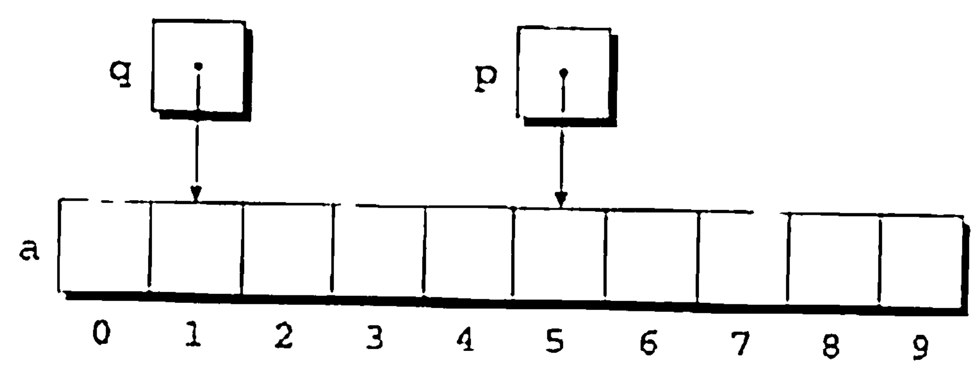
```
p = &a[5];
q = &a[1];
```



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

Performing arithmetic on a pointer that doesn't point to an array element causes undefined behavior. Furthermore, the effect of subtracting one pointer from another is undefined unless both point to elements of the *same* array.

### Comparing Pointers

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.

### C99 Pointers to Compound Literals

compound literals ➤9.3

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];
```

## 12.2 Using Pointers for Array Processing

Pointer arithmetic allows us to visit the elements of an array by repeatedly incrementing a pointer variable. The following program fragment, which sums the elements of an array a, illustrates the technique. In this example, the pointer variable