CS2211b

Software Tools and Systems Programming

Western
UNIVERSITY · CANADA

Week 10b

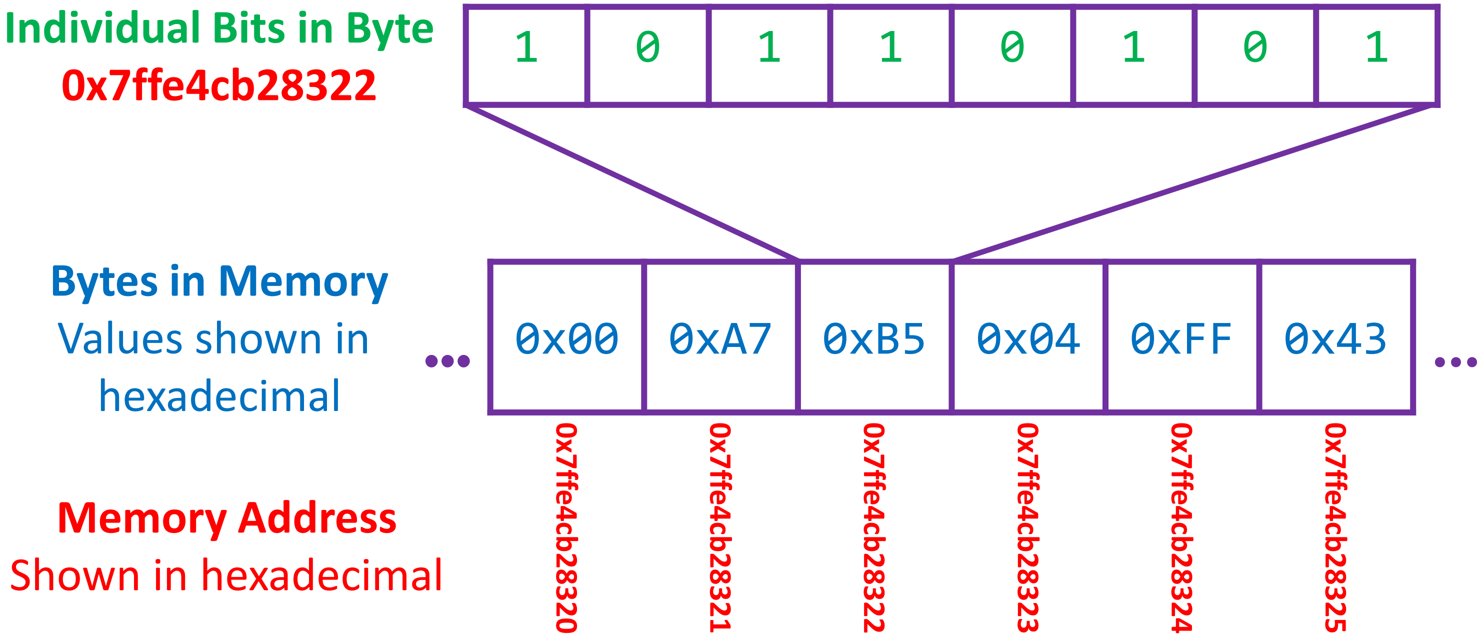
Pointers

Pointers

Western Science

Memory Addresses

- Memory in C can be thought of as a large array of bytes with each byte indexed in order by an unique **memory address**.
- Variables we declare are stored as one or more bytes in this large array of memory. For example a character would take up 1 byte well and integer would take up 4 bytes (on the course sever).



Individual Bits in Byte
0x7ffe4cb28322

1	0	1	1	0	1	0	1

Bytes in Memory
Values shown in hexadecimal

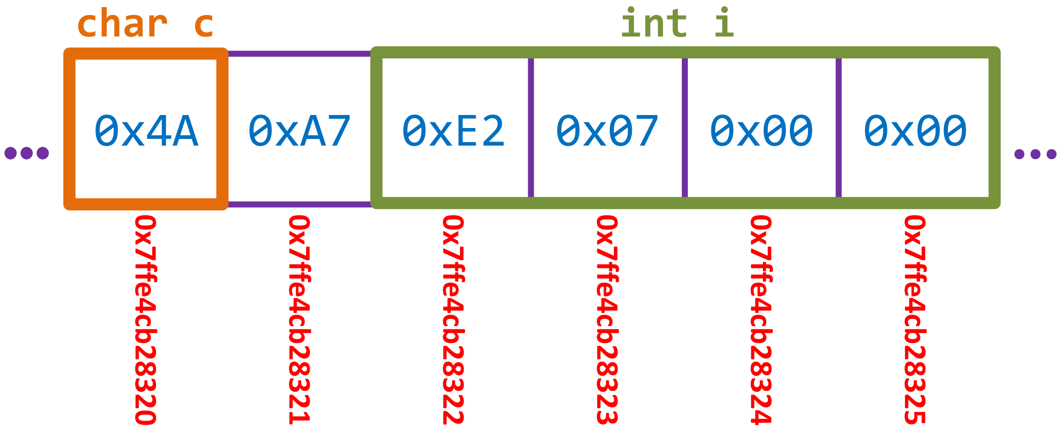
…	0x00	0xA7	0xB5	0x04	0xFF	0x43	…
	0x7ffe4cb28320	0x7ffe4cb28321	0x7ffe4cb28322	0x7ffe4cb28323	0x7ffe4cb28324	0x7ffe4cb28325	

Memory Address
Shown in hexadecimal

Memory Addresses

- Memory in C can be thought of as a large array of bytes with each byte indexed in order by an unique **memory address**.
- Variables we declare are stored as one or more bytes in this large array of memory. For example a character would take up 1 byte well and integer would take up 4 bytes (on the course sever).

```
char c = 'J';
int i = 2018;
```



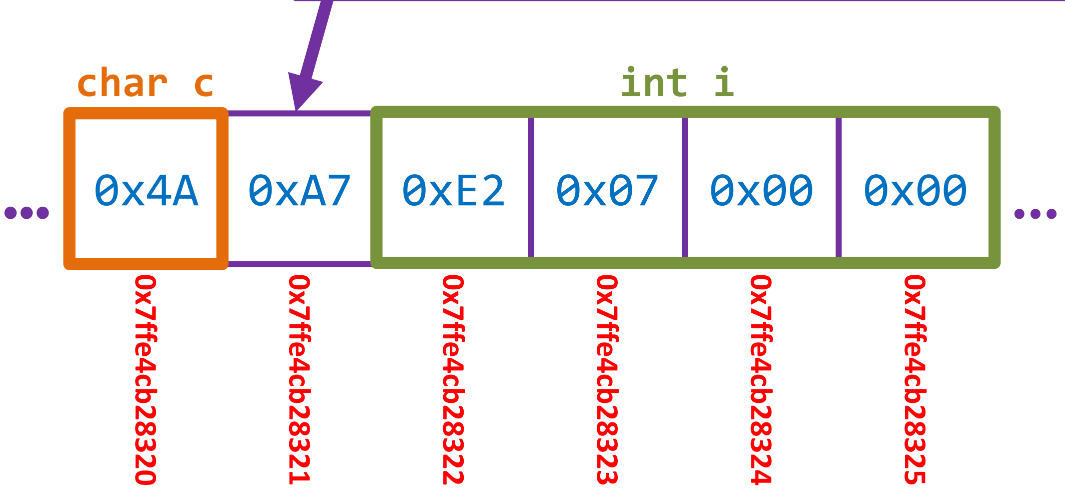
Memory Addresses

- Memory in C can be thought of as a large array of bytes with each byte indexed in order by an unique **memory address**.
- Variables we declare are stored as one or more bytes in this large array of memory. For example a character would take up 1 byte well and integer woul ever).

Important Note

Just because a variable is declared directly after another does not mean it will be directly after the other in memory.

```
char c = 'J';
int i = 2018;
```



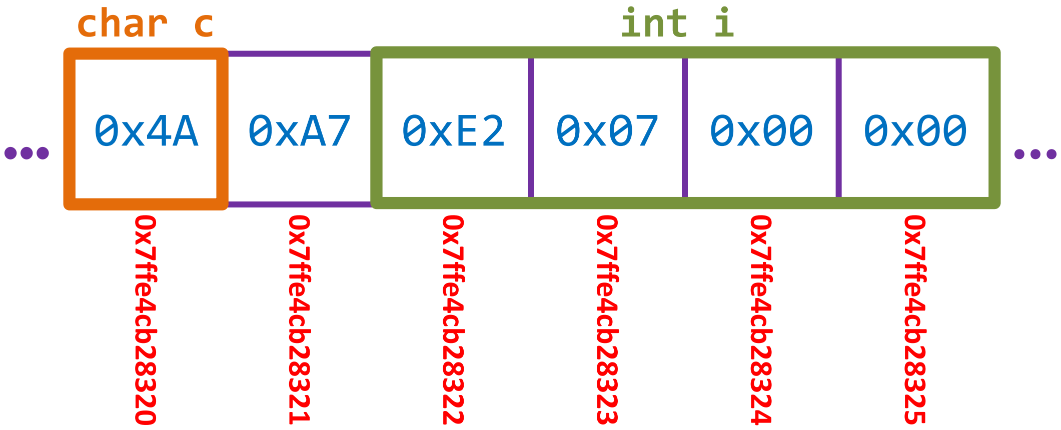
Memory Addresses

- Each variable is given a **memory address** based on the first byte it occupies.

The address of c is the only byte it occupies, **0x7ffe4cb28320**

The address of i is the first byte it occupies, **0x7ffe4cb28322**

```
char c = 'J';     0x7ffe4cb28320
int i = 2018;     0x7ffe4cb28322
```



char c

int i

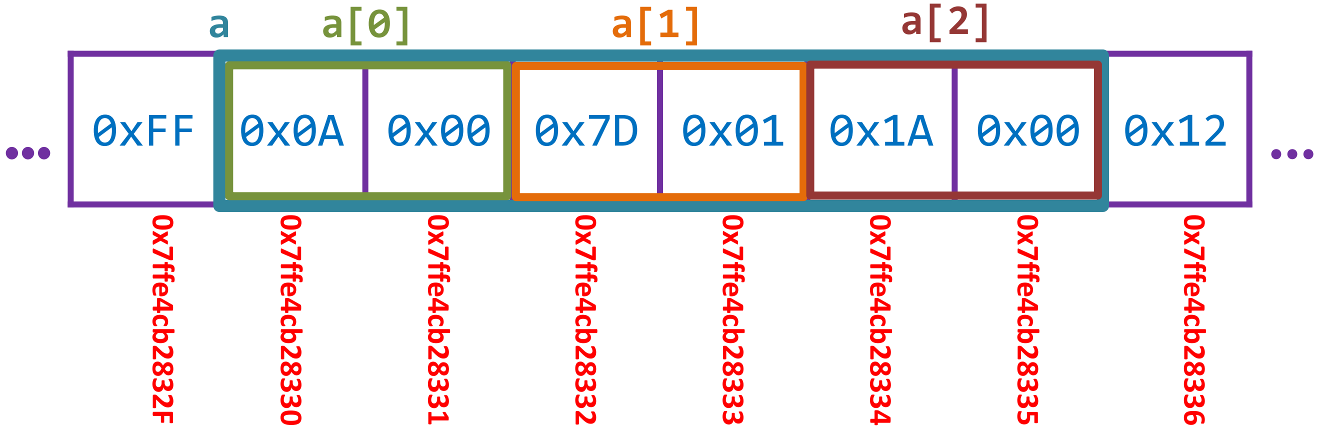
… 0x4A 0xA7 0xE2 0x07 0x00 0x00 …

0x7ffe4cb28320 0x7ffe4cb28321 0x7ffe4cb28322 0x7ffe4cb28323 0x7ffe4cb28324 0x7ffe4cb28325

Memory Addresses

- Array elements in the same array are gratinated to be stored sequentially in memory in **row-major order**.
- The address of the array is the first byte it occupies, the address of an array element is the first byte the individual element occupies.

```
short a[3] = {10, 381, 26};
```

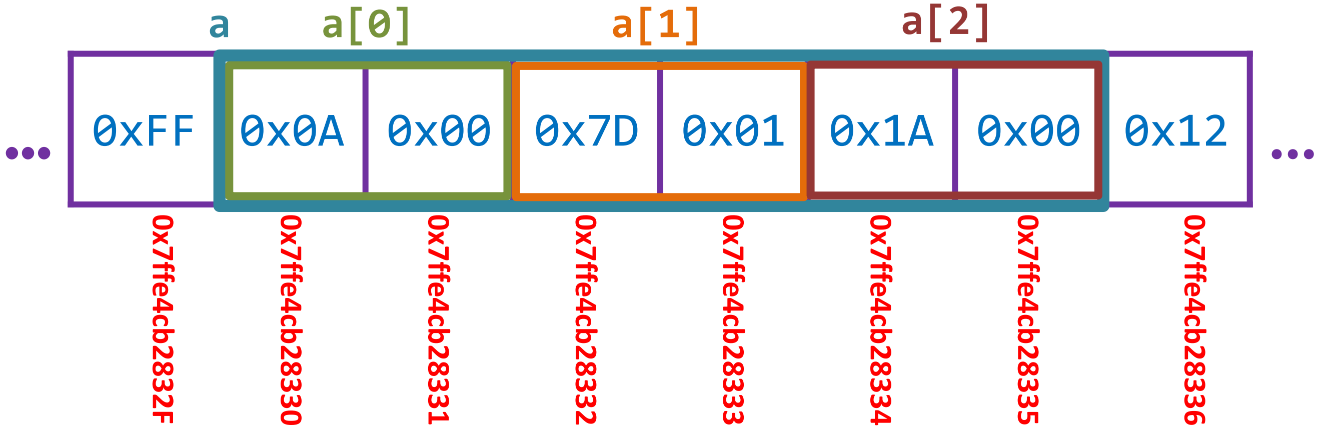


Memory Addresses

- Array elements in the same array are gratinated to be stored sequentially in memory in **row-major order**.
- The address of the array is the first byte it occupies, the address of an array element is the first byte the individual element occupies.

