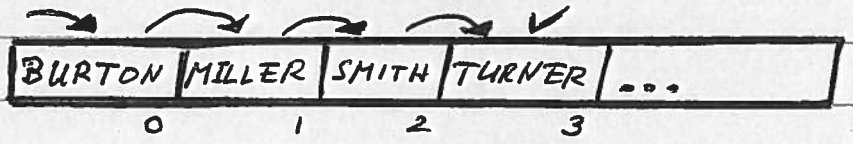
CH. 12

W7

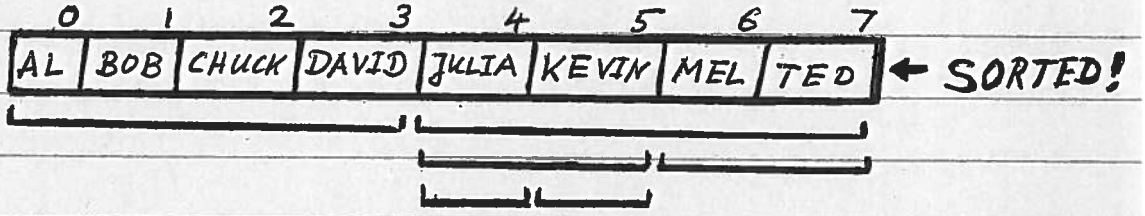
SEARCHING & SORTING

- **SEARCHING - EX:** "Find element in an array"

(i) Find "Turner" — LINEAR SEARCH:



(ii) Find "Kevin" — BINARY SEARCH:



here: $n = 8 \rightarrow n/2 = 4$

4: JULIA < KEVIN

→ Consider 4, 5, 6, 7

6: MEL > KEVIN

→ Consider 4, 5

5: KEVIN == KEVIN → found

- **EX:**

0	2
1	7
2	(9)
3	12
	...

Here: key: 9
$n = 4$, $i = 2$

```
/* LINEAR SEARCH */
int IndexOfIntInArray
    (int key, int arr[], int n)
...
for (i=0; i<n; i++)
{ if (key == arr[i]) return(i);
}
return(-1);
...
```

• **EX.:** BINARY SEARCH (ordered array of cities)

0	ATL
①	**BOS**
2	CHI
3	DEN
4	DET
5	HOU
6	LOA
7	MIA
8	NYC
9	PHI
10	SFR
11	SEA

Here:

```
string ciArr [MaxCi];

Key = "BOS";
```

Index ranges (I1, I2) for binary search:

I1	0	0	0	1	
I2	11	4	1	1	
mid	5	2	0	①	
ciArr[mid]	HOU	CHI	ATL	**BOS**	

↑ done & found

```
mid = (I1+I2)/2;
```

I1=0, I2=11, mid=5, BOS < HOU
" , I2=4, mid=2, BOS < CHI
" , I2=1, mid=0, BOS > ATL
I1=1, " mid=1, BOS == BOS ← done & found

• **Complexity:** (e.g., 16 elements)

- compare with mid of 16 elem.
- " " " " 8 "
- " " " " 4 "
- " " " " 2 "
- " " " " 1 "

no. comparisons: 5

$\log_2 16 + 1 = 4 + 1 = 5$

```
static int FindCity (string key, string arr[], int n)
{ int I1, I2, mid, cmp;

  I1=0;
  I2 = n-1;   /* n = no. of elements */
  while (I1<=I2)
  { mid = (I1+I2)/2;
    cmp = StringCompare (key, arr[mid]);
    if (cmp==0) return (mid);        /*FOUND*/
    if (cmp < 0)
    { ind2 = mid-1;
    }
    else
    { ind1 = mid+1;
    }
  }/* end while */
  return (-1);                       /*NOT FOUND*/
}
```

• **Complexity - Efficiency:** (n = no. of array elements)

n	2^0	2^1	2^2	2^3	2^4	2^5	2^{10}	2^{20}	...
* LIN. SEARCH	1	2	4	8	16	32	1024	1048576	...
* BIN. SEARCH	1	2	3	4	5	6	11	21	...

