

HIGH-LEVEL PROGRAMMING I

Arrays and pointers

by Prasanna Ghali

What do we know about arrays?

2

```
int a[10];
```

Using base type `int` and size `10`, compiler reserves 40 bytes of contiguous memory storage



`a` is base address of array

`a[i]`: subscript operator is used to refer to anonymous array elements

What do we know about pointers?

(1 / 2)

3

```
int x = 1, *pi = &x;
```

address of operator



x	104	1
pi	108	104
	116	?
	⋮	⋮

What do we know about pointers?

(2/2)

4

```
int x = 1, *pi = &x;  
*pi = 3;
```

dereference or indirection operator



x	104	13
pi	108	104
	116	?
	⋮	⋮

Pointers and arrays

5

- Pointers and arrays have close relationship that provides alternative way of referencing array elements
 - ▣ Subscript operator can be replaced with pointer arithmetic and dereference operator
- Understanding this relationship critical for mastering C

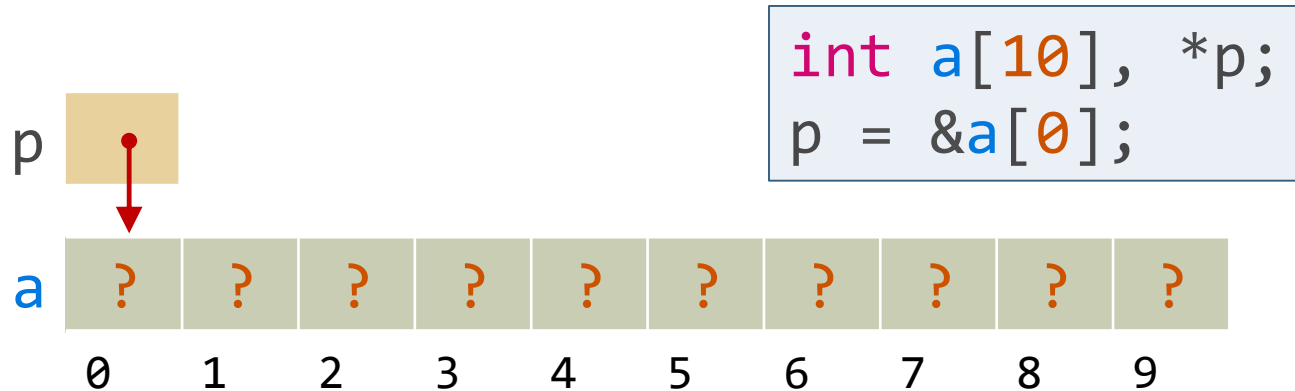
6

```
int a[10], *p;  
p = &a[3];
```

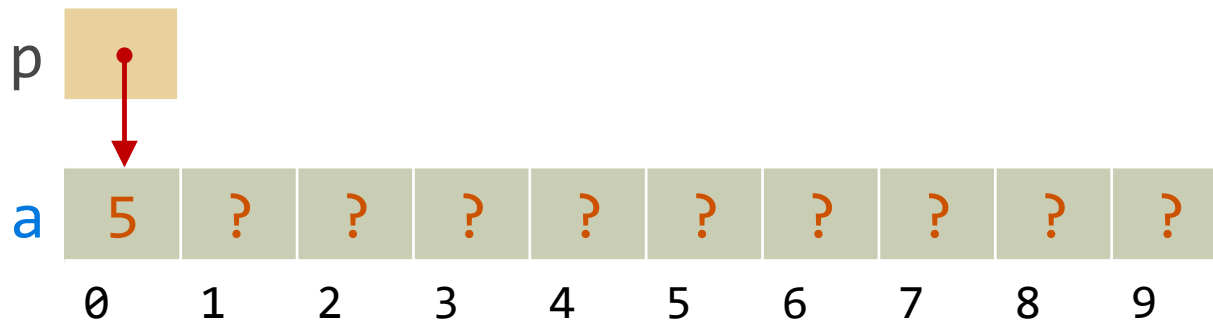
The diagram shows an array `a` with 10 elements, indexed from 0 to 9. Each element is represented by a light green box containing an orange question mark. A pointer variable `p` is shown as a light orange box above the array. A red arrow points from `p` to the element at index 3. A green arrow points to the `p` box from the top right.

Pointer arithmetic (2/4)

7

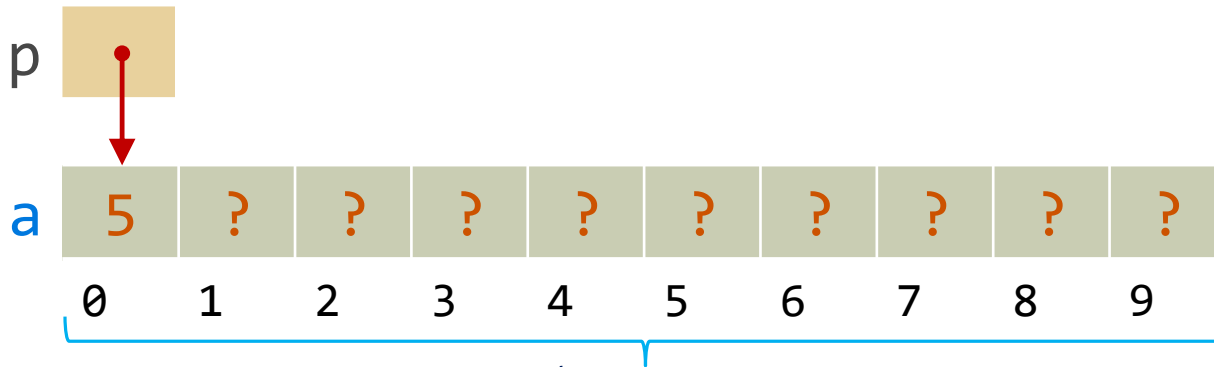


□ Use `p` to indirectly update `a[0]`: `*p = 5;`



Pointer arithmetic (3/4)

8



```
int a[10], *p;  
p = &a[0];  
*p = 5;
```

Array elements given contiguous memory storage.
Each element stored `sizeof(int)` bytes after previous element.

- If pointer `p` points to array element, other array elements can be referenced by performing pointer arithmetic on `p`

Pointer arithmetic (4/4)

9

- Four forms of pointer arithmetic supported:
 - ▣ Adding integer to pointer
 - ▣ Subtracting integer from pointer
 - ▣ Subtracting one pointer from another
 - ▣ Comparing pointers

Adding integer to pointer

10

- Suppose p is pointing to array element $a[i]$
- Adding integer j to pointer p yields pointer to element j places *after* the one that p points to
 - ▣ More precisely: if p points to array element $a[i]$, then $p+j$ points to $a[i+j]$

```
int a[5] = {6, 2, 7, 3, 8};  
int *p = &a[1], *q = p+3;  
printf("%p | %p | %p\n", p, p+3, q);  
printf("%d | %d | %d\n", *p, *(p+3), *q);
```

Examples:

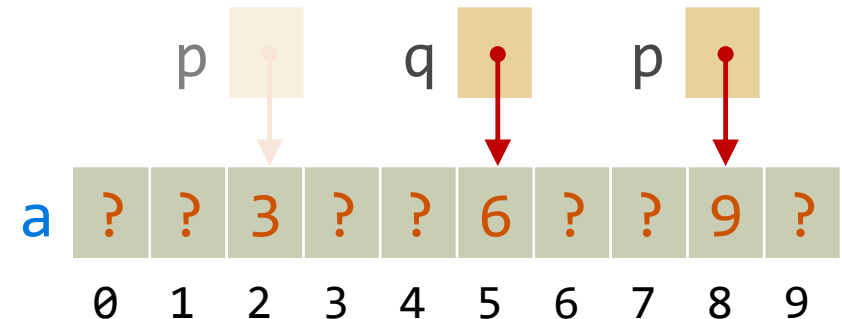
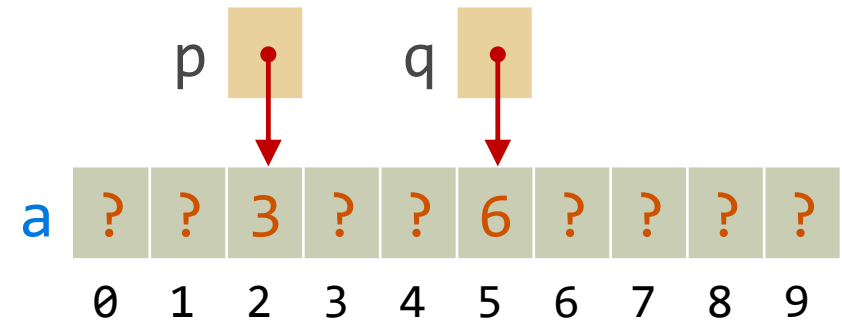
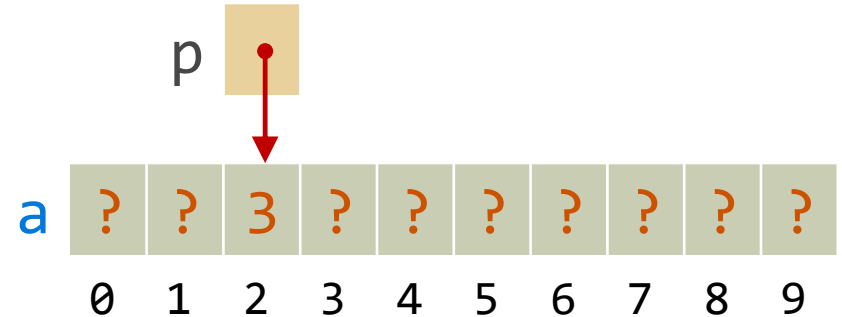
Adding integer to pointer