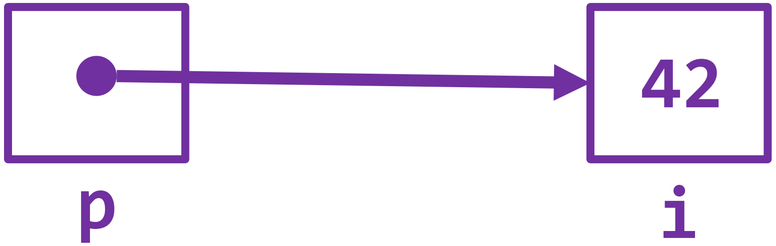
```
short a[3] = {10, 381, 26};
```

Element	Memory Address
a	0x7ffe4cb28330
a[0]	0x7ffe4cb28330
a[1]	0x7ffe4cb28332
a[2]	0x7ffe4cb28334



Pointers

- **Pointers** are a special type of variable that store a **memory address**.
- Normally, we do not give pointers a constant or literal memory address (e.g. 0x7ffe4cb28330) but instead the address of another variable (e.g &x).
- When we store the address of variable `i` in pointer p, we say that p ***points to*** `i`.
- Represented visually (if `i` = 42 and p points to `i`):



Declaring Pointers

- Pointers are declared similar to normal variables, they have a type and name. However its name must be preceded by an asterisk (*).
- **Example:**

```
 int *p;
```

- With pointers, the type does not refer to the value the variable is storing (a memory address), but the type of the **object** the pointer is pointing at.
- In this case, p is a pointer capable of pointing at integer **objects**.
- We use the term **object** here rather than variable, as pointers can point at any arbitrary memory address.
- The type of the pointer tells the compiler how to interpret the values in that memory location and the number of bytes to read.

Declaring Pointers

- **Simple Examples:**

`int *a;`	Integer Pointer
`char *b;`	Character Pointer
`float *c;`	Float Pointer
`double *d;`	Double Pointer
`long *e;`	Long Pointer

Declaring Pointers

- We can declare arrays, variables and pointers of the same type on the same line:

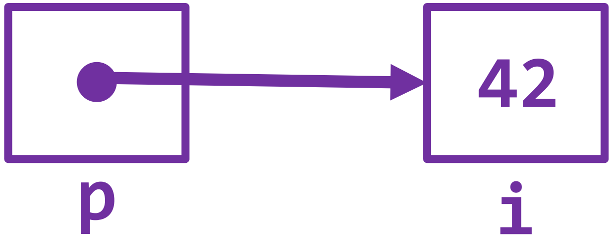
```
int a, *b, c[20], *d, e[13], f, g;
```

Variables a, f and g are regular integers. c and e are arrays of length 20 and 13 respectively. b and d are integer pointers (pointers that point to integer objects).

Assigning Pointers Values

- We can use the **address operator (&)** to assign a memory address of a variable to a pointer.
- Recall that we saw this operator previously when using `scanf` and passing variables by reference.
- The **address operator** returns the address of a variable.
- **Example:**

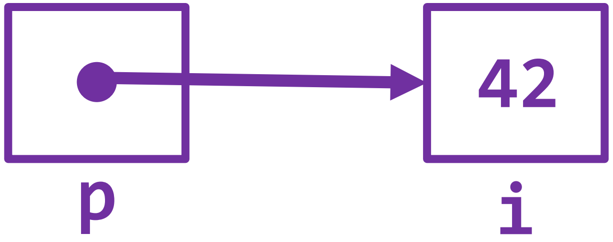
```
int i = 42;
int *p;
p = &i;
```



Assigning Pointers Values

- We can use the **address operator (&)** to assign a memory address of a variable to a pointer.
- Recall that we saw this operator previously when using `scanf` and passing variables by reference.
- The **address operator** returns the address of a variable.
- **Example:**

```
int i = 42;
int *p;
p = &i;
```

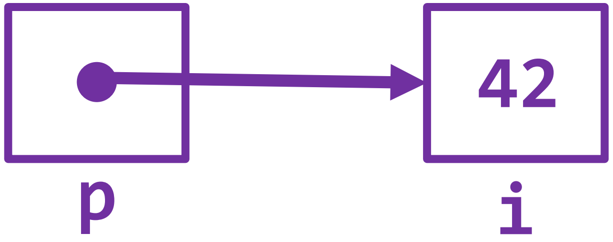


Declares the integer pointer p.

Assigning Pointers Values

- We can use the **address operator (&)** to assign a memory address of a variable to a pointer.
- Recall that we saw this operator previously when using `scanf` and passing variables by reference.
- The **address operator** returns the address of a variable.
- **Example:**

```
int i = 42;
int *p;
p = &i;
```



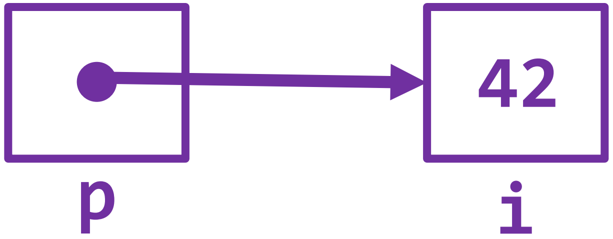
Stores the memory address of the integer variable `i` in `p`.

This points `p` to `i`.

Assigning Pointers Values

- We can use the **address operator (&)** to assign a memory address of a variable to a pointer.
- Recall that we saw this operator previously when using `scanf` and passing variables by reference.
- The **address operator** returns the address of a variable.
- **Example:**

```
int i = 42;
int *p = &i;
```



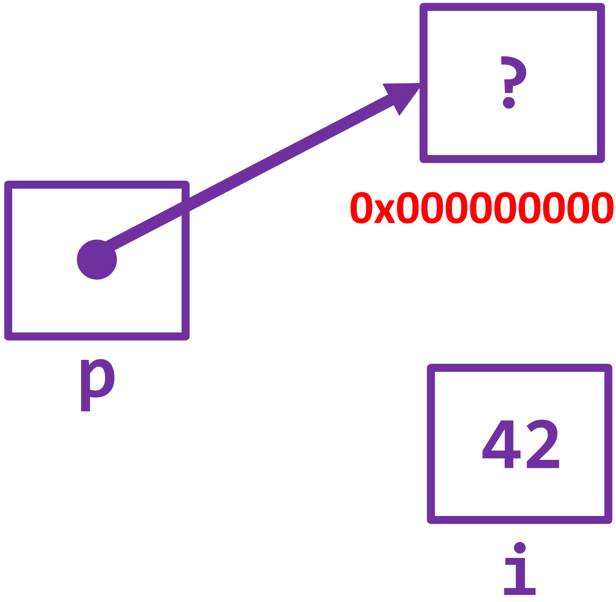
We can do assignment on the same line as a declaration just like we do with normal variables.

Assigning Pointers Values

- We can use the **address operator (&)** to assign a memory address of a variable to a pointer.
- Recall that we saw this operator previously when using `scanf` and passing variables by reference.
- The **address operator** returns the address of a variable.
- **Bad Example:**

```
int i = 42;
int *p;
p = i;
```

Missing &



Assigning Pointers Values

Will compile but probably not what you want.

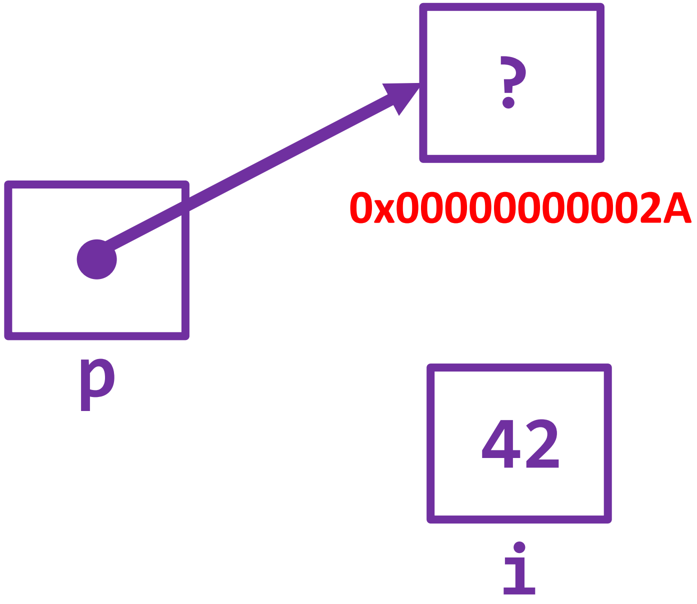
Pointer p is set to memory address 42 (0x000000000002A) and does not point to the variable i.

You would like get a **Segmentation Fault** if you attempted to use this pointer (read value of the memory address or set it).

- **Bad Example:**

```
int i = 42;
int *p;
p = i;
```

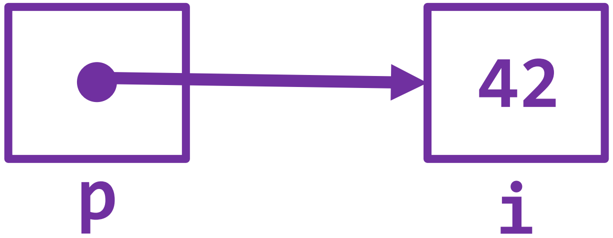
Missing &



Dereferencing Pointers

- We can get the value stored at a memory address using the **indirection operator (*)**.
- This is referred to as "**dereferencing**" the pointer.
- **Example:**

```
int i = 42;
int *p = &i;
printf("%d %d\n", *p, i);
```

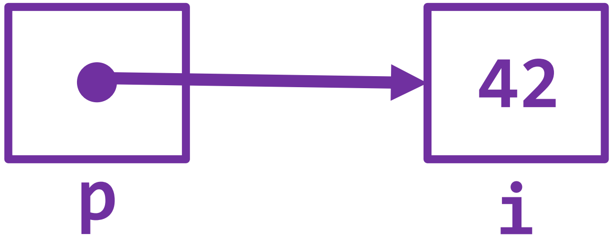


Output: 42 42

Dereferencing Pointers

- We can get the value stored at a memory address using the **indirection operator (*)**.
- This is referred to as "**dereferencing**" the pointer.
- **Example:**

```
int i = 42;
int *p = &i;
printf("%d %d\n", *p, i);
```



Output: 42 42

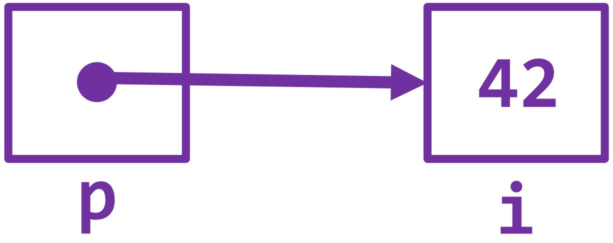
Important Note

These two asterisks are not the same. The first denotes that the variable p is a pointer. Only the second asterisks is the **indirection operator**.

Dereferencing Pointers

- We can get the value stored at a memory address using the **indirection operator (*)**.
- This is referred to as "**dereferencing**" the pointer.
- **Example:**

```
int i = 42;
int *p = &i;
printf("%d %d\n", *p, i);
```



Output: 42 42

*p dereferences the pointer p.

The computer looks at the memory p is pointing to and attempts to interpret it as an integer value (as p is an integer pointer).

In this case the expression *p returns the value 42 as p is pointer at the integer `i` which has the value 42.

Dereferencing Pointers

- We can also use the **indirection operator** to set the value at the memory address the pointer is pointing at.
- **Example:**

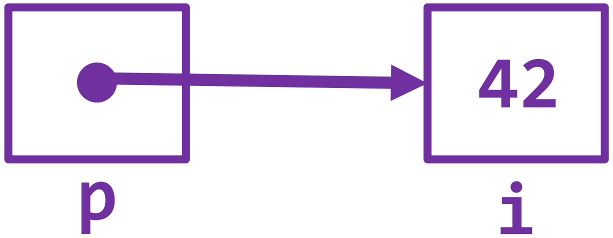
```
int i = 42;
int *p = &i;
*p = 5;
printf("%d %d\n", *p, i);
```

Output: 5 5

Dereferencing Pointers

- We can also use the **indirection operator** to set the value at the memory address the pointer is pointing at.
- **Example:**

```
int i = 42;
int *p = &i;
*p = 5;
printf("%d %d\n", *p, i);
```



Output: 5 5

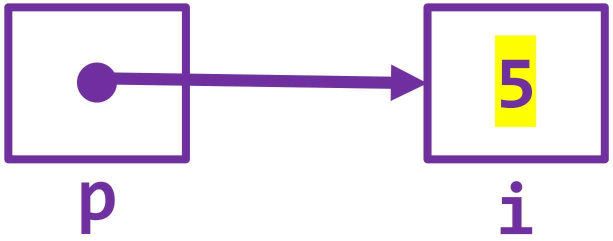
The integer `i` is declared and given the value 42.

The integer pointer p is declared and pointed at `i`.

Dereferencing Pointers

- We can also use the **indirection operator** to set the value at the memory address the pointer is pointing at.

- **Example:**



```
int i = 42;
int *p = &i;
*p = 5;
printf("%d %d\n", *p, i);
```

Output: 5 5

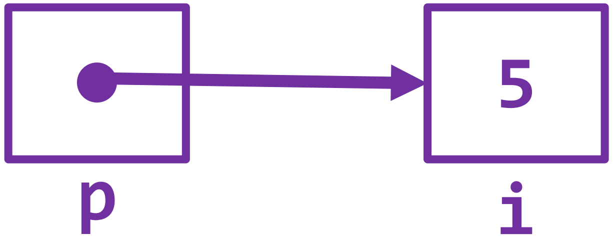
The value of the location p is pointing at is updated to 5.

Dereferencing Pointers

- We can also use the **indirection operator** to set the value at the memory address the pointer is pointing at.

- **Example:**

```
int i = 42;
int *p = &i;
*p = 5;
printf("%d %d\n", *p, i);
```



Output: 5 5

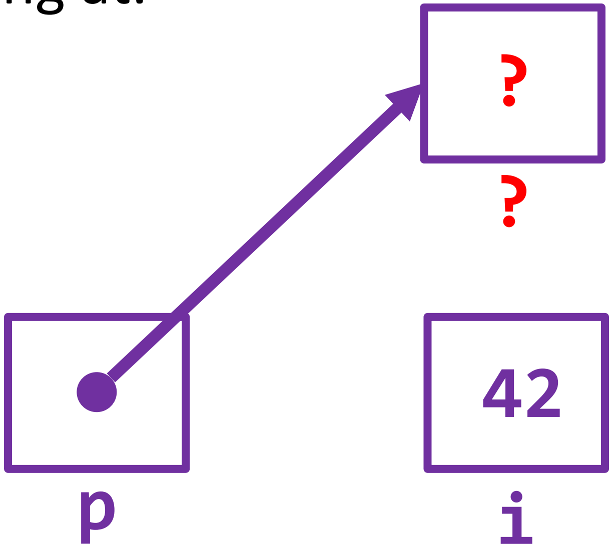
The value of the variable i is now equal to 5.

Dereferencing Pointers

- We can also use the **indirection operator** to set the value at the memory address the pointer is pointing at.
- **Bad Example 1:**

```
int i = 42;
int *p;
*p = 5;
```

p is not given a value



In this case we are trying to set the value of the location p is pointing to but we have failed to initialized p to a memory address.

