Lecture 11: Pointers & Arrays

Sarah Nadi
nadi@ualberta.ca
Department of Computing Science
University of Alberta

CMPUT 201 - Practical Programming Methodology
Winter 2018

[With material/slides from Guohui Lin, Davood Rafei, and Michael Buro. Most examples taken from K.N. King's book]

UNIVERSITY OF ALBERTA

Agenda

- Pointer arithmetic
- Using pointers to process arrays
- Using array name as a pointer
- Pointers & multidimensional arrays
- Pointers & variable-length arrays

Readings

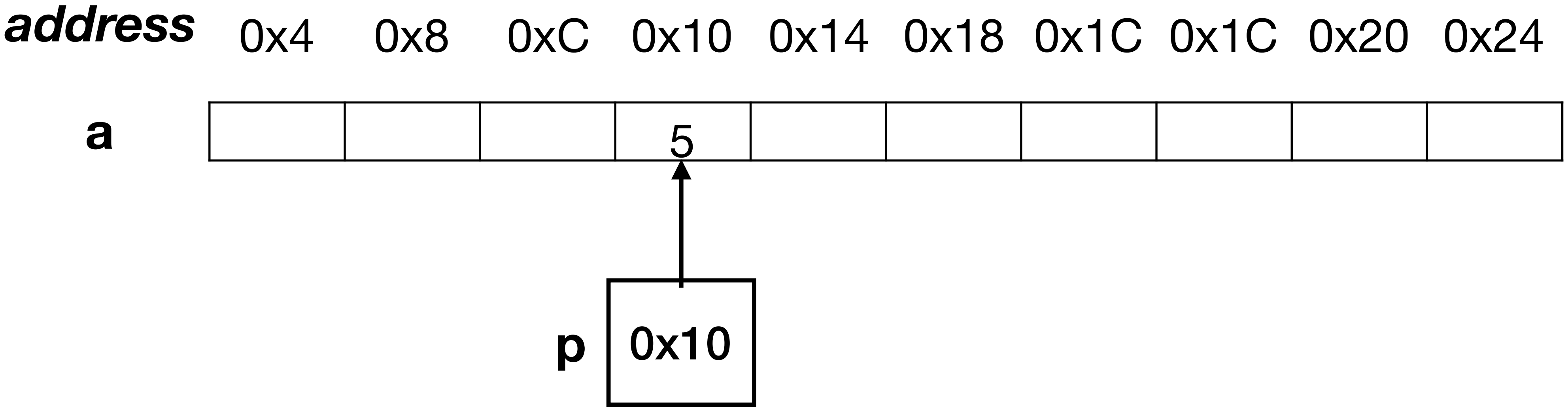
- Textbook Chapter 12

Pointers & Arrays

- There is a close relationship between pointers and arrays
- Historically, using pointers to process arrays was done for efficiency but this is no longer a concern thanks to improvements in compilers
- However, understanding the relationship between pointers and arrays is important to understanding a lot of C programs and how C is designed

Pointers to Array Elements

```
int a[10], *p;
p = &a[3];
*p = 5; // equivalent to a[3] = 5;
```



address	0x4	0x8	0xC	0x10	0x14	0x18	0x1C	0x1C	0x20	0x24
a				5						

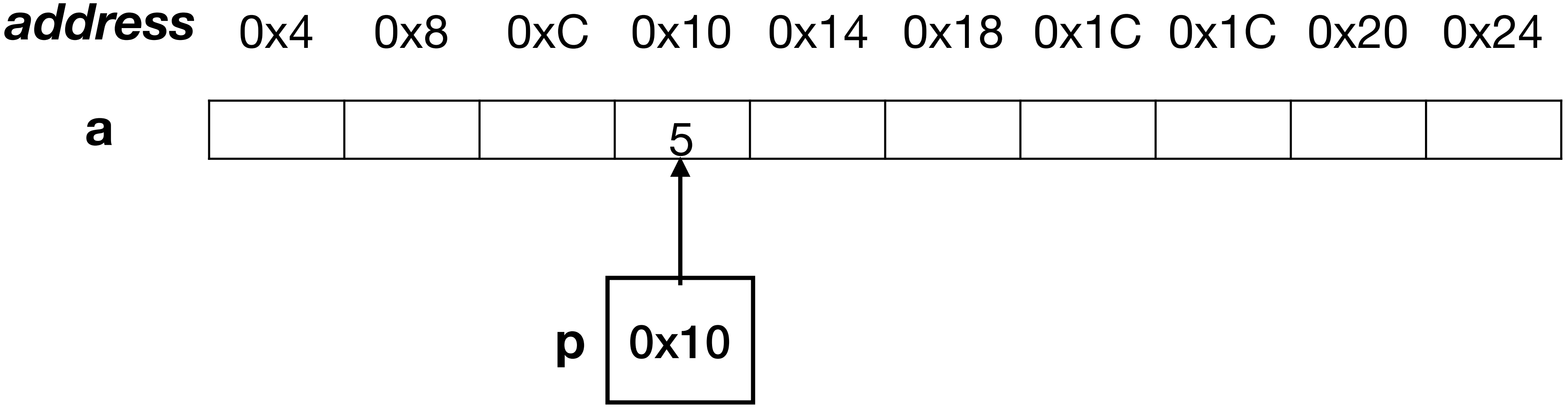
p **0x10**

Pointer Arithmetic

- Three supported arithmetic operations on pointers
  - adding an integer
  - subtracting an integer
  - subtracting one pointer from another (both pointers must point to elements of the same array)
- Pointers can also be compared