11

```
int a[10], *p, *q;  
p = &a[2];  
*p = 3;
```

```
q = p + 3;  
*q = 6;
```

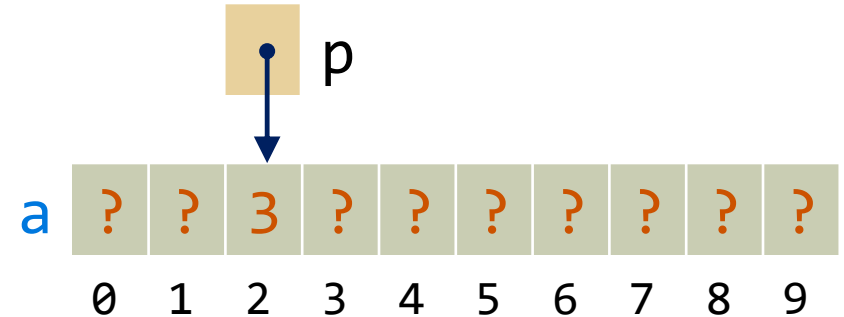
```
p += 6;  
*p = 9;
```



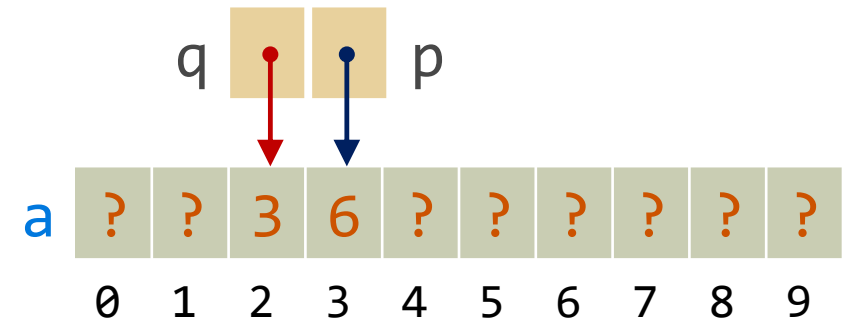
Postfix and prefix increment operators

12

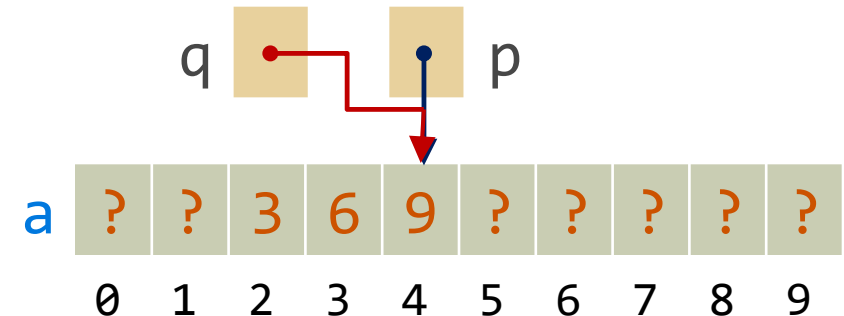
```
int a[10], *p, *q;  
p = &a[2];  
*p = 3;
```



```
q = p++;  
*p = 6;
```



```
q = ++p;  
*p = 9;
```



Subtracting integer from pointer

13

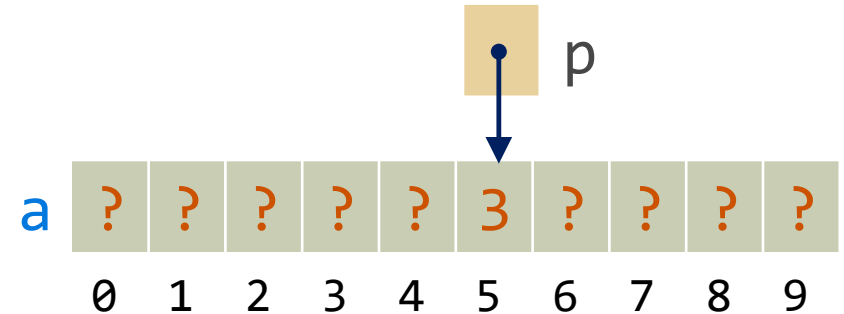
- Suppose p is pointing to array element $a[i]$
- Subtracting integer j from pointer p yields pointer to element j places *before* the one that p points to
 - ▣ More precisely: if p points to array element $a[i]$, then $p-j$ points to $a[i-j]$

```
int a[5] = {6, 2, 7, 3, 8};  
int *p = &a[4], *q = p-3;  
printf("%p | %p | %p\n", p, p-3, q);  
printf("%d | %d | %d\n", *p, *(p-3), *q);
```

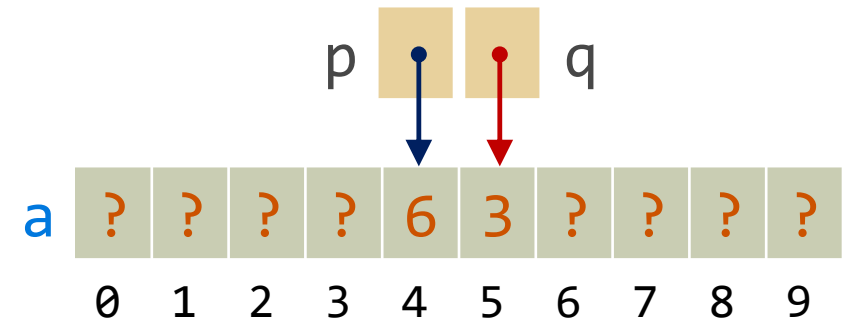
Postfix and prefix decrement operators

15

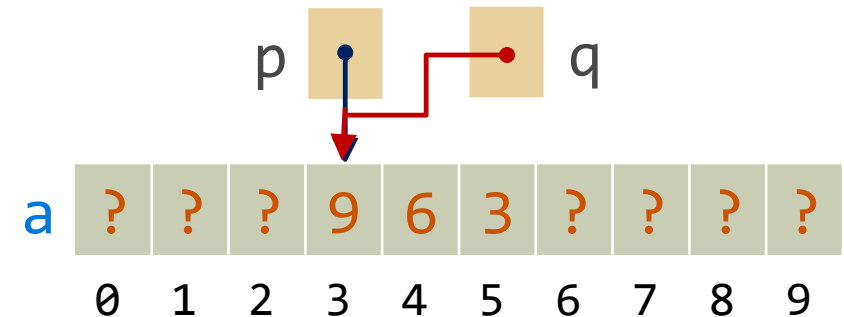
```
int a[10], *p, *q;  
p = &a[5];  
*p = 3;
```



```
q = p--;  
*p = 6;
```



```
q = --p;  
*p = 9;
```



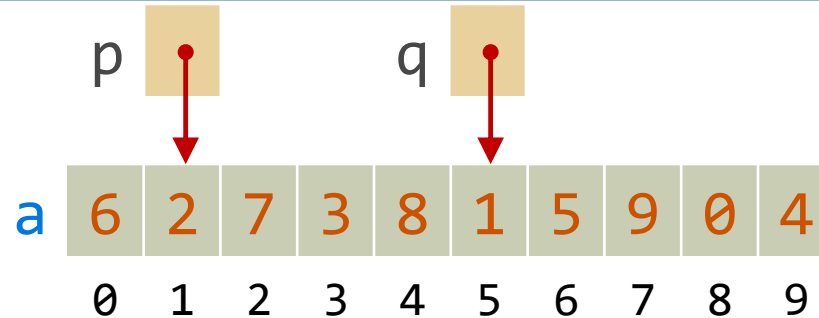
Subtracting pointers

16

- When one pointer is subtracted from another, result is array elements between the pointers
- If p points to $a[i]$ and q points to $a[j]$, then $p - q$ is equal to $i - j$

Subtracting pointers: Example

17



```
int a[10] = {6, 2, 7, 3, 8, 1, 5, 9, 0, 4};  
int *p = &a[1];  
int *q = &a[5];
```

```
int i = q - p; // q is 4 elements after p  
printf("%d\n", i);
```

```
int j = p - q; // p is 4 elements ahead of q  
printf("%d\n", j);
```


Subtracting pointers: Undefined behavior

18

- Undefined behavior caused when:
 - ▣ Arithmetic performed on pointer that doesn't point to array element
 - ▣ Subtracting pointers that both don't point to elements of same array

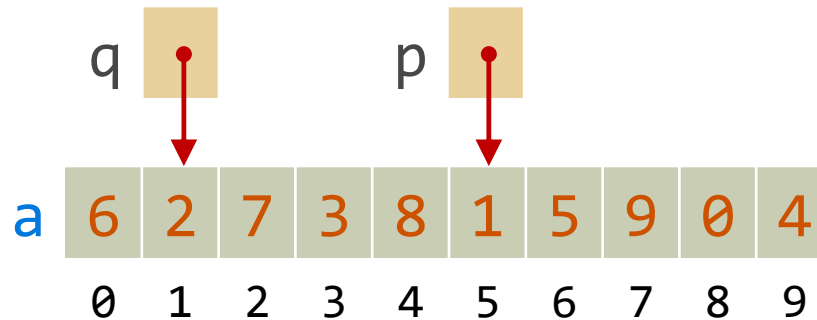
Comparing pointers (1 / 2)

