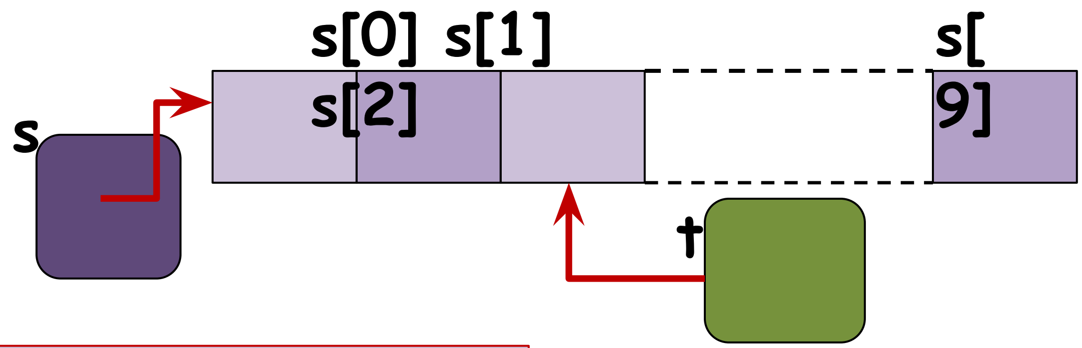
Arrays and Functions

This Class

- Arrays with functions

  (Non-recursive and recursive)

Address Arithmetic




```
int main()  {
  char s[10];
  read_into_array(s+2,8); }
```

```
int read_into_array
        (char t[], int
size);
```

s+2 points to s[2], or,
s+2 is a pointer to s[2].

Passing an actual parameter array s+2 to a formal parameter array t[] makes t now point to the third element of array s.

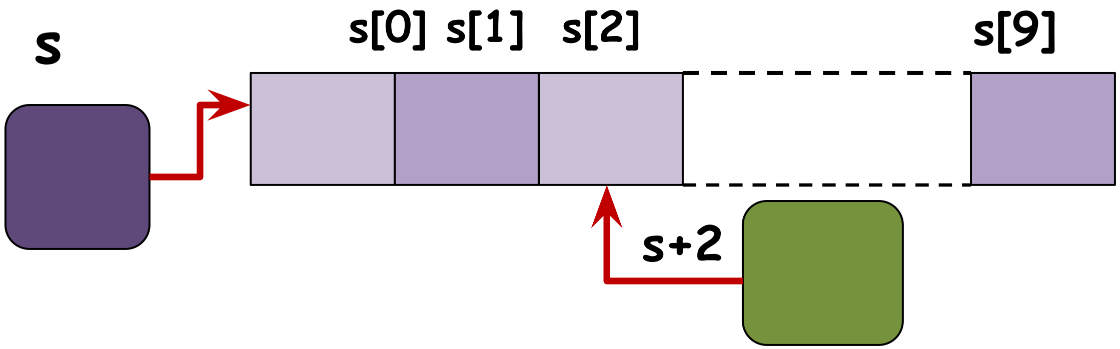
t is declared as char t[], t[0]=s[2] is the box pointed to by t, t[1]=s[3] refers to the box one char further from the box t[0], and so on…

Dereferencing Operators

```
int main()  {
  char s[10];
  read_into_array(s,10);
  ..........
```

- For an array []

  acts as a dereferencing operator.



- Another such operator is * .
  - ✔ Can act on an array address.
- Eg. s[2] is the same as *(s+2).

Example: Dot Product

- Problem: write a function dot_product that takes as argument two integer arrays, a and b, and an integer, size, and computes the dot product of first size elements of a and b.
- Declaration of dot_product

int dot_product(int a[], int b[], int);

OR

int dot_product(int [], int [], int);

```c
#include<stdio.h>
int dot_product (int[], int[], int);
int main(){
  int vec1[] = {2,4,1,7,-5,0, 3, 1};
  int vec2[] = {5,7,1,0,-3,8,-1,-2};
  printf("%d\n", dot_product(vec1, vec1, 8));
  printf("%d\n", dot_product(vec1, vec2, 8));
  return 0;
}
int dot_product (int a[], int b[], int size){
  int p = 0, i;
  for(i=0;i<size; i++)
      p = p + (a[i]*b[i]);
  return p;
}
```

OUTPUT
105
49

Example: Search with Recursion

search(a,n,key)

Base case: If n is 0, then, return **0**.

