ESC101: Introduction to Computing

f(unction)

Midsem

- Saturday 17th September, 13:00 - 15:00
- L1, L2, L7, L16, L17 (OROS)
- Specific seat number for each student, check list on canvas
- No mobile phones, paper allowed during exam
- Syllabus: All topics covered so far including today's class

- Good luck for the exam!

Summary

- Global Variable
  - Visible everywhere
  - Lives everywhere (never destroyed)
- Local Variable
  - Visible in scope
  - Lives in scope (destroyed at the point where we leave the scope)
- Static Variable
  - Visible in Scope
  - Lives everywhere! (but can not be accessed outside scope)

Question

```
// 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;
}
```

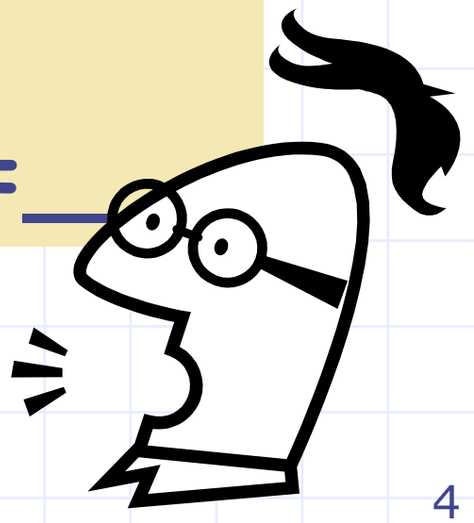
What is the output of the program? (fill the blanks)

OUTPUT

a=__10__ b=__15__

a=__15__ b=__10__

a=___ b=___



Passing arrays as parameters

f(unction[])

Argument Passing: Array vs Simple Type

- When a basic datatype (such as int, char, float, etc) is passed to a function
  - a copy of the value is created in the memory space for that function,
  - after the function completes its execution, these values are lost.
- When an array is passed to a function
  - the address of the first element is copied,
  - any changes to the array elements are visible to the caller of the function.

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);
```

```
#include<stdio.h>
int dot_product (____, ____, 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 size){

}
```

$$p = \sum_{i=1}^{size} (a_i \times b_i)$$

Convert to C

OUTPUT

105

49

```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){
```

$$p = \sum_{i=1}^{size} (a_i \times b_i)$$

Convert to C

```c
}
```

OUTPUT

105

49

```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;
}
```

```c
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

```
#include <stdio.h>
int main() {
      char s[100];
      int count = 0;
      int ch;
      int i;
```

/*read_into_array */

```
while ( (ch=getchar()) != EOF &&
                        count < 100 )
{       s[count] = ch;
        count = count + 1;
}
```

