```
int ch;
int count = 0;
read the next character into ch using getchar();
while (ch is not EOF AND count < 100) {
        s[count] = ch;
        count = count + 1;
        read the next character into ch using getchar();
}
```

initial design pseudo-code

```
int ch;
int count = 0;
ch = getchar();
while ( ch != EOF && count < 100) {
        s[count] = ch;
        count = count + 1;
        ch = getchar();
}
```

Translating the read_into_array pseudo-code into code.

Overall design

```
int main() {
    char s[100];
 /* read_into_array */
 /* print_reverse */
    return 0;
}
```

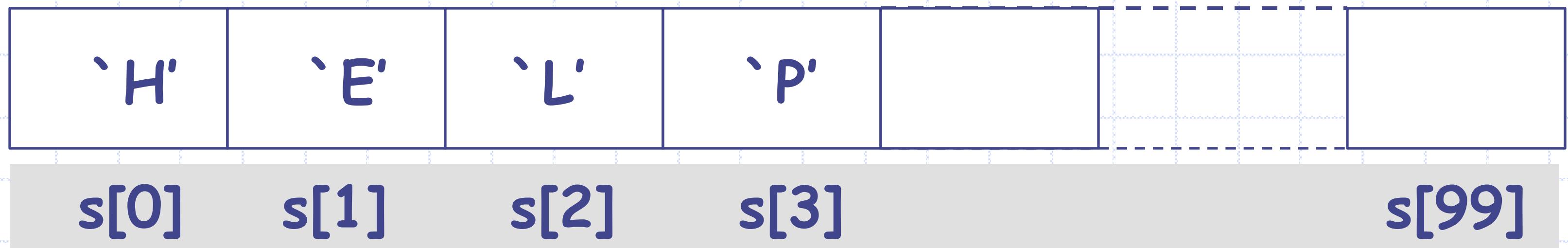
What is the value of count at the end of read_into_array?

Now let us design the code fragment print_reverse

Suppose input is

HELP<eof>

The array char s[100]



i

index i runs backwards in array

count 4

```
int i;
set i to the index of last character read.

while (i >= 0) {
    print s[i]
    i = i-1;        /* shift array index one to left */
}
```

PSEUDO CODE

The array char s[100]

`H'	`E'	`L'	`P'			
s[0]	s[1]	s[2]	s[3]			s[99]

i

index i runs backwards in array

count 4

```
int i;
set i to index of the last character read.

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

PSEUDO CODE

```
int i;
i = count-1;

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

Translating pseudo code to C code: print_reverse

Code for printing characters read in array in reverse

Putting it together

Overall design

```
int main() {
        char s[100];
        /* read_into_array */
        /* print_reverse */
        return 0;
}
```

The code fragments we have written so far.

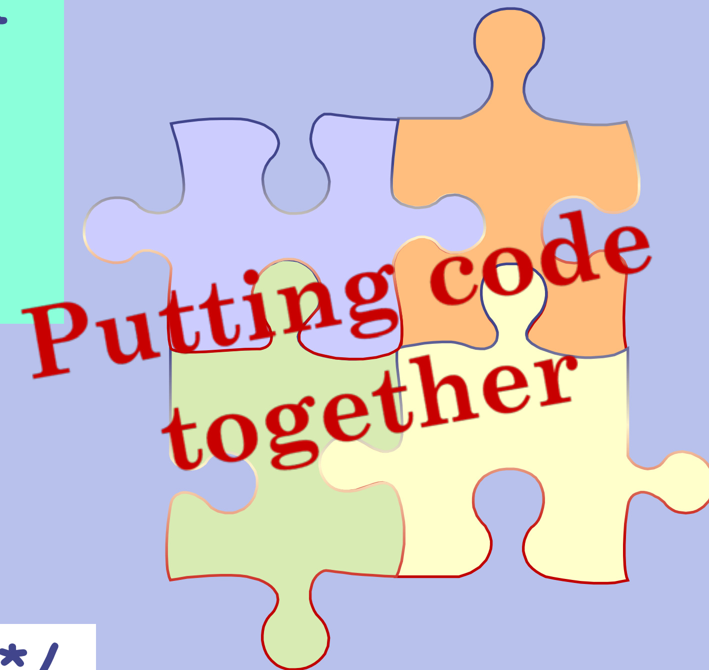
```
int count = 0;
int ch;
ch = getchar();
while ( ch != EOF && count < 100) {
        s[count] = ch;
        count = count + 1;
        ch = getchar();
}
```

read_into_array code.

