

The background of the slide features a series of horizontal, wavy black and white stripes that create a sense of movement. A dark gray rectangular box is positioned in the upper-left quadrant, containing the title text.

POINTERS & ARRAYS

```
int a[10], *p;
```

We can make `p` point to `a[0]` by writing

```
p = &a[0];
```

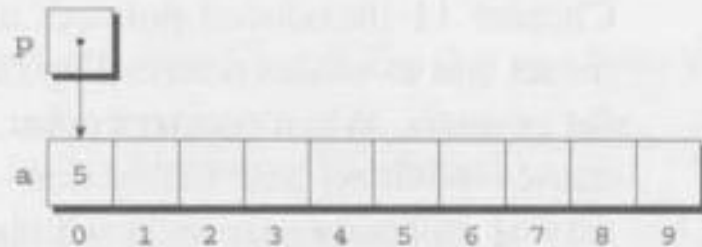
Graphically, here's what we've just done:



We can now access `a[0]` through `p`; for example, we can store the value 5 in `a[0]` by writing

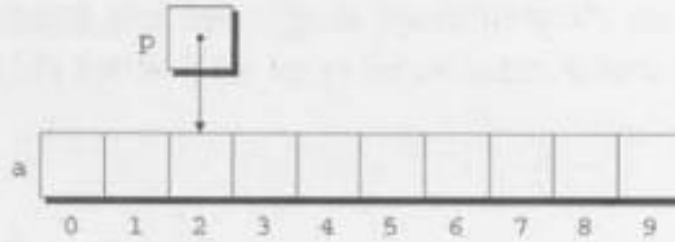
```
*p = 5;
```

Here's our picture now:

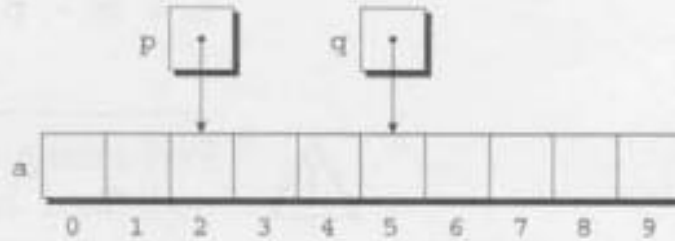


POINTER ARITHMETIC

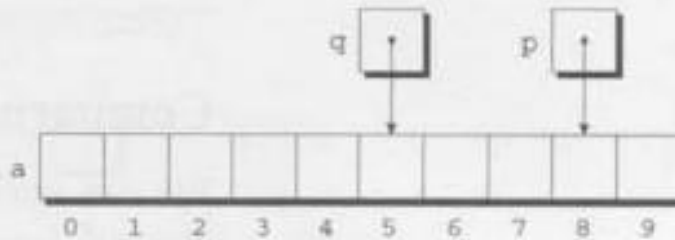
```
p = &a[2];
```



```
q = p + 3;
```



```
p += 6;
```



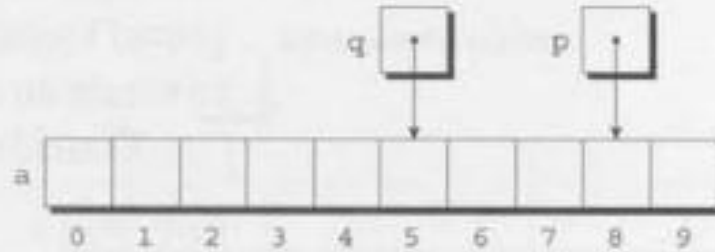
POINTER
ARITHMETIC:
ADDING AN
INTEGER TO
A POINTER

If p points to the array element $a[i]$, then $p - j$ points to $a[i - j]$. For example:

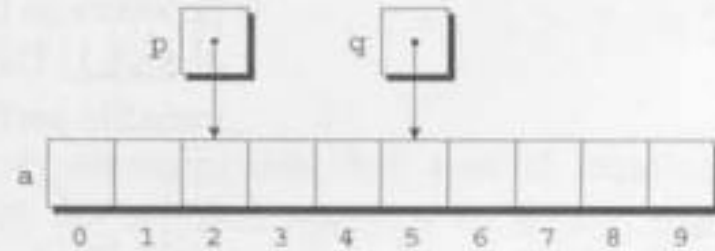
```
p = &a[8];
```



```
q = p - 3;
```