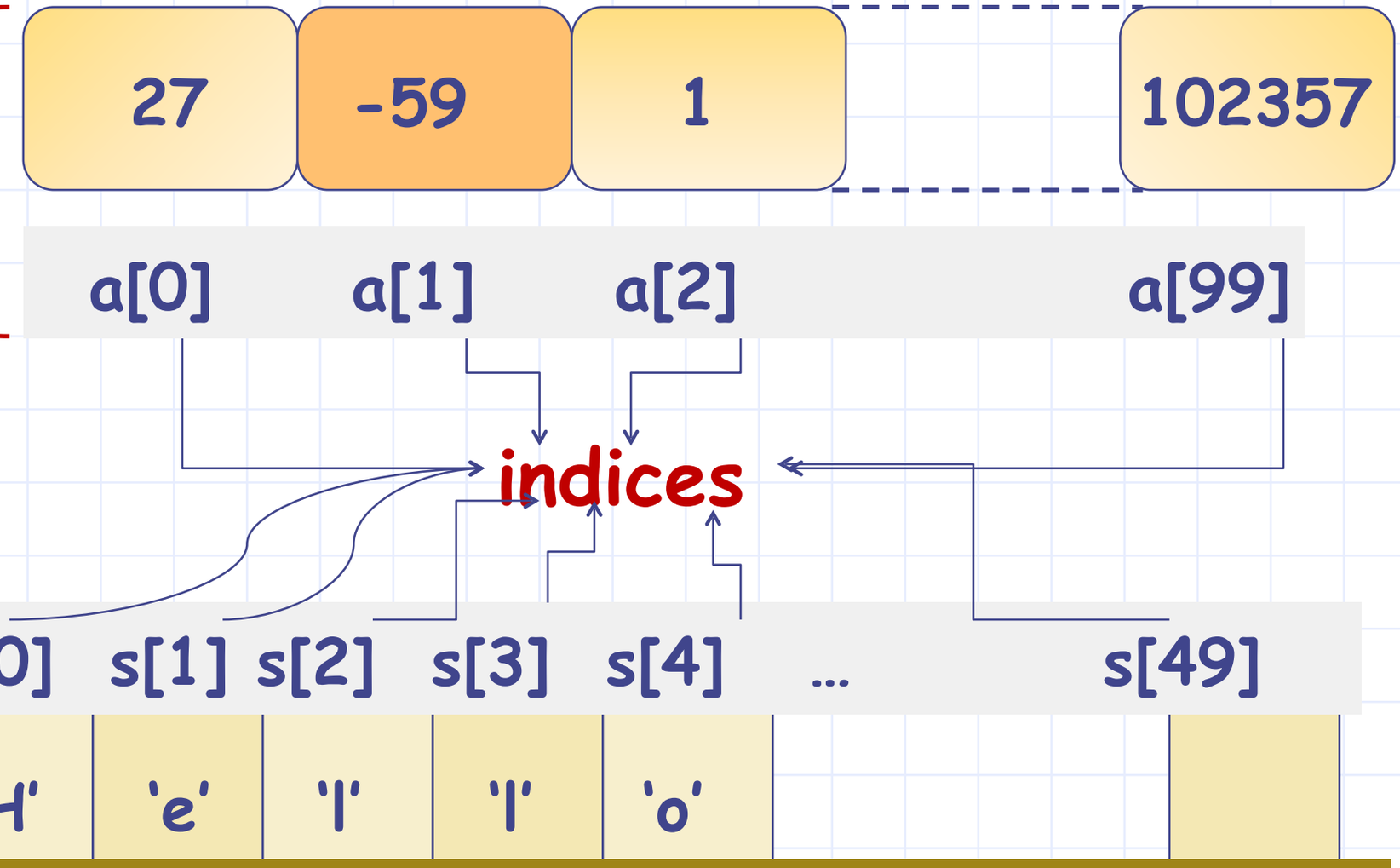

```
i = count-1;
while (i >=0) {
        putchar(s[i]);
        i=i-1;
}
```

/*print_in_reverse */

Arrays: Recap

Arrays are a consecutively allocated group of variables whose names are indexed.

```
int a[100];
a[0]=27;
a[1]=-59;
a[2]=1;
…
char s[50];
s[0]='H';
s[1]='e';
…
```