19

- Pointers can be compared using relational operators ($<$, $<=$, $>$, $>=$) and equality operators ($==$ and $!=$)
 - ▣ *Meaningful only for pointers to elements of same array*
 - ▣ Expression evaluation depends on relative positions of array elements that pointers are pointing to

Comparing pointers (2/2)

20



```
int a[10] = {6, 2, 7, 3, 8, 1, 5, 9, 0, 4};
int *p = &a[5], *q = &a[1];
printf("%d | %d\n", p < q, p > q);
printf("%d | %d\n", *p < *q, *p > *q);
printf("%d | %d\n", p <= q, p >= q);
printf("%d | %d\n", *p <= *q, *p >= *q);
printf("%d | %d\n", p == q, p != q);
printf("%d | %d\n", *p == *q, *p != *q);
```

Processing arrays through pointers (1 / 4)

21

- Individual array elements visited using subscript operator

```
#define SIZE 10
int a[SIZE] = {11, 34, 82, 7, 64, 98, 47, 18, 79, 20};
int sum = 0;
for (int i = 0; i < SIZE; ++i) {
    sum += a[i];
}
```

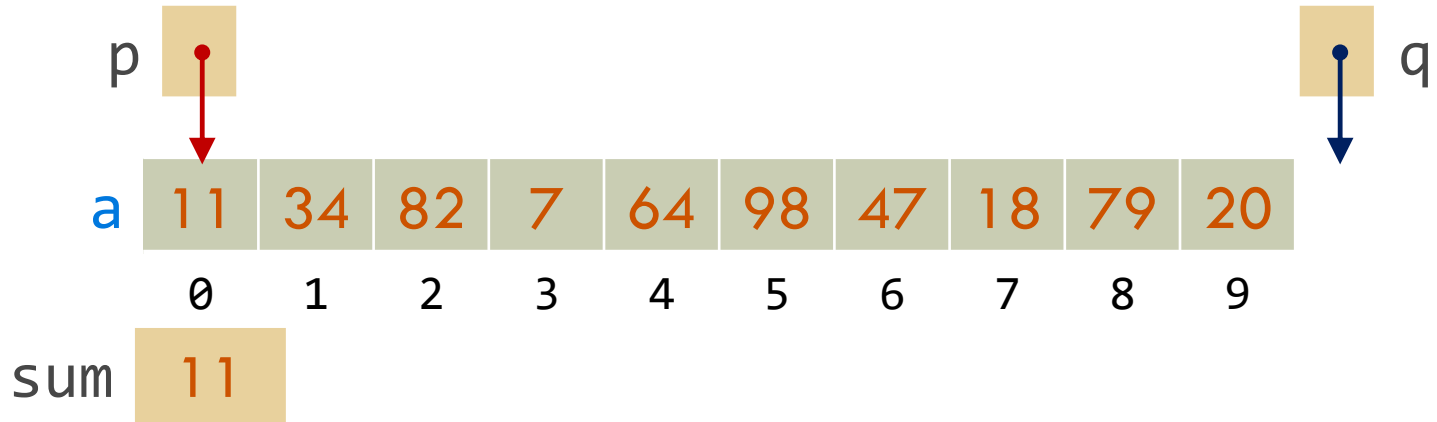
- Pointer arithmetic allows us to visit elements of array by repeatedly incrementing pointer variable

```
#define SIZE 10
int a[SIZE] = {11, 34, 82, 7, 64, 98, 47, 18, 79, 20}, sum = 0;
for (int *p = &a[0], *q = &a[SIZE]; p < q; ++p) {
    sum += *p;
}
```

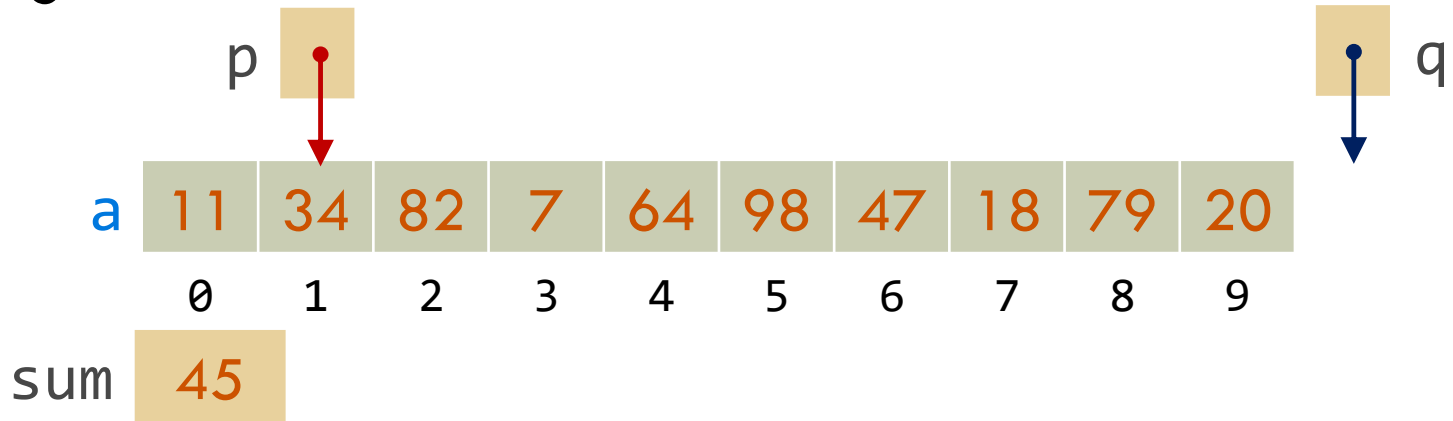
Processing arrays through pointers (2/4)

22

□ During 1st iteration



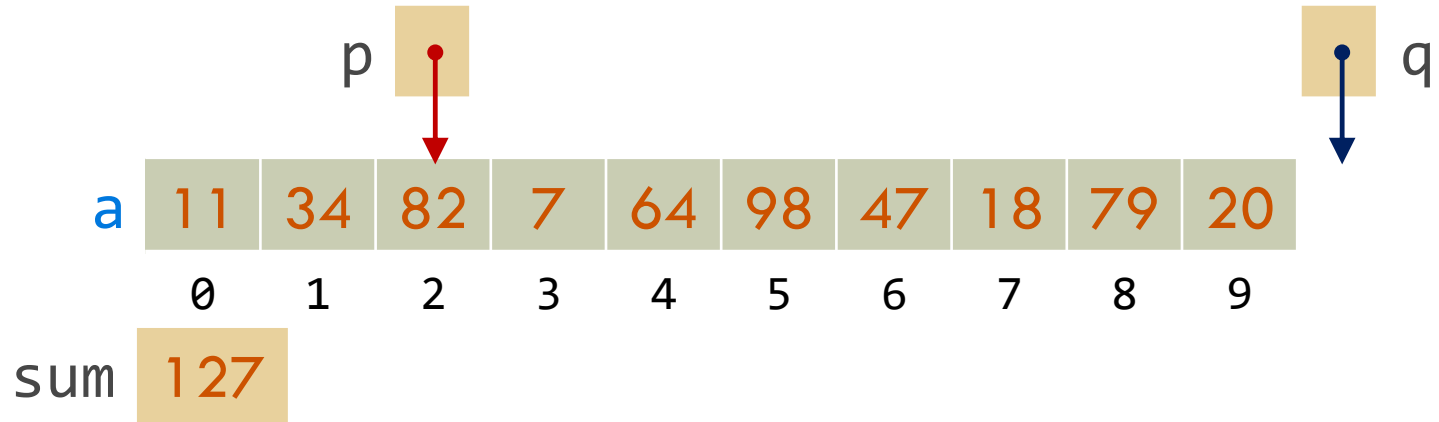
□ During 2nd iteration



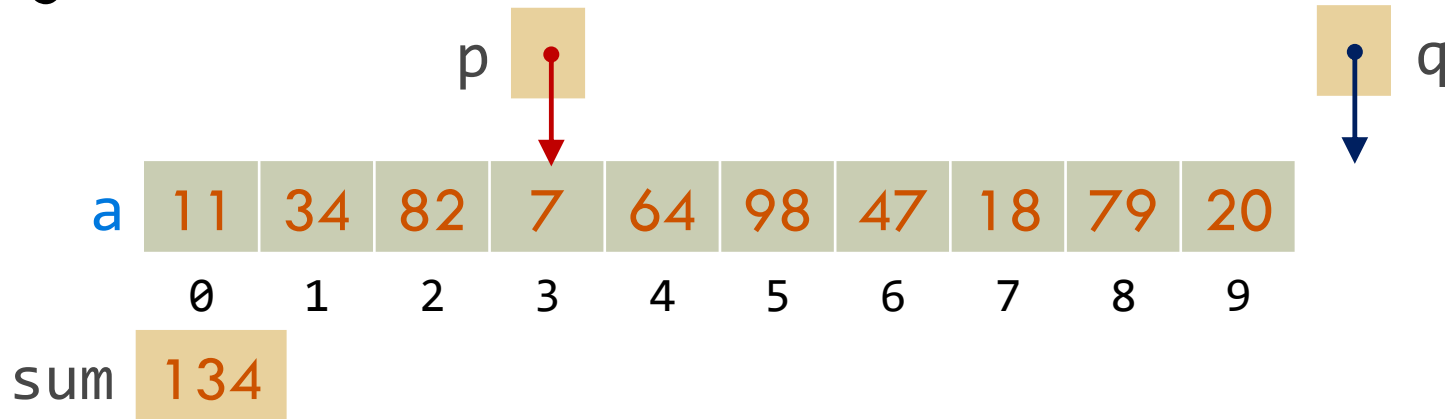
Processing arrays through pointers (3/4)