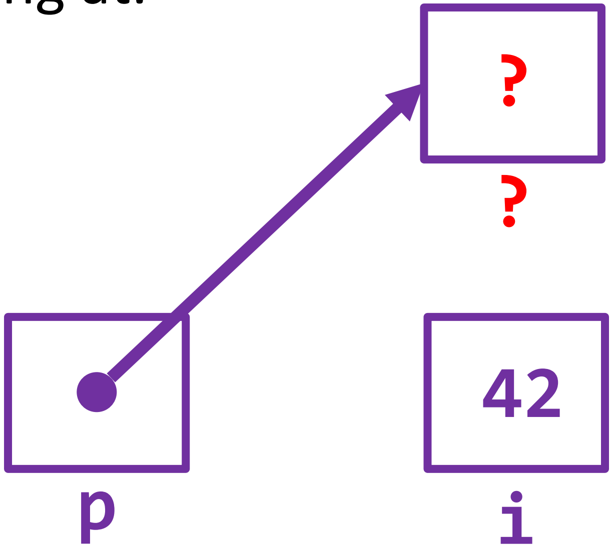
This will compile but we don't know where in the memory we are storing the value 5. Falls into **undefined behaviour** in most cases.

Dereferencing Pointers

- We can also use the **indirection operator** to set the value at the memory address the pointer is pointing at.
- **Bad Example 2:**

```
int i = 42;
int *p;
printf("%d", *p);
```

p is not given a value



It is also a problem if we try to read the value of an uninitialized pointer.

In most cases it is **undefined** where this pointer is pointing.

Copying Pointers

- In addition to setting a point to the address of another variable, we can set a pointer equal to another pointer (i.e. copy it).
- This does not copy the value that the pointer is pointing at, it simply makes another pointer that points to the same memory location.
- **Example:**

```
int i = 42;
int *p, *q;
p = &i;
q = p;
*q = 32;
printf("%d %d %d\n", *p, *q, i);
```

Output: 32 32 32

Copying Pointers

- In addition to setting a point to the address of another variable, we can set a pointer equal to another pointer (i.e. copy it).

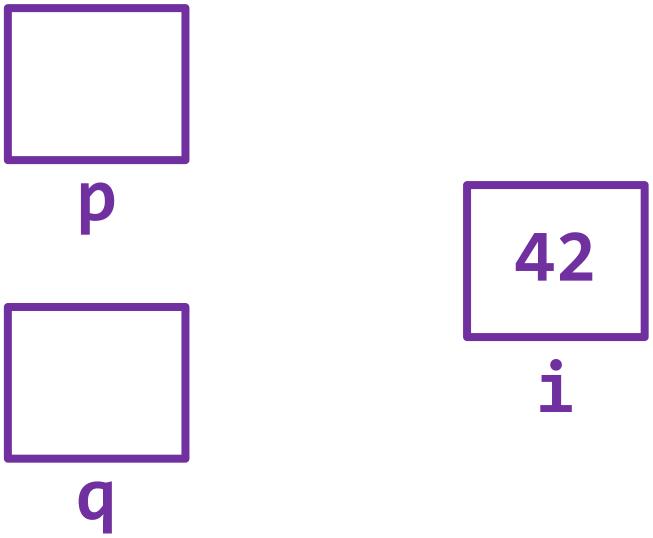
Declare and initialize i to 42.

Declare integer pointers p and q but do not initialize them yet.

location.

- **Example:**

```
int i = 42;
int *p, *q;
p = &i;
q = p;
*q = 32;
printf("%d %d %d\n", *p, *q, i);
```



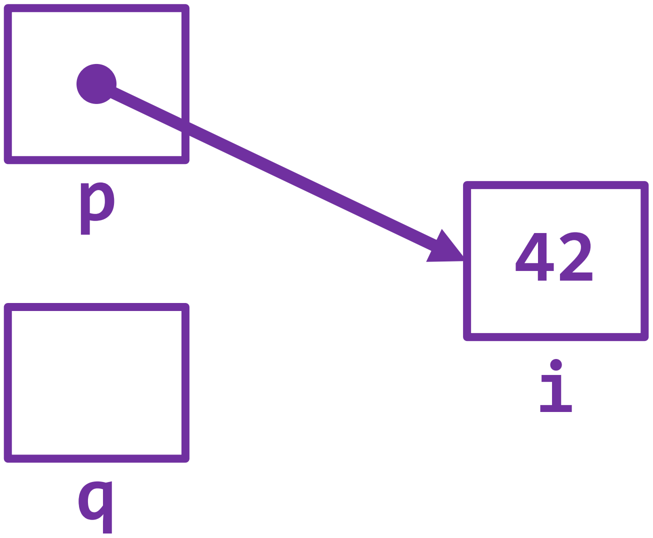
Output: 32 32 32

Copying Pointers

- In addition to setting a point to the address of another variable, we can set a pointer equal to another pointer (i.e. copy it).
- This does not copy the value that the pointer is pointing at, it simply makes another pointer that points to the same memory

Point p to i (i.e. set p equal to the memory address of i).

```
int i = 42;
int *p, *q;
p = &i;
q = p;
*q = 32;
printf("%d %d %d\n", *p, *q, i);
```



Output: 32 32 32

Copying Pointers

Set q equal to p.

This sets q equal to the memory address stored in p.

We could say that this points q to the same place p is pointing.

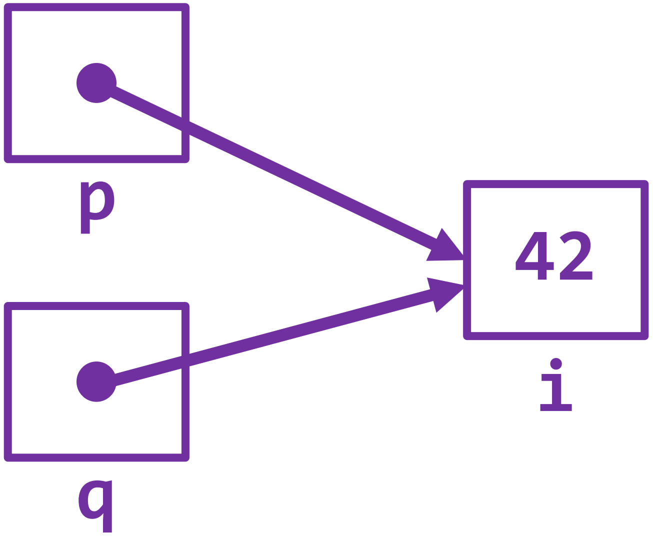
lress of another variable, we
ter (i.e. copy it).

ointer is pointing at, it
nts to the same memory

- **Example:**

```
int i = 42;
int *p, *q;
p = &i;
q = p;
*q = 32;
printf("%d %d %d\n", *p, *q, i);
```

Output: 32 32 32



Copying Pointers

- In addition to setting a point to the address of another variable, we

ter (i.e. copy it).

ointer is pointing at, it

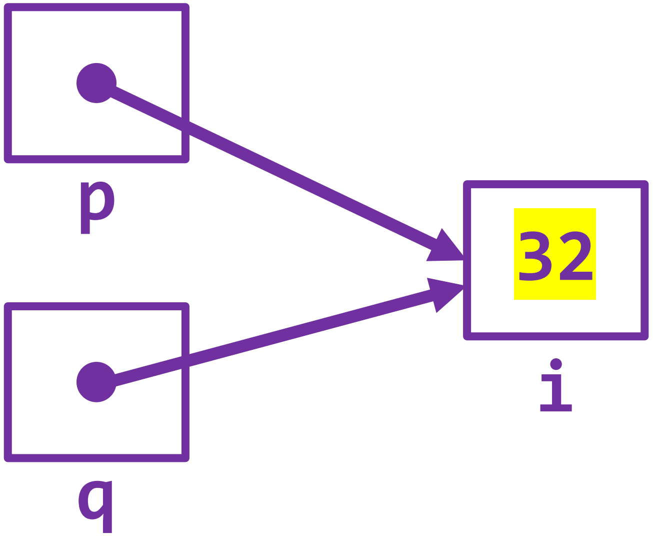
nts to the same memory

Set the value of the location q is pointing to 32.

This changes the value of `i` to 32 as q is pointing at `i`.

- **Example:**

```
int i = 42;
int *p, *q;
p = &i;
q = p;
*q = 32;
printf("%d %d %d\n", *p, *q, i);
```



Output: 32 32 32

Copying Pointers

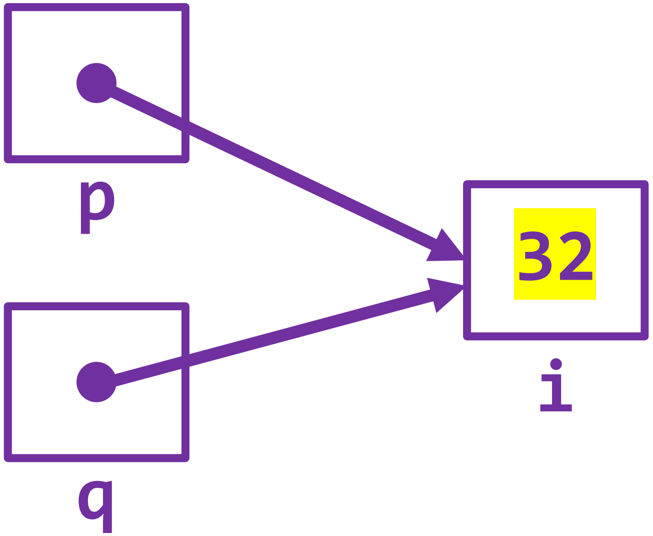
- In addition to setting a point to the address of another variable, we can set a pointer equal to another pointer (i.e. copy it).

- This does not copy the value that the pointer is pointing at, it nts to the same memory

All outputs are 32 as both q and p point to i and i now equals 32.

- **Example:**

```
int i = 42;
int *p, *q;
p = &i;
q = p;
*q = 32;
printf("%d %d %d\n", *p, *q, i);
```



Output: 32 32 32

Copying Pointers

- It is important to note that `q = p` and `*q = *p` have very different meanings.
- The first is pointer assignment as in the last example.
- The second is setting the value of the location q points at.

- **Example:**

```
int j, i = 42;
int *p, *q;
p = &i;
q = &j;
*q = *p;
```

Copying Pointers

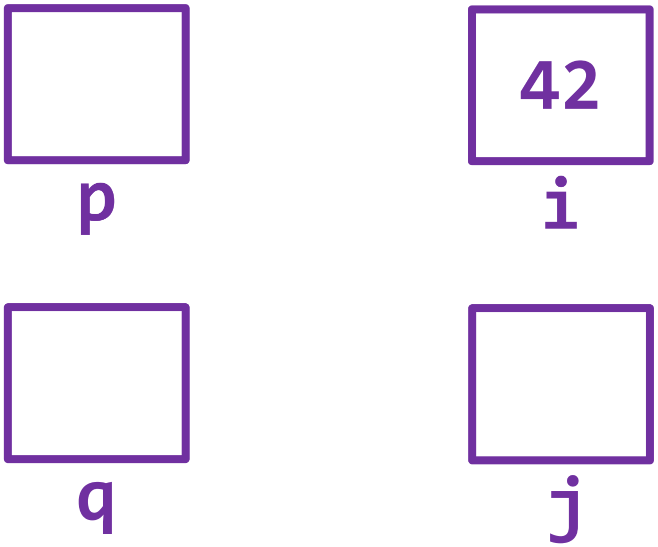
Declare and initialize integer `i` to 42.

Declare but do not initialize integer `j`.

Declare integer pointers p and q but do not initialize them yet.

- **Example:**

```
int j, i = 42;
int *p, *q;
p = &i;
q = &j;
*q = *p;
```



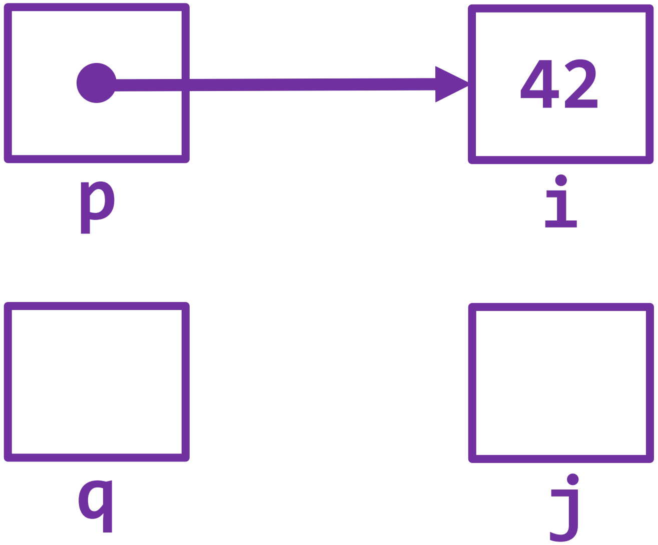
Copying Pointers

- It is important to note that `q = p` and `*q = *p` have very different meanings.
- The first is pointer assignment as in the last example.
- The second is setting the value of the location q points at.

- **Example:**

```
int j, i = 42;
int *p, *q;
p = &i;
q = &j;
*q = *p;
```

Point p to variable `i`.



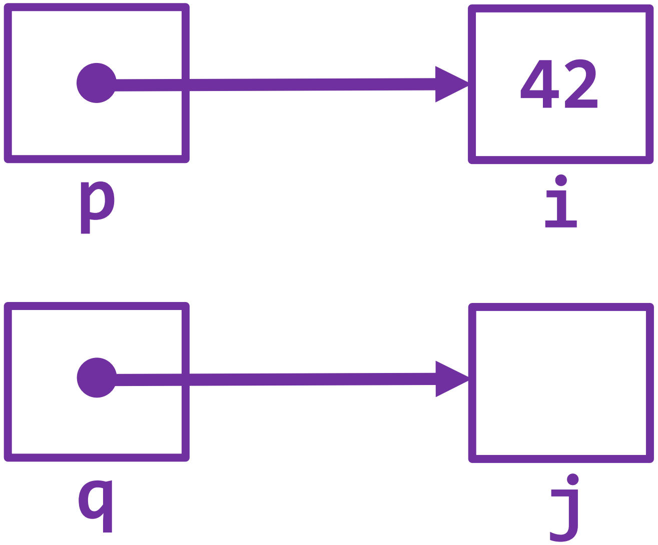
Copying Pointers

- It is important to note that `q = p` and `*q = *p` have very different meanings.
- The first is pointer assignment as in the last example.
- The second is setting the value of the location q points at.

- **Example:**

```
int j, i = 42;
int *p, *q;
p = &i;
q = &j;
*q = *p;
```

Point q to variable j.

