Pointer and Array

Index

01. Pointer and Array

- Introduction
- Pointer Arithmetic
- Pointer Comparison
- Array <-> Pointer

01

Pointer and Array

Pointer and Array

Introduction

The arithmetic, addition, and subtraction, could be performed on pointers to array elements

- ➢ It provides an alternative way of processing arrays in which pointers take the place of array subscripts

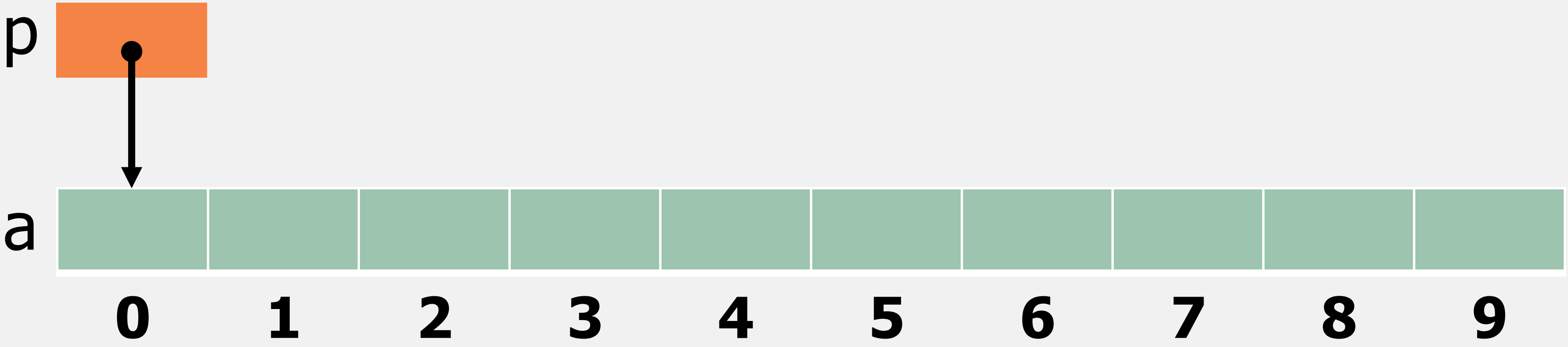
Therefore, understanding the relationship between pointer and array is very critical

Pointer and Array

Pointer Arithmetic

If a pointer points to an array

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

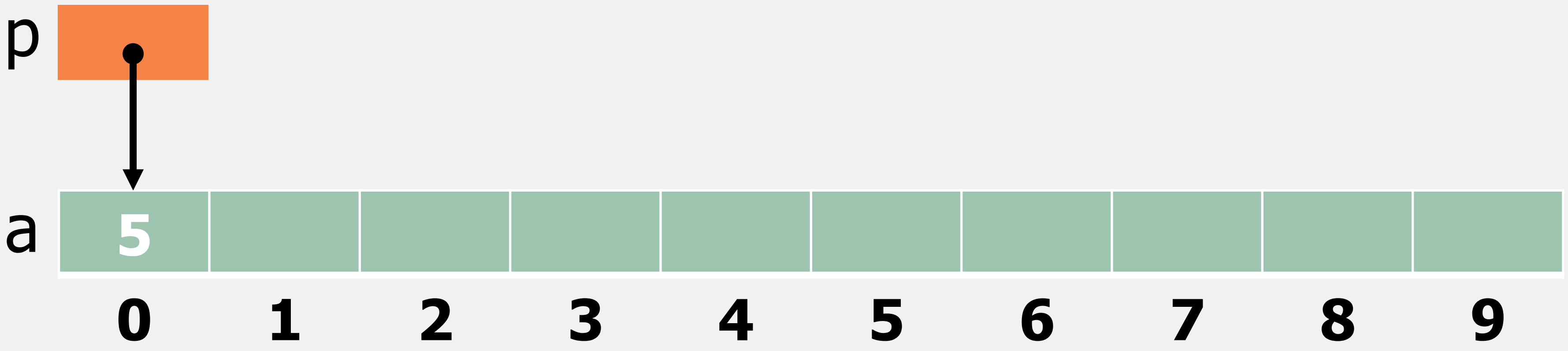


Pointer and Array

Pointer Arithmetic

If a pointer points to an array

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



Pointer and Array

Pointer Arithmetic

If p points to an element of an array a, the other elements of a can be accessed by performing pointer arithmetic (or address arithmetic) on p

