

} To win possibilities!

Simple: 3×3 matrix

Our Computer: Not so brave - Simple enough!

"we can upgrade later!"

X	X	X
X	X	X
X	X	X

0000	0004	0008
0016	0020	0024
0028	0032	0036

0000	0004	0008	0012
0016	0020	0024	0028
0032	0036	0040	0044
0048	0052	0056	0060

1st row = 0000 + 16 (char *)

(int *) → 4 positions away!

(int *)

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15.

2nd row: (1*4) + 0 (Column)

(1*4) + 1.

3rd row: (2*4) + 0

(2*4) + 1.

i → 0 to row

j → 0 to col.

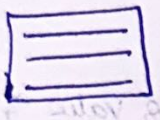
(i * col_count + j)

int → Automatically 4 bytes

0000	4	8	12
16	20	24	28
32	36	40	44
48	52	56	60

=

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15.



flag = 0

arr + 4 * 8 + 3

for loop - if failed → return 0

return completed → return 1

arr * 3 + 2

0 + 0

size = 4 | 0, 3

(0 × 4) + 3 =

(1 × 4) + 3 = 7 (1, 3)

→ row wise

Column wise

0 * 4 + 0 = 0

1 × 4 + 0 = 4

2 × 4 + 0 = 8

Row wise

(i * size) + j

i = 0, i < size, i++

j = 0, j < size, j++

Column wise

(j * size) + i

Pointers

* pointers provide the means by which functions can modify their calling arguments.

* pointers support dynamic allocation.

* Improve efficiency

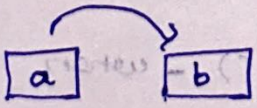
* Support dynamic DS. — Binary tree — Linked lists

* Easy to get error! — Difficult to find bugs! — potential for abuse!

* Pointer — holding address (variable)

* That address — memory location of an object.

"point to another" — one var has address of other



"a" has address of b

"a" points to "b"

declare: type * base ;

eg: int * variable1 ;

Declare, initialize:

variable1 = & count ; & → address of.

» * → complement of &

» v = *m ;

↳ de reference m (m = & count) → dereference address gives value of count.

Assignment

* we can use pointer (RHS) to assign another pointer (LHS) – same type.

```
int *P1, *P2 ;
P1 = & x ;
P2 = P1 ;
```

Conversion

* One type of pointer – can be converted to other type

*

- ↙ involve void *
- ↘ don't involve void *

* In C – allowed to assign any other type of pointer to a void * pointer.
* void * pointer – generic pointer.
* void * → specify a pointer – whose base is unknown.
* void * → allows a function to receive (specify) a pointer parameter – capable of receiving any type of pointer argument without – reporting mismatch.
* Used to refer to raw memory → returned by malloc() – when semantics of the memory – unknown.
* No explicit cast required to convert from/or a void * pointer.

* Except void* → All other pointer types – need to be casted explicitly.
* Sometimes – Casting pointers – results in unexpected behaviours.

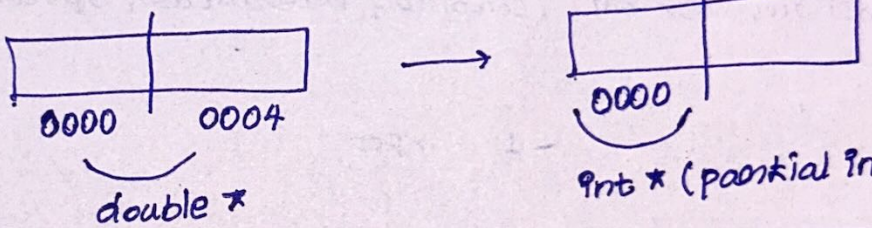
eg: assign x to y through pointer "p"

```
double a = 111.1, b;
int *c = (int *) &a;
printf ("%d\n", *c); → 1717986918 (Not expected one!)
```

---

why?

* Though cast is correct – doesn't act as intended.
* int – 4 bytes
  double – 8 bytes
  » So (int *) gives only 4 bytes of info!
  » Actual double – considers as int (int *) → makes collapse



* A cast is not required – when converting to zero (Null pointer)

Note: Unlike C → In C++ must to explicitly cast – all pointers including void*. So many C programmers – use explicit cast – So that their code is also compatible with C++.