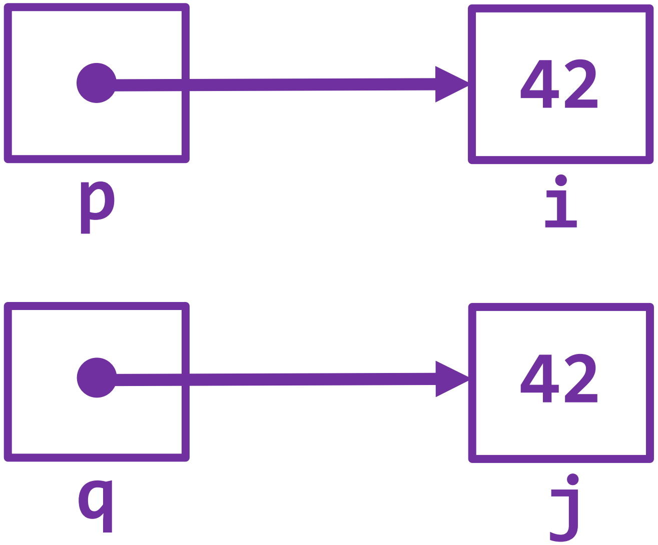


Copying Pointers

- It is important to note that `q = p` and `*q = *p` have very different meanings.
- The first is pointer assignment as in the last example.
- The second is setting the value of the location `q` points at.

- **Example:**

```
int j, i = 42;
int *p, *q;
p = &i;
q = &j;
*q = *p;
```

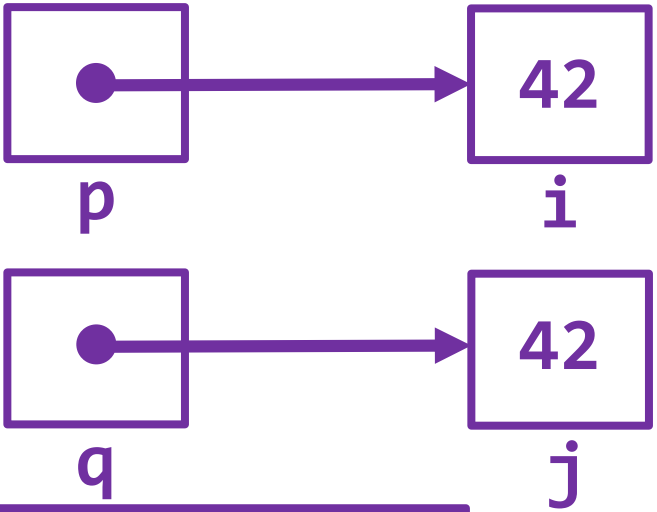


Set the value of what `q` points to (`j`) to the value of what `p` points to (`i`). In this case, `j` is set to 42 as `i` has the value 42.

Copying Pointers

- It is important to note that q = p and *q = *p have very different meanings.
- The first is pointer assignment as in the last example.
- The second is setting the value of the location q points at.
- **Example:**

```
int j, i = 42;
int *p, *q;
p = &i;
q = &j;
*q = *p;
```



It is important to note that the values of i and j are in no way linked together in this case. If you did either of the following after this example:

```
i = 5;
*p = 3;
```

the value of j would unaffected (would remain as 42).

Pointers as Function Arguments

- We have already used pointers to pass variables by reference to functions.
- We can also do this with pointer variables that we have declared.
- **Example:**

/cs2211/week10/ex5.c

```
#include <stdio.h>

void swap(int *a, int *b) {
        int temp = *a;
        *a = *b;
        *b = temp;
}

int main() {
        int i = 5, j = 7;
        int *p = &i, *q = &j;

        printf("Before swap:\n\ti is %d\n\tj is %d\n", i, j);
        swap(p, q);
        printf("After swap:\n\ti is %d\n\tj is %d\n", i, j);

        return 0;
}
```

Pointers as Function Arguments

- We have already used pointers to pass variables by reference to functions.
- We can also do this with pointer variables that we have declared.
- **Example:**

/cs2211/week10/ex5.c

```
#include <stdio.h>
void swap(int *a, int *b) {
        int temp = *a;
        *a = *b;
        *b = temp;
}

int main() {
        int i = 5, j = 7;
        int *p = &i, *q = &j;

        printf("Before swap:\n\ti is %d\n\tj is %d\n", i, j);
        swap(p, q);
        printf("After swap:\n\ti is %d\n\tj is %d\n", i, j);

        return 0;
}
```

Same swap function we saw in ex9b.c from week 9b (*/cs2211/week9/ex9b.c*).

Takes two integer pointers and swaps the values they are pointing to.

Pointers as Function Arguments

- We have already used pointers to pass variables by reference to functions.
- We can also do this with pointer variables that we have declared.
- **Example:**

/cs2211/week10/ex5.c

```
#include <stdio.h>
void swap(int *a, int *b) {
        int temp = *a;
        *a = *b;
        *b = temp;
}
int main() {
        int i = 5, j = 7;
        int *p = &i, *q = &j;

        printf("Before swap:\n\ti is %d\n\tj is %d\n", i, j);
        swap(p, q);
        printf("After swap:\n\ti is %d\n\tj is %d\n", i, j);

        return 0;
}
```

Last time we called this function by using the & operator, for example:

```
swap(&i, &j);
```

This time we are using the integer pointers we declared, p and q, which point to i and j.

Pointers as Function Arguments

- We have already used pointers to pass variables by reference to functions.
- We can also do this with pointer variables that we have declared.
- **Example:**

/cs2211/week10/ex5.c

```
#include <stdio.h>

void swap(int *a, int *b) {
        int temp = *a;
        *a = *b;
        *b = temp;
}

int main() {
        int i = 5, j = 7;
        int *p = &i,

        printf("Befo                                          ;
        swap(p, q);
        printf("After swap:\n\ti is %d\n\tj is %d\n", i, j);

        return 0;
}
```

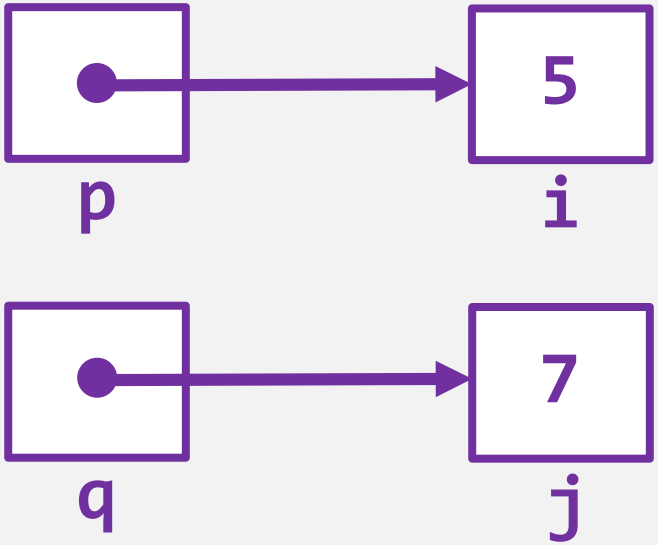
Why don't we need the & in front of p and q here?

Pointers as Function Arguments

- We have already used pointers to pass variables by reference to functions.
- We can also do this with pointer variables that we have declared.
- **Example:**

/cs2211/week10/ex5.c

```
#include <stdio.h>

void swap(int *a, int *b) {
        int temp = *a;
        *a = *b;
        *b = temp;
}

int main() {
        int i = 5, j = 7;
        int *p = &i, *q = &j;

        printf("Before swap:\n\ti is %d\n\tj is %d\n", i, j);
        swap(p, q);
        printf("After swap:\n\ti is %d\n\tj is %d\n", i, j);

        return 0;
}
```

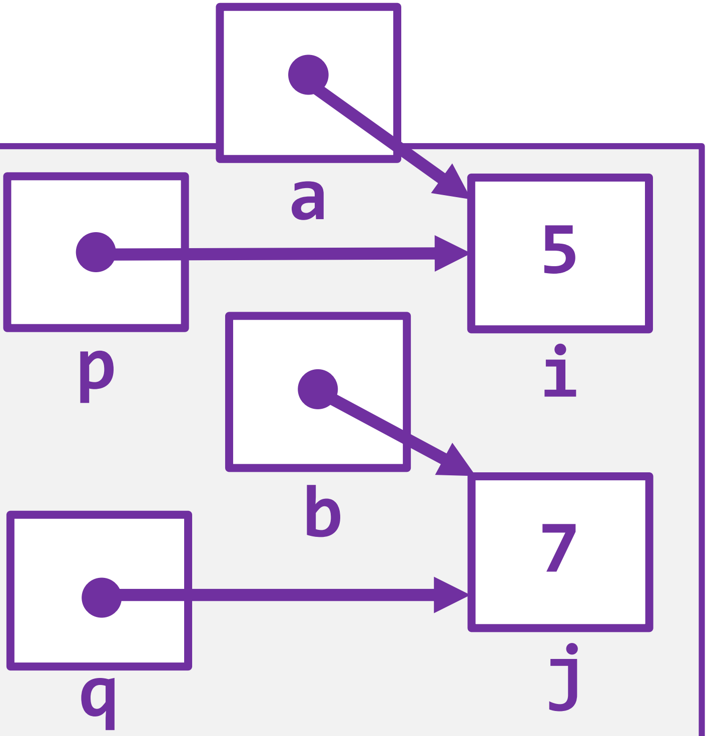


Pointers as Function Arguments

- We have already used pointers to pass variables by reference to functions.
- We can also do this with pointer variables that we have declared.
- **Example:**

/cs2211/week10/ex5.c

```
#include <stdio.h>

void swap(int *a, int *b) {
        int temp = *a;
        *a = *b;
        *b = temp;
}

int main() {
        int i = 5, j = 7;
        int *p = &i, *q = &j;

        printf("Before swap:\n\ti is %d\n\tj is %d\n", i, j);
        swap(p, q);
        printf("After swap:\n\ti is %d\n\tj is %d\n", i, j);

        return 0;
}
```

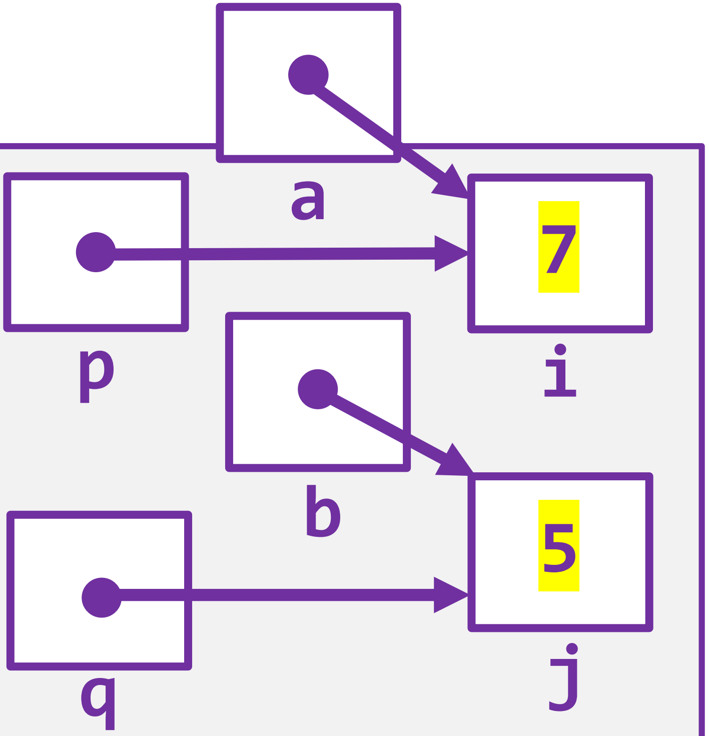


Pointers as Function Arguments

- We have already used pointers to pass variables by reference to functions.
- We can also do this with pointer variables that we have declared.
- **Example:**

/cs2211/week10/ex5.c

```c
#include <stdio.h>

void swap(int *a, int *b) {
        int temp = *a;
        *a = *b;
        *b = temp;
}

int main() {
        int i = 5, j = 7;
        int *p = &i, *q = &j;

        printf("Before swap:\n\ti is %d\n\tj is %d\n", i, j);
        swap(p, q);
        printf("After swap:\n\ti is %d\n\tj is %d\n", i, j);

        return 0;
}
```



s Function Arguments

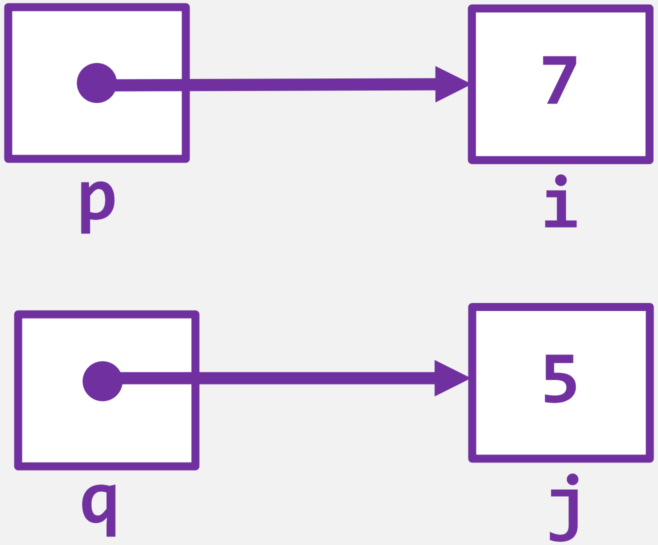
used pointers to pass variables by reference to

his with pointer variables that we have declared.

Output:

```
Before swap:
        i is 5
        j is 7
After swap:
        i is 7
        j is 5
```

```
void swap(int *a, int *b) {
      int temp = *a;
      *a = *b;
      *b = temp;
}

int main() {
      int i = 5, j = 7;
      int *p = &i, *q = &j;

      printf("Before swap:\n\ti is %d\n\tj is %d\n", i, j);
      swap(p, q);
      printf("After swap:\n\ti is %d\n\tj is %d\n", i, j);

      return 0;
}
```



Pointers as Function Arguments

- The variable arguments to `scanf` are also pointers.
- This allows `scanf` to set the value of the variables by reference.
- If we wish to set the value that a pointer is pointing to using `scanf` we would not use the & operator.

- **Example:**

```
int i;
int *p = &i;
scanf("%d", p);
```

Should not use & here, p is already a pointer.

In this example, scanf is setting the value of variable `i` as p points to `i`.

Pointers as Function Arguments

- The variable arguments to `scanf` are also pointers.
- This allows `scanf` to set the value of the variables by reference.
- If we wish to set the value that a pointer is pointing to using `scanf` we would not use the & operator.

- **Bad Example:**

```
int i;
int *p = &i;
scanf("%d", &p);
```

Now `scanf` is setting the value of the pointer (the memory address) and not the value of `i`.