C supports only three forms of pointer arithmetic

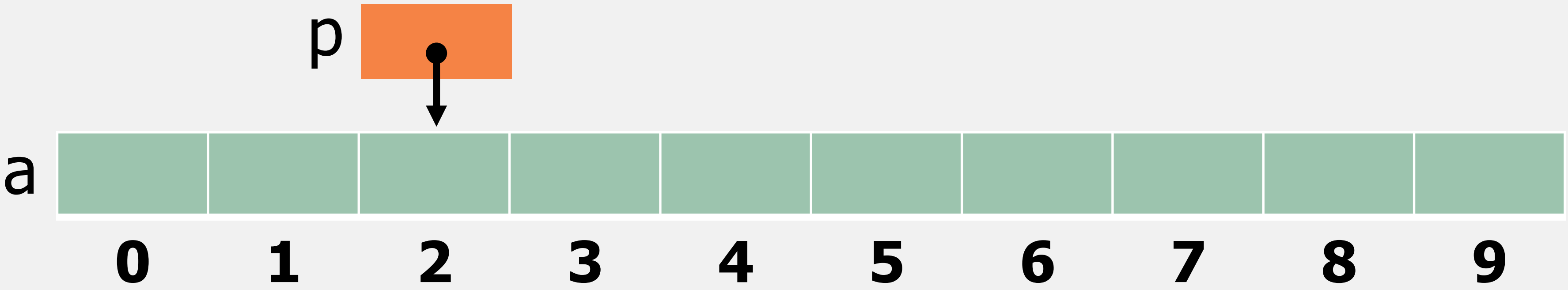
- Adding an integer to a pointer
- Subtracting an integer from a pointer
- Subtracting one pointer from another

Pointer and Array

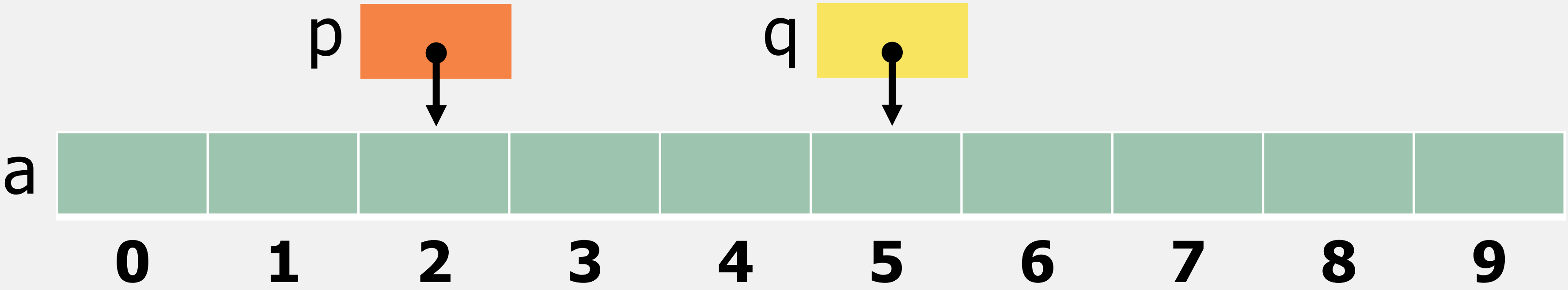
Pointer Arithmetic - Adding Integer to Pointer

int a[10], *p, *q, i;

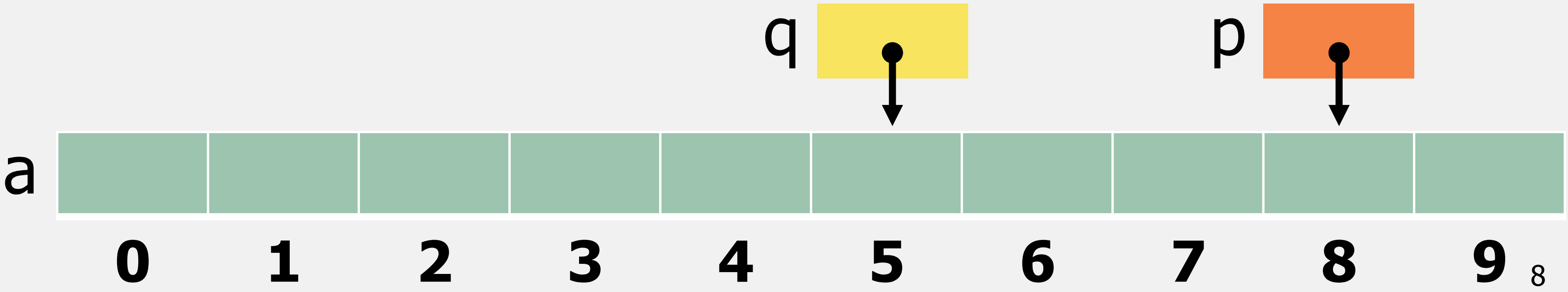
p = &a[2];



q = p + 3;



p+ = 6;

