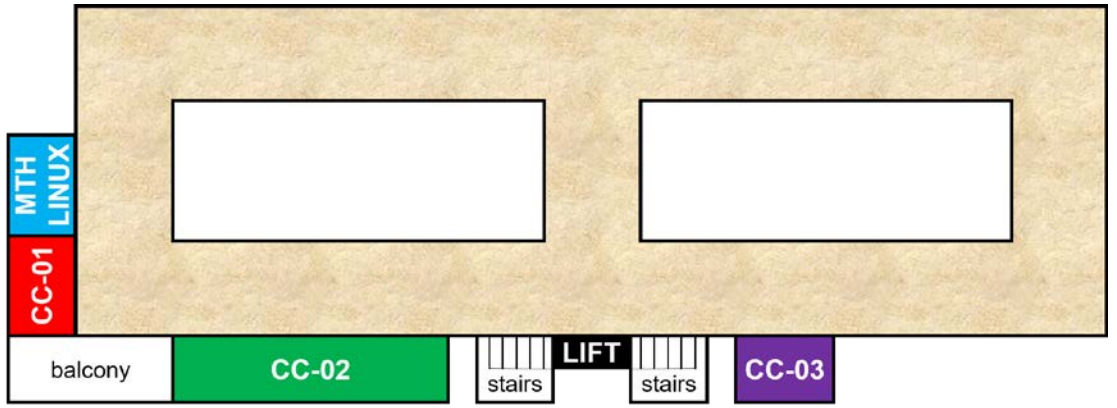
Arrays and functions

ESC101: Fundamentals of Computing

Nisheeth

Mid-sem Lab Exam: February 15 (Saturday)

- Morning exam
  - 10:00 AM - 12:30 PM – starts 10:00 AM sharp
  - **CC-01**: A9, {A14 even roll numbers}
  - **CC-02**: A7, A10, A11
  - **CC-03**: A12
  - **MATH-LINUX**: A8, {A14 odd roll numbers}
- Afternoon exam
  - 12:45 PM – 3:15 PM – starts 12:45 PM sharp
  - **CC-01**: A1, {A2 even roll numbers}
  - **CC-02**: A4, A5, A6
  - **CC-03**: A3
  - **MATH-LINUX**: A13, {A2 odd roll numbers}



Mid-sem Lab Exam: February 15 (Saturday)

- Go see your room during this week's lab
- Be there 15 minutes before your exam time
  - No entry for candidates arriving later than 09:45 for morning exam and 12:30 pm for the afternoon exam
- Cannot switch to another session (morning to afternoon or vice-versa)
- Syllabus – till functions (no arrays)
- Open handwritten notes – However, **NO** printouts, photocopies, slides, websites, mobile phone or tablet
- **POSSESSING ANY OF THESE WILL BE CONSIDERED CHEATING**
- Prutor CodeBook will be unavailable during lab exam
- Exam will be like labs - marks for passing test cases
- Marks for writing clean indented code, proper variable names, a few comments – illegible code = poor marks

Recap: Passing by value

```
// swapping a and b
void swap(int a, int b){
  int temp;
  temp = a;
  a = b;
  b = temp;
  printf("a=%d b=%d\n", a, b);
}
int main(){
  int  a=10, b=15;
  printf("a=%d b=%d\n", a, b);
  swap(a, b);
  printf("a=%d b=%d\n", a, b);
  return 0;
}
```

What is the output of the program?
(fill the blanks)

OUTPUT

a=10 b=15

a=15 b=10

a=____ b=____

Recap: Parameter passing

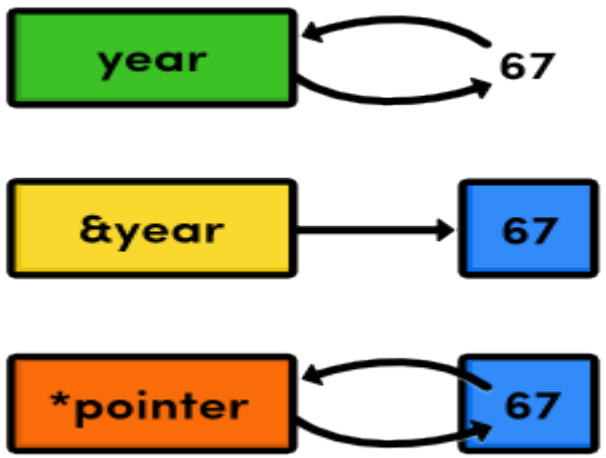
Basic steps:

1. Create new variables (boxes) for each of the formal parameters allocated on a fresh stack area created for this function call.
2. Copy values from actual parameters to the newly created formal parameters.
3. Create new variables (boxes) for each local variable in the called procedure. Initialize them as given.

