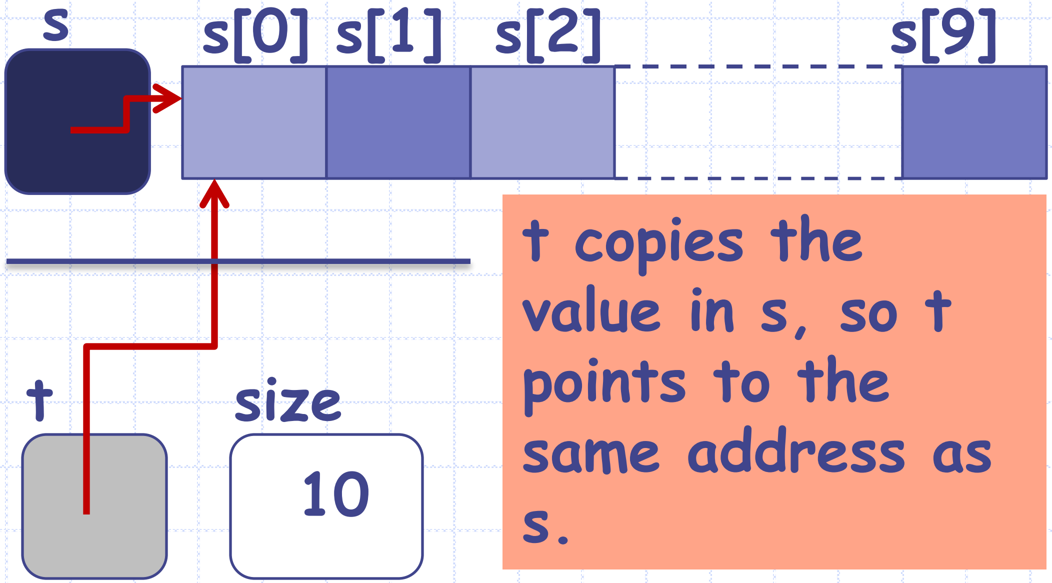
Parameter Passing: Arrays

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!!

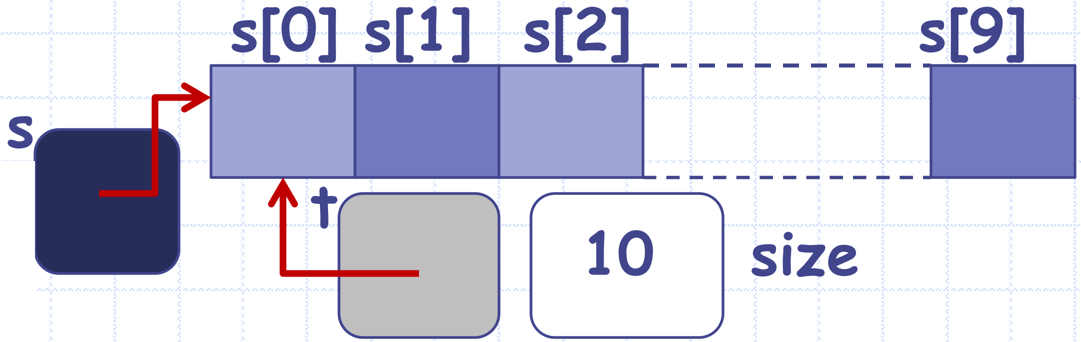
s[0] and t[0] refer to the same variable, etc..

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.

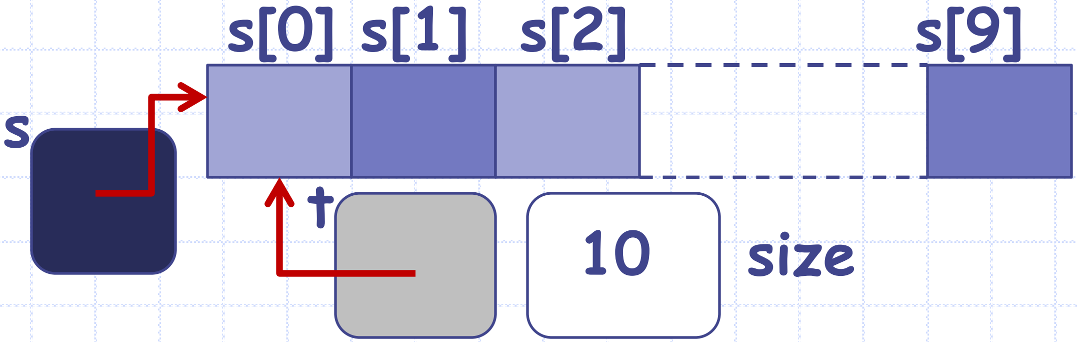
The value of s is copied into t. So the box corresponding to t has the same value as the box corresponding to s.