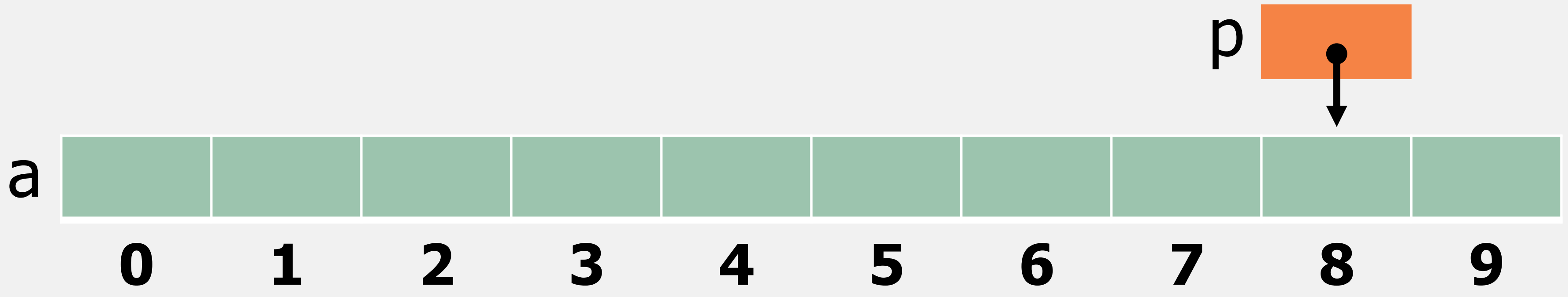


Pointer and Array

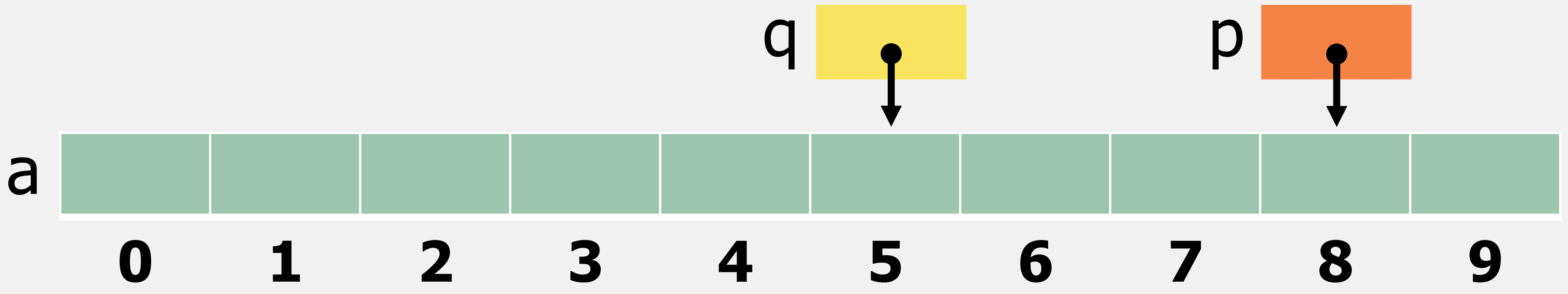
Pointer Arithmetic - Subtracting Integer to Pointer