---

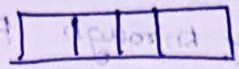
pointer arithmetic

* Addition, Subtraction (alone performable)
* Say int 4 bytes

```
int* a = & b;   → 2004
a++;            → 2008
```

a-- → 2000

↑ Pointer → increments memory it points to.



→ increment ++

4 blocks = 1 unit → int increments by 4 addresses.

---

* "can't add two pointers"
* can't do bitwise operations
* can't add / subtract types eg float / double to or from pointers.

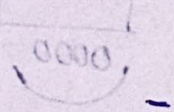
Pointer comparison

```
if(p<q)
    printf("p points to lower memory than q\n");
```

use: when two pointers points to same object (array)

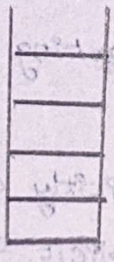
Stack: First in, last out (Compilers, Interpreters, Spreadsheets)

push pop



-1 → pop

Say: 100 element stack



Underflow: no element (element count)

Overflow: No space (101 th element).

Note: In stack we play by rules: first in – last out

last in – first out

* "we only remove last element alone"] No inbetween
* "Add- at last space alone"] Stuffs.

0 → push

-1 → pop

1 → pop.

Pointers & arrays

```
char str[80], *pt;
pt = str;
```

→ pt has been set the address of first ele of str → str holds that

Access array elements → Pointer arithmetic (Done with normal pointer not array pointer)
→ Array indexing (only done with array pointer not with any other pointer)

→ Faster: pointer arithmetic

Pointers & arrays

Array of pointers

```
int *x[10];
x[2] = &val;
```
] Array of pointers.

*x[2] → dereference.

Pass to a function

```
void display_array(int *v[])
{
:
}
```

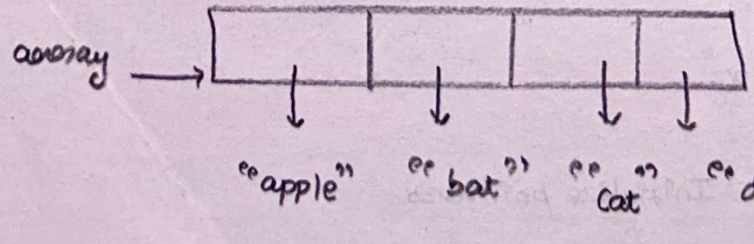
* v → pointer to an array having int pointers.

```
void Syntax_error(int num)
{
static char *error[] = { "Cannot open File\n", "Read Error\n", "Write Errors\n",
                         "Media failure\n" };
printf("%s", error[num]);
}
```

* error → holds pointer to each string

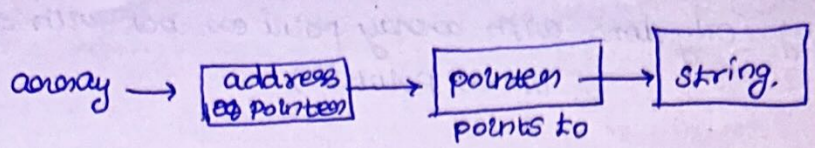
%s → needs a pointer (address)

No need to dereference!



array has the address of first element → Address of a pointer.

* we want pointer that points to string not its address.



* dereference array gives 0th pointer
* dereference again gives string 1st char.

char ** array

what we do!

char * ptr = array[0];

array[0] → pointer points to string (our need)!

Note: string1 & 2 can't be (need not) contiguous

'So double dereference need to be done'

Multiple Indirection

* 'pointer point to a pointer that points to the target'
* pointer of pointers.

float ** newbalance;

newbalance is a pointer to a pointer of type float

*newbalance itself is not a pointer to float!

```
int x, *p, **v;
x=10;
p = &x;     *p → x
v = &p;     *v → p
            **v → *p → x
```

Initialize pointers

* Global & Static local pointers are automatically initialized to NULL.

Use pointers before assigning

```
int *p, x=10;
printf("%p %d\n", p, *p);
```

"core dumped"

Convention

* Assign to a value null - C guarantees no object will exist at the null address assigned to nothing - Shouldn't be used!

```
char *p = 0;
```

using MACRO

'NULL' → In stdio.h.

```
p = NULL;
```

"A pointer that doesn't currently point to a valid memory location is given the value - null"

Say: null → Nothing there - can't be used!

```
int *p;
p=NULL;
```
→ Just a convention - not a rule by C

```
int *p=0;        ⟶ Null pointer
*p = 10; /* Wrong */  ⟶ Null pointer doesn't hold object.
```

Compilation happens but can't run "0 → Not open for sale"!