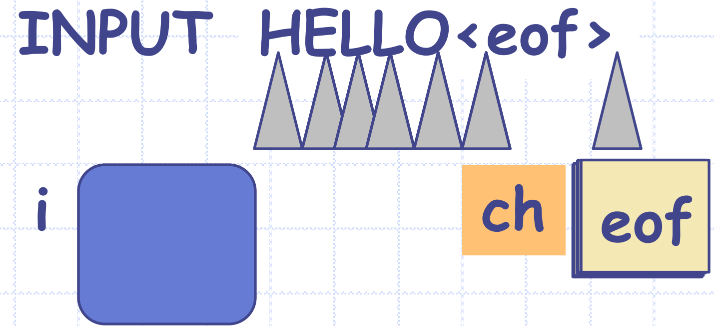
```
int i;
i = count-1;
while (i >=0) {
        putchar(s[i]);
        i=i-1;
}
```

print_reverse code

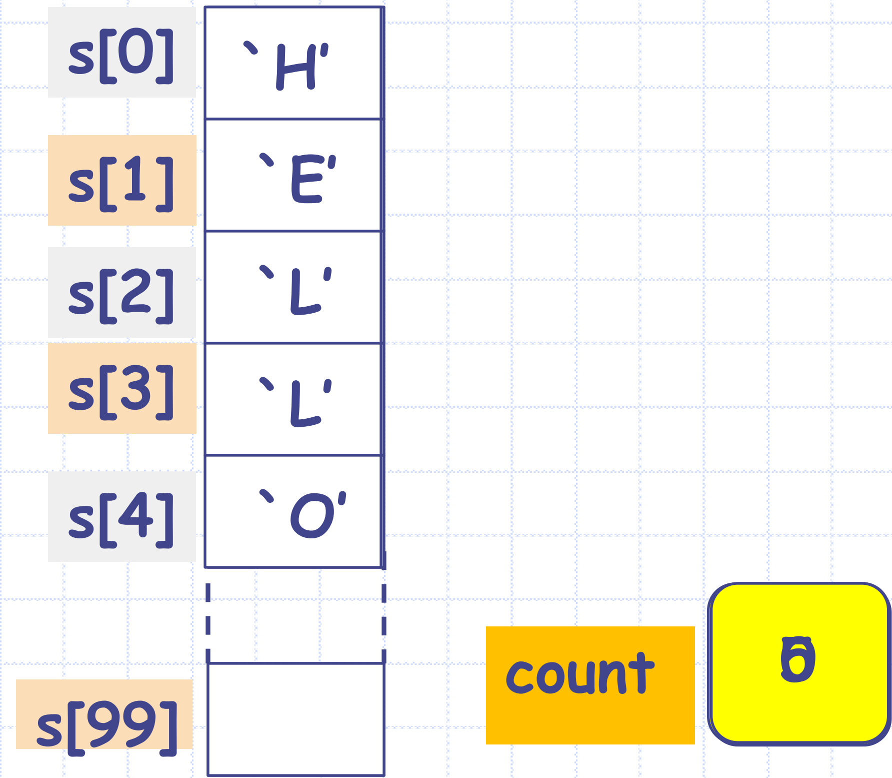
```c
#include <stdio.h>
int main() {
    char s[100];        /* the array of 100 char */
    int count = 0;      /* counts number of input chars read */
    int ch;             /* current character read  */
    int i;              /* index for printing array backwards */

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

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

    return 0;
}
```



Let us trace the execution.
We will do this for part
read_into_array




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

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



```
#include <stdio.h>
int main() {
    char s[100];
    int count = 0;
    int ch;
    int i;
    while ( (ch=getchar()) != EOF &&
                        count < 100 )
    {       s[count] = ch;
            count = count + 1;
    }

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

/*read_into_array */

/*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';
…
```

27	-59	1		102357
a[0]	a[1]	a[2]		a[99]

indices

s[0]	s[1]	s[2]	s[3]	s[4]	…	s[49]
'H'	'e'	'l'	'l'	'o'		