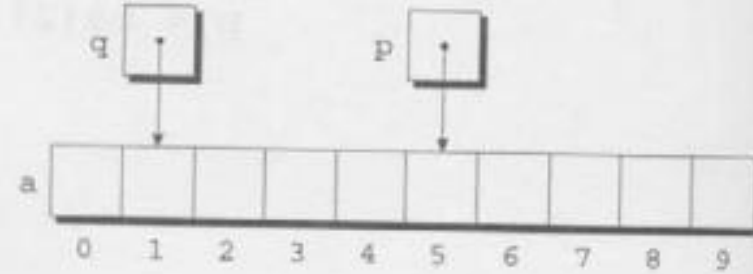
```
p -= 6;
```



POINTER ARITHMETIC: SUBTRACTING AN INTEGER TO A POINTER

```
p = &a[5];  
q = &a[1];
```

```
i = p - q;    /* i is 4 */  
i = q - p;    /* i is -4 */
```




POINTER ARITHMETIC: ADDING/SUBTRACTING
ONE POINTER TO ANOTHER

We can compare pointers using the relational operators (<, <=, >, >=) and the equality operators (== and !=). Using the relational operators to compare two pointers is meaningful only when both point to elements of the same array. The outcome of the comparison depends on the relative positions of the two elements in the array. For example, after the assignments

```
p = &a[5];  
q = &a[1];
```

the value of `p <= q` is 0 and the value of `p >= q` is 1.

COMPARING POINTERS



It's legal for a pointer to point to an element within an array created by a compound literal. A compound literal, you may recall, is a C99 feature that can be used to create an array with no name.

Consider the following example:

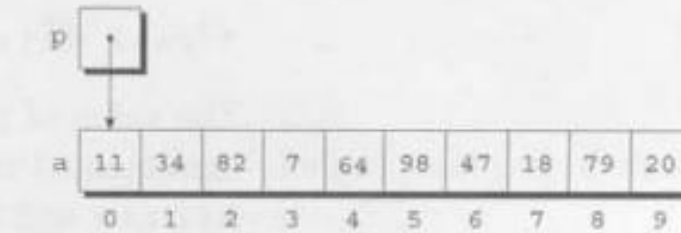
```
int *p = (int []){3, 0, 3, 4, 1};
```

`p` points to the first element of a five-element array containing the integers 3, 0, 3, 4, and 1. Using a compound literal saves us the trouble of first declaring an array variable and then making `p` point to the first element of that array:

```
int a[] = {3, 0, 3, 4, 1};  
int *p = &a[0];
```

POINTERS TO COMPOUND LITERALS

At the end of the first iteration:



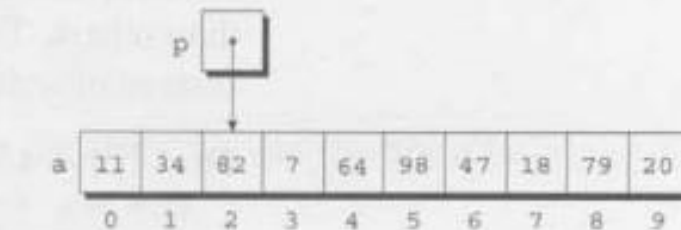
sum 11

At the end of the second iteration:




sum 45

At the end of the third iteration:



sum 127

USING POINTERS TO ARRAY PROCESSING



<i>Expression</i>	<i>Meaning</i>
<code>*p++</code> or <code>*(p++)</code>	Value of expression is <code>*p</code> before increment; increment <code>p</code> later
<code>(*p)++</code>	Value of expression is <code>*p</code> before increment; increment <code>*p</code> later
<code>*++p</code> or <code>*(++p)</code>	Increment <code>p</code> first; value of expression is <code>*p</code> after increment
<code>++*p</code> or <code>++(*p)</code>	Increment <code>*p</code> first; value of expression is <code>*p</code> after increment

COMBINING THE * AND ++ OPERATORS

PROGRAM **Reversing a Series of Numbers (Revisited)**

The `reverse.c` program of Section 8.1 reads 10 numbers, then writes the numbers in reverse order. As the program reads the numbers, it stores them in an array. Once all the numbers are read, the program steps through the array backwards as it prints the numbers.

USING AN
ARRAY NAME AS
A POINTER
& ARRAY
ARGUMENT



THE END