→ Reason: NULL pointer assumed to be unused.

→ use null pointer to mark end of the pointer array.

→ Routine accessing that know - encounters end.

eg: search.

for (q=0; 0; q++) ⟶ False ∴ 0 means failed.

for (q=0; NULL; q++) ⟶ Same!

p[k] → false when pk is NULL

Questions

Char *p = "Hello World";

>> p → pointer - not array (holds address of string not a string)

① where "Hello World is saved" ?

'How compilers handle string constants'

* Compiler creates string table - Stores data (String constants)
* preceding declaration statement - places address of "hello world" in to pointer p.

where: In some memory land : where strings are saved.

pointers to functions

→ Powerful feature: function pointer.
→ Function has a physical location in memory - assigned to a pointer.
→ Address - entry point of the function - (address used when func. is called).
→ once a pointer points to a function - that func. can be called through that pointer.
→ Allow to pass as arguments to other functions.

get address of a function

* use function name without any ()/arguments.
* Simpler to array.

'Really useful'

```
#Include <stdio.h>
#Include <stdlib.h>
#Include <string.h>
void check (char *a, char *b, int (*cmp)(const char *, const char *));
int main (void)
{
  Char s1[80], s2[80];
  int (*p)(const char * , const char *) ; // function pointer.
  p=strcmp; // assign address of strcmp to p
  printf ("Enter two strings. \n");
  gets(s1);
  gets(s2);
  check(s1, s2, p); // pass address of strcmp via p.
  return 0;
}
  void check(char *a, char *b, int (*cmp) (const char *, const char *))
  {
   printf ("Test for equality. \n");
   if (! (*cmp) (a, b)) printf ("Equal");
   else printf ("Not equal");
  }
```

---

int (*p) (const char *, const char *);

* p → pointer to function that has two constant char * parameters
* Returns int result.
* parenthesis (*p) → must to interpret declaration

Same format: even though declaring other function pointers - even return type & parameter differ!

↓

Same as main()

Note: Function pointer is declared using same format as was p inside main()

→ int o/p → 2 arguments.

>> int (*p) (const char *, const char *)

>> p = Strcmp;

Assigns the address of Strcmp to P

(*cmp)(a,b)

one way to call a function through a pointer!

mostly used: Easy to understand - a function pointer is used.

two ways

(*cmp)(a,b) (Preferred)

cmp(a,b)

>> Check (S_1, S_2, P); → giving function pointer

>> Check (S_1, S_2, Strcmp);

eliminates the need for an additional pointer.

① digits given as i/p → Compare digits

② string given as i/p → Compare strings.

int (*cmp) (const char *a, const char *b)

name (reference)

parameters

o/p

Just like Compare (S_1, S_2, Strcmp) function.

>> function_pointer_equality_check.c

Dynamic allocation functions

* Obtain memory while running
* Global variables – get memory while compilation time – (stack – local variable)
* Global, local variables – can't be added during execution
* use: when program can't be known (storage) ahead of time.

eg: dynamic DS – linked list, binary tree – inherently dynamic – in nature (grow | shrink as needed – allocate as needed, free as needed)

* dynamic allocation – done in heap – free memory – not used by OS, program/ anyother functions / programs in the computer.
* size of heap – not known in advance – fairly large enough
  ↳ But finite, exhausted.

---

* malloc() → allocates – remaining (portion) of free memory (heap) – allocated.
* free() → Releases. – memory returned to system.

<Stdlib.h>

```
void *malloc (size_t number_of_bytes);
```

→ returns pointer!

number_of_bytes – want allocate.

no pointer details

(Cast to req. type)

→ Successful execution returns pointer to first byte reign

→ Not enough memory → failure → returns NULL.

```
char *p;
p = malloc(1000);
```

→ Contiguous 1000 bytes.

↳ Raw (∴ no type cast) → automatically converted to the type of left side variable.

→ In C++ → type conversion must

```
int *p;
p = malloc(50 * sizeof(int));   → 50 integers
```

Check: value returned by malloc → is not null → else crash your program

Test for success

```
p = malloc(100);
if(!p) {
    printf("out of memory.\n");
    exit(1);
}
```

```
void free(void *p);
          ↳ pointer
```

→ "free" → if not no space for further function - manage memory.