Indices always start at 0 in C

Passing arrays to functions

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  */
```

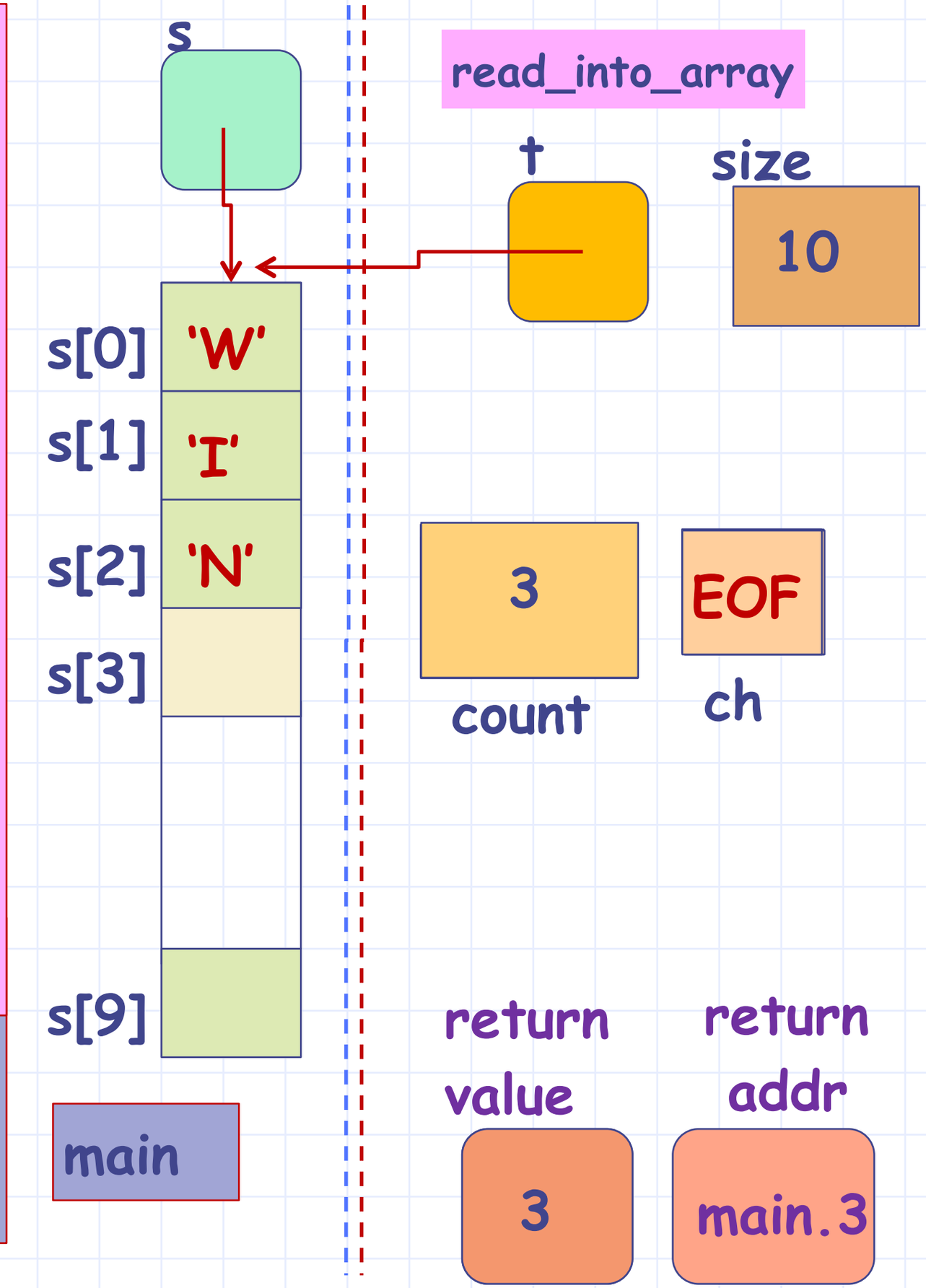
read_into_array takes an array t as an argument and **size** of the array and reads the input into array.

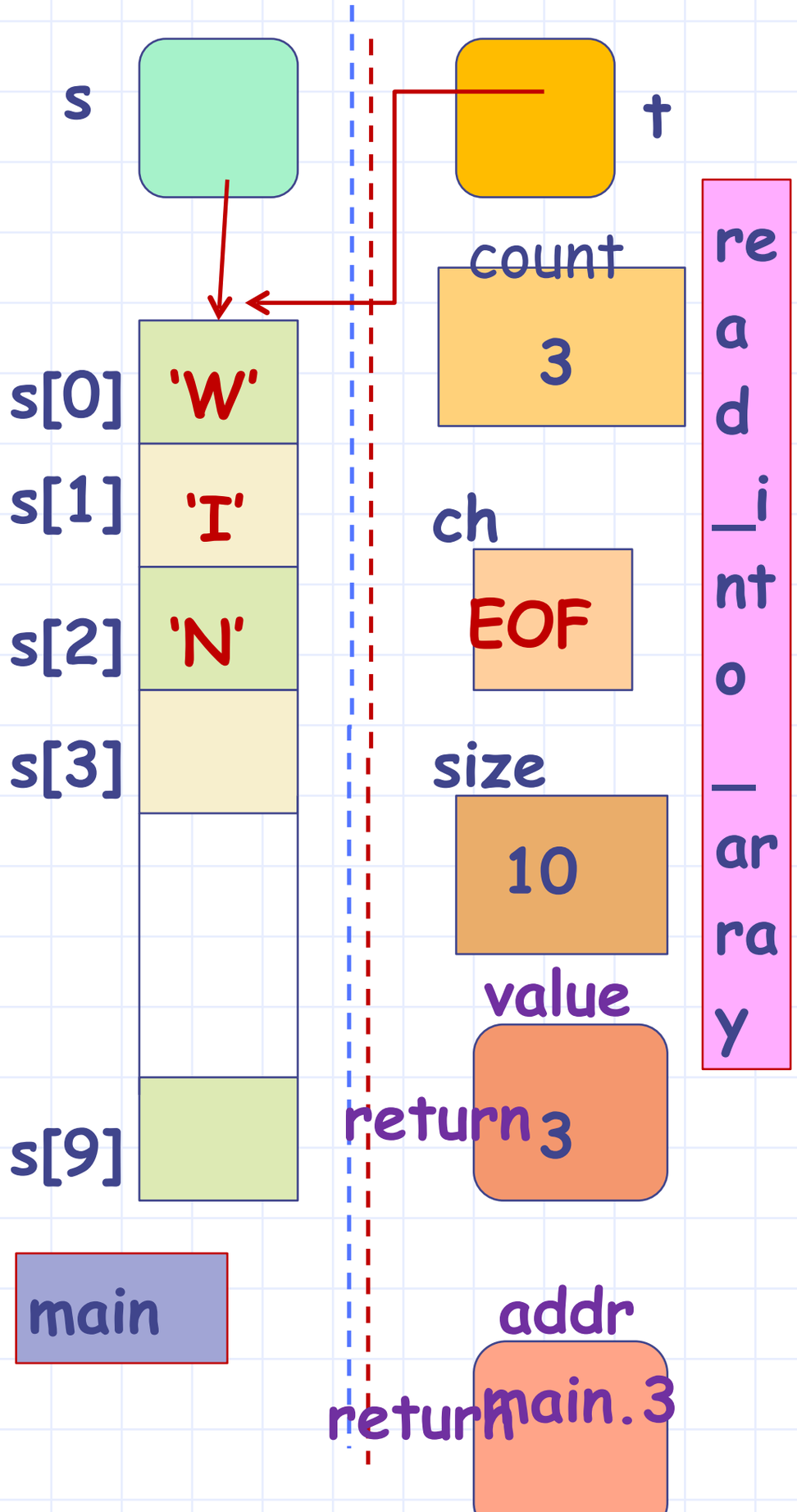
```
int main()  {
  char s[100];
  read_into_array(s,100);
  /* process */
}
```