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

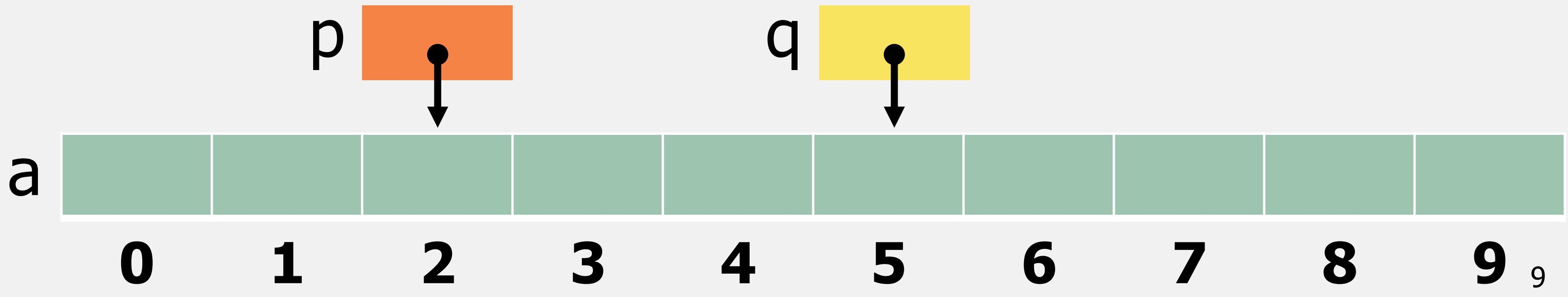
```
p = &a[8];
```



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



```
p -= 6;
```

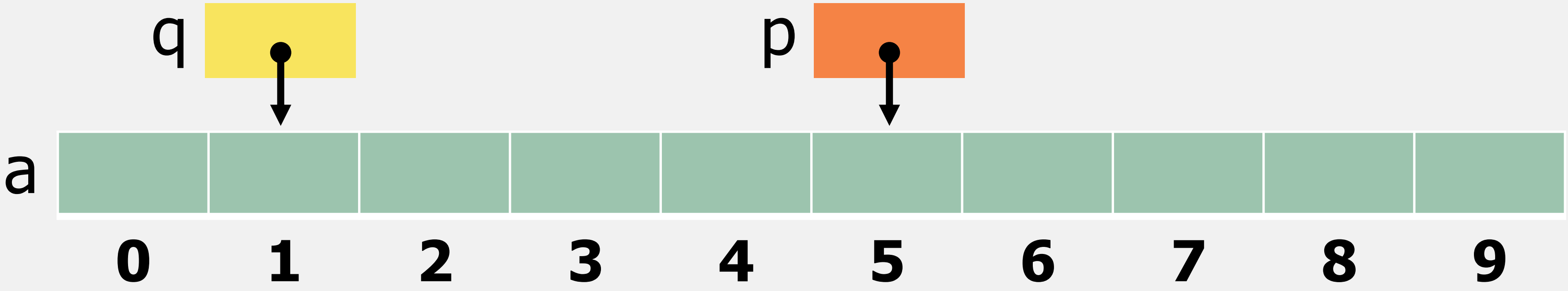


Pointer and Array

Pointer Arithmetic - Subtracting Pointer from Another

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


p = &a[5];
q = &a[1];
i = p - q;      // i = 4
i = q - p;      // i = -4
```



Pointer and Array

Pointer Comparison

Pointers can be compared using the relational operations and the equality operators

- <, <=, >, >=
  - Using the relational operators is meaningful only for pointers to elements of the same array
- == and !=

The outcome of the comparison depends on the relative positions of the two elements in the array

After the assignments

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

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

Pointer and Array

Pointer Comparison

It's legal for a pointer to point to an element within an array created by a compound literal such as

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

But, using a compound literal makes us the trouble of first declaring an array variable and then making up point to the first element of that array

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

Pointer and Array

Pointer Comparison

Suppose that the following declarations are in effect:

```
int a[] = {5, 15, 34, 54, 14, 2, 52, 72};
int *p = &a[1], *q = &a[5];
```

(a) What is the value of *(p+3)? 14

(b) What is the value of *(q-3)? 34

(c) What is the value of q-p? 4

(d) Is the condition p < q true or false? Y

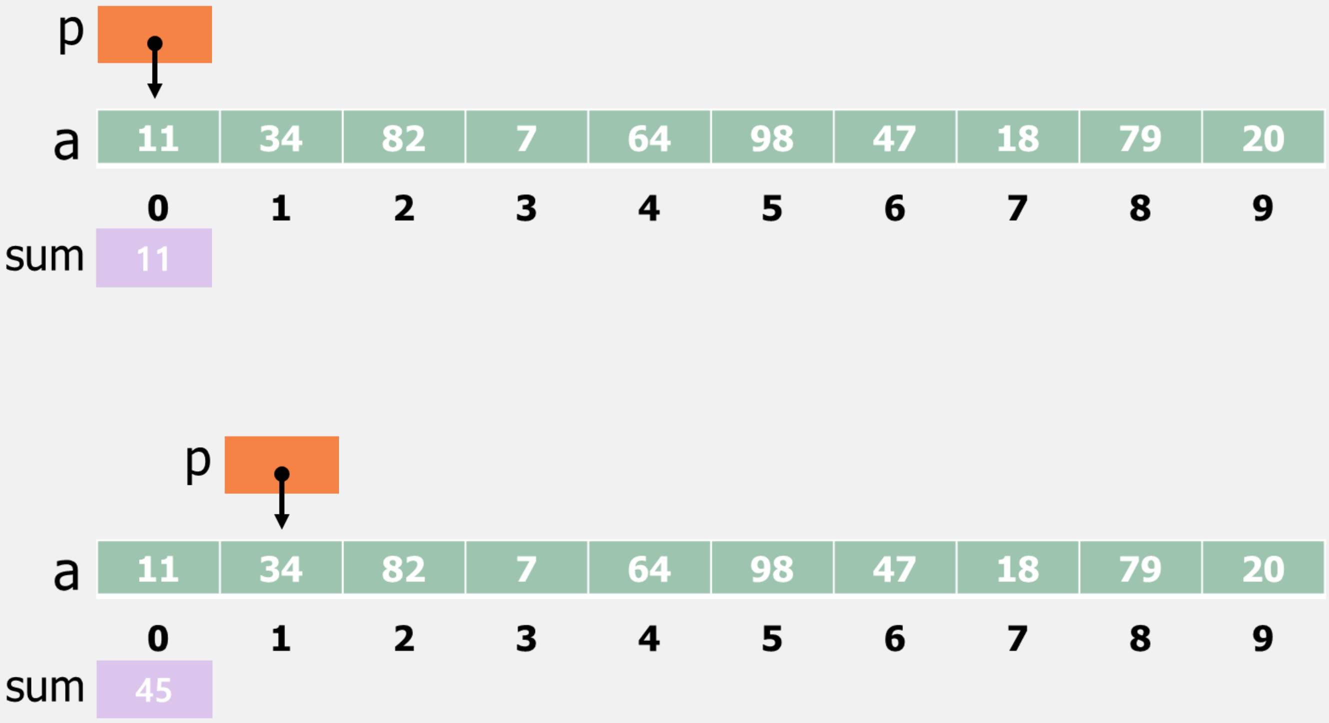
(e) Is the condition *p < *q true or false? N

Pointer and Array

Array <-> Pointer

Pointer arithmetic allows us to visit the elements of an array by incrementing a pointer variable repeatedly

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



Pointer and Array

Array <-> Pointer

The * and ++ operators are often combined in C

```
a[i++] = j;
   ‖
p = &a[i];
*p++ = j;
   ‖
p = &a[i];
*(p++) = j;
```

Because the postfix version ++ takes precedence over *

Pointer and Array

Array <-> Pointer

Possible combinations of * and ++

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

Pointer and Array

Array <-> Pointer

The most common combinations of * and ++ is *p++, which is handy in loops

```
for (p = &a[0]; p < &a[N];
   p++)
     sum += *p;
```

‖

```
p = &a[0];
while (p < &a[N])
     sum += *p++;
```

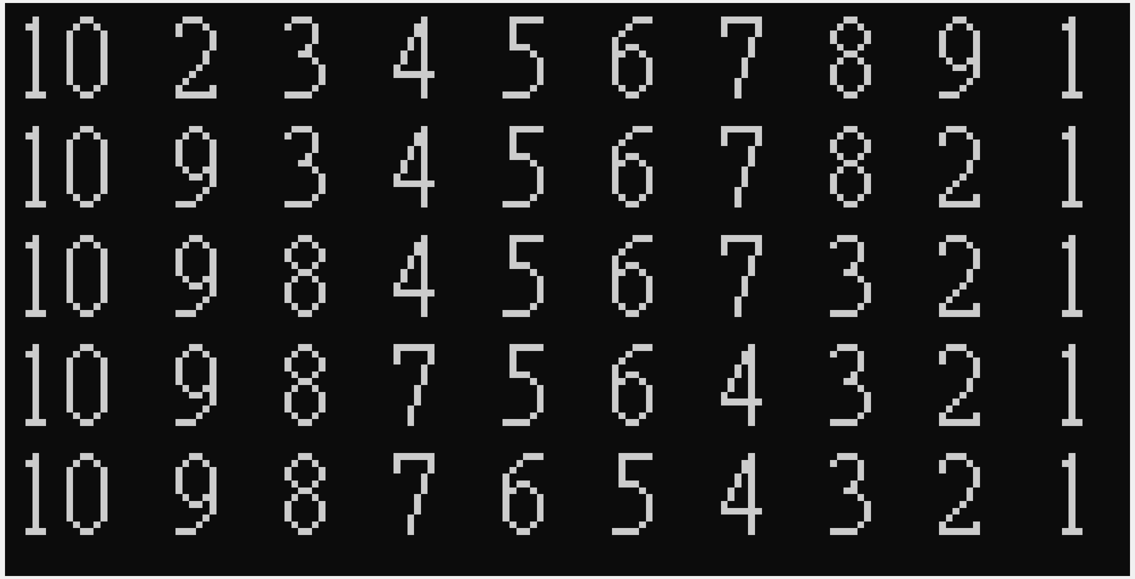
The * and -- operators mix in the same way as * and ++

Pointer and Array

Array <-> Pointer

What will be the contents of the a array after the following statements are executed?