Today, we will look at parameter passing more carefully. Pay attention!

Values and addresses

- Pointers are special variables that store memory addresses
- We will cover pointers in much greater depth soon



A variable transparently stores a value with no notion of memory addresses.

The reference operator returns the memory address of a variable.

The dereference operator accesses the value stored in a memory address.

Argument passing by value and reference

```
#include <stdio.h>

long sum(long a, long b){
        printf("%ld\t%ld\n", &a, &b);
        printf("%ld\t%ld\n", a, b);
        return a+b;
}
int main(){
        long a=2,b=3;
        printf("%ld\t%ld\n", a, b);
        printf("%ld\t%ld\n", &a, &b);
        printf("%ld\n",sum(a,b));
        printf("%ld\n",sum(&a,&b));
        return 0;
}
```

Output:

```
2               3
140732008792672  140732008792664
140732008792616  140732008792608
2               3
5
140732008792616  140732008792608
140732008792672  140732008792664
281464017585336
```

Passing by reference

- Telling compiler you will be passing a memory address, not a value
- Pass address using reference operator (&) during function call
  - So far, we have thought of variables x as values
  - More accurate to think of x as ‘the value stored at x’
  - &x is the memory address of x

Passing arrays by value

- Can pass array elements to functions
  - Treated like normal variables
- This is passing an array *by value*
- We are passing the values stored in the array to a function
- What else could we be passing?

```
#include <stdio.h>
void shift( char ch) {
        printf("%c ", ch+4);
}

int main() {
        char arr[] = {'a', 'b', 'c'};
        for (int x=0; x<3; x++) {
                shift (arr[x]);
        }
        return 0;
}
```

Passing arrays by reference

Write a function that reads input into an array of characters until EOF is seen or array is full.

```
int read_into_array
        (char t[], int size);
/* returns number of chars
   read  */
```

read_into_array:

- array t (arg.)
- size of the array (arg.)
- reads the input in array

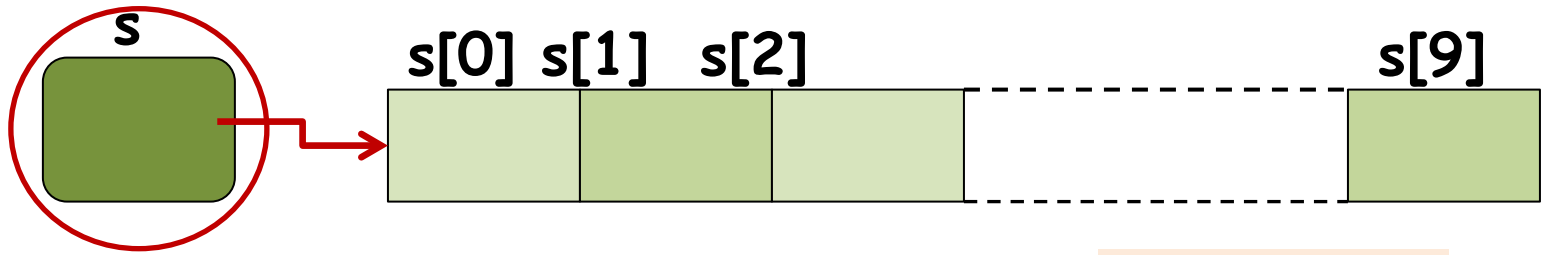
```
int main()  {
  char s[100];
  read_into_array(s,100);
  /* process */
}
```

```
int read_into_array
        (char t[], int size) {
    int ch;
    int count = 0;
    ch = getchar();
    while (count < size
            && ch != EOF) {
            t[count] = ch;
            count = count + 1;
        ch = getchar();
    }
    return count;
}
```

But what's the point of this code? Counting inputs?

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

```
int read_into_array
        (char t[], int size) {
     int ch;
     int count = 0;
     /* …   */
}
```



Array variables store address!!

s is an array. It is a variable and it has a box.

The value of this box is the address of the first element of the array.