23

□ During 3rd iteration



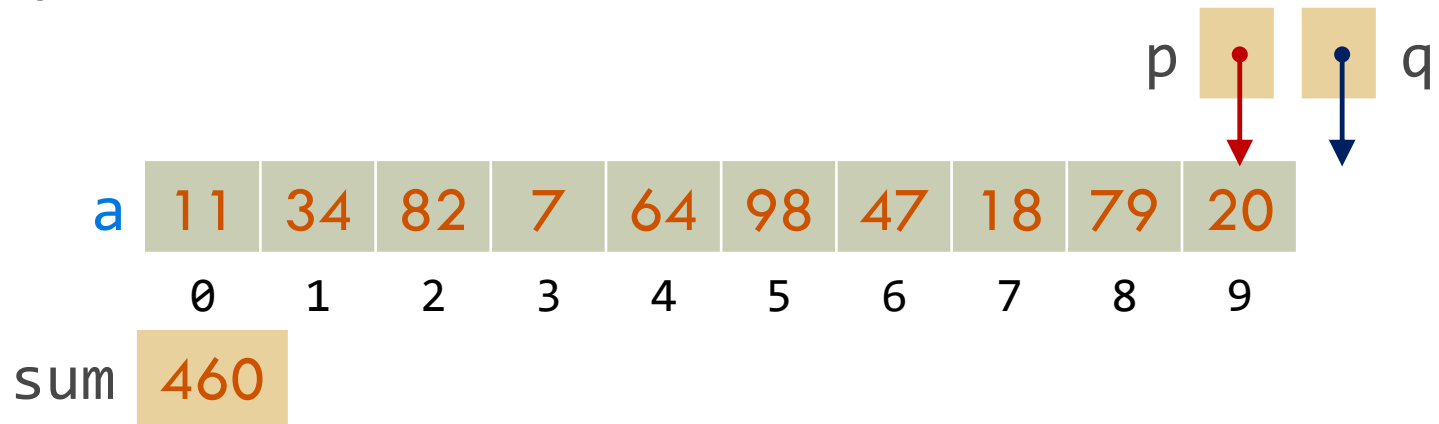
□ During 4th iteration



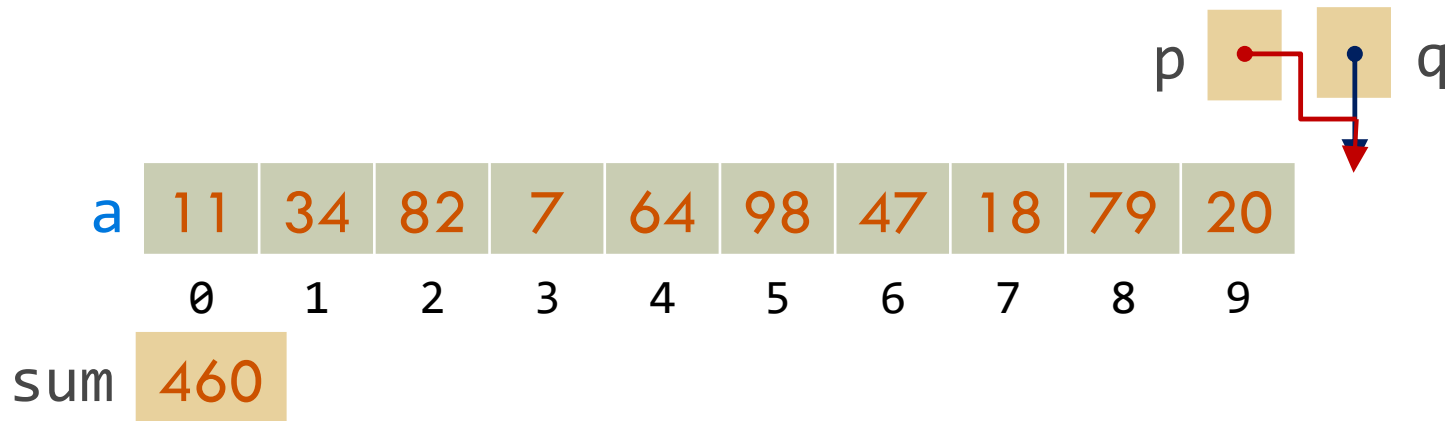
Processing arrays through pointers (4/4)

24

□ During 10th iteration



□ After 10th iteration



Compact pointer expressions (1 / 6)

25

- Combinations of indirection operator and pre/post-fix increment/decrement operators

high to low precedence order ↓	Operator	Associativity
	postfix ++	Left to right
	postfix --	
	prefix ++	Right to left
	prefix --	
	indirection *	
	address &	

Compact pointer expressions (2/6)

26

- Where do compact expressions come in handy?

```
#define SIZE 10
int a[SIZE] = {11, 34, 82, 7, 64, 98, 47, 18, 79, 20};
int sum = 0;
for (int i = 0; i < SIZE; ++i) {
    sum += a[i];
}
```

- Can rewrite as:

```
#define SIZE 10
int a[SIZE] = {11, 34, 82, 7, 64, 98, 47, 18, 79, 20};
int i = 0, sum = 0;
while (i < SIZE) {
    sum += a[i];
    ++i;
}
```

Compact pointer expressions (3/6)

27

□ Previously written as ...

```
#define SIZE 10
int a[SIZE] = {11, 34, 82, 7, 64, 98, 47, 18, 79, 20};
int i = 0, sum = 0;
while (i < SIZE) {
    sum += a[i];
    ++i;
}
```

□ Can further rewrite as:

```
#define SIZE 10
int a[SIZE] = {11, 34, 82, 7, 64, 98, 47, 18, 79, 20};
int i = 0, sum = 0;
while (i < SIZE) {
    sum += a[i++];
}
```

Compact pointer expressions (4/6)

28

□ Previously written as ...

```
#define SIZE 10
int a[SIZE] = {11, 34, 82, 7, 64, 98, 47, 18, 79, 20};
int i = 0, sum = 0;
while (i < SIZE) {
    sum += a[i++];
}
```

□ Can further rewrite using pointers as:

```
#define SIZE 10
int a[SIZE] = {11, 34, 82, 7, 64, 98, 47, 18, 79, 20};
int *p = &a[0], *q = &a[SIZE], sum = 0;
while (p < q) {
    sum += *p;
    ++p;
}
```

Compact pointer expressions (5/6)

29

- Previously written using pointers as ...

```
#define SIZE 10
int a[SIZE] = {11, 34, 82, 7, 64, 98, 47, 18, 79, 20};
int *p = &a[0], *q = &a[SIZE], sum = 0;
while (p < q) {
    sum += *p;
    ++p;
}
```

- Using compact pointer expression:

```
#define SIZE 10
int a[SIZE] = {11, 34, 82, 7, 64, 98, 47, 18, 79, 20};
int *p = &a[0], *q = &a[SIZE], sum = 0;
while (p < q) {
    sum += *p++;
}
```

Compact pointer expressions (6/6)

30

- If pointer **p** is pointing to array element, combinations of compact pointer expressions:

Expression	Operation	Affects	Read as
*p++	post increment	pointer	*(p++)
*p--	post decrement	pointer	*(p--)
*++p	pre increment	pointer	*(++p)
*--p	pre decrement	pointer	*(--p)
++*p	pre increment	pointee object	++(*p)
--*p	pre decrement	pointee object	--(*p)
(*p)++	post increment	pointee object	(*p)++
(*p)--	post decrement	pointee object	(*p)--

Compact pointer expressions:

Example (1 / 3)

31

□ Given

```
char str[] = "SeaToShiningC";  
char *p = &str[5];
```

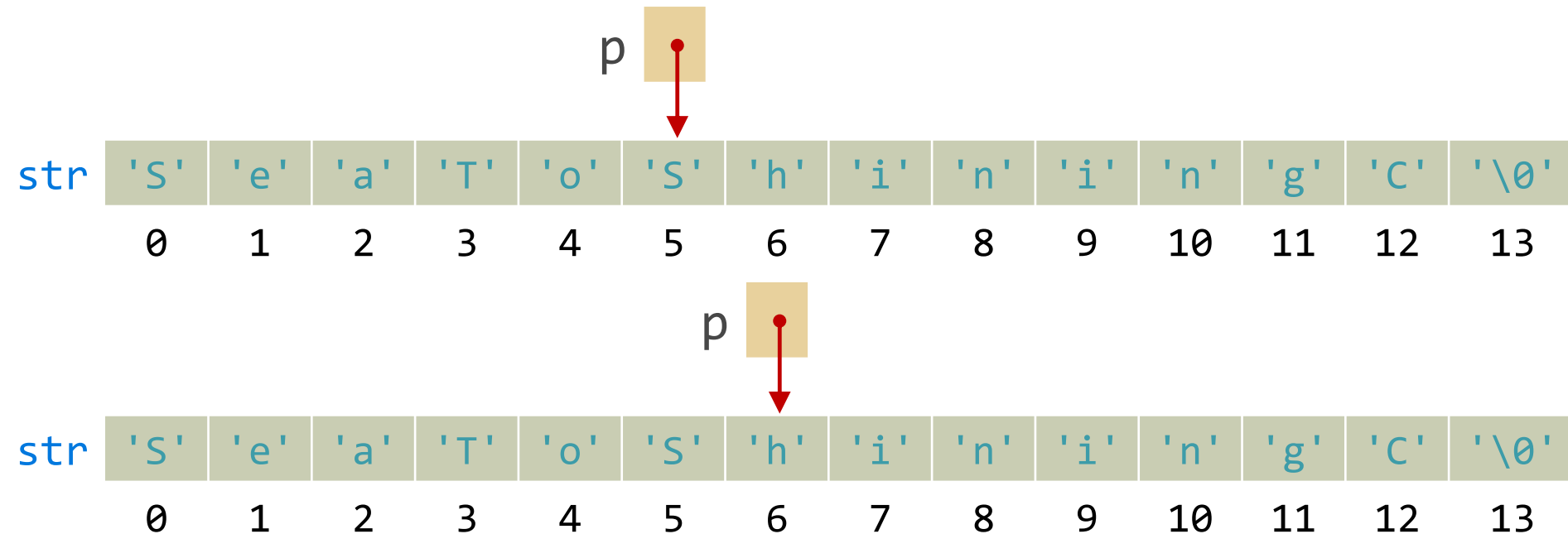
Expression	Result	Pointer pointing to	Resultant string
*p++	'S'	'h'	"SeaToShiningC"
*p--	'S'	'o'	"SeaToShiningC"
*++p	'h'	'h'	"SeaToShiningC"
*--p	'o'	'o'	"SeaToShiningC"
++*p	'T'	'T'	"SeaToThiningC"
--*p	'R'	'R'	"SeaToRhiningC"
(*p)++	'S'	'T'	"SeaToThiningC"
(*p)--	'S'	'R'	"SeaToRhiningC"