```
#define N 10
int a[N] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int *p = &a[0], *q = &a[N-1], temp;
while (p < q)
{
        temp = *p;
        *p++ = *q;
        *q-- = temp;
}
```



```
10  2  3  4  5  6  7  8  9  1
10  9  3  4  5  6  7  8  2  1
10  9  8  4  5  6  7  3  2  1
10  9  8  7  5  6  4  3  2  1
10  9  8  7  6  5  4  3  2  1
```

Pointer and Array

Array <-> Pointer

Pointer arithmetic is one way in which arrays and pointer are related

Another critical relationship

- ➢ The name of an array can be used as a pointer to the first element in the array

This relationship simplifies pointer arithmetic and makes both arrays and pointers more versatile

If the array a is declared as

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

Using a as a pointer

```
*a = 7;            //stores 7 in a[0]
*(a+1) = 12;       //stores 12 in a[1]
```

Pointer and Array

Array <-> Pointer

In fact, the array name can serve as a pointer makes it easier to write loops that step through an array

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

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

Pointer and Array

Array <-> Pointer

Although an array name can be used as a pointer, it's not possible to assign it a new value

```
#define N 10
int a[N];
while (*a != 0)
    a++;                //Error
```

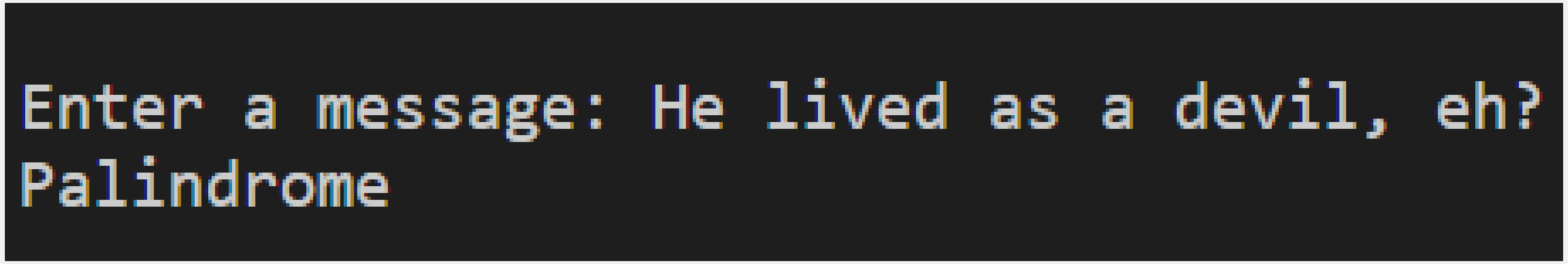
We can use a pointer variable to point to a and change it

```
#define N 10
int a[N];
int *p = a;
while (*p != 0)
    p++;
```

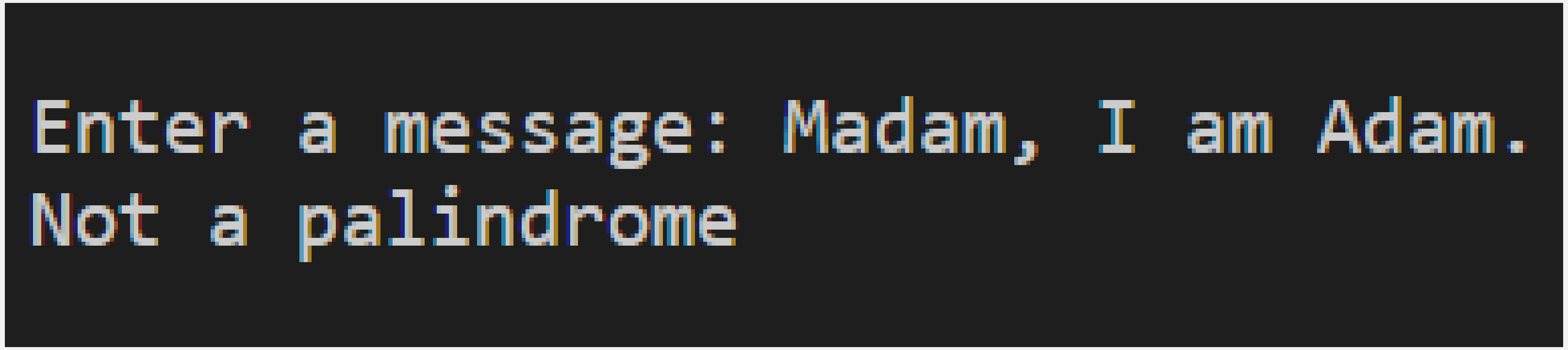
Pointer and Array

Array <-> Pointer

Write a program that reads a message and checks whether it's a palindrome or not using pointer and function "isalpha"