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;
}
```

```c
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;
}
```

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

s

read_into_array

t

size

10

s[0] 'W'

s[1] 'I'

s[2] 'N'

s[3]

s[9]

3

count

EOF

ch

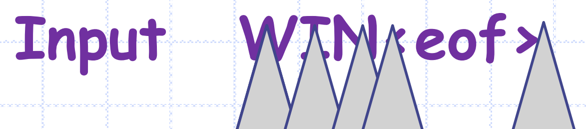
main

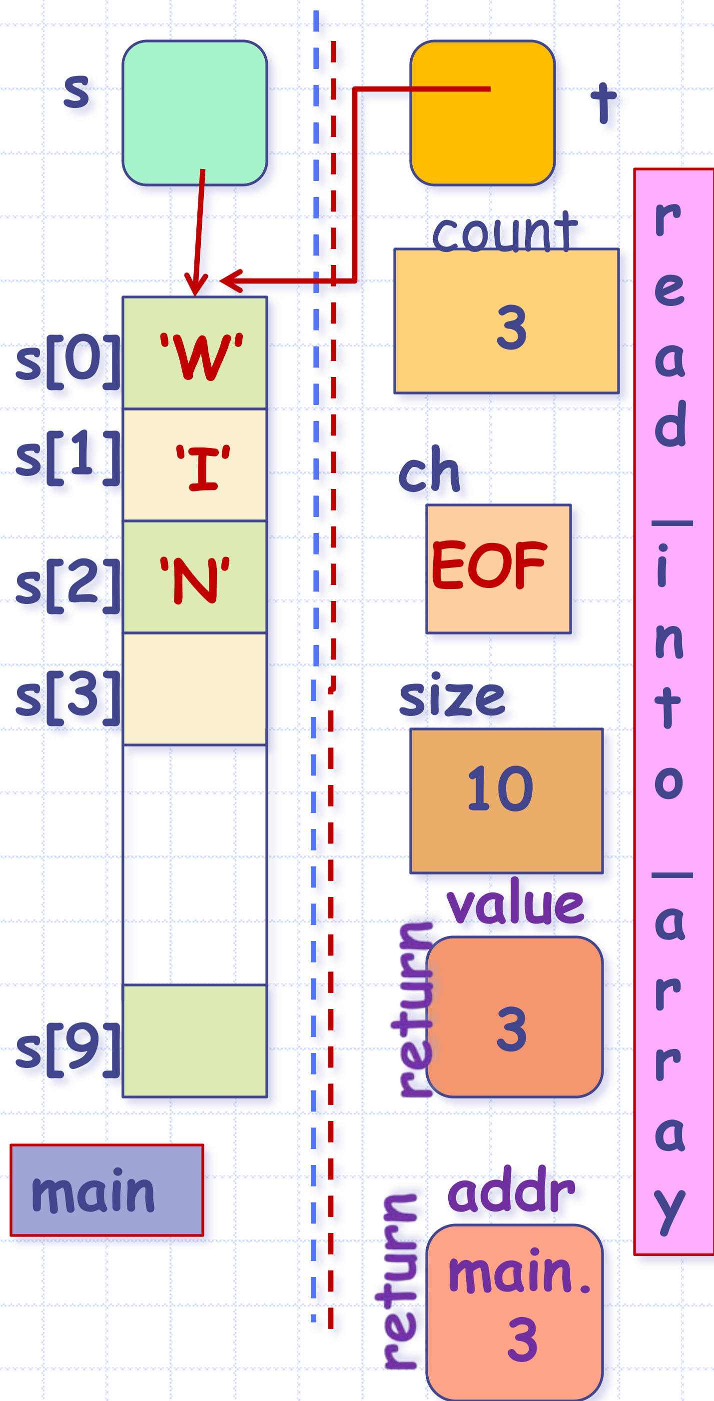
return value

3

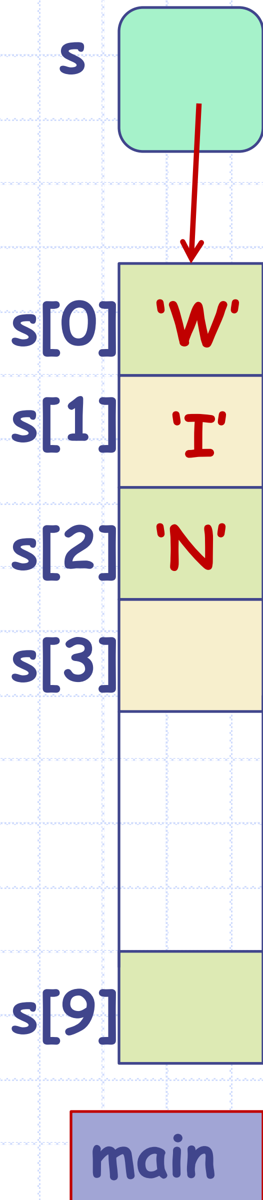
return addr

main.3





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!!