Returning a Pointer

- Functions can return pointers like they would normal values.
- **Example:**

/cs2211/week10/ex6.c

```
#include <stdio.h>

int * max(int *a, int *b) {
        if(*a > *b)
                return a;
        else
                return b;
}

int main() {
        int i = 46, j = 24, *p;

        p = max(&i, &j);

        printf("max is %d\n", *p);
        return 0;
}
```

The function `max` takes two integer pointers and returns an integer pointer.

The pointer returned will point to the larger of the values a and b point to.

When called with `&i` and `&j`, `max` will return a pointer to either `i` or `j`.

In this case a pointer to `i` is returned as `i > j`.

Returning a Pointer

- Functions can return pointers like they would normal values.
- **Bad Example:**

```
#include <stdio.h>
int * max(int a, int b) {
        int big;
        if(a > b)
                big = a;
        else
                big = b;
        return &big;
}
int main() {
        int i = 46, j = 24, *p;
        p = max(i, j);
        printf("max is %d\n", *p);
        return 0;
}
```

You should **never** return a pointer to a local function variable that is declared like this.

The variable big stops existing after the function finishes.

The compiler might reuse this memory, clear it, etc.

You can no longer be sure of what will happen to it.

Returning a Pointer

- It is ok to return a pointer to an array element that was passed to the function and can be quite useful.
- **Example:**

/cs2211/week10/ex7.c

```
#include <stdio.h>

int * max_a(int a[], int n) {
        int i, *max = &a[0];

        for(i = 1; i < n; i++)
                if(a[i] > *max)
                        max = &a[i];

        return max;
}

int main() {
        int *p, a[5] = {77, 89, 90, 38, 74};

        p = max_a(a, 5);

        printf("max is %d\n", *p);
        return 0;
}
```

The function max_a returns a pointer to the largest element in the array.

Printing a Pointer

- Sometimes it can be useful to print the address a pointer is pointing to (especially for debugging purposes).
- `printf` supports the format specifier **%p** which prints a memory address in hexadecimal format.
- **Example:**

```
int i;
int *q = &i;
printf("%p", q);
```

Output:

```
0x7ffe1481cd54
```

The exact address output will depend on a larger number of factors. In many cases it will be different each time you run the program.

In-class Activity

Fix the errors in the following function and write a main method to test and call the function:

/cs2211/week10/q1.c

```
void avg_sum(double a[], int n, double *avg, double *sum) {
        int i;

        sum = 0.0;
        for (i = 0; i < n; i++)
                sum += a[i];
        avg = sum / n;
}
```

This function should compute the sum and average of the values in the array a. The results should be stored in the variables sum and avg point to.

In-class Activity

Fix the errors in the following function and write a main method to test and call the function:

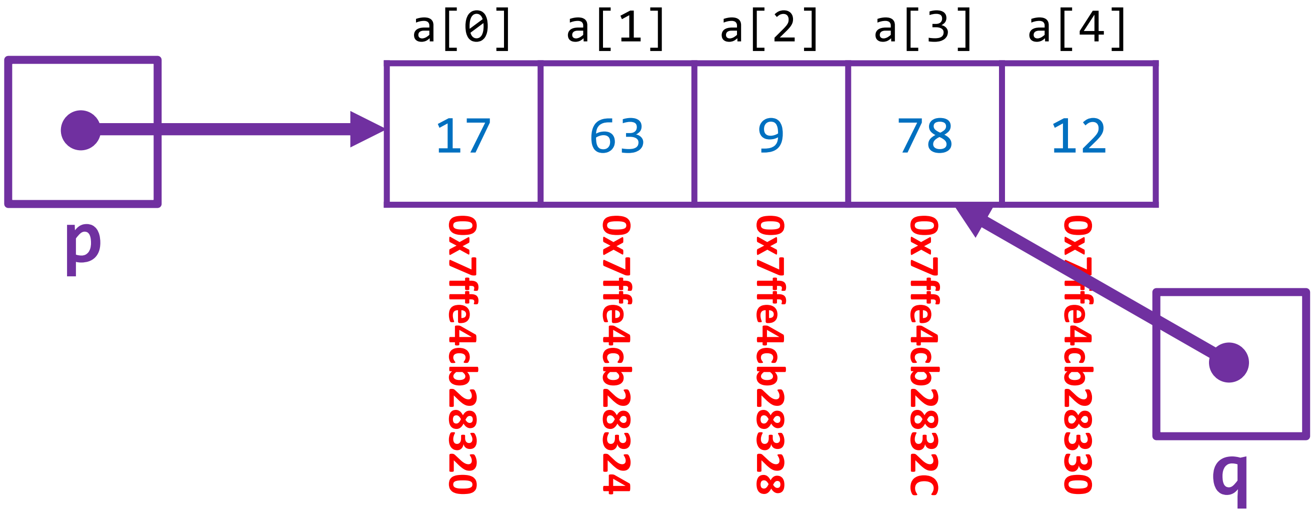
```
void avg_sum(double a[], int n, double *avg, double *sum) {
        int i;

        *sum = 0.0;
        for (i = 0; i < n; i++)
                *sum += a[i];
        *avg = *sum / n;
}

int main() {
        double a[7] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
        double sum, avg;

        avg_sum(a, 7, &avg, &sum);
        printf("avg: %f   sum: %f\n", avg, sum);
        return 0;
}
```

Pointer Arithmetic & Arrays

Pointers to Arrays

- We have already seen that pointers can point to individual array elements.
- **Example:**

```
int a[5] = {17, 63, 9, 78, 12};
int *p = &a[0];
int *q = &a[3];
```



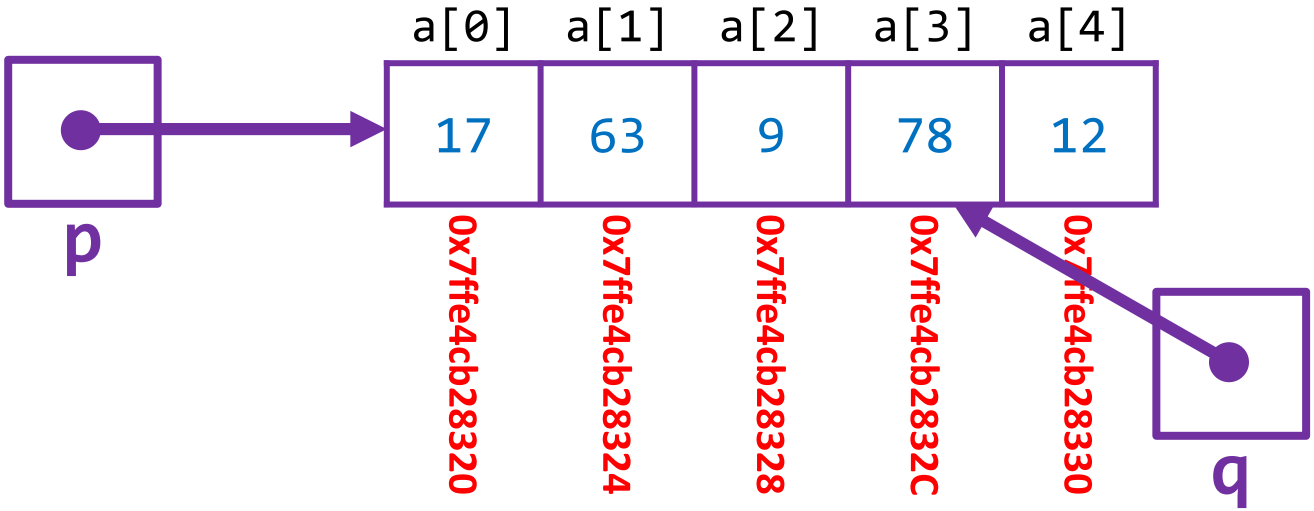
Pointers to Arrays

- We have already
  elements.

- **Example:**

In this example, the pointer p points to the 1st element of a (`a[0]`) and the pointer q points to the 4th element of a (`a[3]`).

```
int a[5] = {17, 63, 9, 78, 12};
int *p = &a[0];
int *q = &a[3];
```

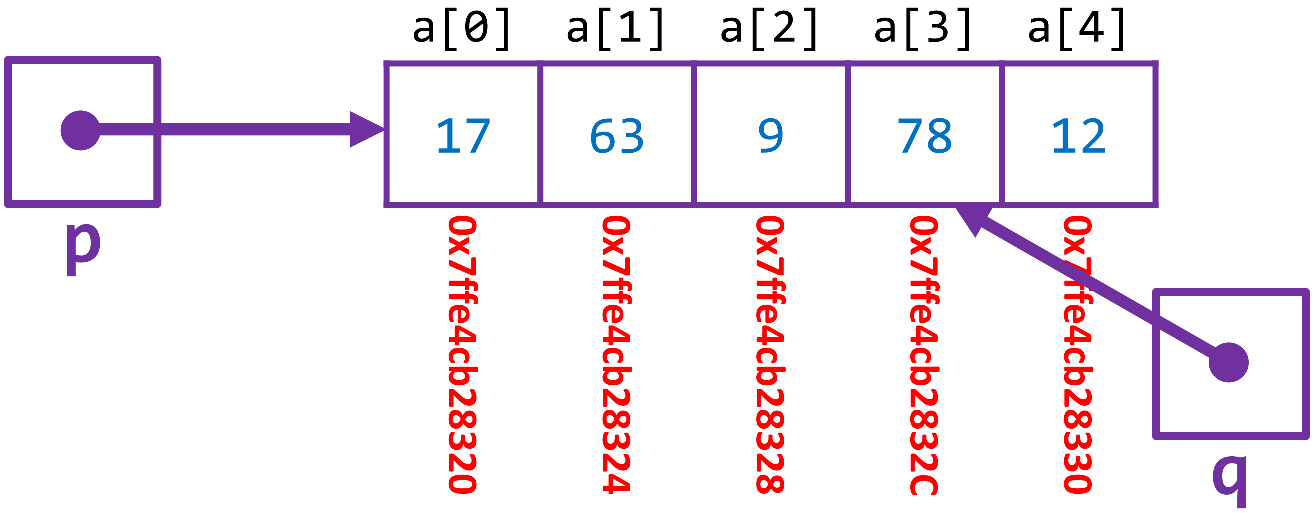


Pointers to Arrays

- We have alrea
  elements.
- **Example:**

The value of p is the memory address of a[0], that is 0x7ffe4cb28320 and the value of q is the memory address of a[3], that is 0x7ffe4cb2832C.

```
int a[5] = {17, 63, 9, 78, 12};
int *p = &a[0];
int *q = &a[3];
```



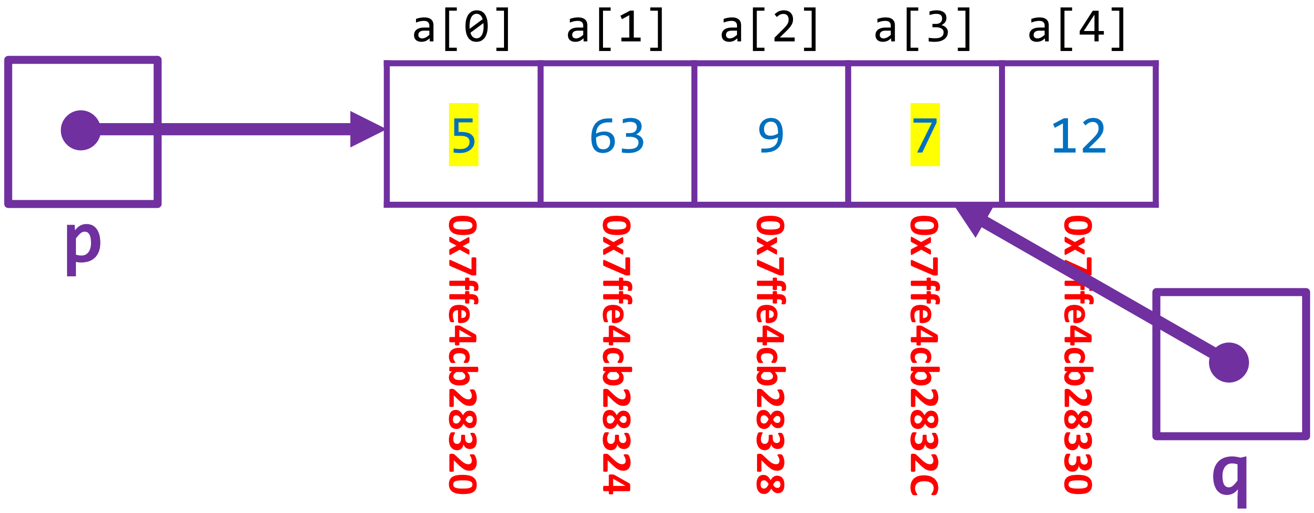
Pointers to Arrays

- We have alrea elements.

If we set `*p = 5` and `*q = 7` the values in the array would be updated (as shown).

- **Example:**

```
int a[5] = {17, 63, 9, 78, 12};
int *p = &a[0];
int *q = &a[3];
```



Pointer Arithmetic

- When we have pointers to arrays, C allows us to do a limited number of arithmetic operations on the pointers.
- The following operations are supported:
  - Adding an integer to a pointer.
  - Subtracting an integer from a pointer.
  - Subtracting one pointer from another.
- The result of adding or subtracting an integer to/from a pointer is a new pointer.
- The result of subtracting one pointer from another is an integer.
- Adding 1 to a pointer would increase the address it points to by "one". In this case "one" is not one byte but one element in the array. For an integer pointer, this would increase the address by 4 bytes (assuming an integer is 4 bytes).

Pointer Arithmetic

Example:

```
int a[5] = {17, 63, 9, 78, 12};
int *q, *p = &a[0];

p += 1;
p += 3;
p -= 2;
q = p - 1;
q = p + 2;
```

a[0]	a[1]	a[2]	a[3]	a[4]
17	63	9	78	12
0x7ffe4cb28320	0x7ffe4cb28324	0x7ffe4cb28328	0x7ffe4cb2832C	0x7ffe4cb28330

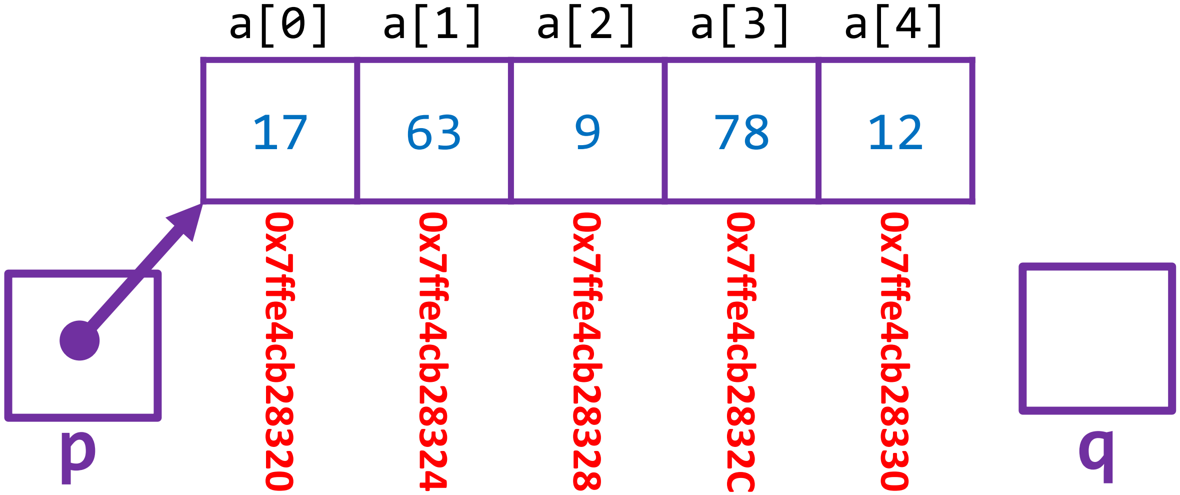
Pointer Arithmetic

Variable	Value
p	0x7ffe4cb28320
q	?