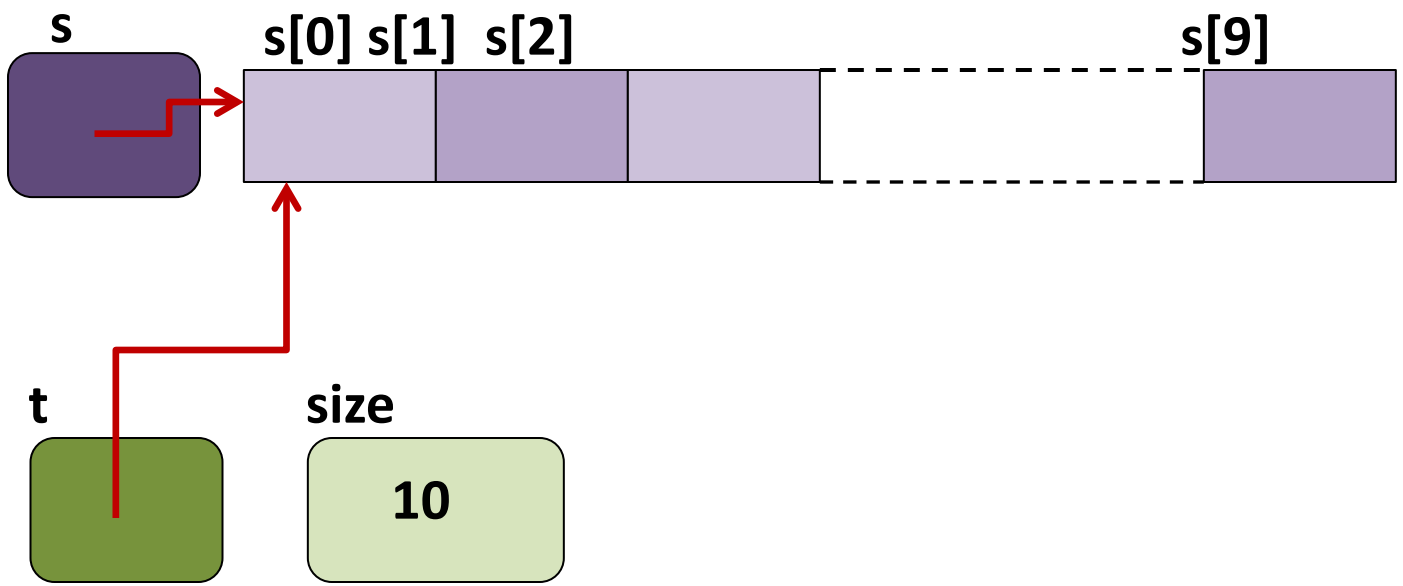
The stack of main just prior to call

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

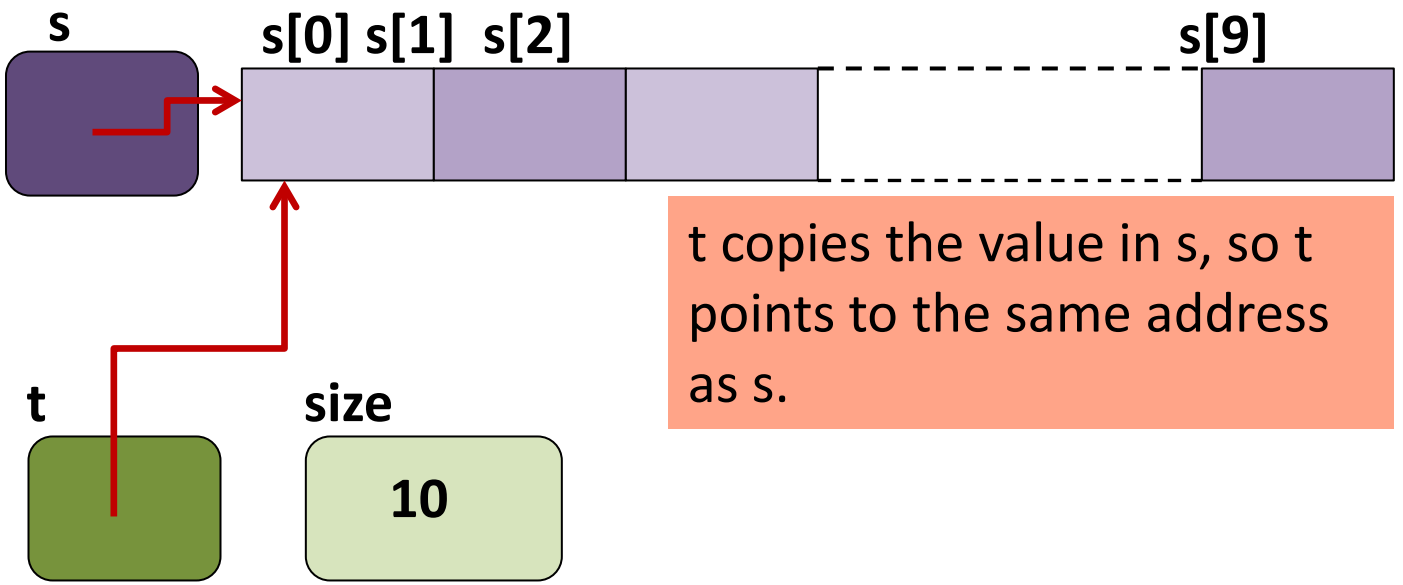
Create new variables (boxes) for each of the formal parameters allocated on a fresh stack created for this function call.

```
int read_into_array
       (char t[], int size) {
    int ch;
    int count = 0;
    /* …   */
}
```