```
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;
}
```

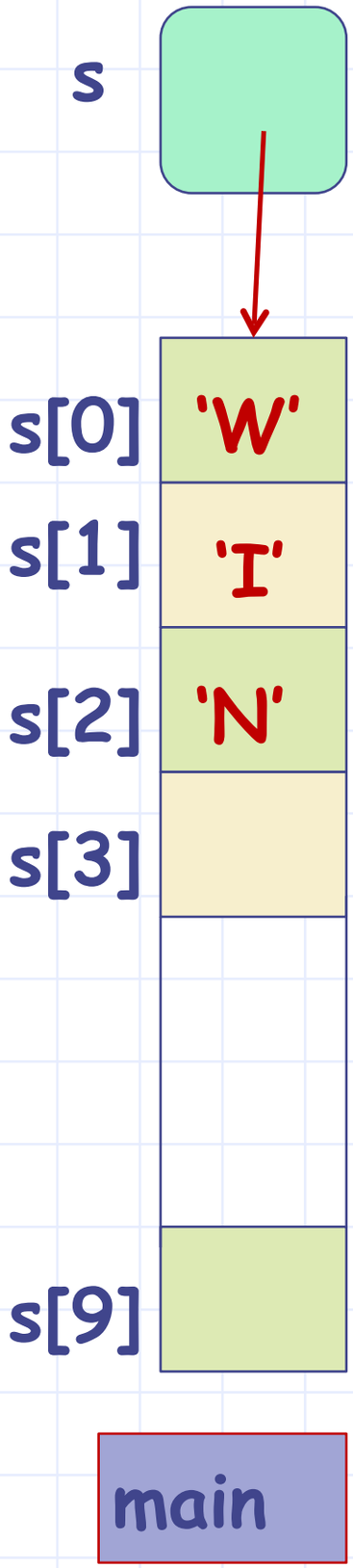
```
int read_into_array
            (char t[], int size) {
      int ch;
      int count = 0;
      ch = getchar();

    while (count < size
                && ch != EOF) {
          t[count] = ch;
          count = count + 1;
          ch = getchar();
    }   return count;
}
      char s[10];
      ead_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[] as a parameter!!

Parameter Passing

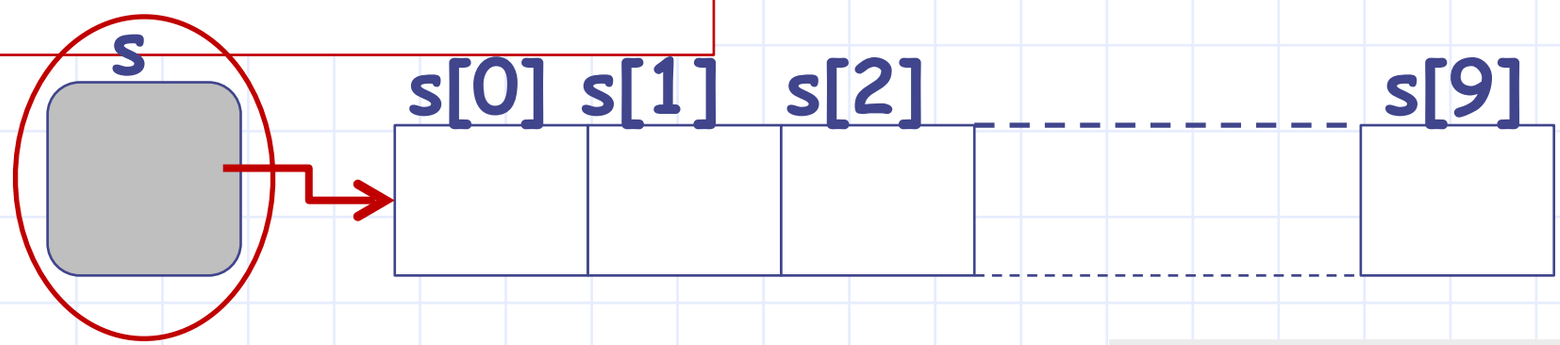
Basic steps:

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.

Let us look at parameter passing more carefully.

```
int main()  {
   int 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.

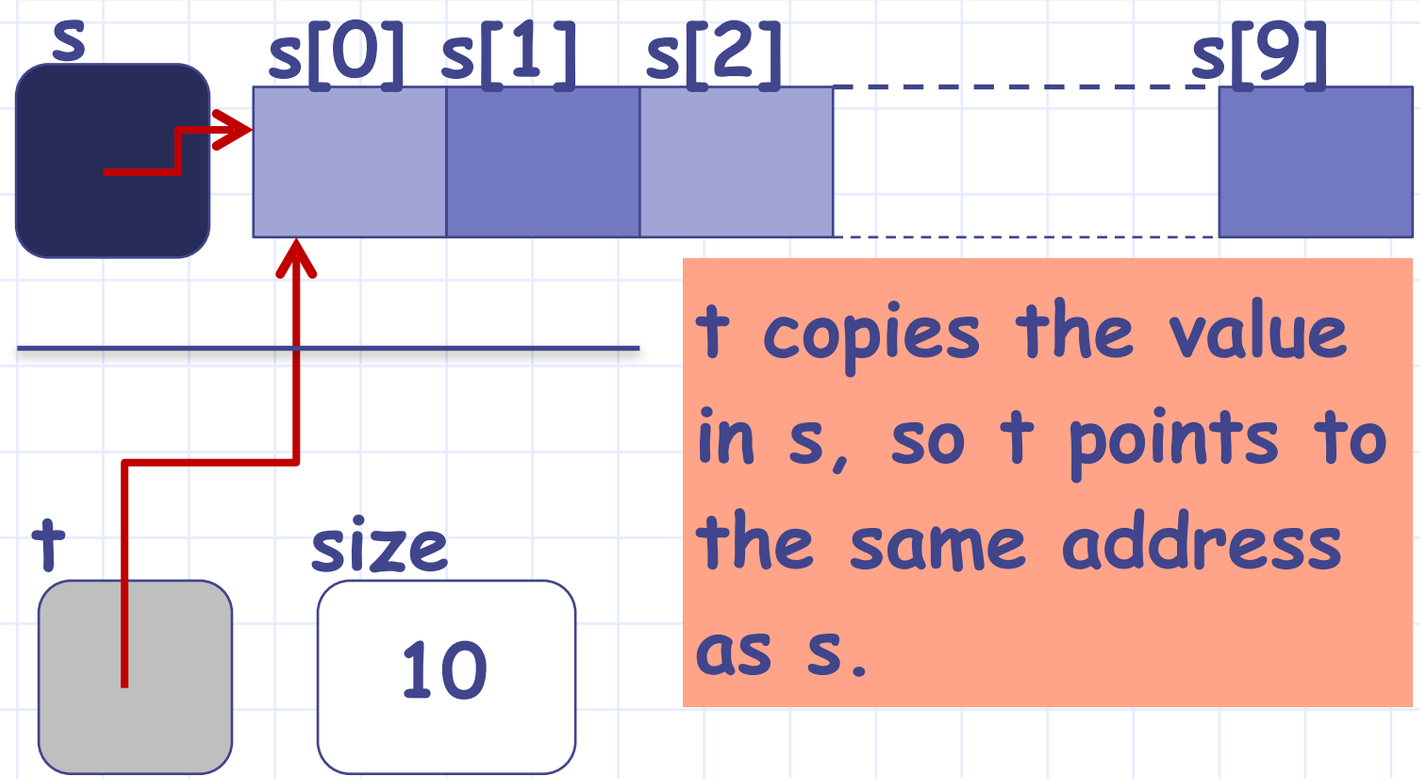
The stack of main just prior to call

Parameter Passing: Arrays

1. Create new variables (boxes) for each of the formal parameters allocated on a fresh stack created for this function call.
2. Copy values from actual parameters to the newly created formal paramters.

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

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



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, etc..

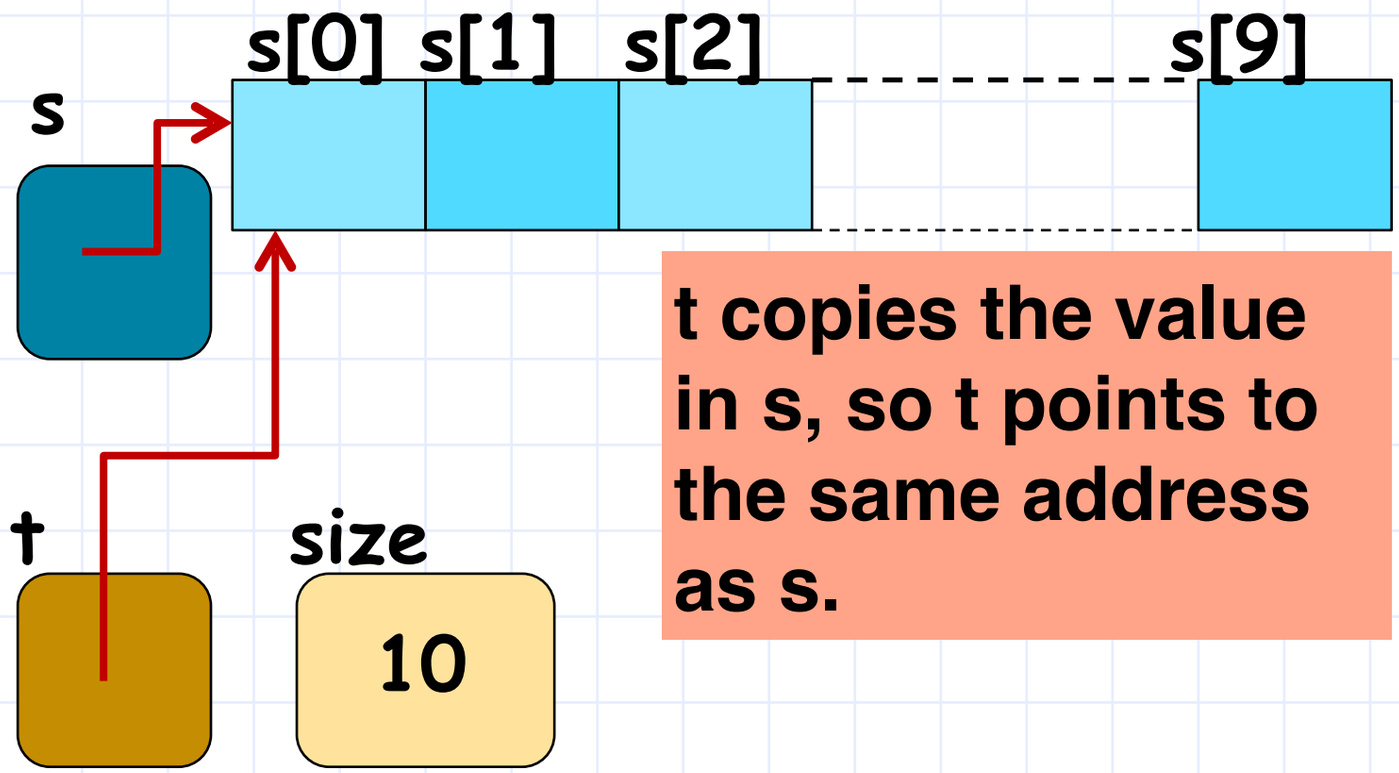
Parameter Passing: Arrays

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

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