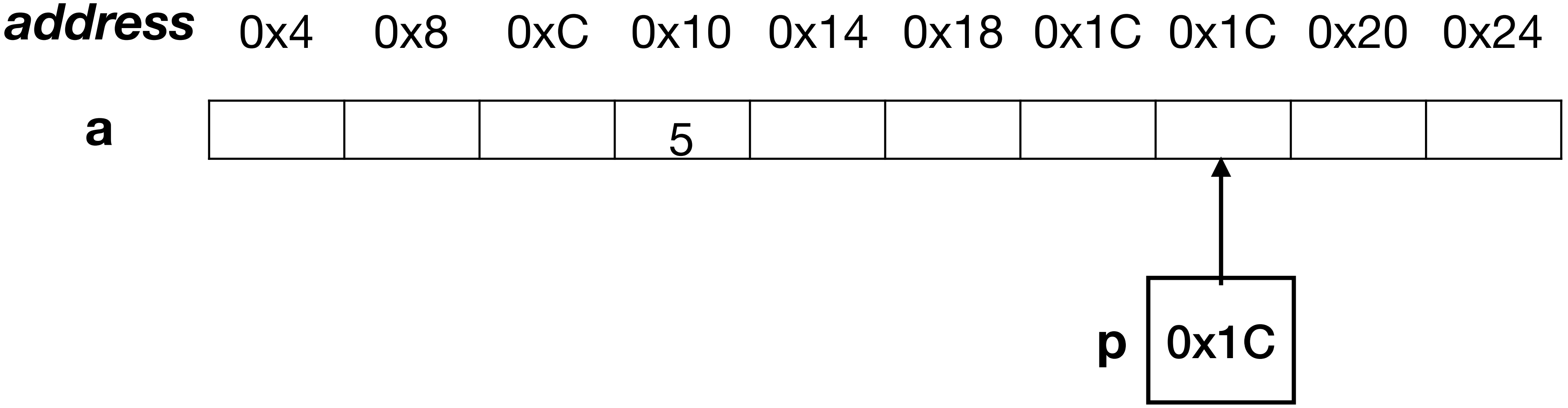
Adding an Integer to a pointer

```
int a[10], *p;
p = &a[3];
*p = 5; // equivalent to a[3] = 5;
```



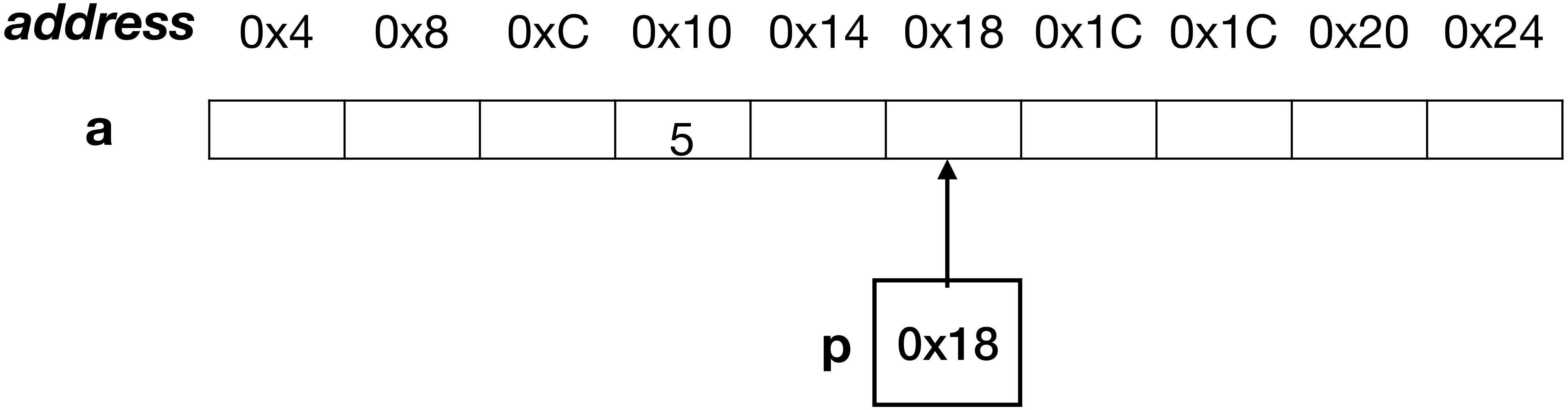
Adding an Integer to a pointer

```
int a[10], *p;
p = &a[3];
*p = 5; // equivalent to a[3] = 5;
p += 4; //p now points to a[7]
```