→ memory freed → use for (reuse) subsequent calls.

Dynamically allocated arrays

* Allocate malloc()
* use like an array. – In addition to create dynamic array for indexing.

> "Any pointer can be indexed – if it was an array" – contiguous memory

```
s = malloc(80);
```

>> Get array print backwards

1	1	1	1
2	4	8	16
:	9	27	81
10	100	1000	10000.

"Each row has 10 columns"

int (*P)[10];

↳ must – parantheses.

each pointer holds 10 ints (address) array

* P is a pointer to an array of 10 ints.
* Base type: 10-int array.

If P incremented → points to next 10 ints

* P → pointer to 2d array → 10 elements in each row!
* P → indexed as 2d array.

C++ → more specific

P = (int (*)[10]) malloc (40 * sizeof (int));

int *P → holds first element's address

int (*P)[10] → first element has 10 elements.

1	1	1
2	4	8
3	9	27
4	:	:
:	:	:
10	100	1000

i → from 1 to 10.

Pow (i, 2 or 3)

display (1 row, 2nd row, 3, 4)

Remember: int (*P)[10];

P → pointer (having) – pointed to 10-int array!

Restricted array

→ C99 → New features – apply only to pointers

→ restrict

→ pointers qualified by restrict → feature

why?

→ Pointer qualified by restrict is initially the only means by which the object it points to be accessed.

→ Access to the object by another pointer - can occur only if the second is based on the first.

→ Access to the object is restricted by that one pointer.

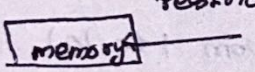
Access to the object - restricted to expressions based on the restrict - qualified pointer

* Use: function parameter / point memory allocated via malloc()
* Use: Better (compiler) able to optimize certain types of routines.

eg: If function has two restrict - qualified pointer parameters

Compiler - Assumes - they point different object - no overlapping.

"doesn't change semantics"



"All accesses to that object must through p'

Directly not possible.

Problems with pointers

* mixed blessing - hard to debug - (Pointer - not itself not the problem - but object accessing through that pointer - wrong piece of memory)
* Garbage value - overwrite - etc..
* Lose sleep - troublesome - Important to know Common errors.

```
Int  x, *P;
x = 10;
*P = x;
```

Int *P → not initialized with any address - so

Can't ask it to save something in memory

Unknown memory location.

* Sometimes unnoticed → program grows - less memory - overwrite - some thing very important.

Note: most compilers issue a warning

misunderstand - how to use a pointer

```
int x, *P;
x = 10;
P = x;
printf("%d", *P);
```

x → int variable

P → int pointer variable. (doesn't hold value but address).

"So prints unknown value"

Correct: P = &x; (Atleast Compilers - warn).

Incorrect assumptions - about placement of vars in memory

Note: we can't know where our data is placed in memory.

(same, different - not sure - depends upon OS, Compilers).

"Any comparisons b/w pointers that don't point to a common object may yield - problems"

```
char x[80], y[80];
char *p1, *p2;
p1 = x;
p2 = y;
if (p1 < p2)
```

Invalid concept - to determine position of pointers (rare).

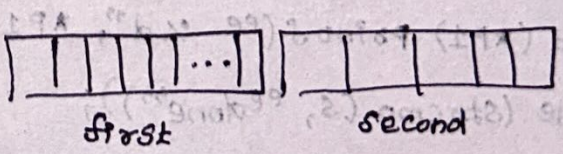
(Sometimes - rare - we use this)

```
int first[10], second[10];
int *p, t;
p = first;
for (t=0; t<20; ++t)
    *p++ = t;
```

Not a good way!

Some compilers work - but assumes both arrays will be placed back in memory with first first.

Why? Assumed case



we are not using 2 pointers - just use first

Theory: After completing first 10, goes to second 10.

Exercise

int (*p)[10] → Told a pointer pointing to 10 element array

↓

9458654072899 2

p++;

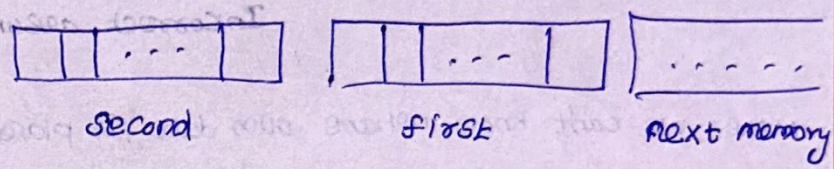
↓

9458654072903 2

40 bytes shifted.