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

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

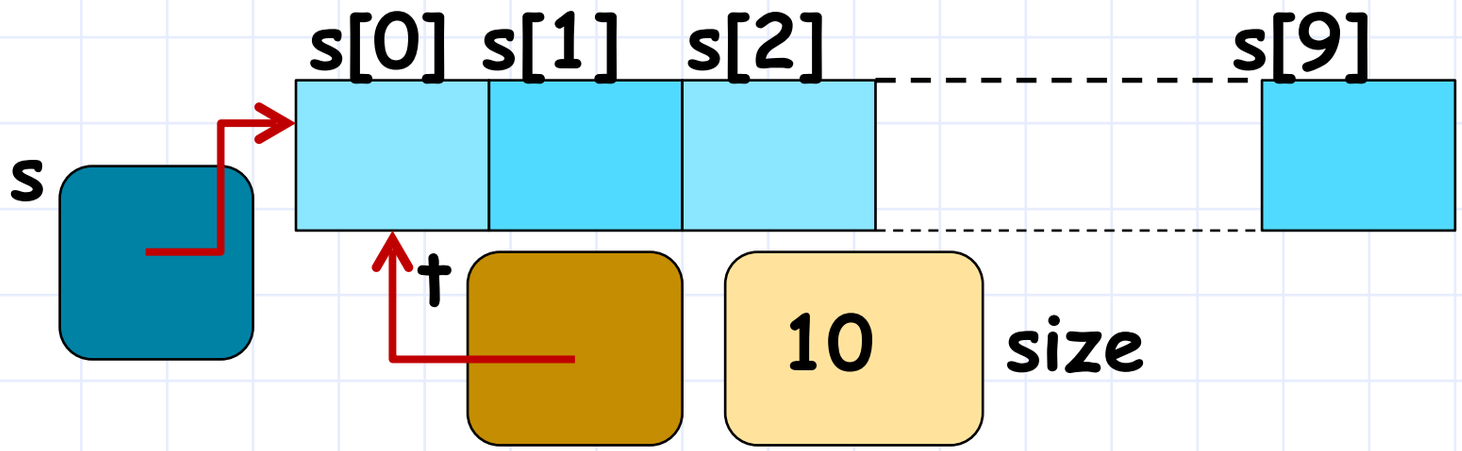


t copies the value in s, so t points to the same address as s.

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

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

Implications of copying content of array variable during parameter passing



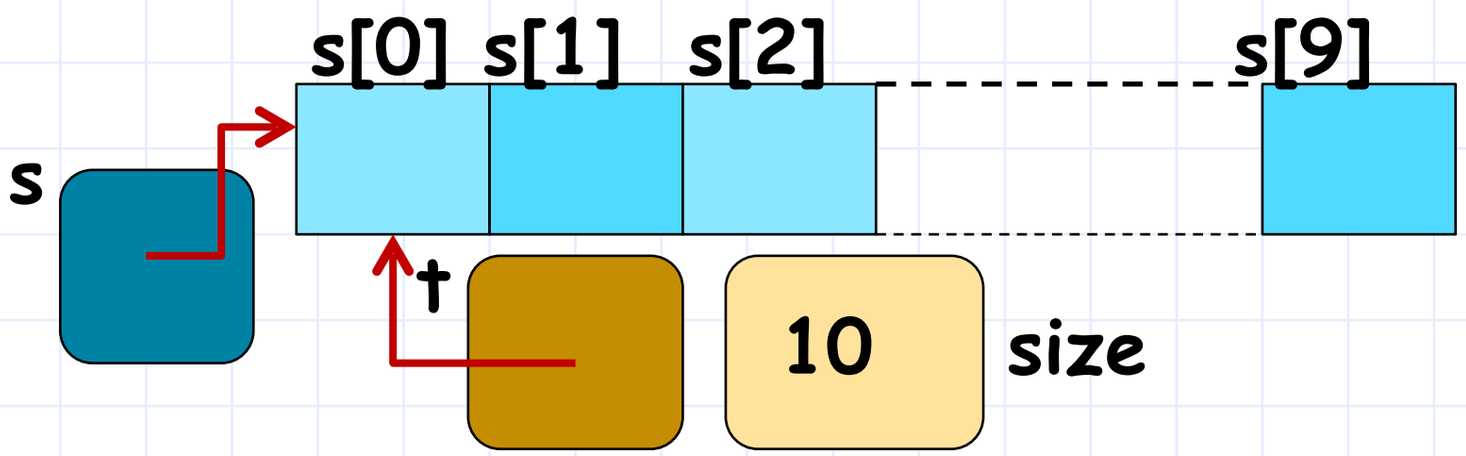
s is an array. In C an array is identified with a box whose value is the address of the first element of the array.

The value of s is copied into t. So the box corresponding to t has the same value as the box corresponding to s. They both now contain the address of the first element of the array.

1. **In the computer, an address is simply the value of a memory location.**
2. **For e.g., the value in the box for s would be the memory location of s[0].**