Adding an Integer to a pointer

```
int a[10], *p;
p = &a[3];
*p = 5; // equivalent to a[3] = 5;
p += 4; //p now points to a[7]
p -= 2; //p now points to a[5]
```



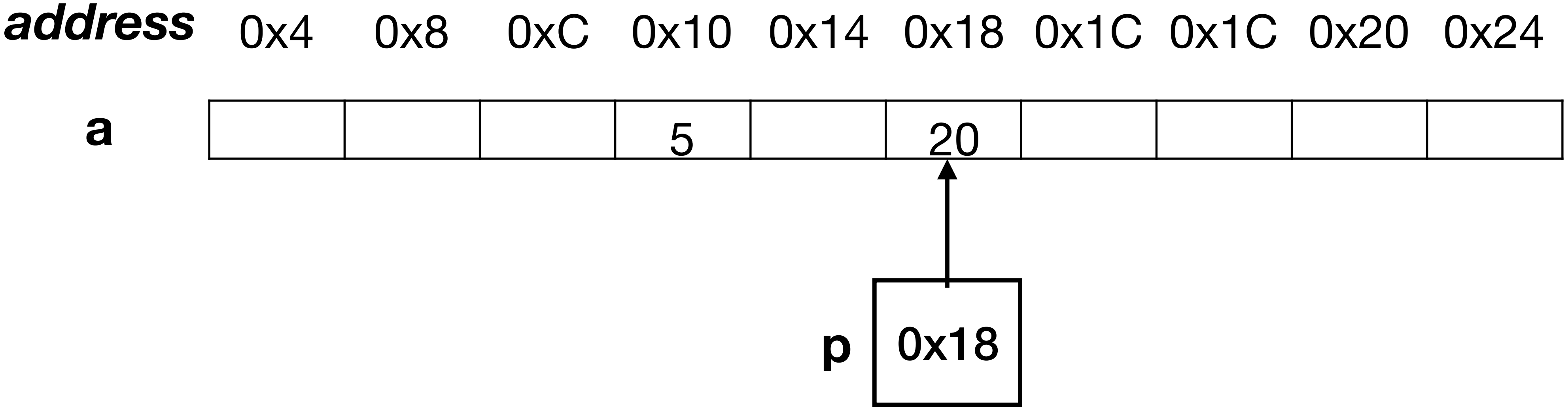
address 0x4 0x8 0xC 0x10 0x14 0x18 0x1C 0x1C 0x20 0x24

a 5

p 0x18

Adding an Integer to a pointer

```
int a[10], *p;
p = &a[3];
*p = 5; // equivalent to a[3] = 5;
p += 4; //p now points to a[7]
p -= 2; //p now points to a[5]
*p = 20; //equivalent to a[5] = 20;
```



address	0x4	0x8	0xC	0x10	0x14	0x18	0x1C	0x1C	0x20	0x24
a				5		20				

p **0x18**

Adding an Integer to a Pointer — More Formally

- Only makes sense when pointing to array elements

- Adding an integer `j` to a pointer `p` yields a pointer to the element `j` places after the one `p` currently points to. If p points to `a[i]` then `p+j` points to `a[i+j]`

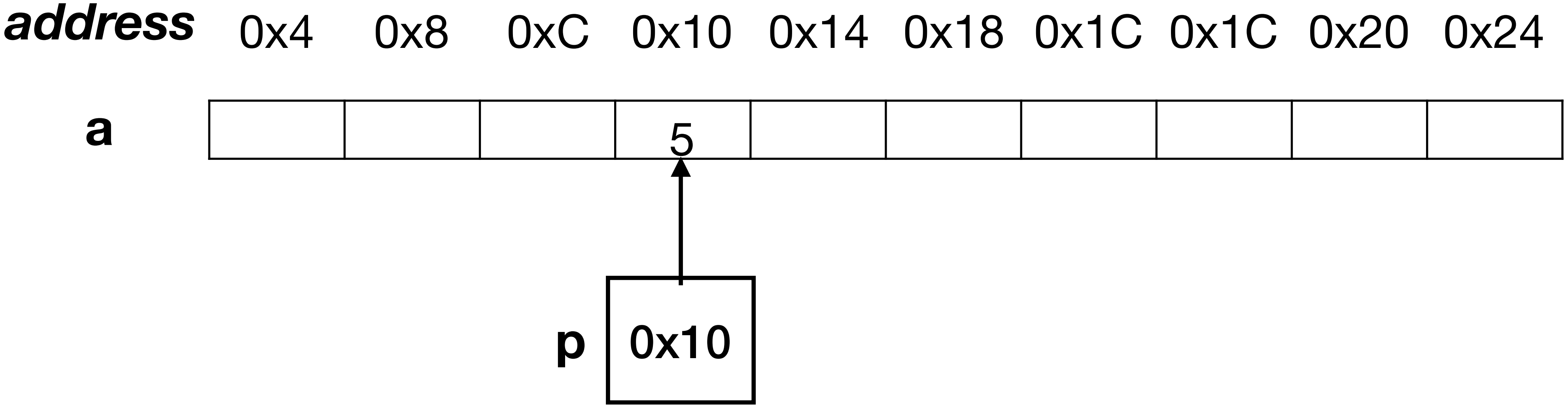
- While the `j` is an integer, the address that `p` points to depends on the type of the array. For example:

    - Adding 1 to a pointer that points to an element in an integer array moves the pointer to an address that is 4 bytes later

    - Adding 1 to a pointer that points to an element in a character array moves the pointer to an address that is 1 byte later

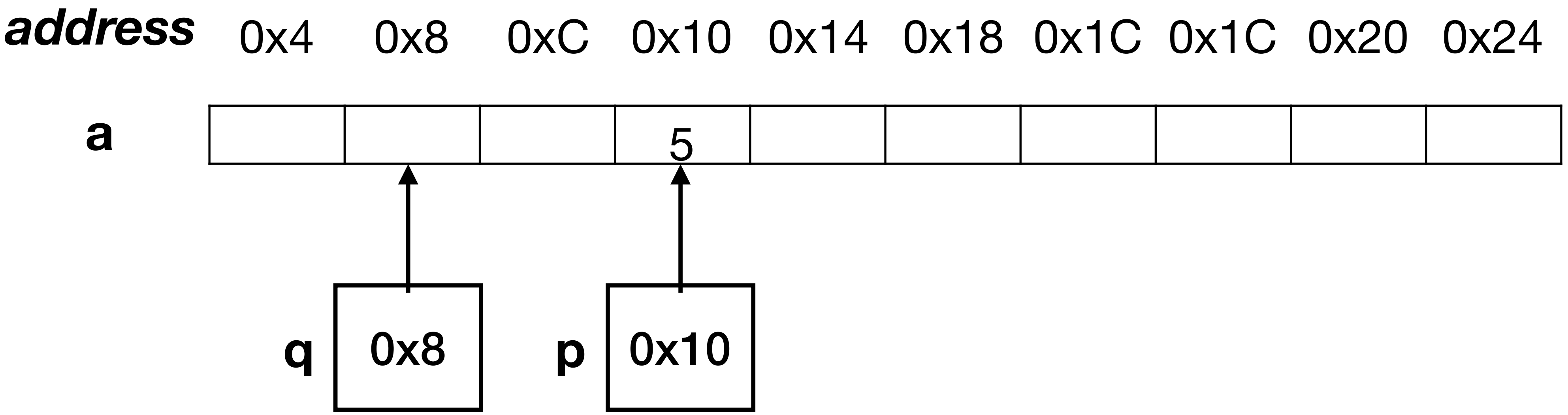
Subtracting an Integer From a Pointer

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



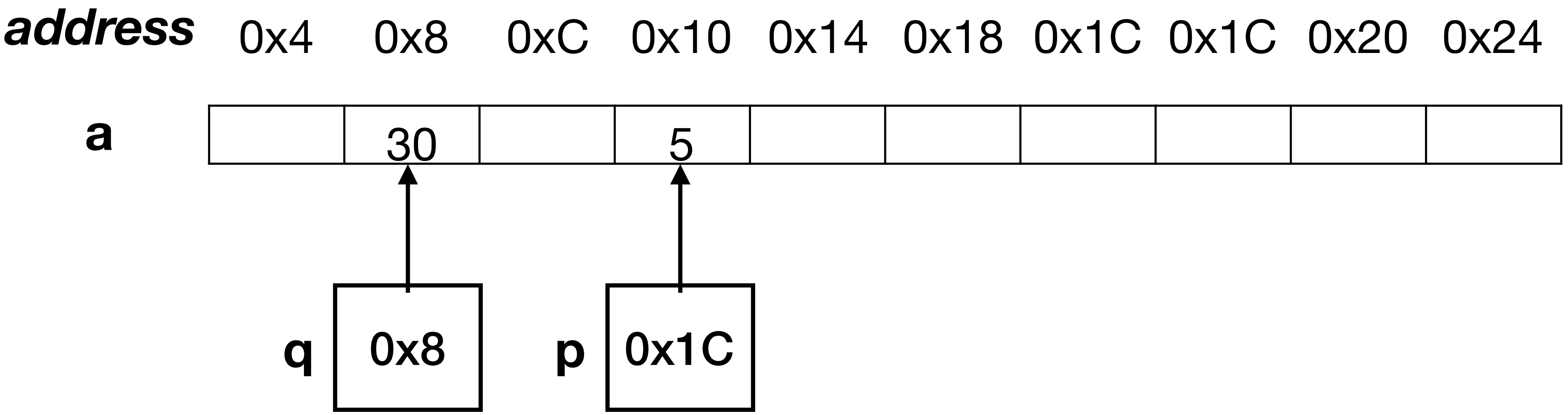
Subtracting an Integer From a Pointer

```
int a[10], *p, *q;
p = &a[3];
*p = 5; // equivalent to a[3] = 5;
q = p - 2; //p now points to a[1]
```