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].

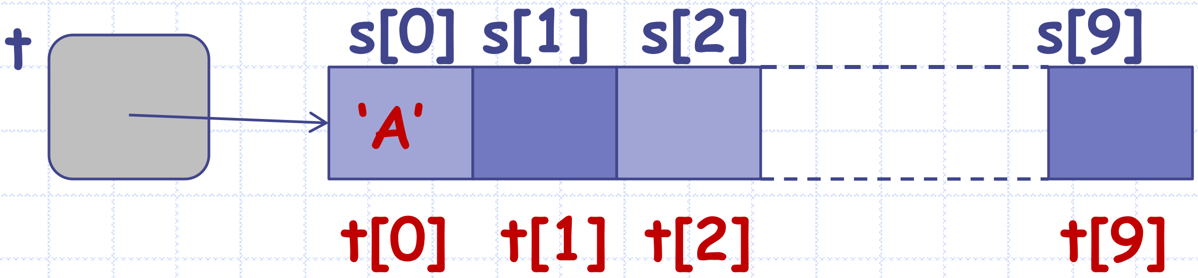
```
int main() {
  int s[10];
  read_into_array(s,10); }
```

```
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 steps next to the box whose address is stored in t; this is same as s[2], etc..

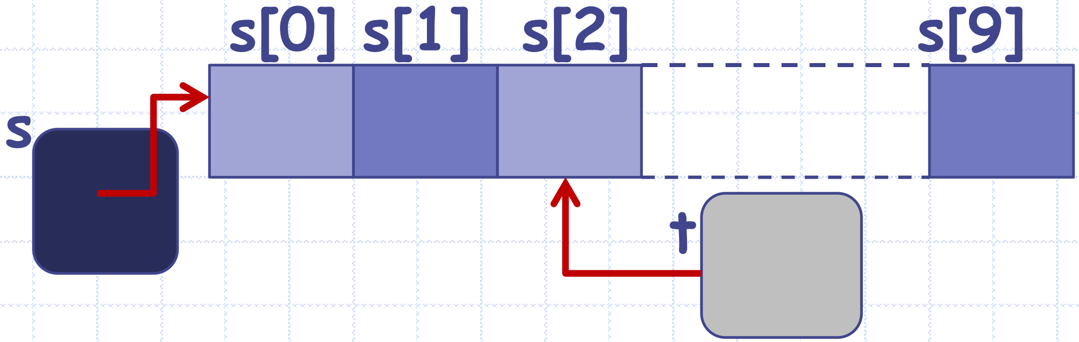
Now suppose we change t[0] using

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

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

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

Address Arithmetic



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.

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

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

Since 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], t[2]=s[4] refers to the box that is 2 chars further from the box t[0] and so on...

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[], 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