Copy values from actual parameters to the newly created formal parameters.



Parameter Passing: Arrays

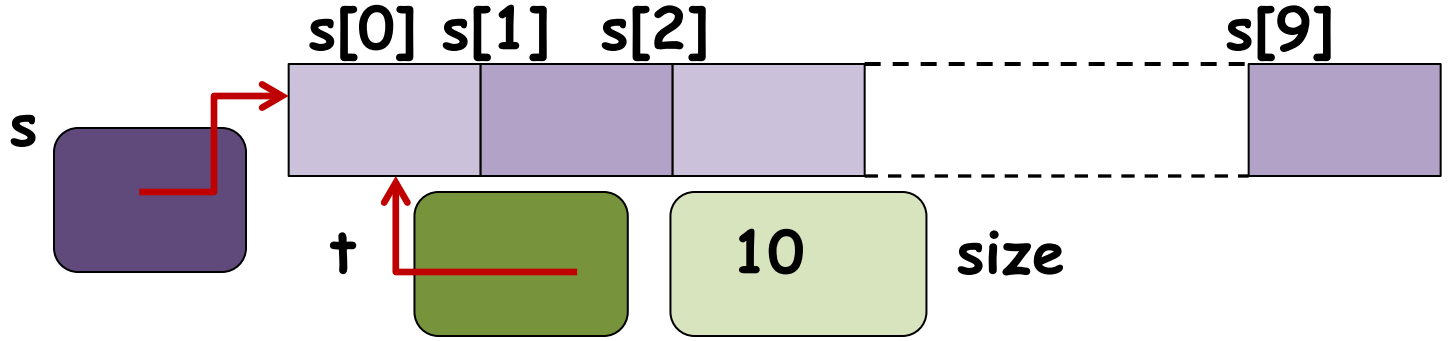


t copies the value in s, so t points to the same address as s.

s and t are the same array now, with two different names!

s[0] and t[0] refer to the same variable.

Implications of copying content of array variable during parameter passing



An array (s) is identified with a box whose value is the address of the first element of the array.

The value of s is copied into t.
Value in the box of t
=
Value in the box of s.

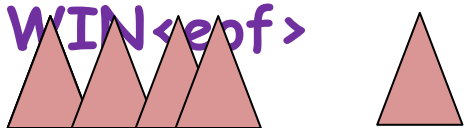
1. In the computer, an address is simply the value of a memory location.
2. The value in the box for s would be the memory location of s[0].