Compact pointer expressions:

Example (2/3)

32

```
char str[] = "SeaToShiningC", *p = &str[5];  
char ch = *p++;
```



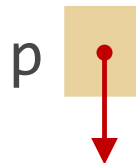
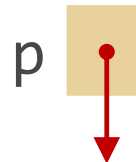
Expression	Result	Pointer pointing to	Resultant string
*p++	'S'	'h'	"SeaToShiningC"

Compact pointer expressions:

Example (3/3)

33

```
char str[] = "SeaToShiningC", *p = &str[5];  
char ch = ++*p;
```



str

'S'	'e'	'a'	'T'	'o'	'S'	'h'	'i'	'n'	'i'	'n'	'g'	'C'	'\0'
0	1	2	3	4	5	6	7	8	9	10	11	12	13

str

'S'	'e'	'a'	'T'	'o'	'T'	'h'	'i'	'n'	'i'	'n'	'g'	'C'	'\0'
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Expression	Result	Pointer pointing to	Resultant string
++*p	'T'	'T'	"SeaToThiningC"

Array name as pointer (1 / 7)

34

- Pointer arithmetic relates arrays and pointers:
 - ▣ *Pointer to array element can use pointer arithmetic to access other array elements*
- Another key relationship:
 - ▣ *Array name in expression evaluates to address of element with subscript 0*
 - ▣ Sole exception: when array name is operand to sizeof operator

Array name as pointer (2/7)

35

```
int a[5] = {3, 1, 5, 7, 9};
*a = 4;           // assign 4 to a[0]
*(a+3) = 23;      // assign 23 to a[3]
++*a;             // increments a[0] not a!!!

int *p = a;       // p points to a[0]
printf("%lu\n", sizeof(a)); // print 20 to stdout
printf("%lu\n", sizeof(p)); // prints 8 to stdout

a++; // error - cannot relocate array
```

Array name as pointer (3/7)

36

- Actually, subscript operator is syntactic sugarcoating
- C always replaces subscript operator with expression involving pointer arithmetic and dereference operator

Array name as pointer (4/7)

37

- In general, $a[i]$ is equivalent to $*(a+i)$
 - ▣ Both represent element with subscript i
- In general, $\&a[i]$ is equivalent to $\&*(a+i)$ which is equivalent to $(a+i)$
 - ▣ Both $\&a[i]$ and $(a+i)$ represent address of element with subscript i

Array name as pointer (5/7)

38

- Using array name as pointer to 1st array element makes loops easier to write because address of **&** operator is not required!!!

```
int a[10] = {11, 34, 82, 7, 64, 98, 47, 18, 79, 20}, sum = 0;
int *p = &a[0], *q = &a[10];
while (p < q) {
    sum += *p++;
}
```

```
int a[10] = {11, 34, 82, 7, 64, 98, 47, 18, 79, 20}, sum = 0;
int *p = a, *q = a+10; // & and [] operators not required
while (p < q) {
    sum += *p++;
}
```

Array name as pointer (6/7)

39

- Array names are non-modifiable
- Compiler treats array name as `const` pointer to first element of array
- Meaning of `const` pointer:
 - ▣ Pointer value cannot be modified to reference another address
 - ▣ Pointer can be used to modify value at address it references

Array name as pointer (7/7)

40

```
int a[10] = {11, 34, 82, 7, 64, 98, 47, 18, 79, 20};  
// find first array element with value 0  
int *q = a+10;  
while (*a != 0 && a < q) {  
    ++a; // flagged as compile-time error  
}  
// if a == q, then no element with value 0 ...
```

```
int a[10] = {11, 34, 82, 7, 64, 98, 47, 18, 79, 20};  
// find first array element with value 0  
// not a great loss that value of a cannot be changed  
// we just copy value of a into variable p ...  
int *p = a, *q = a+10;  
while (*p != 0 && p < q) {  
    ++p;  
}  
// if p == q, then no element with value 0 ...
```

Example: Array reversal using subscripts (1 / 8)

41

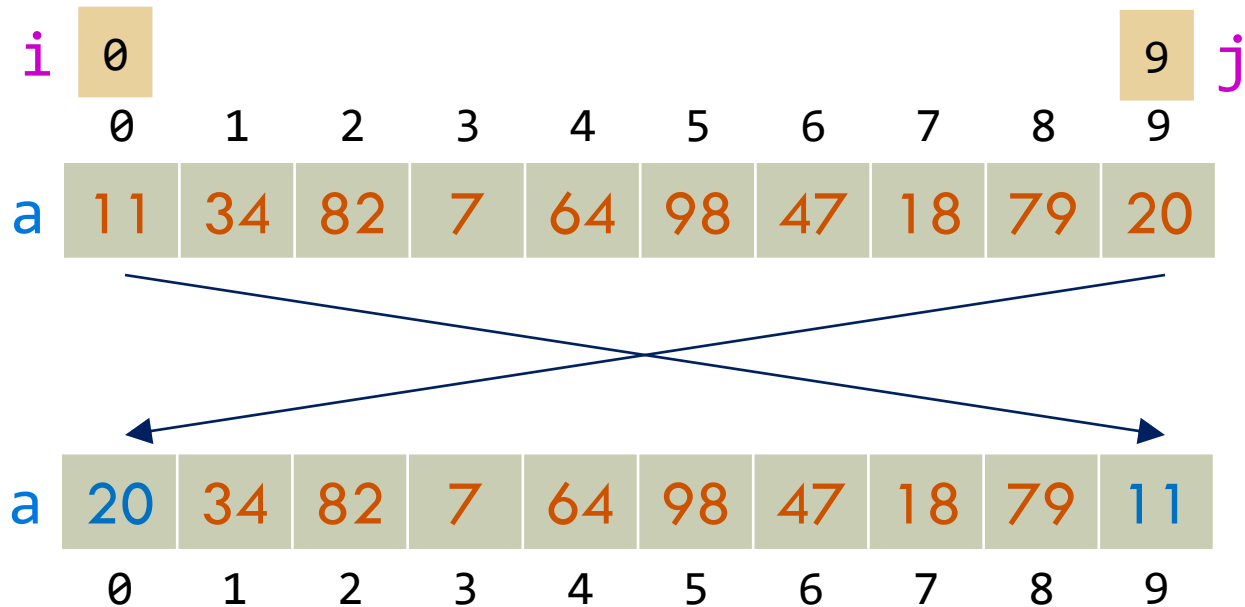
- We want to reverse elements of array **a** in-place using subscripts **i** and **j** and function **swap**

i	?									?	j
	0	1	2	3	4	5	6	7	8	9	
a	11	34	82	7	64	98	47	18	79	20	

a	11	34	82	7	64	98	47	18	79	20
	0	1	2	3	4	5	6	7	8	9

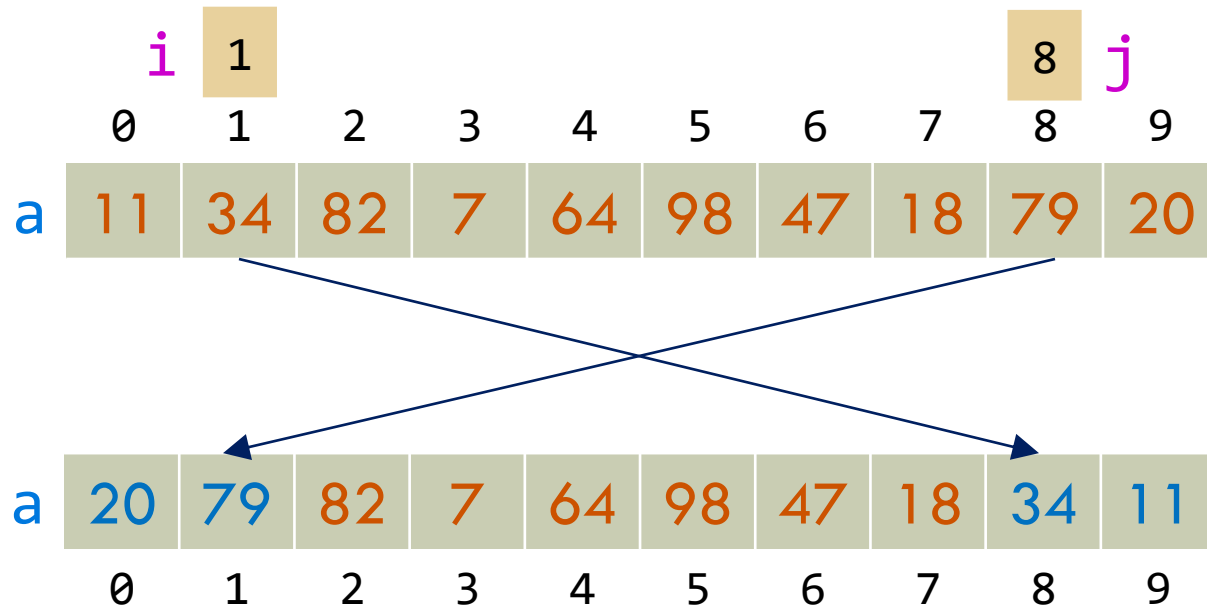
Example: Array reversal using subscripts (2/8)