3. **When we draw figures, we will show this by an arrow.**

Pointers

The arrow from inside box s to s[0] indicates that s stores address of s[0].



Referred to as :

s points to s[0], or,
s is a pointer to s[0].

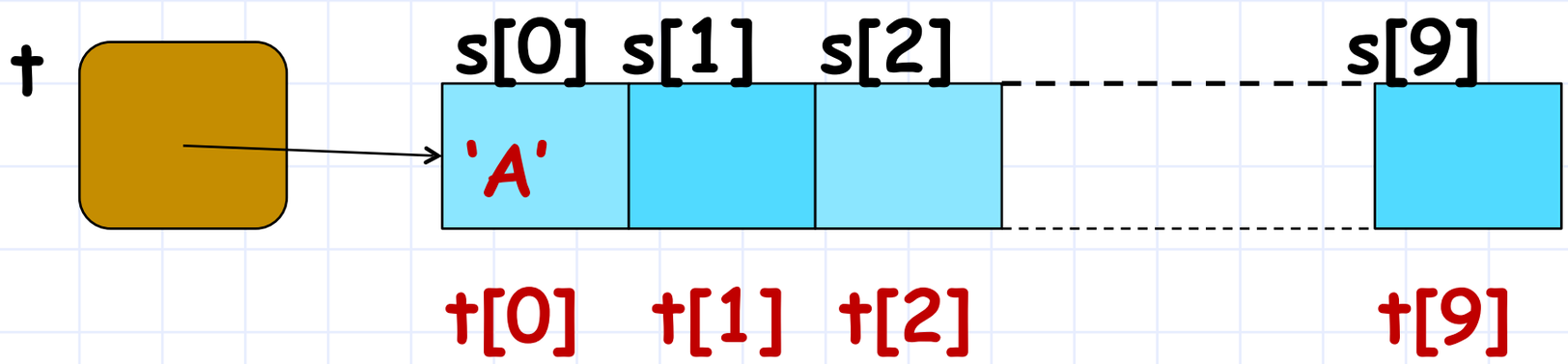
```
int main()  {
  int s[10];
  read_into_array(s,10); }
```

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

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

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

Let us see this now.



➢**t[0]** is the box whose address is stored in t. This is same as s[0].

➢**t[1]** is the box next to (successor to) the box whose address is stored in t. This is the same as s[1].

➢**t[2]** is the box 2 removed from the box whose address is stored in t; this is same as s[2], etc..

Now suppose we change t[0] using