```
Enter a message: He lived as a devil, eh?
Palindrome
```

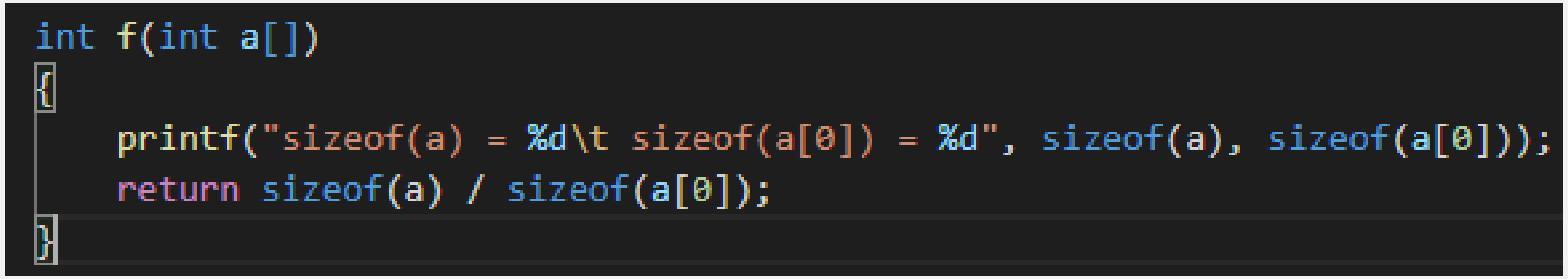


```
Enter a message: Madam, I am Adam.
Not a palindrome
```

Pointer and Array

Array <-> Pointer

Now you can understand why the following code can't compute the length of the array argument



```
int f(int a[])
{
    printf("sizeof(a) = %d\t sizeof(a[0]) = %d", sizeof(a), sizeof(a[0]));
    return sizeof(a) / sizeof(a[0]);
}
```

sizeof(a) = 4   sizeof(a[0]) = 4

In fact, an array argument is treated as a pointer has some important consequences

Pointer and Array

Array <-> Pointer

Consequence 1

- When an ordinary variable is passed to a function, its value is copied and any changes to the corresponding parameter don't affect the variable

In contrast, an array used as an argument isn't protected against change

```
void initial_zeros(int a[], int n)
{
    int i;

    for (i = 0; i < n; i++)
      a[i] = 0;
}
```

Pointer and Array

Array <-> Pointer

To ensure that an array parameter won't be changed, the word const can be used in its declaration

```
void initial_zeros(const int a[], int n)
{
      int i;

      for (i = 0; i < n; i++)
        a[i] = 0;                    //Error: assignment of read-only location '*a'
}
```

If *const* is present, the compiler will check that no assignment to an element of a appears in the body of initial_zeros

Pointer and Array

Array <-> Pointer

Consequence 2

- The time required to pass an array to a function doesn't depend on the size of the array
- Actually, there is no penalty for passing a large array, since no copy of the array is made

Consequence 3

- An array parameter can be declared as a pointer if desired
- initial_zeros could be defined as

```
void initial_zeros(int *a, int n)
{
   …
}
```

Pointer and Array

Array <-> Pointer

Consequence 4

- A function with an array parameter can be passed an array "slice" - a sequence of consecutive elements