↑ Ratio of ≈ 50000

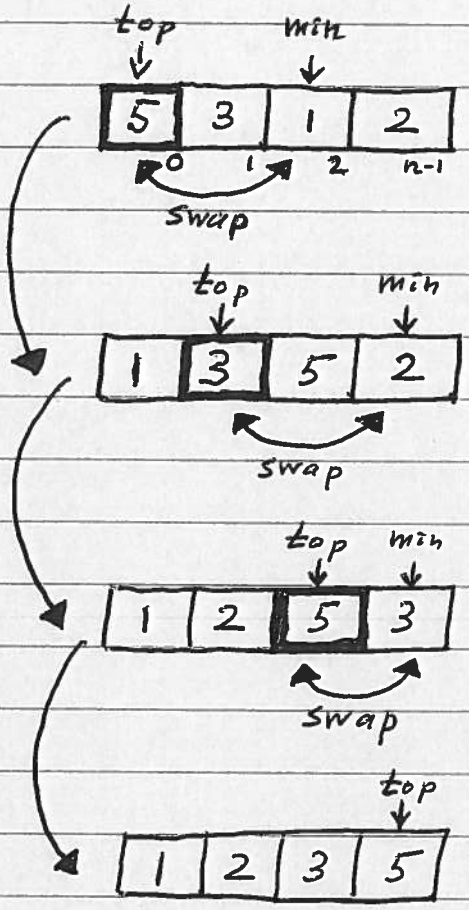
* Max. no. of comparisons done

$O(n)$ vs. $O(\log n)$

➡ Use BINARY search for large data sets!
BUT: Must SORT/ORDER data first...

• SORTING – Selection Sort:

Idea: Place smallest element/datum at top of remaining unordered sub-array



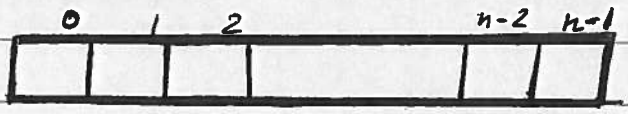
done.

```
static void SelectionSort
            (int arr[], int n)
{ int top, min;

  for (top=0; top<(n-1); top++)
  { min = FindMinInt
          (arr, top, n-1);
    /* min is index of min */
    /* int in arr [top...n-1]*/
    if (min != top)
    { SwapElem (arr, min, top);
    }
  } /* end for */
}
```

...

• COMPLEXITY:



n elements

(i) Max. no. of SWAPS: $n-1$

(ii) No. of comparisons:

top = 0	→	n-1	comp.
= 1	→	n-2	"
= 2	→	n-3	"
...		...	
= n-2	→	1	"
= n-1	→	0	"

→ $\frac{1}{2} n (n-1) = \frac{1}{2} (n^2 - n)$ comparisons

$O(n^2)$ complexity

⟵|

CH. 13

W7

POINTERS

• **DEF.:** **Pointer** = data item whose value is the **address** of another data item.

EX:

```
int x, y;

int *p1, *p2;
```

(indicating "pointer")

address:

1000	-42	x
1004	163	y
1008	1000	p1
1012	1004	p2

```
x = -42;
y = 163;

p1 = &x;
p2 = &y;
```

("address of")

• **Declaration:**

```
int *p;      /* "*p" is pointer */
             /* to an integer   */

char *cp;    /* "cp" is pointer */
             /* to a character  */
```

• **Operations:**

"&" – address-of operator: returns address of a variable

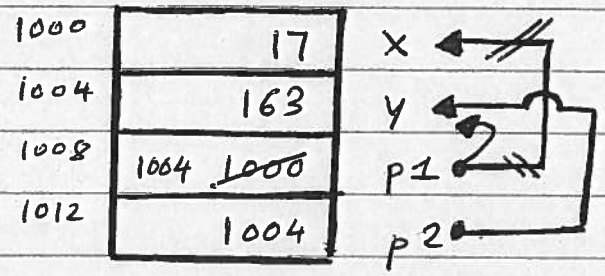
"*" – value-pointed-to operator: returns the value to which a pointer points

(above example:

```
int z;
z = x;
z = *p1;
```

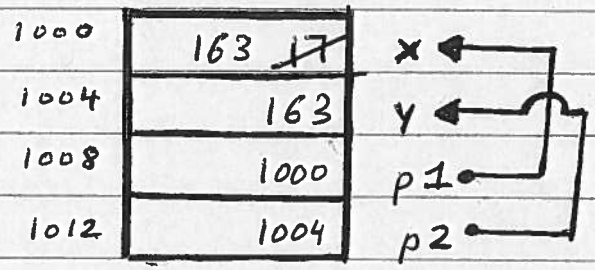
} equivalent!)

- **Pointer assignment:**



p1 = p2;

- **Value assignment:**



*p1 = *p2;

- **NULL pointer:** Pointer not pointing to any valid datum

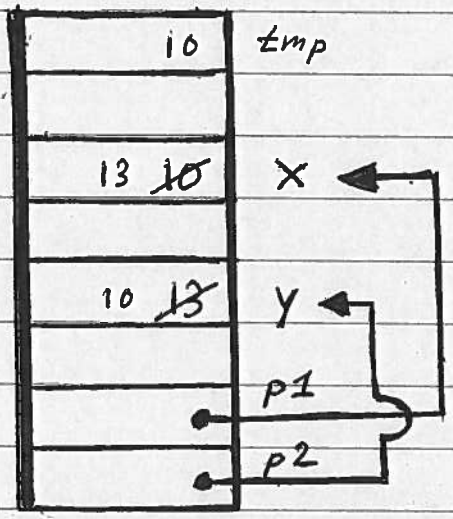
! - **CALL-BY-REFERENCE:**

> Passing a pointer to a function enables manipulation of data in main memory (inside the calling function)