```
}
```

```
OUTPUT
105
49
```

```
#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
```

Generating Prime Numbers

- Problem: Given a positive integer N, generate all prime numbers up to N.
- A Greek mathematician Eratosthenes came up with a simple but fast algorithm
- Sieve of Eratosthenes

Sieve of Eratosthenes

- On a piece of paper, write down all the integers starting from 2 till N.
- Starting from 2 strike off all multiples of 2, except 2.
- Next, find the first number that has not been struck and strike off all its multiples, except the number.
- Continue until you cannot strike out any more numbers.
- The numbers that have not been struck, are PRIMES.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

	2	3	~~4~~	5	~~6~~	7	~~8~~	9	~~10~~
11	~~12~~	13	~~14~~	15	~~16~~	17	~~18~~	19	~~20~~
21	~~22~~	23	~~24~~	25	~~26~~	27	~~28~~	29	~~30~~
31	~~32~~	33	~~34~~	35	~~36~~	37	~~38~~	39	~~40~~
41	~~42~~	43	~~44~~	45	~~46~~	47	~~48~~	49	~~50~~
51	~~52~~	53	~~54~~	55	~~56~~	57	~~58~~	59	~~60~~
61	~~62~~	63	~~64~~	65	~~66~~	67	~~68~~	69	~~70~~
71	~~72~~	73	~~74~~	75	~~76~~	77	~~78~~	79	~~80~~
81	~~82~~	83	~~84~~	85	~~86~~	87	~~88~~	89	~~90~~
91	~~92~~	93	~~94~~	95	~~96~~	97	~~98~~	99	~~100~~

	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	25	~~26~~	~~27~~	~~28~~	29	~~30~~
31	~~32~~	~~33~~	~~34~~	35	~~36~~	37	~~38~~	~~39~~	~~40~~
41	~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	49	~~50~~
~~51~~	~~52~~	53	~~54~~	55	~~56~~	~~57~~	~~58~~	59	~~60~~
61	~~62~~	~~63~~	~~64~~	65	~~66~~	67	~~68~~	~~69~~	~~70~~
71	~~72~~	73	~~74~~	~~75~~	~~76~~	77	~~78~~	79	~~80~~
~~81~~	~~82~~	83	~~84~~	85	~~86~~	~~87~~	~~88~~	89	~~90~~
91	~~92~~	~~93~~	~~94~~	95	~~96~~	97	~~98~~	~~99~~	~~100~~

	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29	~~30~~
31	~~32~~	~~33~~	~~34~~	~~35~~	~~36~~	37	~~38~~	~~39~~	~~40~~
41	~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	49	~~50~~
~~51~~	~~52~~	53	~~54~~	~~55~~	~~56~~	~~57~~	~~58~~	59	~~60~~
61	~~62~~	~~63~~	~~64~~	~~65~~	~~66~~	67	~~68~~	~~69~~	~~70~~
71	~~72~~	73	~~74~~	~~75~~	~~76~~	77	~~78~~	79	~~80~~
~~81~~	~~82~~	83	~~84~~	~~85~~	~~86~~	~~87~~	~~88~~	89	~~90~~
91	~~92~~	~~93~~	~~94~~	~~95~~	~~96~~	97	~~98~~	~~99~~	~~100~~

	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29	~~30~~
31	~~32~~	~~33~~	~~34~~	~~35~~	~~36~~	37	~~38~~	~~39~~	~~40~~
41	~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	~~49~~	~~50~~
~~51~~	~~52~~	53	~~54~~	~~55~~	~~56~~	~~57~~	~~58~~	59	~~60~~
61	~~62~~	~~63~~	~~64~~	~~65~~	~~66~~	67	~~68~~	~~69~~	~~70~~
71	~~72~~	73	~~74~~	~~75~~	~~76~~	~~77~~	~~78~~	79	~~80~~
~~81~~	~~82~~	83	~~84~~	~~85~~	~~86~~	~~87~~	~~88~~	89	~~90~~
~~91~~	~~92~~	~~93~~	~~94~~	~~95~~	~~96~~	97	~~98~~	~~99~~	~~100~~

	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29	~~30~~
31	~~32~~	~~33~~	~~34~~	~~35~~	~~36~~	37	~~38~~	~~39~~	~~40~~
41	~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	~~49~~	~~50~~
~~51~~	~~52~~	53	~~54~~	~~55~~	~~56~~	~~57~~	~~58~~	59	~~60~~
61	~~62~~	~~63~~	~~64~~	~~65~~	~~66~~	67	~~68~~	~~69~~	~~70~~
71	~~72~~	73	~~74~~	~~75~~	~~76~~	~~77~~	~~78~~	79	~~80~~
~~81~~	~~82~~	83	~~84~~	~~85~~	~~86~~	~~87~~	~~88~~	89	~~90~~
~~91~~	~~92~~	~~93~~	~~94~~	~~95~~	~~96~~	97	~~98~~	~~99~~	~~100~~

Generating Prime Numbers using Sieve of Eratosthenes

- No more numbers can be marked. Algorithm terminates.
- Primes up to 100 are 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97.
- Going up to $\sqrt{N}$ is enough.

Sieve of Eratosthenes: Program

```
int prim[10000]; // global array
void sieve(int n) {
  int i, j = 2;
  prim[0]=0; prim[1]=0;
  for (i=2; i<=n; i++)  prim[i] = 1;

  while (j <= n) {
      if (prim[j] == 0) { // composite
         j++; continue;
      }
      for (i= j*j; i<=n; i=i+j)
               prim[i] = 0;
      j++;
  }
}
```

```
int main() {
  int i, n;
  scanf("%d", &n);
  // check n < 10000

  sieve(n); // set primes

  for (i=2; i<=n; i++) {
       if (prim[i] == 1)
            printf("%d\n", i);
  }

  return 0;
}
```