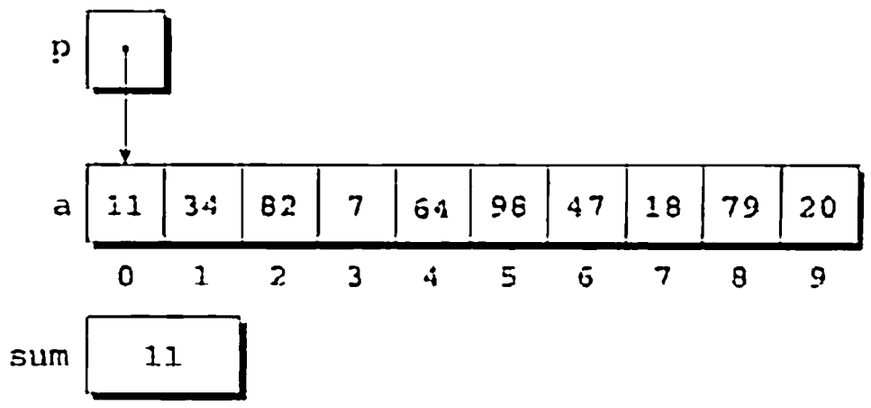
p initially points to a[0]. Each time through the loop, p is incremented; as a result, it points to a[1], then a[2], and so forth. The loop terminates when p steps past the last element of a.

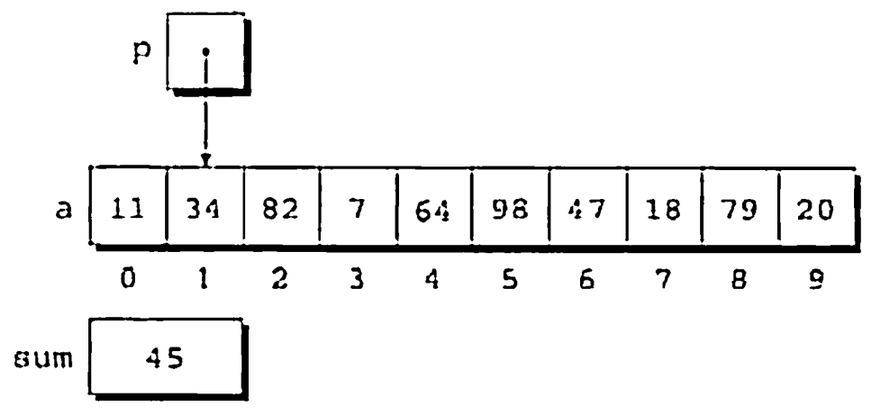
```
#define N 10
...
int a[N], sum, *p;
...
sum = 0;
for (p = &a[0]; p < &a[N]; p++)
  sum += *p;
```

The following figures show the contents of a, sum, and p at the end of the first three loop iterations (before p has been incremented).

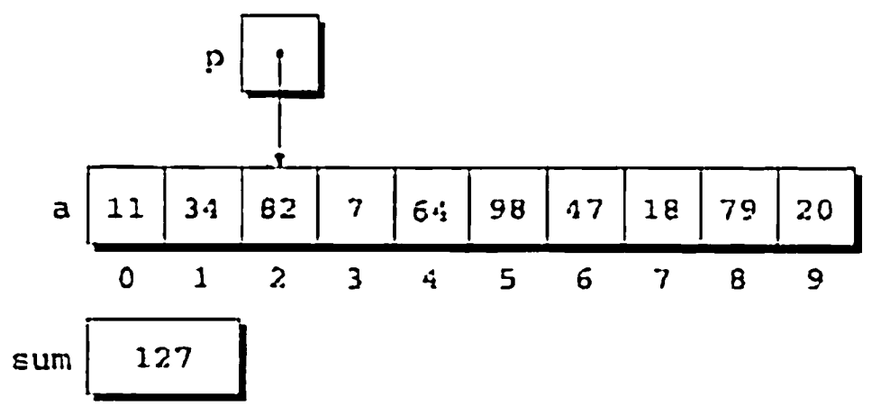
At the end of the first iteration:



At the end of the second iteration:



At the end of the third iteration:



The condition p < &a[N] in the for statement deserves special mention. Strange as it may seem, it's legal to apply the address operator to a[N], even though this element doesn't exist (a is indexed from 0 to N – 1). Using a[N] in this fashion is perfectly safe, since the loop doesn't attempt to examine its value. The body of the loop will be executed with p equal to &a[0], &a[1], ..., &a[N-1], but when p is equal to &a[N], the loop terminates.

We could just as easily have written the loop without pointers, of course, using subscripting instead. The argument most often cited in support of pointer arithmetic is that it can save execution time. However, that depends on the implementation—some C compilers actually produce better code for loops that rely on subscripting.

Q&A