Otherwise: /* n > 0 */

1. compare last item, a[n-1], with key.
2. if a[n-1] == key, return **1**.
3. search in array a, up to size n-1.
4. return the result of this “smaller” search.

a

search(a,10,3)

31	4	10	35	59	31	3	25	35	11

Either 3 is a[9]; or search(a,10,3) is same as the result of search for 3 in the array starting at a and of size 9.

```c
int search(int a[], int n, int key) {
    if (n==0) return 0;
    if (a[n-1] == key)  return 1;
    return search(a,n-1,key);
}
```

Let us do a quick trace.

a

E.g., (0) search(a,5,10)

31	4	10	35	59

a[4] is 59, not 10. call search(a,4,10)

a

(2) search(a,3,10)

31	4	10	35	59

a[2] is 10, return 1

a

(1) search(a,4,10)

31	4	10	35	59

a[3] is 35, calls search(a,3,10)

```
int search(int a[], int n, int key) {
    if (n==0) return 0;
    if (a[n-1] == key)  return 1;
    return search(a,n-1,key);
}
```

Let us do another quick trace.

E.g., (0) search(a,5,3)

a

31	4	10	35	59

a[4] is 59, not 3. call search(a,4,3)

(1) search(a,4,3)

a

31	4	10	35	59

a[3] is 35, calls search(a,3,3)

(2) search(a,3,3)

a

31	4	10	35	59

a[2] is 10, calls search(a,2,3)

(3) search(a,2,3)

a

31	4	10	35	59

a[1] is 4, calls search(a,1,3)

(4) search(a,1,3)

a

31	4	10	35	59

a[0] is 31, calls search(a,0,3)

31	4	10	35	59

a

(5) search(a,0,3)

returns 0

```
1.   int search(int a[], int n, int key) {
2.       if (n==0) return 0;
3.       if (a[n-1] == key)  return 1;
4.       return search(a,n-1,key);
5.   }
```

a

search(a, 5,3)

31	4	10
35	59	

tack

function	called by	return address	return value
search(a,5,3)	main()	---	
search(a,4,3)	search(a,5,3)	search.5	
search(a,3,3)	search(a,4,3)	search.4	
search(a,2,3)	search(a,3,3)	search.3	
search(a,1,3)	search(a,2,3)	search.2	
search(a,0,3)	search(a,1,3)	search.1	

recursion

A state of

```
1.   int search(int a[], int n, int key) {
2.       if (n==0) return 0;
3.       if (a[n-1] == key)  return 1;
4.       return search(a,n-1,key);
5.   }
```

a

search(a,5,3)

31	4	10
35	59	

tack

function	called by	return address	return value
search(a,5,3)	main()	---	
search(a,4,3)	search(a,5,3)	search.5	
search(a,3,3)	search(a,4,3)	search.4	
search(a,2,3)	search(a,3,3)	search.3	
search(a,1,3)	search(a,2,3)	search.2	
search(a,0,3)	search(a,1,3)	search.1	0

recursion

A state of the stack

```
1.   int search(int a[], int n, int key) {
2.       if (n==0) return 0;
3.       if (a[n-1] == key)  return 1;
4.       return search(a,n-1,key);
5.   }
```

a

search(a, 5,3)

31	4	10
35	59	

tack

function	called by	return address	return value
search(a,5,3)	main()	---	
search(a,4,3)	search(a,5,3)	search.5	
search(a,3,3)	search(a,4,3)	search.4	
search(a,2,3)	search(a,3,3)	search.3	
search(a,1,3)	search(a,2,3)	search.2	0

state of the stack

```
1.   int search(int a[], int n, int key) {
2.       if (n==0) return 0;
3.       if (a[n-1] == key)  return 1;
4.       return search(a,n-1,key);
5.   }
```

a

search(a, 5,3)

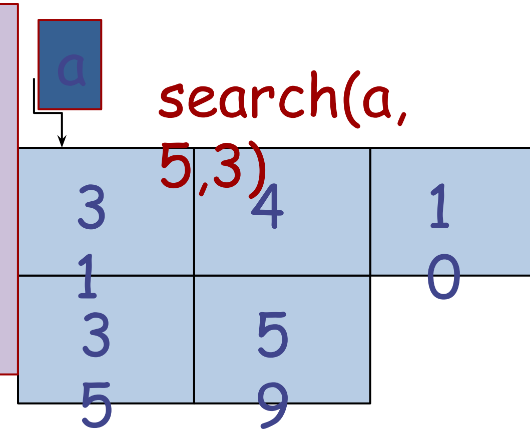
31	4	10
35	59	

tack

function	called by	return address	return value
search(a,5,3)	main()	---	
search(a,4,3)	search(a,5,3)	search.5	
search(a,3,3)	search(a,4,3)	search.4	
search(a,2,3)	search(a,3,3)	search.3	0

A state of the stack

```
1.  int search(int a[], int n, int key) {
2.      if (n==0) return 0;
3.      if (a[n-1] == key)  return 1;
4.      return search(a,n-1,key);
5.  }
```



Stac

function	called by	return address	return value
search(a,5,3)	main()	---	
search(a,4,3)	search(a,5,3)	search.5	0

A state of the stack

```
int search(int a[], int n, int key) {
    if (n==0) return 0;
    if (a[n-1] == key)  return 1;
    return search(a,n-1,key);
}
```

a

search(a, 5,3)