42



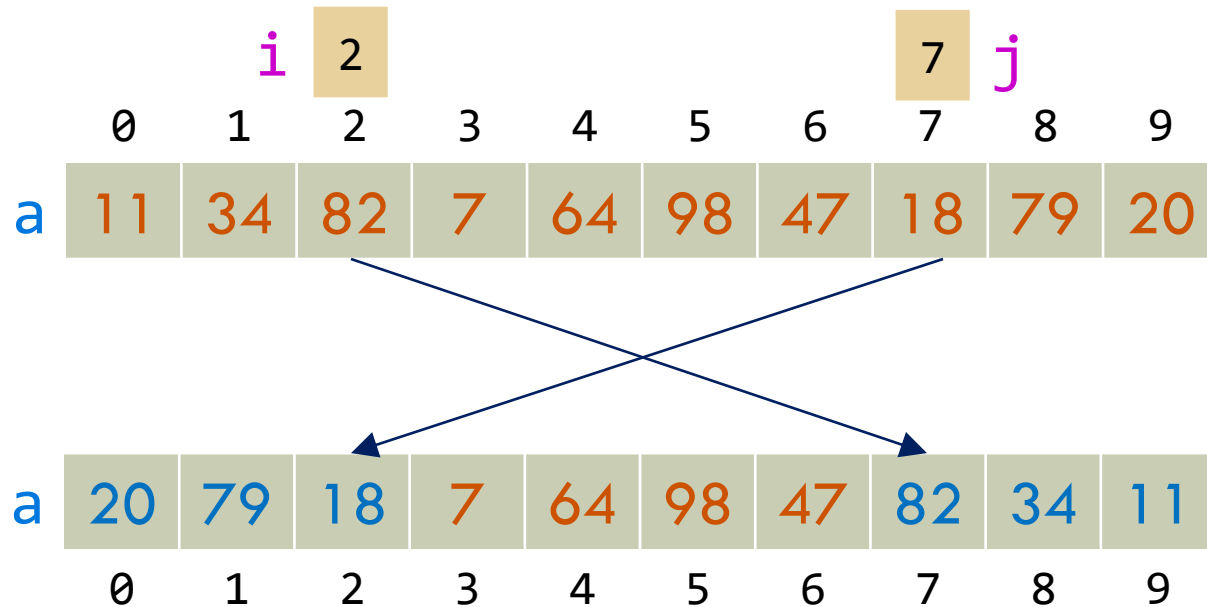
Example: Array reversal using subscripts (3/8)

43



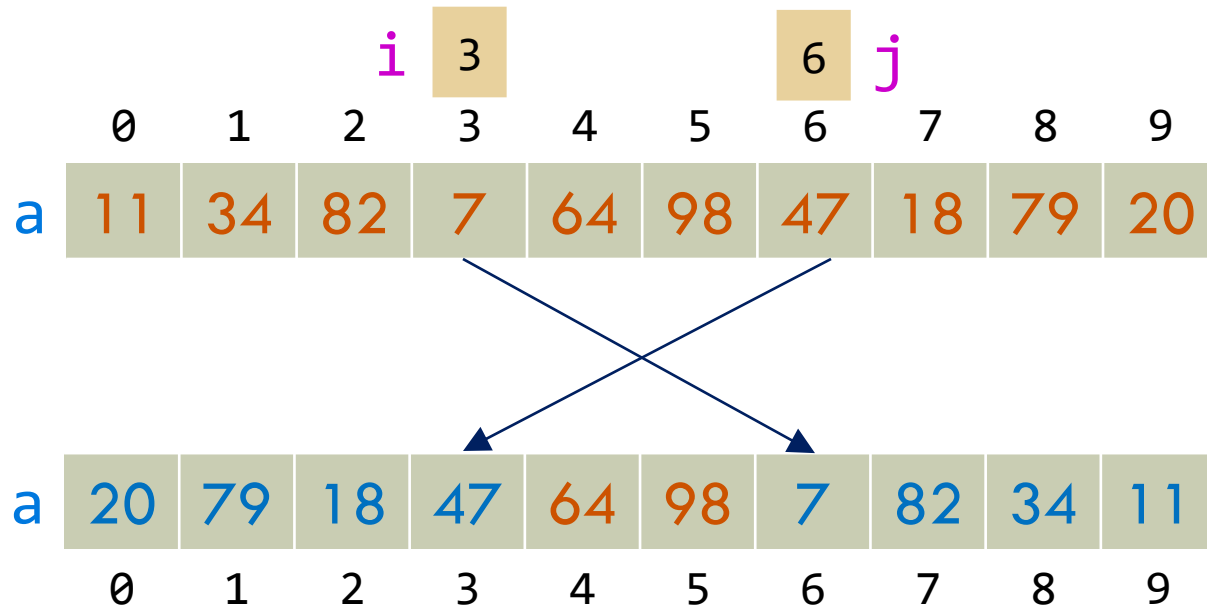
Example: Array reversal using subscripts (4/8)

44



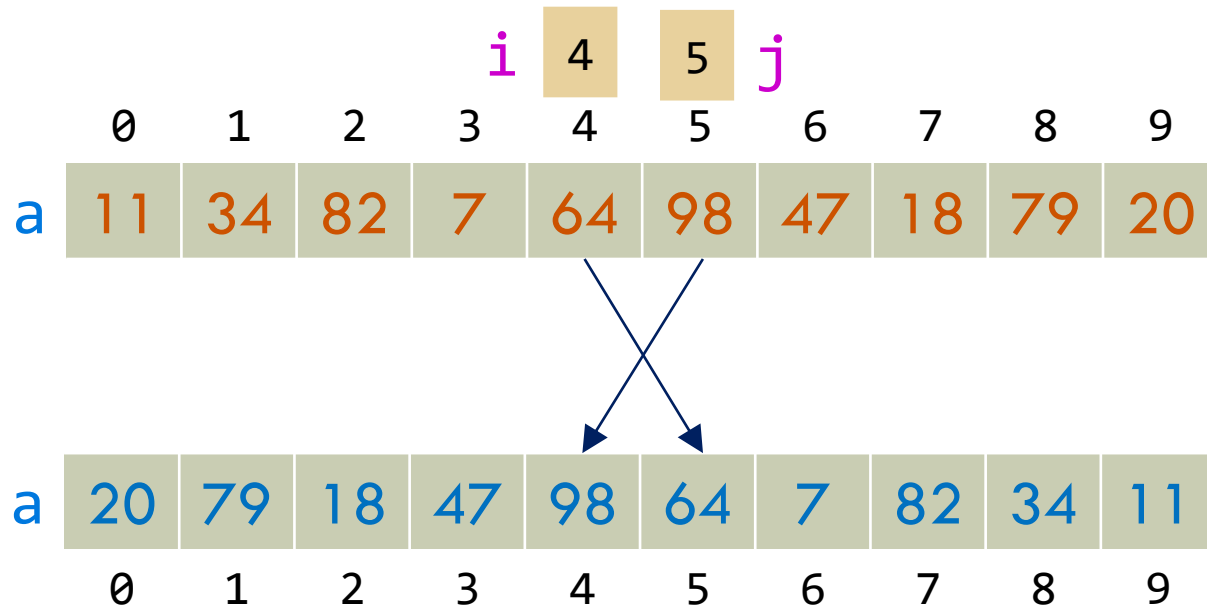
Example: Array reversal using subscripts (5/8)

45



Example: Array reversal using subscripts (6/8)

46



Example: Array reversal using subscripts (7/8)

47

				j	4	5	i			
	0	1	2	3	4	5	6	7	8	9
a	11	34	82	7	64	98	47	18	79	20

a	20	79	18	47	98	64	7	82	34	11
	0	1	2	3	4	5	6	7	8	9

Example: Array reversal using subscripts (8/8)

48

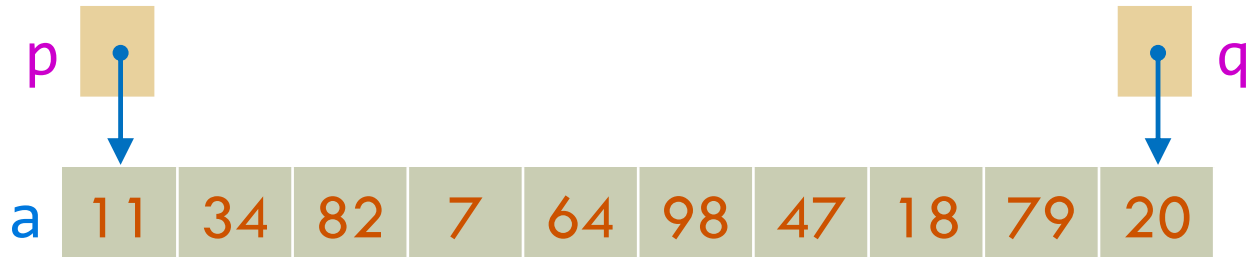
```
void swap(int *p, int *q) {
    int tmp = *p;
    *p = *q;
    *q = tmp;
}

int main(void) {
#define SIZE 10
    int a[SIZE] = {11, 34, 82, 7, 64, 98, 47, 18, 79, 20};
    for (int i = 0, j = SIZE-1; i < j; ++i, --j) {
        swap(&a[i], &a[j]);
    }
    for (int i = 0; i < SIZE; ++i) {
        printf("%d%c", a[i], i==SIZE-1 ? '\n' : ' ');
    }
    return 0;
}
```