Subtracting an Integer From a Pointer

```
int a[10], *p, *q;
p = &a[3];
*p = 5; // equivalent to a[3] = 5;
q = p - 2; //p now points to a[1]
*q = 30; //equivalent to a[1] = 30
```



address 0x4 0x8 0xC 0x10 0x14 0x18 0x1C 0x1C 0x20 0x24

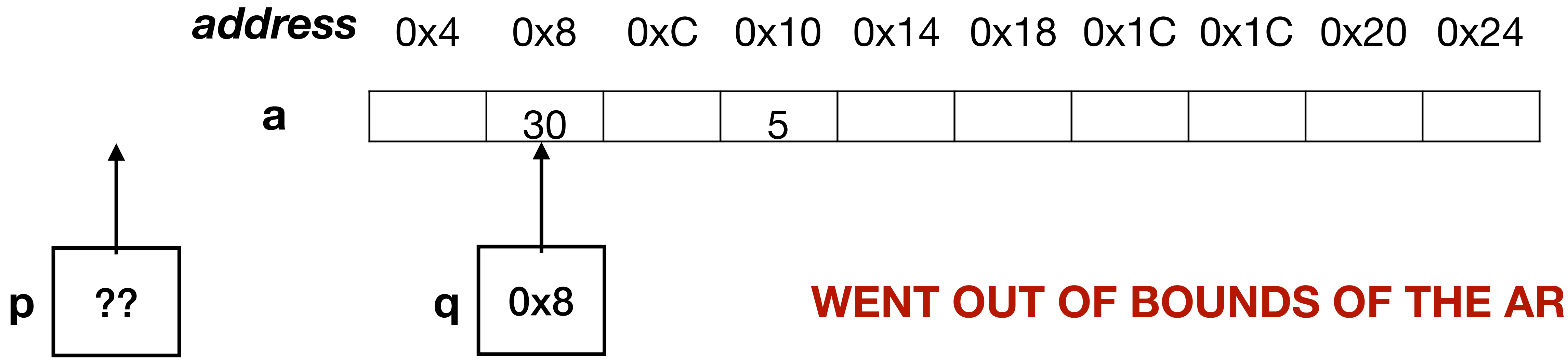
a | | 30 | | 5 | | | | | |

q **0x8**

p **0x1C**

Subtracting an Integer From a Pointer

```
int a[10], *p, *q;
p = &a[3];
*p = 5; // equivalent to a[3] = 5;
q = p - 2; //p now points to a[1]
*q = 30; //equivalent to a[1] = 30
p -= 6;
```

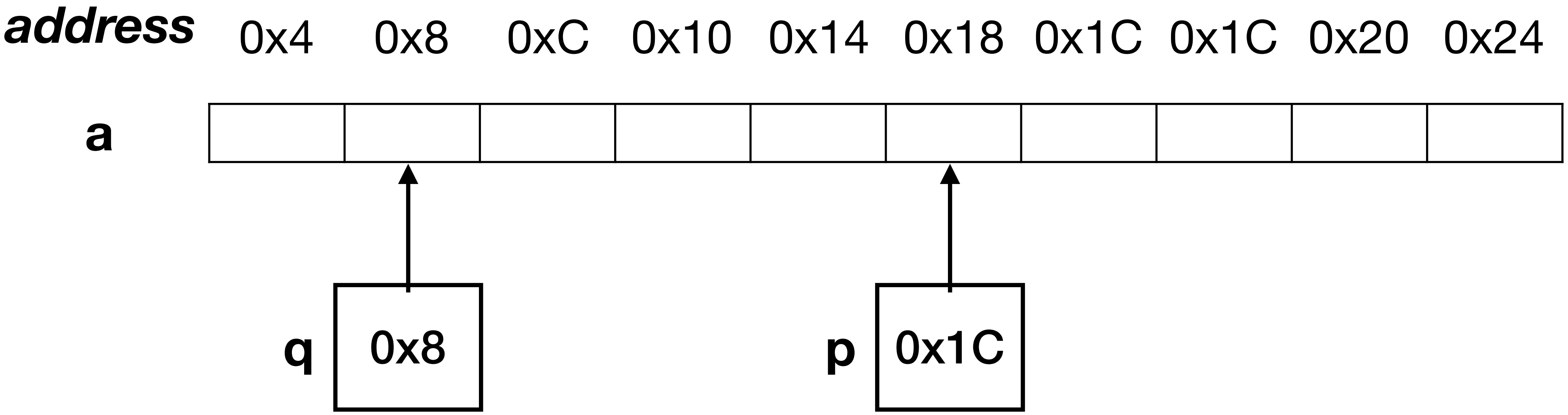


address	0x4	0x8	0xC	0x10	0x14	0x18	0x1C	0x1C	0x20	0x24
a		30		5						

WENT OUT OF BOUNDS OF THE ARRAY!!!!