∴ int → 4 bytes, 10 elements → 40 bytes.

Contd Problem: Suppose



for(t=0 → t<20; t++) → Alright up to first

second 10 → Some other memory

Problem: type not assigned memory blocks

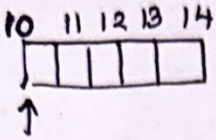
second [10] → not initialized.

'Bad way'

Dangerous bug

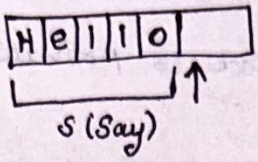
```
char *p1;
char s[80];
p1=s;
do {
    gets(s);
    while (*p1) printf("%d", *p1++);
} while (strcmp(s, "done"));

return 0;
}
```

Problem: P1 assigned to address of s once



① get string until s = "done"

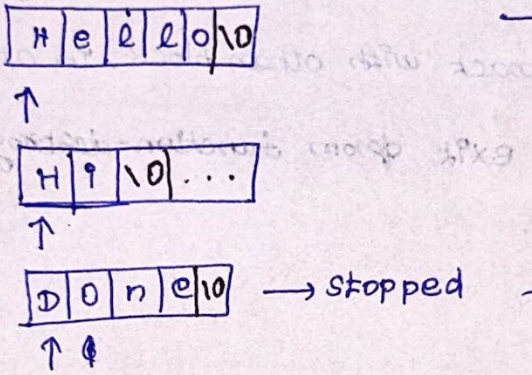
Hello → pointed now pointer points to



over a period of time

Pointer (P) - points out of s.

expectation:



Solution

```
do {
    P1 = s;  ⟶ Initialize each times
    gets(s);        (That's how we store at s each time)
    :
}
:
```

Functions - Building blocks of C

```
return_type function_name ( parameters list)
{
    body
}
```

* Parameters: optional.

eg: f (int i, int k, int j)

Scope

* Access to data / code.
* function - discrete block of code
* Function defines a block scope - function's code is private to that func & only accessable by it until returned.

Note: Can't use jump to statements b/w functions

Unless - global variable - need to pass as arguments.

Code in one block - can't interact with other block. in other function.

* Local scope - once exit from function - destroyed
* Exception: Static.

→ All functions have file scope

function arguments

```
int fs_in (char *s, char c)
{
   :
}
```

Character pointer & a character.

* Call by value, Reference:

* call by value - Just pass the copy of the value as argument
  → changes made to arguments - don't affect actual variables
* call by reference - pass address. - anything changed will be changed.

'call by value' -] see hub (Stanford)
call by reference -

C++ → allows to fully automate a call by reference through reference parameters - not supported by C.

Calling functions with arrays

* Exception to call by value - parameter passing
* when array is used as func. argument - reference is passed.
* Exception!

Let's see - Upper case!

A → 65	a → 97
Z → 90	z → 122

65 + 32

If I want - unchanged

* Don't alter!

```
{
for (i=0; string[i]; ++i)
    putchar (toupper (string[i]));
}
```

↳ Just returns (Inbuilt) - not alters.

Recreate - toupper() / gets()

Say: xgets()

`char *xgets(char *s)` → i/p string (fgets), returns pointer!

'\b' → backspace (go to previous index)

---

argc and argv - Arguments to main()

* Sometimes - useful to pass info into a program - running it.
* pass info to main() → via Cmd.

cc program_name

cmd argument specifying name of program

argc → no. of arg passed

argv → pointer to an array of character pointers.

gcc -o out argc_argv.c

./out hello	./out hello, hi	./out hello hi
Hello! hello	Hello! hello, hi	Forgot your name.
② argc	② argc	③ argc

why?

, (Comma) — not a legal seperation

Hello Tom → 2 strings.
run Spot, run → 2 strings
run Spot, run → 3 strings

Heart, Stand, hello → 1 String.

'hello hi how' → single string.

Format

char * argv[]
↳ 'undetermined length'

argv[0] → first string

hello_argv.c

gcc -o out hello_argv.c
./out Ram kumar prem

Hello Ram
Hello kumar
Hello Prem

argv

(gn)

char * argv[]

argv[0] → ./out [name] of program
argv[1] → Ram
argv[2] → kumar
argv[3] → prem