```
void initial_zeros(int *a, int n)
{
    …
}

initial_zeros(&b[5], 10);
```

From element 5 to 14 of array b

Pointer and Array

Array <-> Pointer

C allows us to subscript a pointer as though it were an array name

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

The compiler treats p[i] as *(p+i)

Pointer and Array

Array <-> Pointer

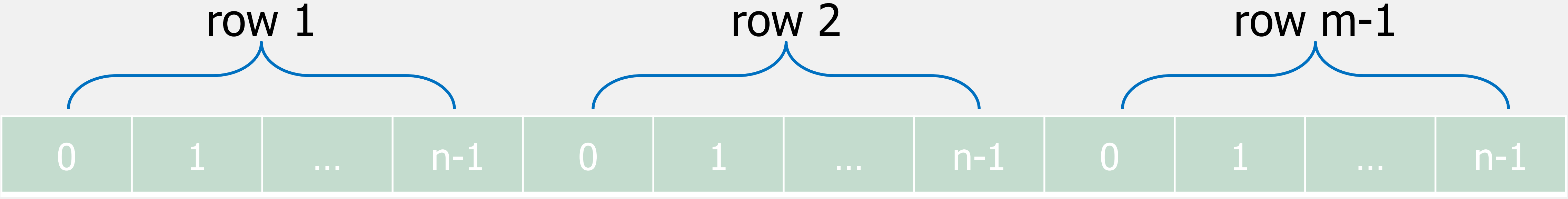
Suppose that a is a one-dimensional array and p is a pointer variable. Assuming that the assignment p = a has just bee performed, which of the following expressions are illegal? Of the remaining expressions, which are true?

(a) p == a[0]	Illegal
(b) p == &a[0]	Legal, true
(c) *p == a[0]	Legal, true
(d) p[0] == a[0]	Legal, true

Pointer and Array

Array <-> Pointer

As pointers can point to elements of one-dimensional arrays, they can also point to elements of multidimensional arrays



row 1 | row 2 | row m-1

0	1	…	n-1	0	1	…	n-1	0	1	…	n-1

If p initially points to the element in row 0, column 0, every element can be visited by incrementing p repeatedly

Pointer and Array

Array <-> Pointer

Consider the problem of initializing all elements of the following array to zero

```
int a[Num_Rows][Num_Cols];
```

Using nested for loops is a obvious technique

```
int row, col;
for (row = 0; row < Num_Rows; row++)
    for (col = 0; col < Num_Cols; col++)
        a[row][col] = 0;
```

If we view array a as a one-dimensional array of integers, a single loop is sufficient

```
int *p;
for (p = &a[0][0]; p <= &a[Num_Rows-1][Num_Cols-1]; p++)
     *p = 0;
```

Pointer and Array

Array <-> Pointer

For any two-dimensional array a, the expression a[i] is a pointer to the first element in row i

```
int a[Num_Rows][Num_Cols];
```

Recall that a[i] is equal to *(a + i)

Therefore, &a[i][0] = &(*(a[i] + 0)) = a[i]

A loop that clears row i of the array a

```
int a[Num_Rows][Num_Cols], *p, i;
for (p = a[i]; p < a[i] + Num_Cols; p++)
      *p = 0;
```

Pointer and Array

Array <-> Pointer

The name of any array can be used as a pointer, regardless of how many dimensions it has, but some care is required

```
int a[Num_Rows][Num_Cols];
```

a is not a pointer to a[0][0]; instead, it's a pointer to a[0]

C regards a as a one-dimensional array whose elements are one-dimensional arrays

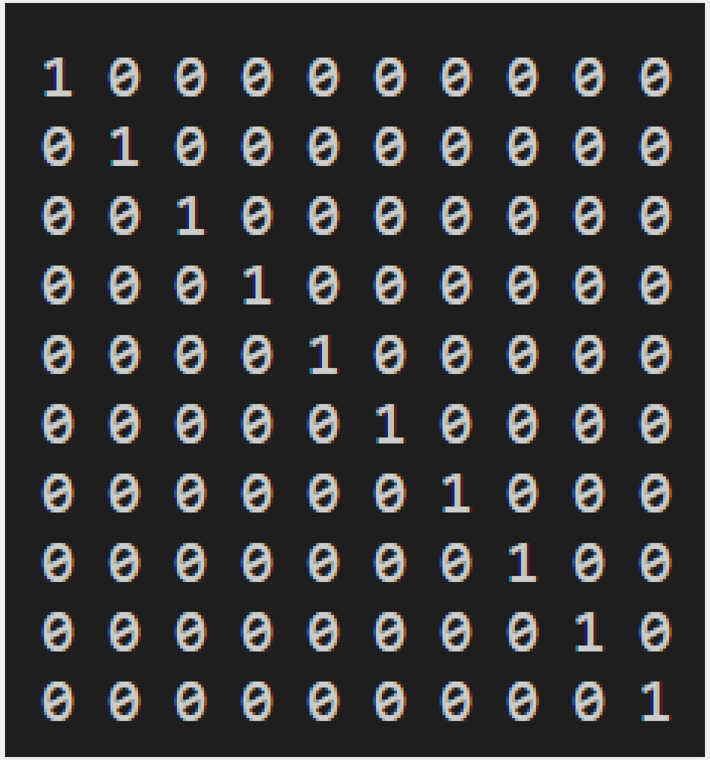
When used as pointer, a has type int (*) [Num_Cols]

Pointer and Array

Array <-> Pointer

Write a program to initialize an 10×10 identity array using a single pointer

```
int *p;
for (p = &a[0][0]; p <= &a[Num_Rows-1][Num_Cols-1]; p++)
    *p = 0;
```



```
1 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 1
```