```
t[0] = 'A';
```

Later on, in main(), when we access s[0], we see that s[0] is 'A'.

The box was the same, but it had two names, s[0] in main() and t[0] in read_into_array()

Practice Problem

- Write a program to read in a string that contains capital or small alphabets. Write a function CapToSmall that converts any Capital character to small characters

- Input: ESC101
- Output: ESC101 is now esc101

- Input: Esc101
- Output: Esc101 is now esc101

Solution for Practice problem

```
#include <stdio.h>

void CapToSmall(char src[], char dst[], int size)
{
   //convert cap to small
}

int main()
{
   char arr[10];
   char dst[10];
   scanf("%s",arr);
   CapToSmall( ___, ____, ___);
   printf("%s is now %s\n",arr, dst);
   return 0;
}
```

Solution for Practice problem

```
#include <stdio.h>

void CapToSmall(char src[], char dst[], int size)
{
   //convert cap to small
}

int main()
{
   char arr[10];
   char dst[10];
   scanf("%s",arr);
   CapToSmall( arr, dst, 10);
   printf("%s is now %s\n",arr, dst);
   return 0;
}
```

Solution for Practice problem

```
#include <stdio.h>

void CapToSmall(char src[], char dst[], int size)
{
   int i;
   for(i=0; i<size && src[i]!='\0'; i++)
      dst[i] = (src[i]>='A' && src[i]<='Z')?src[i]-'A'+'a':src[i];
   dst[i] = '\0';
}

int main()
{
   char arr[10];
   char dst[10];
   scanf("%s",arr);
   CapToSmall( arr, dst, 10);
   printf("%s is now %s\n",arr, dst);
   return 0;
}
```

Assignment operators

- i = i+10; can be shortened to

  i += 10;

- += is a new *assignment operator*.
- Similarly, +=, -=, *=, /=, %=
- expr1 op= expr2 is equivalent to

  expr1 = (expr1) op (expr2).
- Eg. x %= y+1 is x = x%(y+1) .
- Precedence rules are the same as that of = (Right to left associativity).

(In)(De)crement operators

- What is the difference between i++ and ++i in C ?
- The expression (i++) has the
  - value i
  - side-effect i=i+1
- The expression (++i) has the
  - value i+1
  - side-effect i=i+1
- Eg. (i == ++i) is always FALSE.
- Eg. (i == i++) is always TRUE.

Next Week

- Midsem
- Good luck for the exam!