Example: Array reversal using pointers (1 / 8)

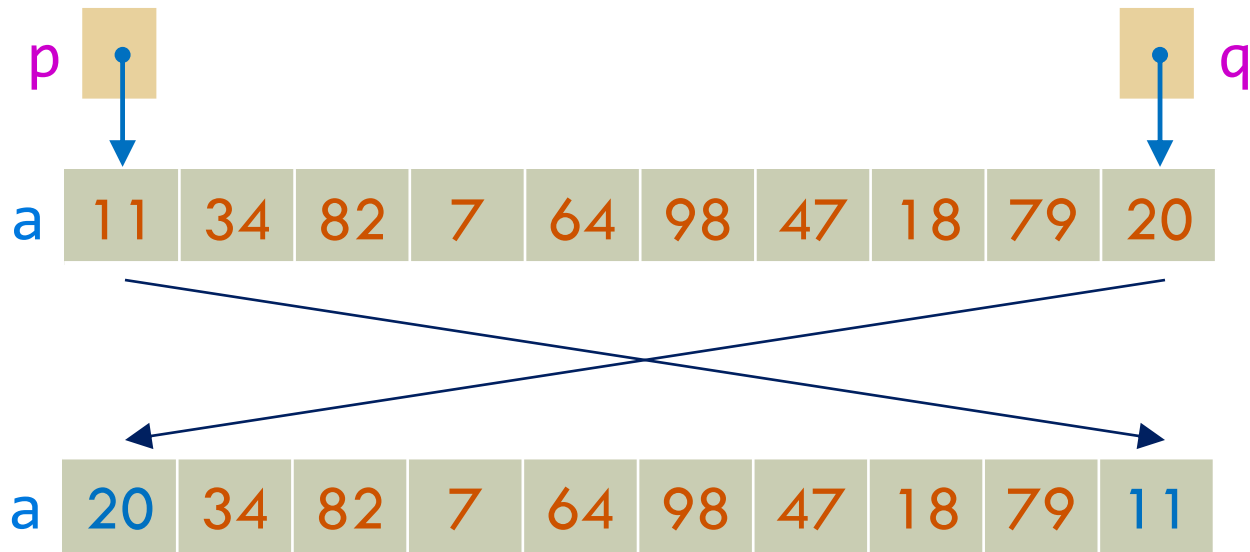
49

We want to reverse elements of array **a** in-place using pointers **p** and **q** and function **swap**



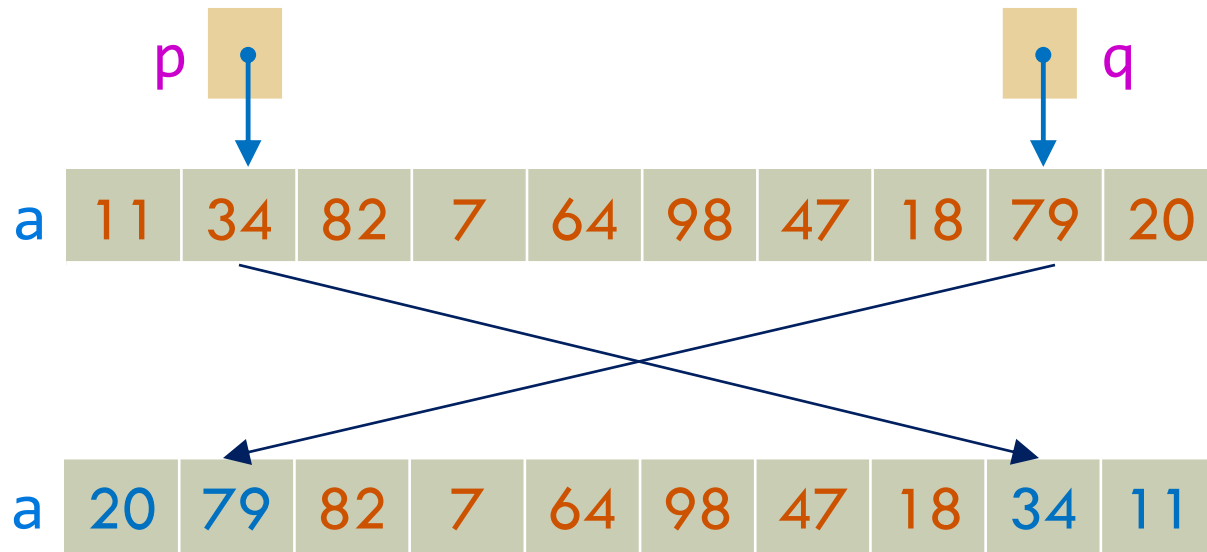
Example: Array reversal using pointers (2/8)

50



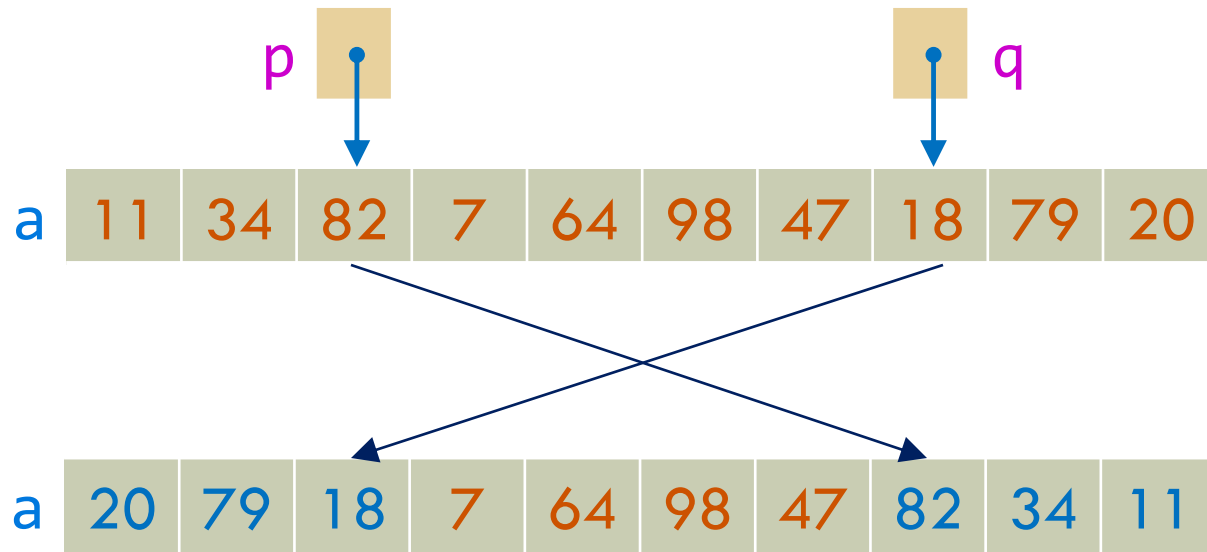
Example: Array reversal using pointers (3/8)

51



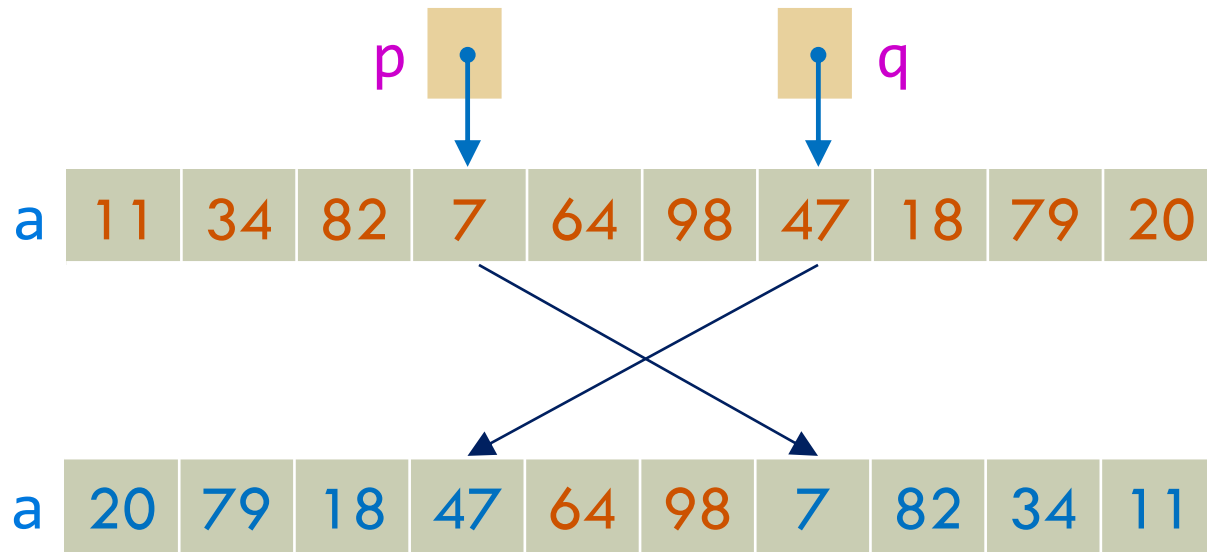
Example: Array reversal using pointers (4/8)