Subtracting One Pointer From Another

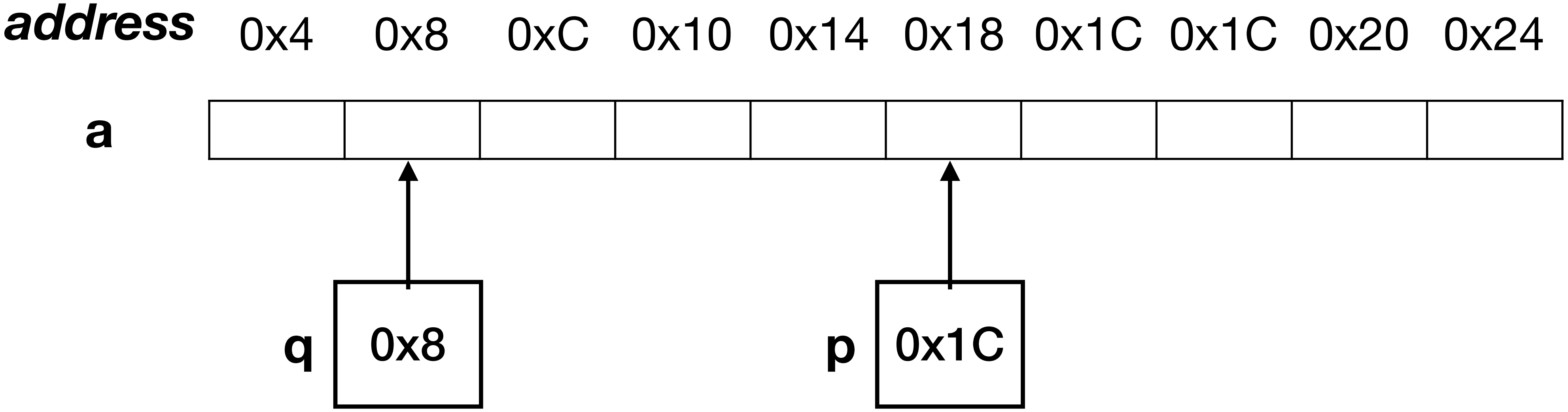
```
int a[10], *p, *q, i;
p = &a[5];
q = &a[1];
i = p - q; //i is now 4
i = q - p; //i is now -4
```



Only makes sense when both pointers point to elements of the same array!

Comparing Pointers

```
int a[10], *p, *q, i;
p = &a[5];
q = &a[1];
if (p < q) {…} //evaluates to false (i.e., 0)
if (q < p) {…} //evaluates to true (i.e., 1)
```



Using Pointers for Array Processing

```
#define N 10

int main(){
  int a[N], sum =0, *p;

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

Combining * and ++/-- Operators

```
#define N 10

int main(){
  int a[N], sum =0, *p;
  p = &a[0];
  while( p < &a[N]){
    sum += *p++;
  }
}
```

- Postfix version of `++/--` takes precedence over `*`
- Therefore `*p++` is equivalent to `*(p++)`
- ..but, we know that `p++` means we first use the value of p then increment
- Therefore `sum += *p++;` is equivalent to:

```
sum += *p;
p++;
```

Combining * and ++/-- Operators

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

Same applies for prefix/postfix -- operator

Additional Examples of Combinations of ++/-- and *

```
int *top_ptr = &contents[0];

void push(int i){
  if (is_full())
    stack_overflow();
  else
    *top_ptr++ = i;
}

int pop(){
  if (is_empty())
    stack_underflow();
  else
    return *--top_ptr;
}
```

Using An Array Name as a Pointer

- An array name can be used as a pointer to the first element in the array

- However, you **CANNOT** change the value of that pointer, because it is reserved to always point to the first element of the array