Example:

```
int a[5] = {17, 63, 9, 78, 12};
int *q, *p = &a[0];

p += 1;
p += 3;
p -= 2;
q = p - 1;
q = p + 2;
```

Declares integer pointers q and p. Only p is initialized (to point to the first element of a).



a[0]	a[1]	a[2]	a[3]	a[4]
17	63	9	78	12
0x7ffe4cb28320	0x7ffe4cb28324	0x7ffe4cb28328	0x7ffe4cb2832C	0x7ffe4cb28330

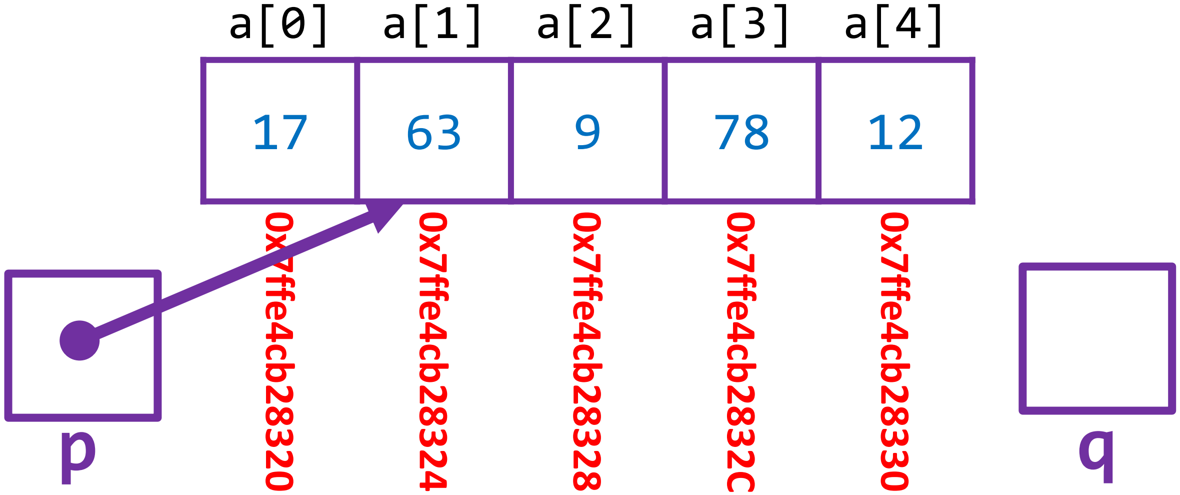
Pointer Arithmetic

Variable	Value
p	0x7ffe4cb28324
q	?

Example:

```
int a[5] = {17, 63, 9, 78, 12};
int *q, *p = &a[0];

p += 1;
p += 3;
p -= 2;
q = p - 1;
q = p + 2;
```

The pointer p is incremented by one. This makes it point at the next element in a. Note that the address was increased by 4 not 1, as integers in this case are 4 bytes.



Pointer Arithmetic

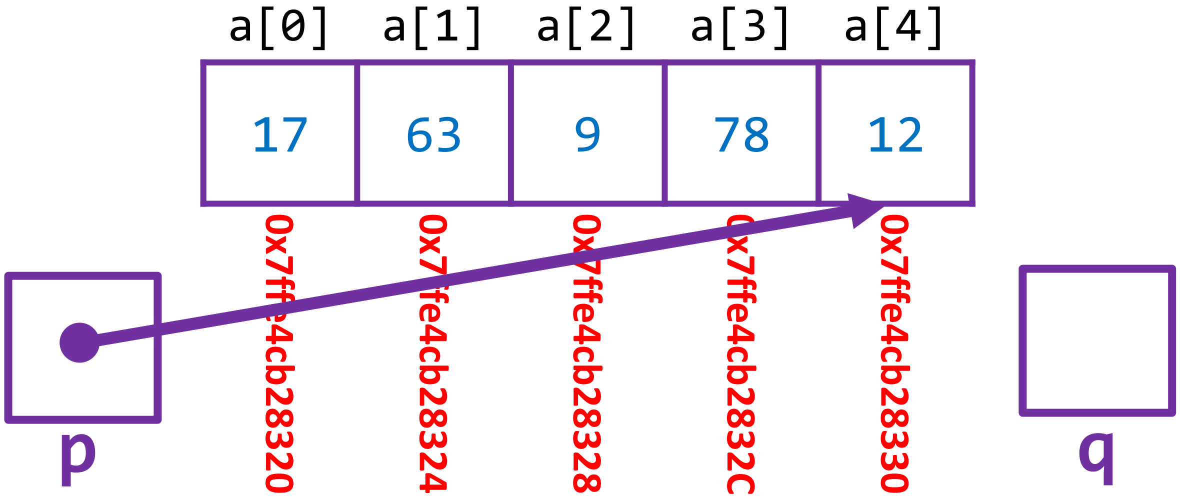
Variable	Value
p	0x7ffe4cb28330
q	?

Example:

```
int a[5] = {17, 63, 9, 78, 12};
int *q, *p = &

p += 1;
p += 3;
p -= 2;
q = p - 1;
q = p + 2;
```

The pointer p is further incremented by 3, making it point to the last element of a.

If we did `printf("%d", *p);` after this line the output would be 12.



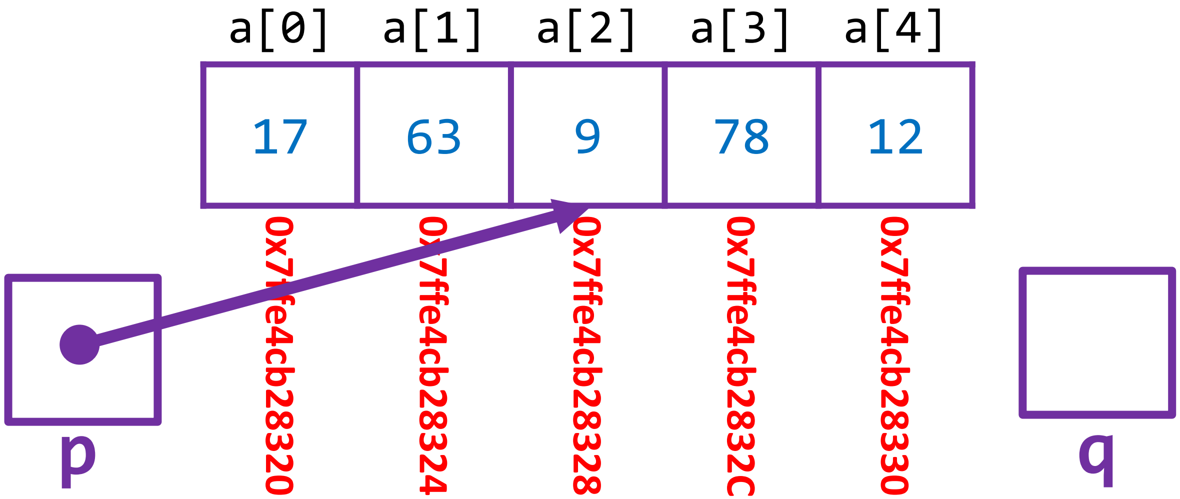
Pointer Arithmetic

Variable	Value
p	0x7ffe4cb28328
q	?

Example:

```
int a[5] = {17, 63, 9, 78, 12};
int *q, *p = &a[0];

p += 1;
p += 3;
p -= 2;
q = p - 1;
q = p + 2;
```

The pointer p is decremented by 2, making it point to a[2].



a[0]	a[1]	a[2]	a[3]	a[4]
17	63	9	78	12
0x7ffe4cb28320	0x7ffe4cb28324	0x7ffe4cb28328	0x7ffe4cb2832C	0x7ffe4cb28330

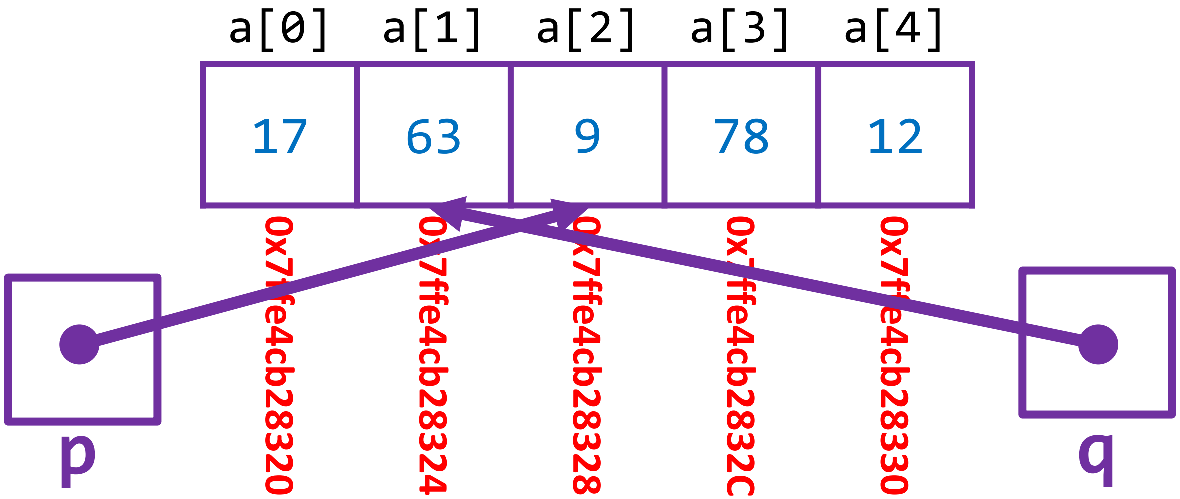
Pointer Arithmetic

Variable	Value
p	0x7ffe4cb28328
q	0x7ffe4cb28324

Example:

```
int a[5] = {17, 63, 9, 78, 12};
int *q, *p = &a[0];

p += 1;
p += 3;
p -= 2;
q = p - 1;
q = p + 2;
```

The pointer q is set to the address p is pointing to minus one element. This result in q pointing to a[1].



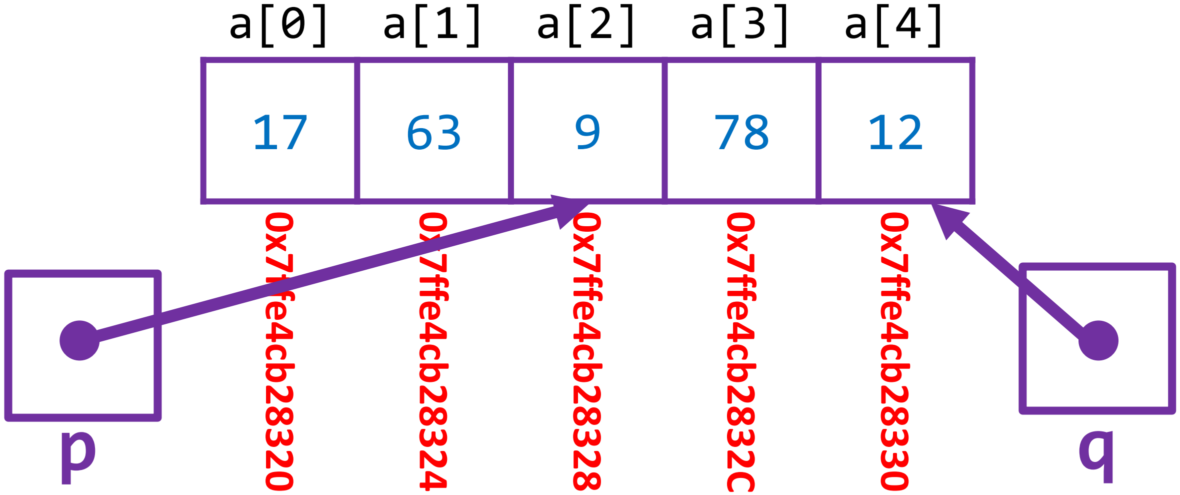
Pointer Arithmetic

Variable	Value
p	0x7ffe4cb28328
q	0x7ffe4cb28330

Example:

```
int a[5] = {17, 63, 9, 78, 12};
int *q, *p = &a[0];

p += 1;
p += 3;
p -= 2;
q = p - 1;
q = p + 2;
```

The pointer q is set to the address p is pointing to plus two elements. This results in q pointing to a[4].