```
paint_hostel(char hostel[], int number-of-rooms)
{
    int r;
    for (r = 0; r < number-of-rooms; r++)
        paint_room(hostel[r]);
}

iit ()
{
   char hostel1[200];
   char hostel2[300];
   char hostel3[300];
   ...

   if ( ... ) paint_hostel(hostel1, 200);
   if ( ... ) paint_hostel(hostel2, 200); // OK: last 100 rooms
                                          //     may not need painting
   if ( ... ) paint_hostel(hostel3, 400); // Not OK: There are no
                                                  rooms beyond 300
   ...
}
```

```
paint_hostel200(char hostel[200])
{   int r;
    for (r = 0; r < 200; r++)
          paint_room(hostel[r]);
}
paint_hostel300(char hostel[300])
{   int r;
    for (r = 0; r < 300; goto-next-room)
          paint_room(hostel[r]);
}
iit ()
{
    char hostel1[200];
    char hostel2[300];
    char hostel3[300];
     // Are these correct? EXERCISE!!
    if ( ... ) paint_hostel200(hostel1);
    if ( ... ) paint_hostel300(hostel2);
    if ( ... ) paint_hostel300(hostel1);
    if ( ... ) paint_hostel200(hostel3);
}
```

Parameter Passing

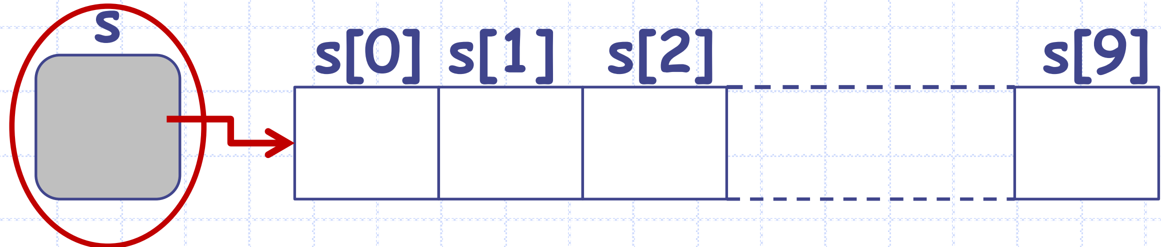
Basic steps:

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

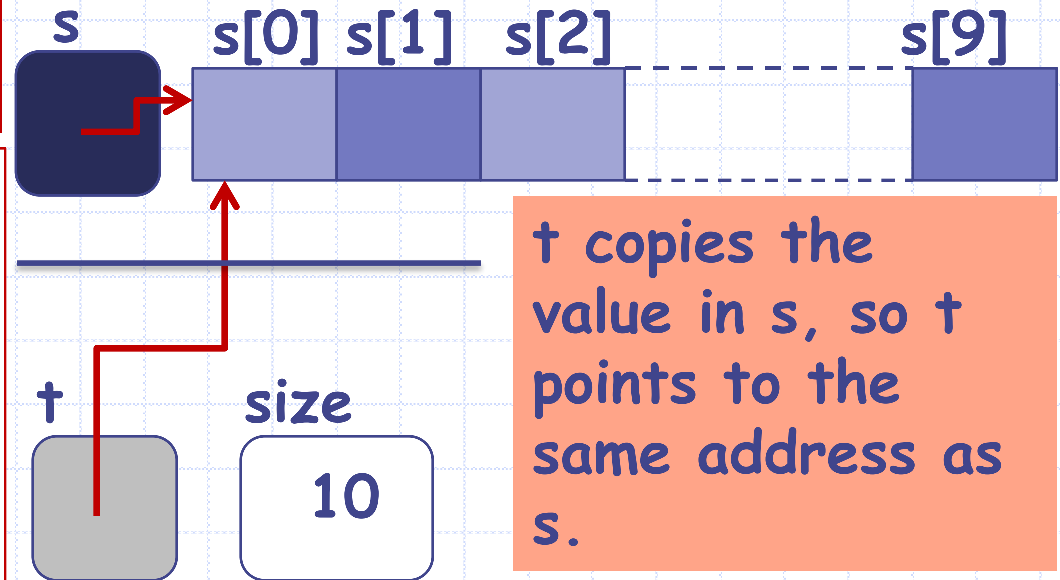
Parameter Passing: Arrays

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;
     /*  …    */
}
```



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

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