52



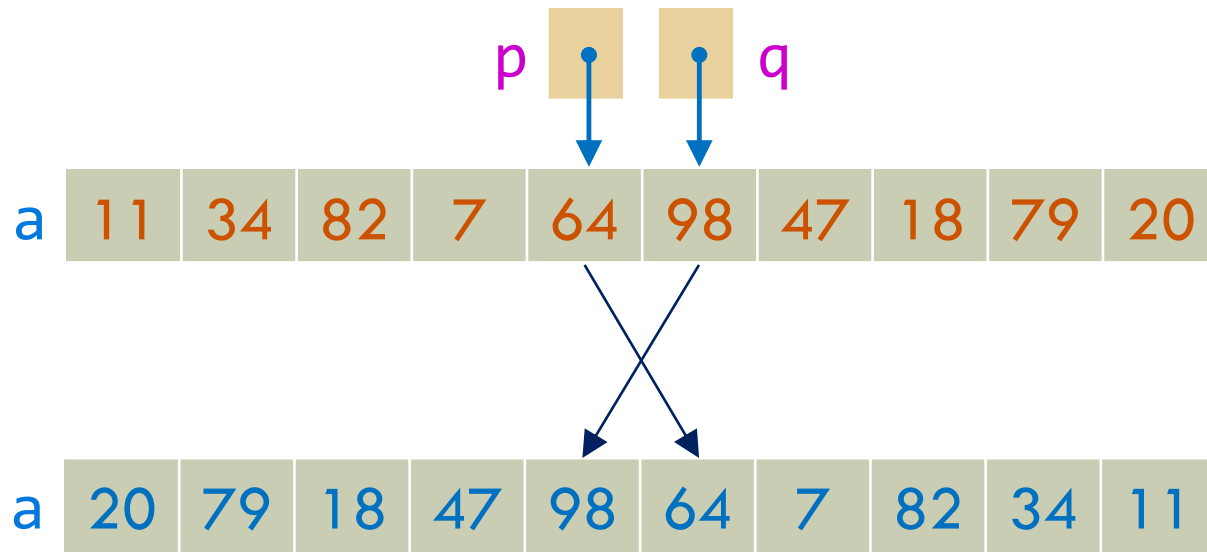
Example: Array reversal using pointers (5/8)

53



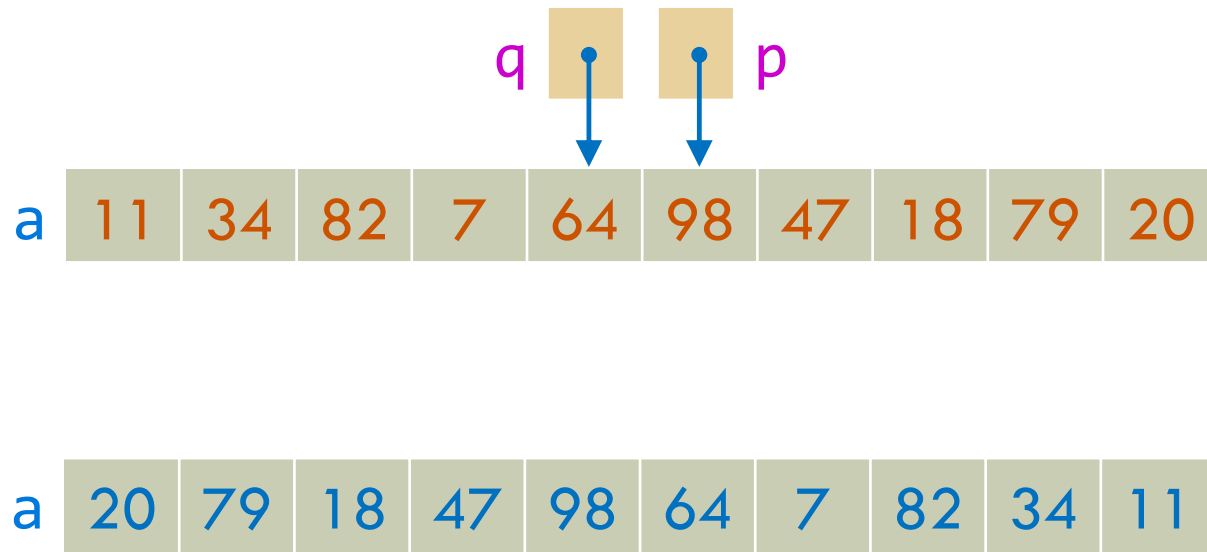
Example: Array reversal using pointers (6/8)

54



Example: Array reversal using pointers (7 / 8)

55



Example: Array reversal using pointers (8/8)

56

```
void swap(int *p, int *q) {
    int tmp = *p;
    *p = *q;
    *q = tmp;
}

int main(void) {
#define SIZE 10
    int a[SIZE] = {11, 34, 82, 7, 64, 98, 47, 18, 79, 20};
    for (int *p = a, *q = a+SIZE-1; p < q; ++p, --q) {
        swap(p, q);
    }
    for (int i = 0; i < SIZE; ++i) {
        printf("%d%c", a[i], i==SIZE-1 ? '\n' : ' ');
    }
    return 0;
}
```

Arrays as function arguments (1 / 2)

57

- We've previously seen arrays can be passed to functions

```
void zero_out_array(int p[], int size) {  
    for (int i = 0; i < size; ++i) {  
        p[i] = 0;  
    }  
}
```

```
// find subscript of element whose value is val  
int find(int p[], int size, int val) {  
    for (int i = 0; i < size; ++i) {  
        if (val == p[i]) return i;  
    }  
    return -1;  
}
```


Arrays as function arguments (2/2)

58

- Copy source array to destination array ...

```
void copy(int dst[], int src[], int size) {  
    int i = 0;  
    while (i < size) {  
        dst[i] = src[i];  
        ++i;  
    }  
}
```

Arrays as function arguments: pointer parameter (1 / 3)

59

- When array is passed to function, what is actually passed is *array's base address* which is also *address of array's first element*

```
#define SIZE 10
int a[SIZE] = {11, 34, 82, 7, 64, 98, 47, 18, 79, 20};
int total = summation(a, SIZE);
```

```
int summation(int p[], int size) {
    int sum = 0;
    for (int i = 0; i < size; ++i) {
        sum += p[i];
    }
    return sum;
}
```

Arrays as function arguments: pointer parameter (2/3)

60

- Syntactic sugar coating can be discarded by replacing array parameter with pointer parameter

```
#define SIZE 10
int a[SIZE] = {11, 34, 82, 7, 64, 98, 47, 18, 79, 20};
int total = summation(a, SIZE);
```

```
int summation(int *p, int size) {
    int sum = 0;
    for (int i = 0; i < size; ++i) {
        sum += *p++;
    }
    return sum;
}
```

Arrays as function arguments: pointer parameter (3/3)

61

- Compiler treats array parameter and pointer parameter as identical declarations
- It is left to author to decide which declaration to use

```
void zero_out_array(int p[], int size) {  
    for (int i = 0; i < size; ++i) {  
        p[i] = 0;  
    }  
}
```

```
void zero_out_array(int *p, int size) {  
    int *q = p + size;  
    while (p < q) {  
        *p++ = 0;  
    }  
}
```

Arrays as function arguments:

`const` type specifier (1 / 3)

62

- When ordinary variable is passed to function, its value is copied; changes to corresponding parameter don't affect variable
- In contrast, when argument is array name, function has copy of array's base address and can therefore modify individual subscripted variables of array

Arrays as function arguments:

const type specifier (2/3)

63

- This function can modify array by storing zero into each of its elements since array parameter **p** is pointer to first element of array in calling function ...

```
void zero_out_array(int p[], int size) {  
    for (int i = 0; i < size; ++i) {  
        p[i] = 0;  
    }  
}
```

Arrays as function arguments:

`const` type specifier (3/3)

64

- In contrast, this function *must not* modify its array parameter
 - ▣ Author can make compiler enforce this contract by adding type qualifier `const` to array declaration
 - ▣ In this case, parameter `p` is *pointer to read-only int*

```
int summation(int const p[], int size) {  
    int sum = 0;  
    for (int i = 0; i < size; ++i) {  
        sum += p[i];  
    }  
    return sum;  
}
```

Arrays as function arguments:

cost of passing arrays

65

- Inexpensive to pass arrays between functions – only need to pass base address and number of elements
- Time required to pass arrays is same irrespective of their size

Arrays as function arguments: range-based functions (1 / 2)

66

- Can pass array slices to functions
- Example: zero out array elements in half-open range specified by element with address **start** and up to but not including element with address **end**

```
void zero_out_array(int *start, int *end) {  
    while (start != end) {  
        *start++ = 0;  
    }  
}
```

Arrays as function arguments: range-based functions (2/2)

67

- Example: return sum of array elements in half-open range specified by element having address `start` and up to but not including element having address `end`

```
int summation(int const *start, int const *end) {  
    int sum = 0;  
    while (start != end) {  
        sum += *start++;  
    }  
    return sum;  
}
```

Summary

68

- Pointer arithmetic allows adding and subtracting of integers to/from pointers
- Pointers used as arguments to functions that modify values
- When arrays are sent to functions as arguments, a pointer to first element is passed to function
- `const` type qualifier can protect data