Pointer Arithmetic

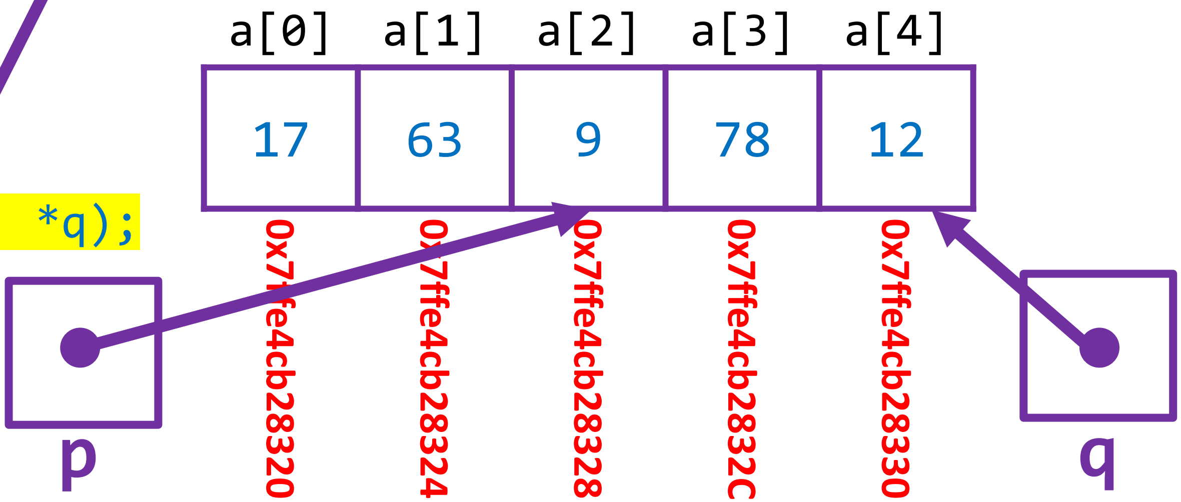
Variable	Value
p	0x7ffe4cb28328
q	0x7ffe4cb28330

Example:

```
int a[5] = {17, 63, 9, 78, 12};
int *q, *p = &a[0];

p += 1;
p += 3;
p -= 2;
q = p - 1;
q = p + 2;
```

If we added a printf statement at the end to output the values of *p and *q the result would be: 9 12

```
printf("%d %d", *p, *q);
```



a[0]	a[1]	a[2]	a[3]	a[4]
17	63	9	78	12
0x7ffe4cb28320	0x7ffe4cb28324	0x7ffe4cb28328	0x7ffe4cb2832C	0x7ffe4cb28330

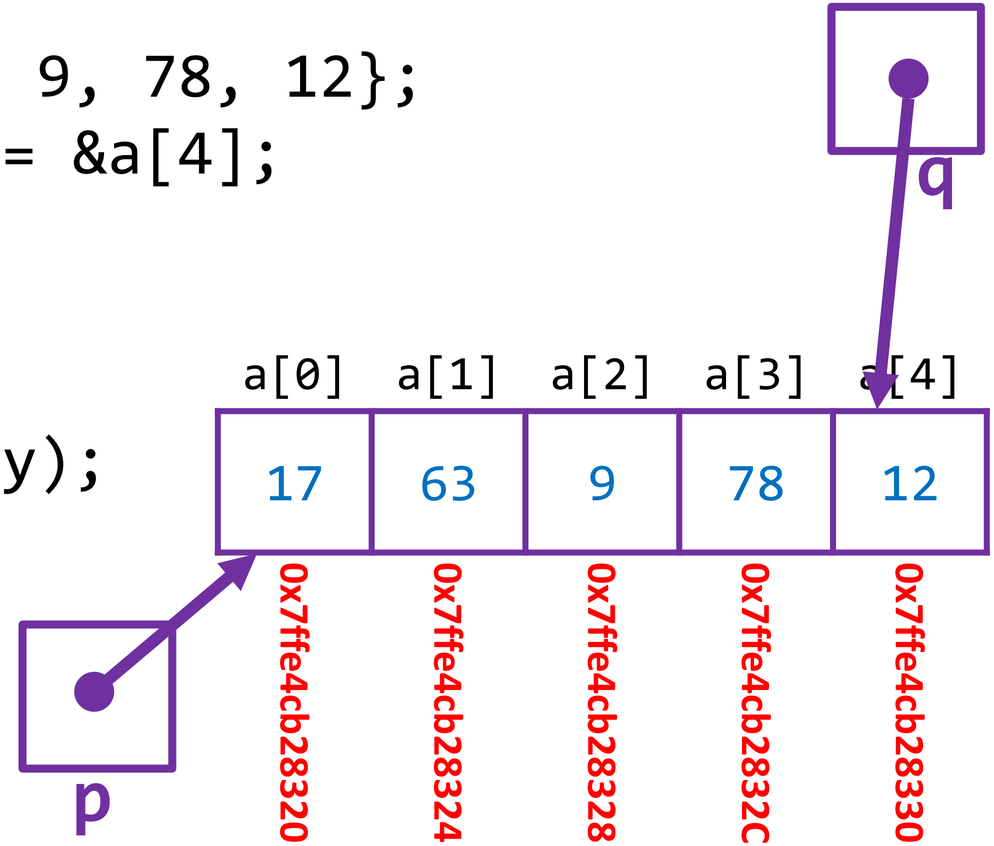
Pointer Arithmetic

- Subtracting two pointers gives us the difference between them in number of elements.

- **Example:**

```
int a[5] = {17, 63, 9, 78, 12};
int *p = &a[0], *q = &a[4];

int x = q - p;
int y = p - q;
printf("%d %d", x, y);
```

Output: 4 -4



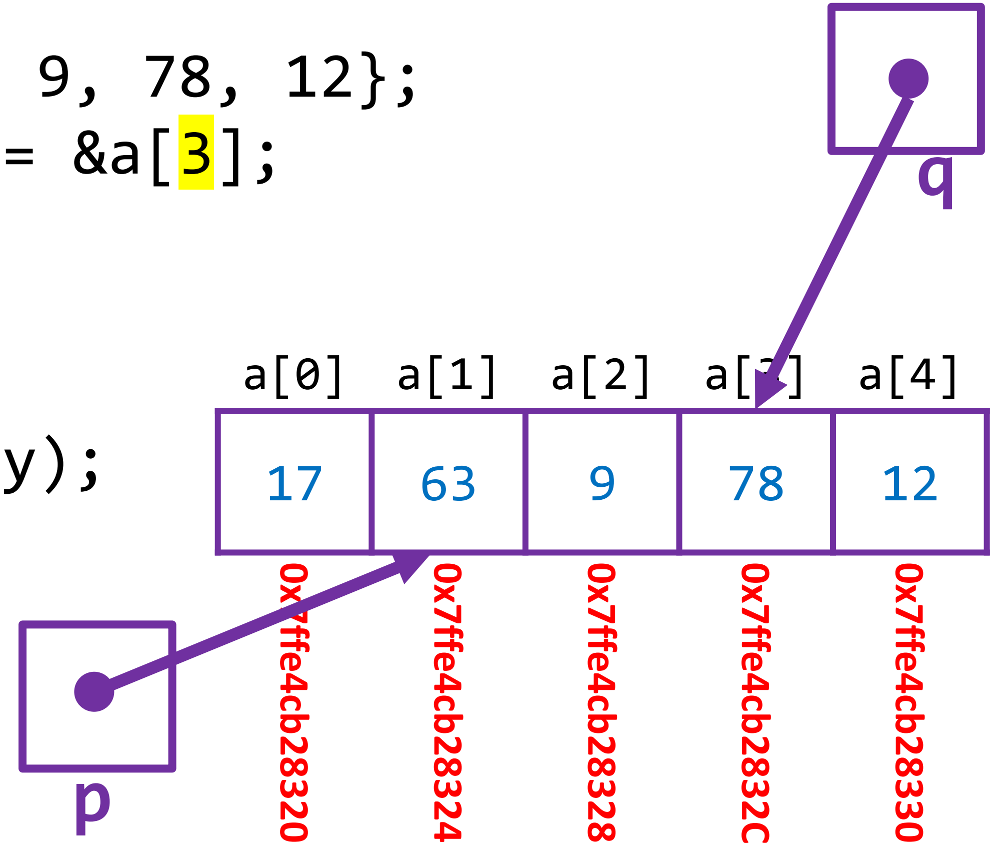
Pointer Arithmetic

- Subtracting two pointers gives us the difference between them in number of elements.

- **Example:**

```
int a[5] = {17, 63, 9, 78, 12};
int *p = &a[1], *q = &a[3];

int x = q - p;
int y = p - q;
printf("%d %d", x, y);
```

Output: `2 -2`



Pointer Arithmetic

- In most cases, the following will lead to **undefined behaviour**:
  - Adding an integer to a pointer that is not pointing to an array.
  - Subtracting an integer to a pointer that is not pointing to an array.
  - Adding to or subtracting from a pointer such that it is now pointing outside of the bounds of the array.
  - Subtracting two pointers that are not part of the same array.
- The issues is that we don't have a guarantee of what might be in these memory locations.

Comparing Pointers

- The relational operators <, <=, >, >= and the equality operators == and != can be used on pointers.
- The behaviour of the relational operators on pointers is only defined for pointers of the same array.
- Result depends on the address stored in the pointers.
- Larger address is considered to be a later element in the array.
- **Example:** For some integer array a

```
int *p = &a[4], *q = &a[2];
printf("%d %d\n", p > q, p == q);
```

Output: 1 0

Comparing Pointers

- The relational operators <, <=, >, >= and the equality operators == and != can be used on pointers.
- The behaviour of the relational operators on pointers is only defined for pointers of the same array.
- Result depends on the address stored in the pointers.
- Larger address is considered to be a later element in the array.
- **Example:** For some integer array a

```
int *p = &a[4], *q = &a[2];
printf("%d %d\n", p > q, p == q);
```

Output: 1 0

It is true that p is greater than q. As p is pointing to a later element in the array, it would have a larger address than q.

Comparing Pointers

- The relational operators <, <=, >, >= and the equality operators == and != can be used on pointers.
- The behaviour of the relational operators on pointers is only defined for pointers of the same array.
- Result depends on the address stored in the pointers.
- Larger address is considered to be a later element in the array.
- **Example:** For some integer array a

```
int *p = &a[4], *q = &a[2];
printf("%d %d\n", p > q, p == q);
```

Output: `1 0`

It is not true that p is equal to q as they do not point to the same address.

Using Pointer Arithmetic in Loops

- Pointer arithmetic can be very useful for processing and dealing with arrays when combined with loops (for loops in particular).

- **Example:**

/cs2211/week10/ex8.c

```c
#include <stdio.h>
#define N 10

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

        printf("Input %d numbers:\n", N);
        for(p = &a[0]; p < &a[N]; p++)
                scanf("%d", p);

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

        printf("Sum is %d\n", sum);
        return 0;
}
```

Using Pointer Arithmetic in Loops

- Pointer arithmetic can be very useful for processing and dealing with arrays when combined with loops (for loops in particular).

- **Example:**

/cs2211/week10/ex8.c

```
#include <stdio.h>
#define N 10

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

        printf("Input %d numbers:\n", N);
        for(p = &a[0]; p < &a[N]; p++)
                scanf("%d", p);

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

        printf("Sum is %d\n", sum);
        return 0;
}
```

Start the loop by setting p to point at the first element of a.

Using Pointer Arithmetic in Loops

- Pointer arithmetic can be very useful for processing and dealing with arrays when combined with loops (for loops in particular).

- **Exampl**

/cs2211/week1

```
#include <
#define N

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

        printf("Input %d numbers:\n", N);
        for(p = &a[0]; p < &a[N]; p++)
                scanf("%d", p);

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

        printf("Sum is %d\n", sum);
        return 0;
}
```

The loop will keep running as long as p is less than the first address outside of the bounds of a.

Note that a is of size N and the last element is a[N-1] so a[N] would be just outside of the bounds of the array.

Using Pointer Arithmetic in Loops

- Pointer arithmetic can be very useful for processing and dealing with arrays when combined with loops (for loops in particular).

- **Example:**

/cs2211/week10/ex8.c

```
#include <stdio.h>
#define N 10

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

        printf("Input %d numbers:\n", N);
        for(p = &a[0]; p < &a[N]; p++)
                scanf("%d", p);

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

        printf("Sum is %d\n", sum);
        return 0;
}
```

At the end of each loop iteration, increment the value of p. This moves p to the next element of a.

Using Pointer Arithmetic in Loops

- Pointer arithmetic can be very useful for processing and dealing with arrays when combined with loops (for loops in particular).

- **Example:**

/cs2211/week10/ex8.c

```c
#include <stdio.h>
#define N 10

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

        printf("Input %d numbers:\n", N);
        for(p = &a[0]; p < &a[N]; p++)
                scanf("%d", p);

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

        printf("Sum is %d\n", sum);
        return 0;
}
```

Read an integer in from the user and save it in the location p points to.

Note that we do not use an & here, as p is already a pointer/memory address.

Using Pointer Arithmetic in Loops

- Pointer arithmetic can be very useful for processing and dealing with arrays when c

- **Example:**

/cs2211/week10/ex8.c

```c
#include <stdio.h>
#define N 10

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

        printf("Input %d numbers:\n", N);
        for(p = &a[0]; p < &a[N]; p++)
                scanf("%d", p);

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

        printf("Sum is %d\n", sum);
        return 0;
}
```

Same idea as the first for loop but we are summing all of the elements in array a.

Need to dereference the pointer p (using the * operator) to get the value of the array element rather than the memory address.

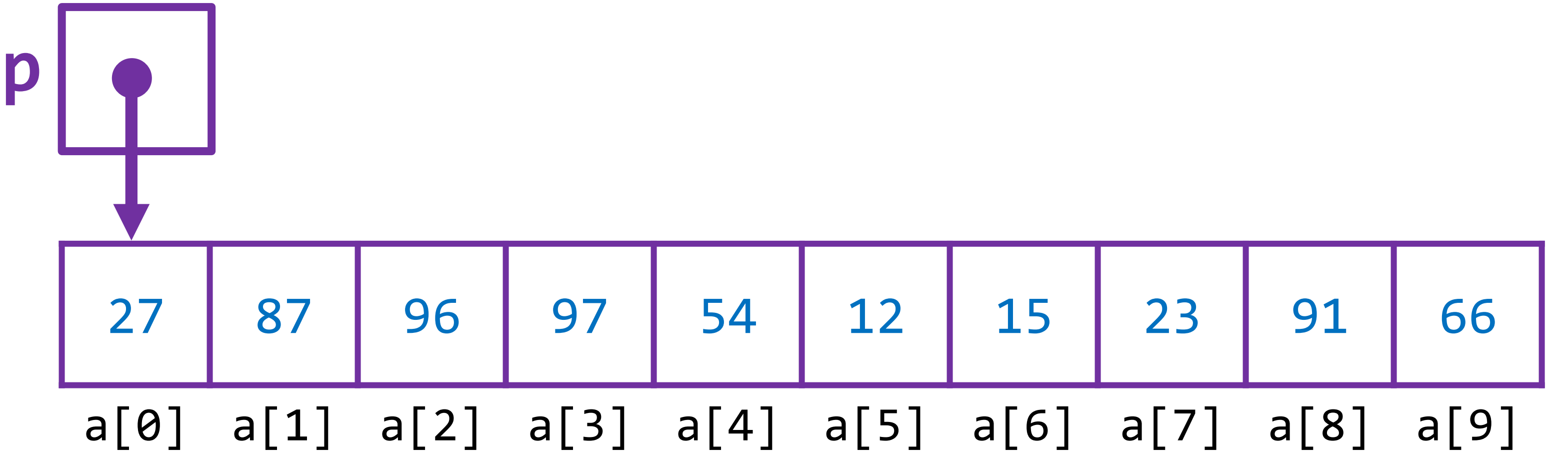
Using Pointer Arithmetic in Loops

- Pointer arithmetic can be very useful for processing and dealing with arrays when combined with loops (for loops in particular).