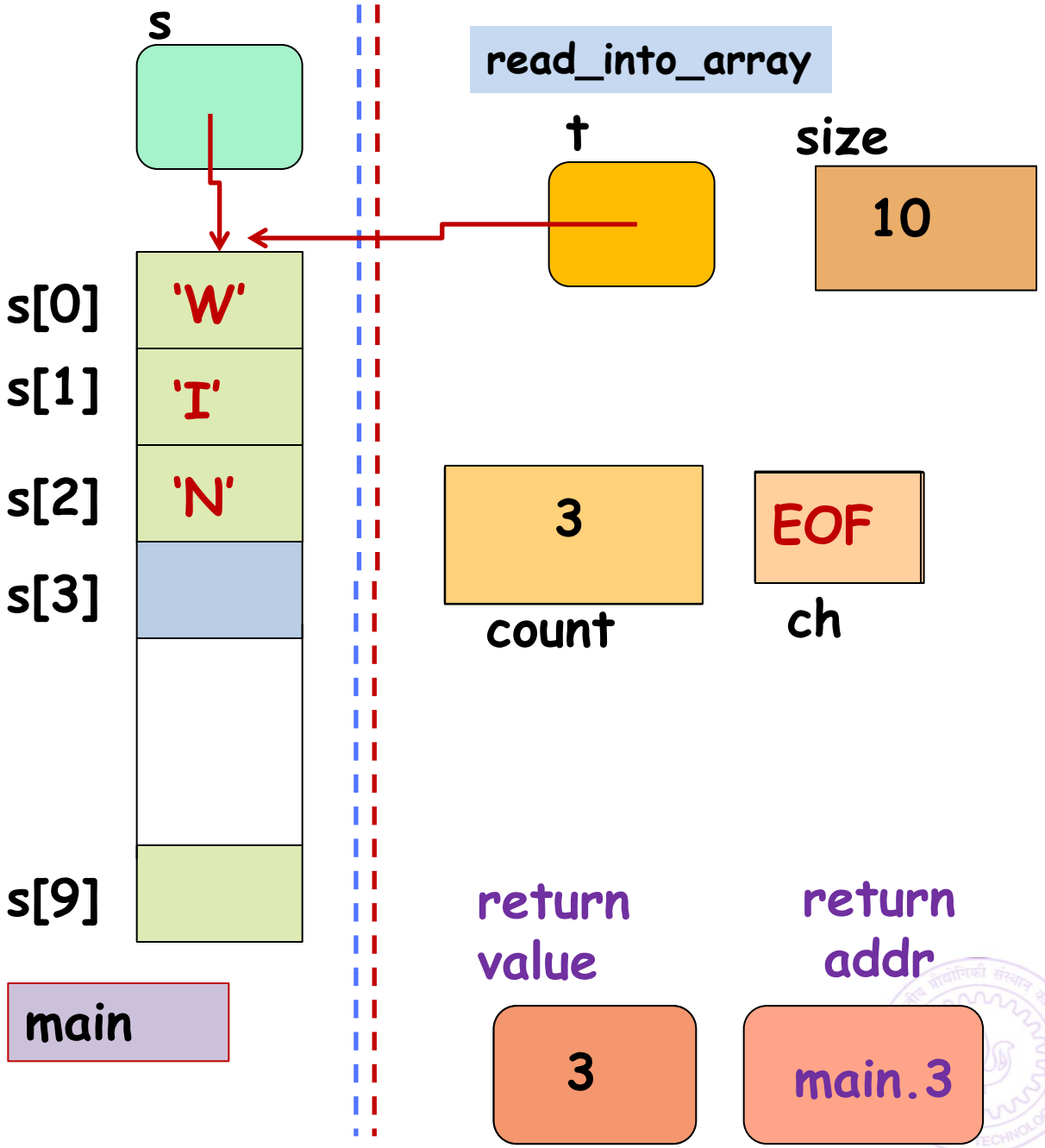
They both now contain the address of the first element of the array.