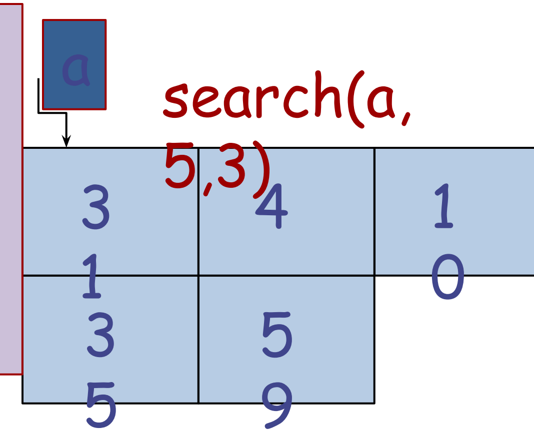
31	4	10
35	59	

Stac

function	called by	return address	return value
search(a,5,3)	**main()**	---	0

A state of the stack

```c
int search(int a[], int n, int key) {
    if (n==0) return 0;
    if (a[n-1] == key)  return 1;
    return search(a,n-1,key);
}
```



search(a,5,3) returns 0. Recursion call stack terminates.

```
int search(int a[], int n, int key) {
    if (n==0) return 0;
    if (a[n-1] == key)  return 1;
    return search(a,n-1,key);
}
```

search(a, 5,3)

31	4	10
35	59	

search(a,5,3) returns 0. Recursion call stack terminates.

Searching in an Array

- We can have other recursive formulations
- Search1: search (a, start, end, key)
  - Search key between a[start]…a[end]

if start > end, return 0;

if a[start] == key, return 1;

Searching in an Array

- One more recursive formulation
- Search2: search (a, start, end, key)
  - Search key between a[start]…a[end]

if start > end, return 0;

mid = (start + end)/2 ;

if a[mid]==key, return 1;

return search(a, start, mid-1, key)

Time Cost

- Two types of operations
  - Function calls
  - Other operations (call them simple operations)
- Assume each simple operation takes fixed amount of time (1 unit) to execute
  - Really a very crude assumption, but will simplify calculations
-

Time Cost

```
1. if start > end, return 0;
2. if a[start] == key, return 1;
3. return search(a, start+1, end, key);
```

- Search1
  - Let $T(n)$ denote the time taken by search on an array of size n.
  - Line 1 takes 1 unit (or 2 units if you consider if check and return as two operations)
  - Line 2 takes 1 unit (or 3 units if you consider if check, array access and return as three operations)
  - But what about line 3?

Time Cost

```
1. if start > end, return 0;
2. if a[start] == key, return 1;
3. return search(a, start+1, end, key);
```

- Search1
  - What about line 3?
  - Remember the assumption: *Let T(n) denote the time taken by search on an array of size n.*
  - Line 3 is searching in n-1 sized array => takes T(n-1) units
  -

Time Cost

```
1. if start > end, return 0;
2. if a[start] == key, return 1;
3. return search(a, start+1, end, key);
```

- Search1
  - But what about the value of T(n) ?
  - Looking at the body of search, and the information we gathered on previous slides, we can come up with a recurrence relation:
  -

$$T(n) = T(n-1) + C$$

$$T(n-1)$$

Time Cost

- Search1
  - Solution to the recurrence?

$$T(n) = T(n-1) + C, \quad T(1) = C$$

$$T(n) = Cn$$

  - The wor l to the size of array
    - Bigger the array, slower the search
  - What is the best case run time?
  - Which one is more important to consider?

Time Cost

- Search1
  - Solution to the recurrence?

$$T(n) = T(n-1) + C\ ,\ T(0) = C$$

$$
\begin{aligned}
T(n) &= T(n-2) + C + C = T(n-2) + 2C \\
&= T(n-3) + 3C \\
&\dots \\
&= T(n-(n-1)) + (n-1)C \\
&= T(1) + (n-1)C
\end{aligned}
$$

$$T(n) = Cn$$

Time Cost

```
if start > end, return 0;
mid = (start + end)/2 ;
if a[mid]==key, return 1;
return search(a, start, mid-1, key)
         || search(a, mid+1, end, key);
```

- Search2
  - Recurrence?

$$T(n)\ T(n/2) + T(n/2) + C$$

  - Solution?

$$T(n)\ \ n$$

  - The worst case run time of Search2 is also proportional to the size of array
    - Can we do better?

Time Cost

- Search2
  - Solution to the recurrence?

$$T(n) <= T(n/2) + T(n/2) + C,$$
$$T(1) = C$$

$T(n) <= 2\ T(n/2) + C$

$<= 4\ T(n/4) + 2C$

$<= n\ T(1) + (n/2)\ C$

$= nC + (n/2)C$

$= (3/2)Cn$

$$T(n) \approx n$$