```
int a[10], *p, sum;
*a = 7;           /* stores 7 in a[0] */
*(a + 1) = 12;    /* stores 12 in a[1] */
sum = 0;

for (p = &a[0]; p < &a[10]; p++)
   sum += *p;
sum = 0;

for (p = a; p < a + 10; p++)
   sum += *p;
```

Using An Array Name as a Pointer *Cont'd*

- If you assign a pointer to an array, you can then subscript the pointer as if it were an array

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

In other words, p[i] is equivalent to *(p+i)

Reversing Numbers in an Array Using Pointers

- Given an array, use pointers to print the array in reverse order

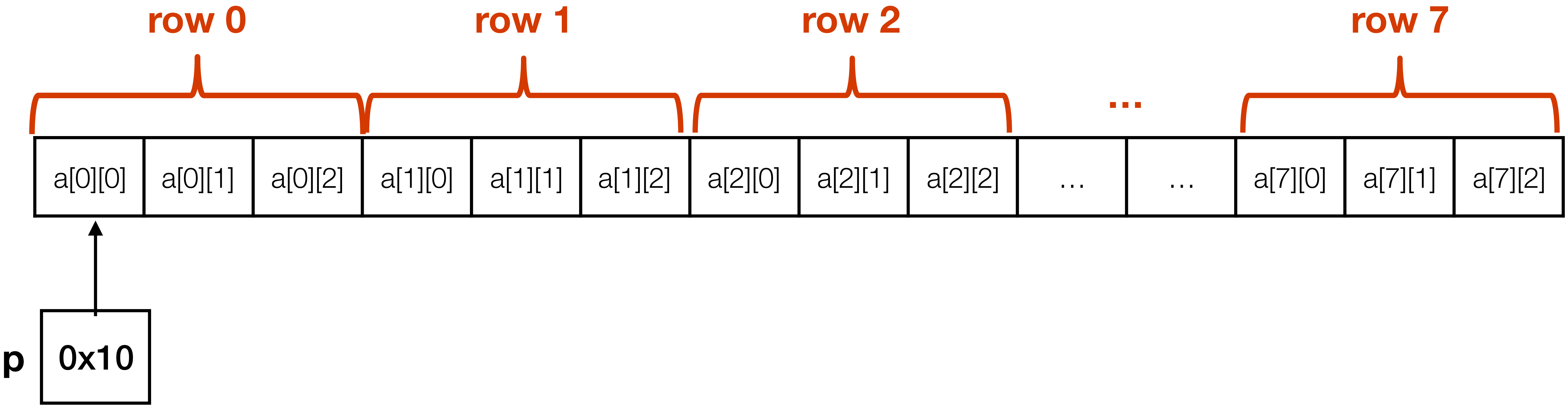
Group Exercise!

Array Arguments

- When passed to a function, an array name is **always** treated as a pointer
- Since parameters in C are passed by value, this means that the value passed is the address of the array. The address of the array is actually the address of the first element of the array.
- This is why the values of array elements can be changed in a function (what we have seen earlier but perhaps hadn't understood exactly why)
- Use the keyword `const` if you want to prevent a function from changing the elements of an array

Pointers & Multidimensional Arrays