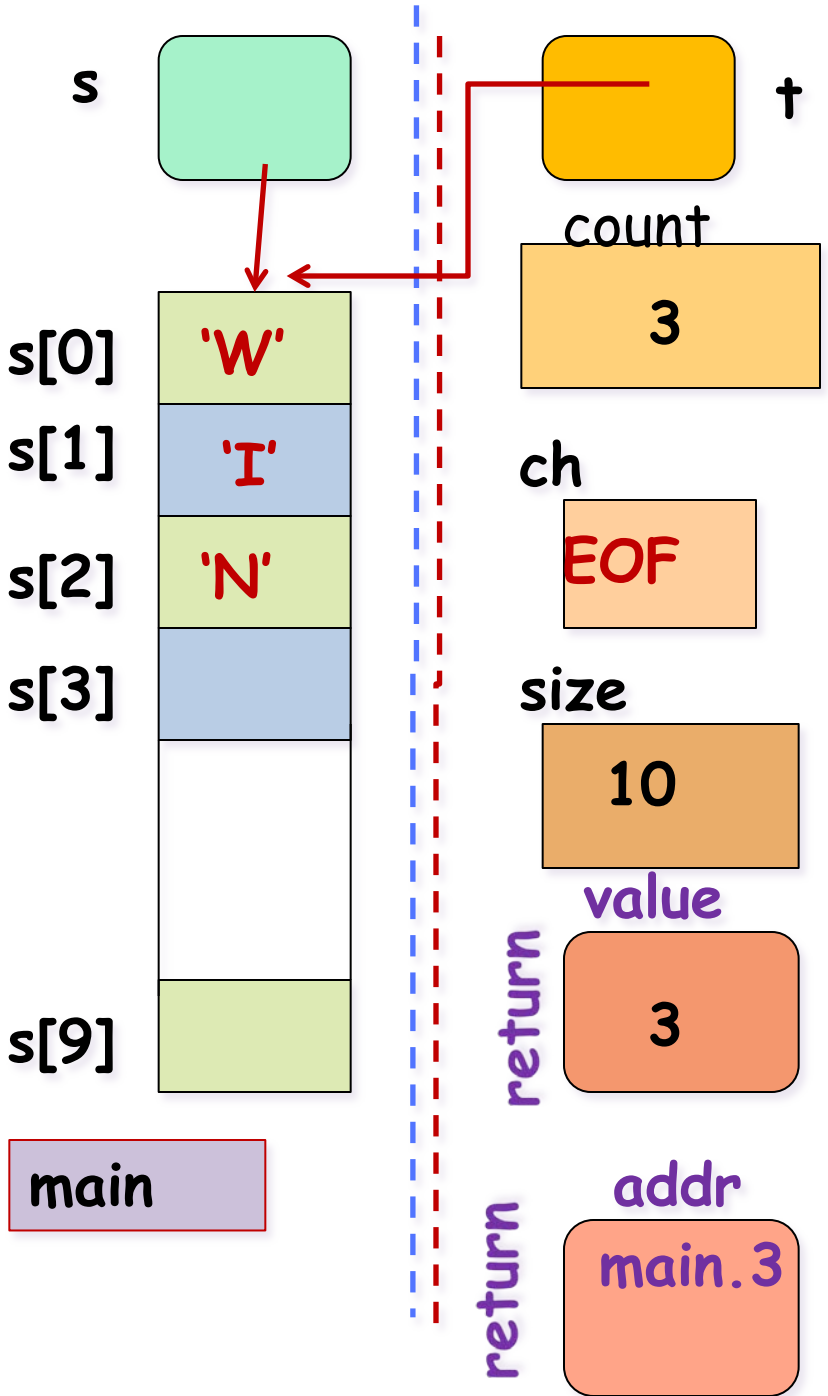
```
int read_into_array
            (char t[], int size) {
        ch;
    int count = 0;
        = getchar();

    while (count < size
                && ch != EOF) {
        t[count] = ch;
        count = count + 1;
        ch = getchar();
        turn count;
}
```

```
int main()  {
    char s[10];
    ad_into_array(s,10);
    …
```

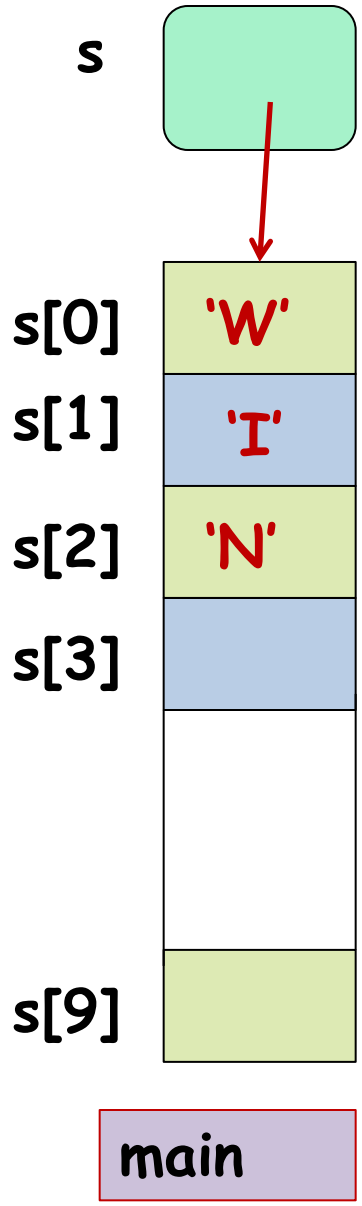
Input WIN<eof>







State of memory just prior to returning from the call read_into_array()



State of memory just after returning from the call read_into_array().

All local variables allocated for read_into_array() on stack may be assumed to be erased/de-allocated.

Only the stack for main() remains, that is, all local variables for main() remain.

Behold !!

The array s[] of main() has changed!

THIS DID NOT HAPPEN BEFORE!
WHAT DID WE DO DIFFERENTLY?

Ans: we passed the array s[] by reference

Next Class

- After the mid-sem
- We will talk about arrays and functions some more