EX:
```
void SetToZero (int *p)
{
  *p = 0;
}
```
↑ "Call-by-reference"

CALL: "SetToZero (&x);"

EX: Swapping values




```
void SwapInt
     (int *p1, int *p2)
{ int tmp;

  tmp = *p1;
  *p1 = *p2;
  *p2 = tmp;
}
```

```
SwapInt(&x, &y);   /* swapping values of x and y */
```

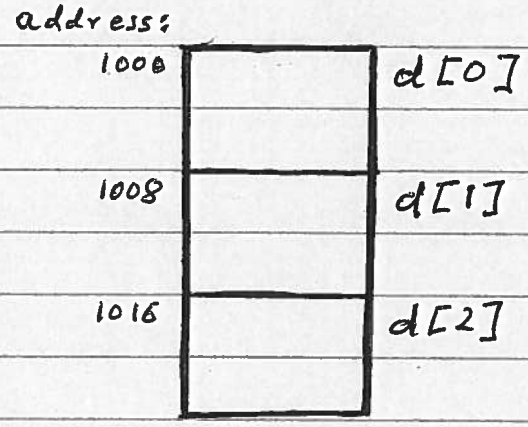
EX: Manipulation of MULTIPLE data via ONE function call using call-by-reference

```
/* input: 125 min, output: 2 h 5 min */

void MinToHourAndMin
     (int min, int *phour, int *pmin)

{ *phour = min / 60;
  *pmin = min % 60;
}
```

• EX: Pointers and arrays

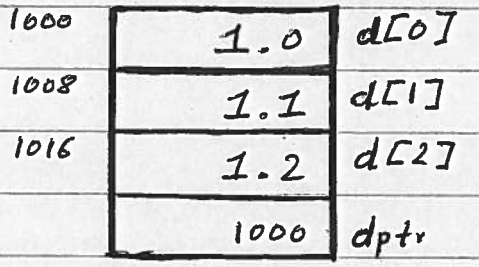


double d[3];

&d[1] is 1008.

&d[i] is starting byte address of i^{th} element

• POINTER ARITHMETIC (+ and − for pointers)



double d[3];
double *dptr;

dptr = &d[0];

ex: dptr+2 is address of d[2] (== 1016)

! ex: *dptr++; /* pointer now */
/* points to d[1], */
/* if it pointed to */
/* d[0] before. */
/* see p. 476: *(dptr++) */

ex: dptr−1 is...

ex: double *p1, *p2;

p2 = &d[2]; /*1016*/
p1 = &d[0]; /*1000*/

p2 - p1 yields 2

"d[0] and d[2] are "2 array cells apart" from each other."

• DYNAMIC MEMORY ALLOCATION

→ STATIC Alloc.: Compiler assigns fixed locations in memory for all variables.

→ DYNAMIC Alloc.: New memory is assigned during program's execution!

→ In strlib.h: "malloc"

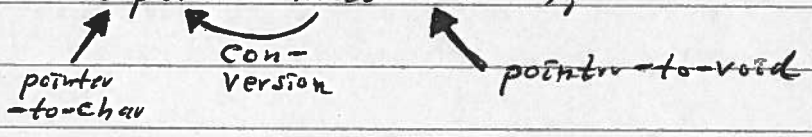
/*allocates block of mem. of spec. size*/

ex: malloc(10)

- returns pointer to 10 consecutive bytes of mem.
- result must be assigned to a pointer variable!

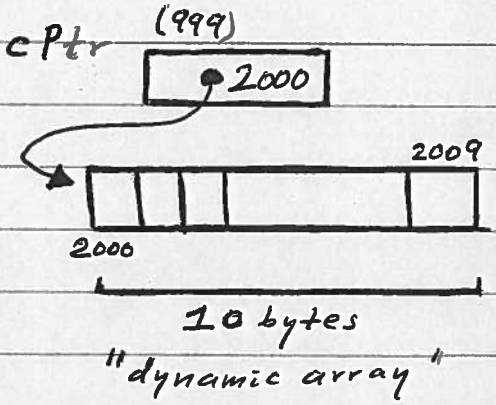
- malloc returns pointer to type "void", 'pointer-to-void';
  C prforms necessary conversion

ex: `char *cptr;`
`cptr = malloc(10);`



• Dynamic arrays

EX1: `char *cPtr;`
`cPtr = malloc(10);`
`/* 1 char ≙ 1 byte */`



EX2:
```
int *intArr;
intArr = malloc(10 * sizeof(int));

/* intArr can be of any size as needed */
/* by a program and its input and state. */
free(intArr); /* "frees" memory when no longer needed */
```

! EX3: RUNNING OUT OF MEMORY

```
intArr = malloc(10 * sizeof(int));
if (intArr == NULL) Error("Out of memory!\n");
```