```
int a[8][3];
int *p, *q;
p = &a[0][0]; // p points to the first element
```



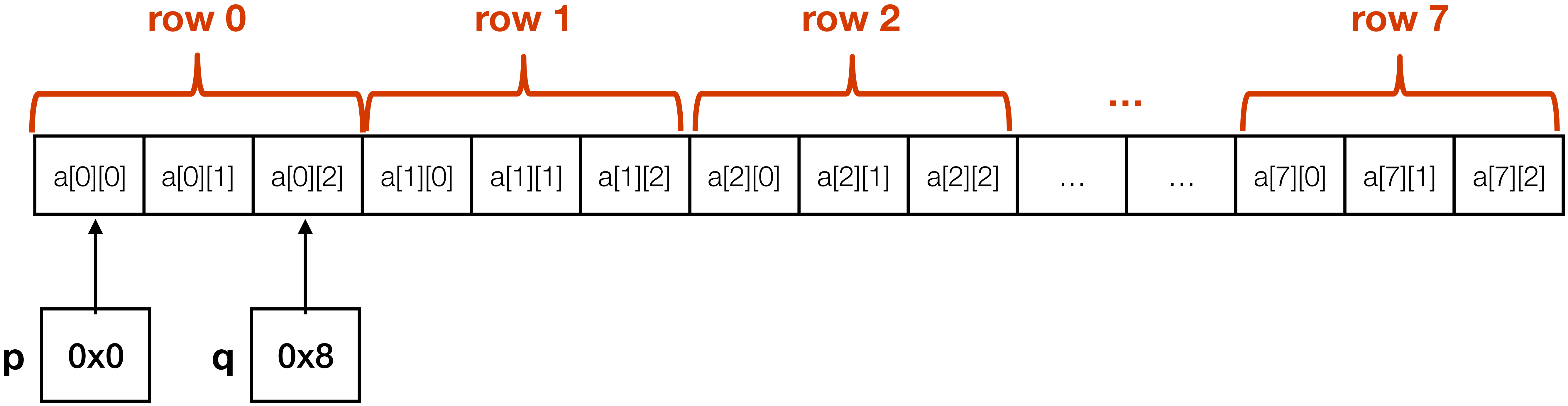
row 0 row 1 row 2 … row 7

a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]	a[2][0]	a[2][1]	a[2][2]	…	…	a[7][0]	a[7][1]	a[7][2]

p 0x10

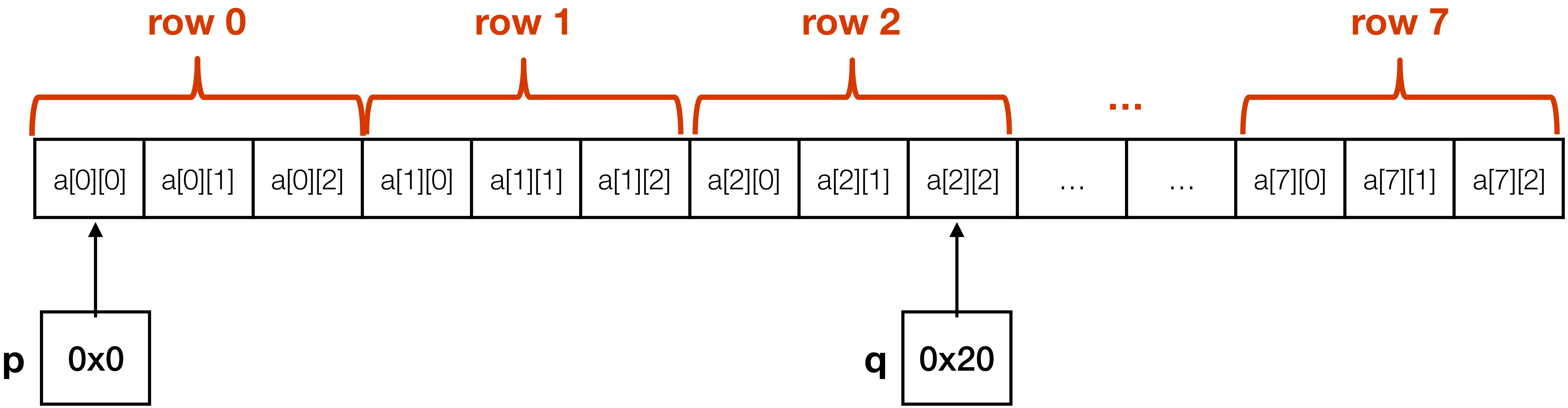
Pointers & Multidimensional Arrays

```
int a[8][3];
int *p, *q;
p = &a[0][0]; // p points to the first element
q = p + 2; //q points to a[0][2]
```



Pointers & Multidimensional Arrays

```
int a[8][3];
int *p, *q;
p = &a[0][0]; // p points to the first element
q = p + 2; //q points to a[0][2]
q = p + 8; // q points to a[2][2]
```



Using Pointers to Process a Multidimensional Array

```
…
int a[NUM_ROWS][NUM_COLUMNS], i, sum = 0, *p;
…
for (p = &a[0][0]; p < &a[NUM_ROWS][NUM_COLUMNS]; p++)
   sum += *p;
```

Pointers & Multidimensional Arrays *Cont'd*

```
…
int a[NUM_ROWS][NUM_COLUMNS], *p;
int i = 2;
p = &a[i][0]; // p points to the first element of row i (=2 here)
p = a[i]; //p points to row 2. Think about what the type of a[i] is
p = a; //WRONG! incompatible types.. How can we assign a pointer to a 2D
         array?
```

Pointers & Multidimensional Arrays *Cont'd*

- So if you did want to have `p` point to the beginning of the array, we need to think about what `a` itself points to? `a` points to `a[0]`
- C regards `a[rows][cols]` as a one-dimensional array whose elements are 1D arrays. Thus, `a` has the type `int (*) [cols]` (pointer to an integer array of length cols)

```
int a[10][20], (*p)[20], i;
i = 4;

p = a; //this is now valid and is equiv to p = &a[0]
(*p)[i] = 5; //same as a[0][i] = 5;
p = a + 3; //this is equivalent to p = &a[3]
(*p)[i] = 20; //same as a[3][i] = 20
```

Note: int *p[20]; would declare p as an array of 20 int pointers (similar to how argv works)

Processing Rows/Columns of a Multidimensional Array

```
int a[NUM_ROWS][NUM_COLUMNS];
```

- Write a loop that assigns 0 to all elements of row i of a 2D array
- Write a loop that assigns 0 to all elements of column j of a 2D array

Group Exercise!

Pointers and Variable-length Arrays (c99)

```
void f(int n){
 int a[n], *p;
 p = a;
 …
}
```

```
void f(int m, int n){
 int a[m][n], (*p)[n];
 p = a;
 …
}
```

Pointers and Variable-length Arrays (c99)

```
void f(int n){
 int a[n], *p;
 p = a;
 …
}
```

```
void f(int m, int n){
 int a[m][n], (*p)[n];
 p = a;
 …
}
```

p is a variably modified type