- **Example:**

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

Expression	Value
*p	27
sum	0



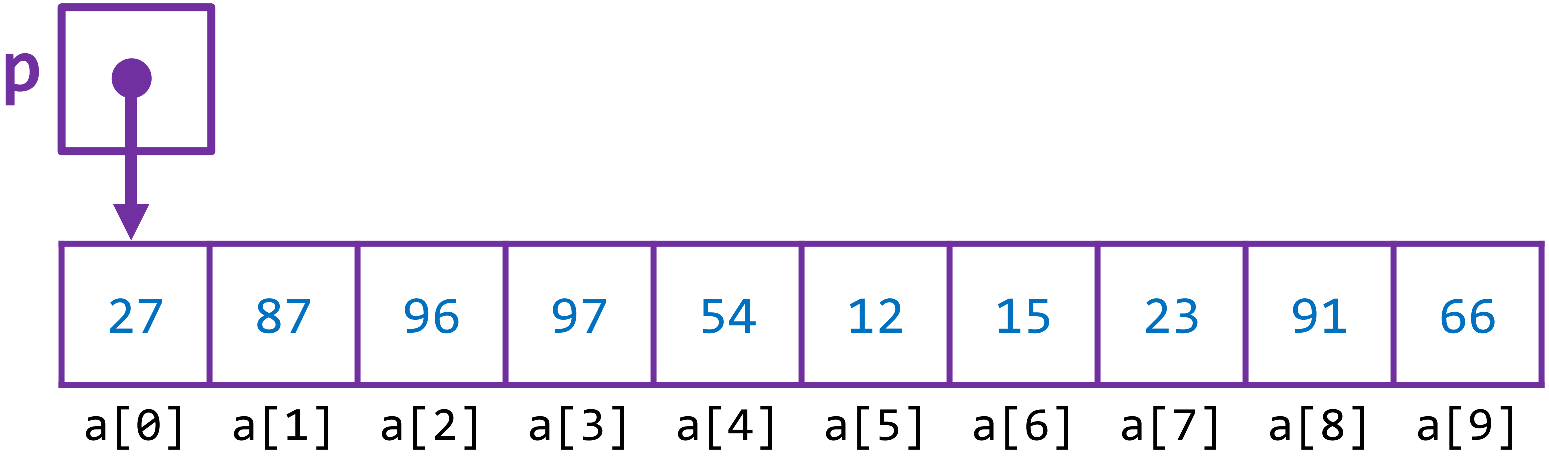
27	87	96	97	54	12	15	23	91	66
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

Using Pointer Arithmetic in Loops

- Pointer arithmetic can be very useful for processing and dealing with arrays when combined with loops (for loops in particular).
- **Example:**

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

Expression	Value
*p	27
sum	27



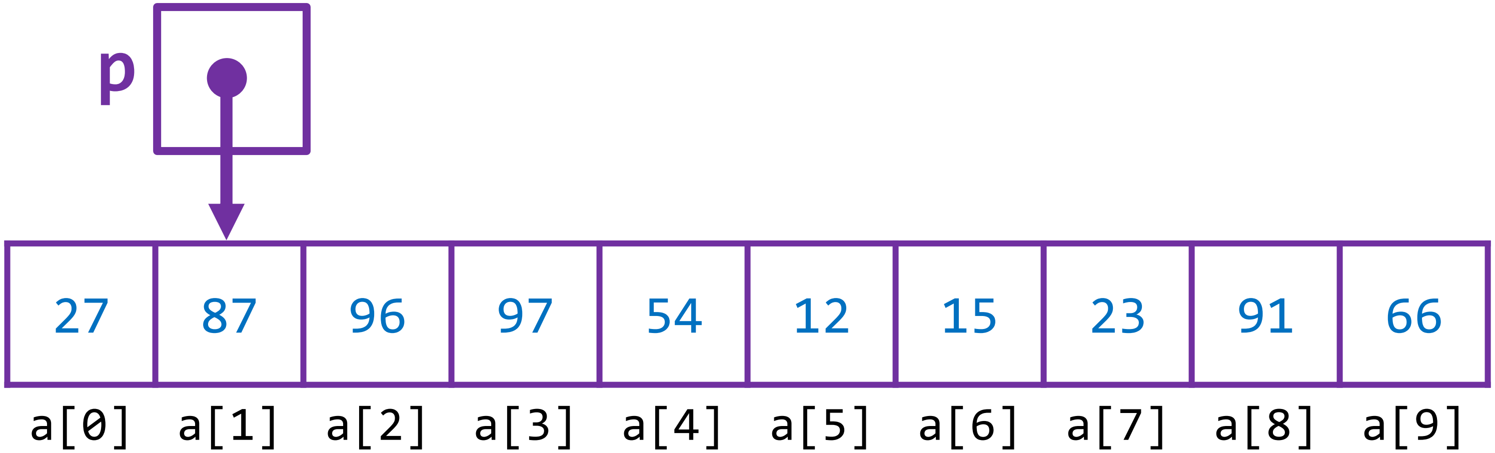
Using Pointer Arithmetic in Loops

- Pointer arithmetic can be very useful for processing and dealing with arrays when combined with loops (for loops in particular).

- **Example:**

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

Expression	Value
*p	87
sum	27



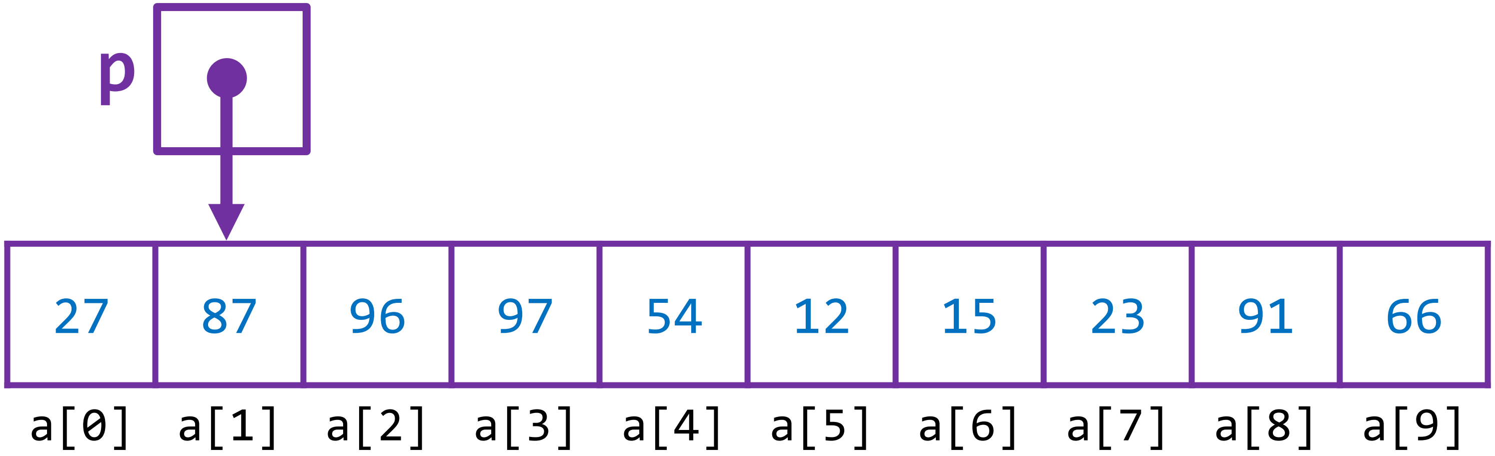
Using Pointer Arithmetic in Loops

- Pointer arithmetic can be very useful for processing and dealing with arrays when combined with loops (for loops in particular).

- **Example:**

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

Expression	Value
*p	87
sum	114



p

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
27	87	96	97	54	12	15	23	91	66

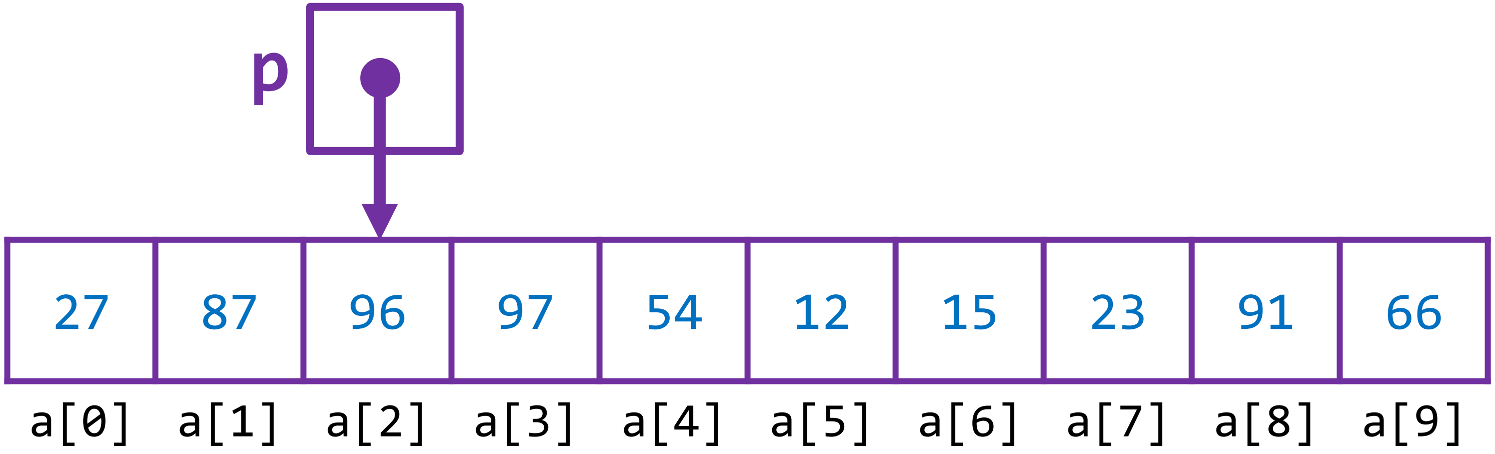
Using Pointer Arithmetic in Loops

- Pointer arithmetic can be very useful for processing and dealing with arrays when combined with loops (for loops in particular).

- **Example:**

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

Expression	Value
*p	96
sum	114



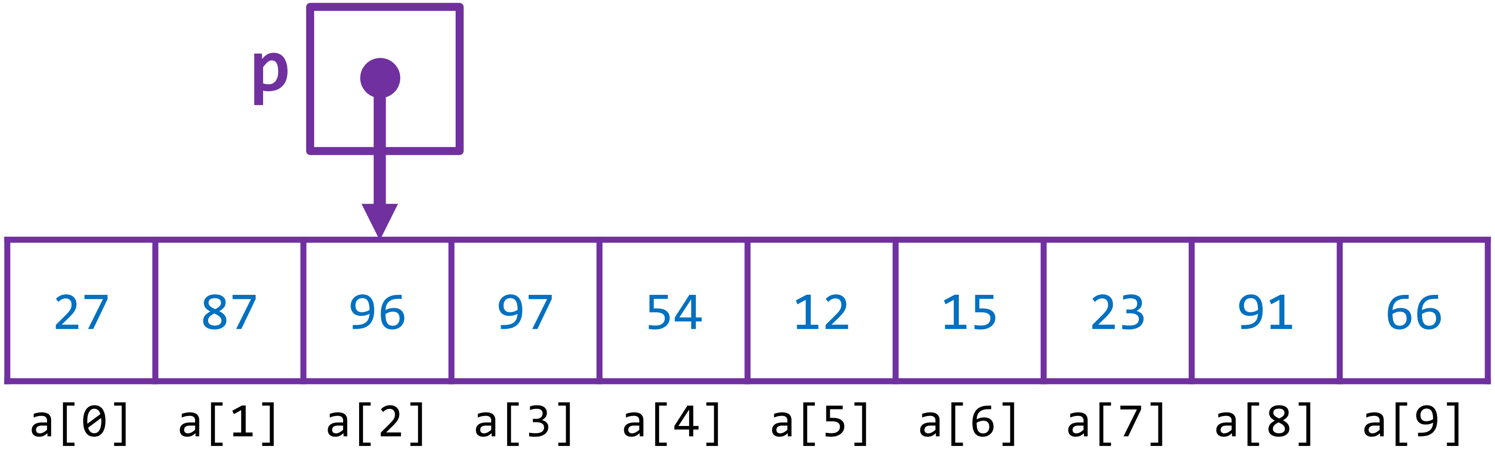
Using Pointer Arithmetic in Loops

- Pointer arithmetic can be very useful for processing and dealing with arrays when combined with loops (for loops in particular).

- **Example:**

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

Expression	Value
*p	96
sum	210



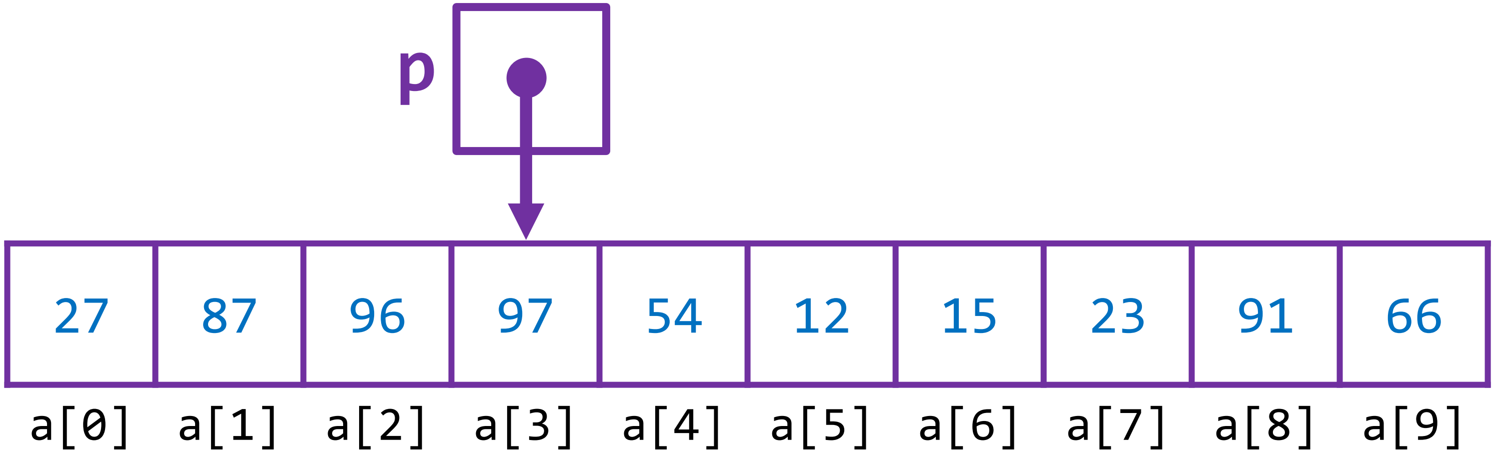
Using Pointer Arithmetic in Loops

- Pointer arithmetic can be very useful for processing and dealing with arrays when combined with loops (for loops in particular).

- **Example:**

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

Expression	Value
*p	97
sum	307



27	87	96	97	54	12	15	23	91	66
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

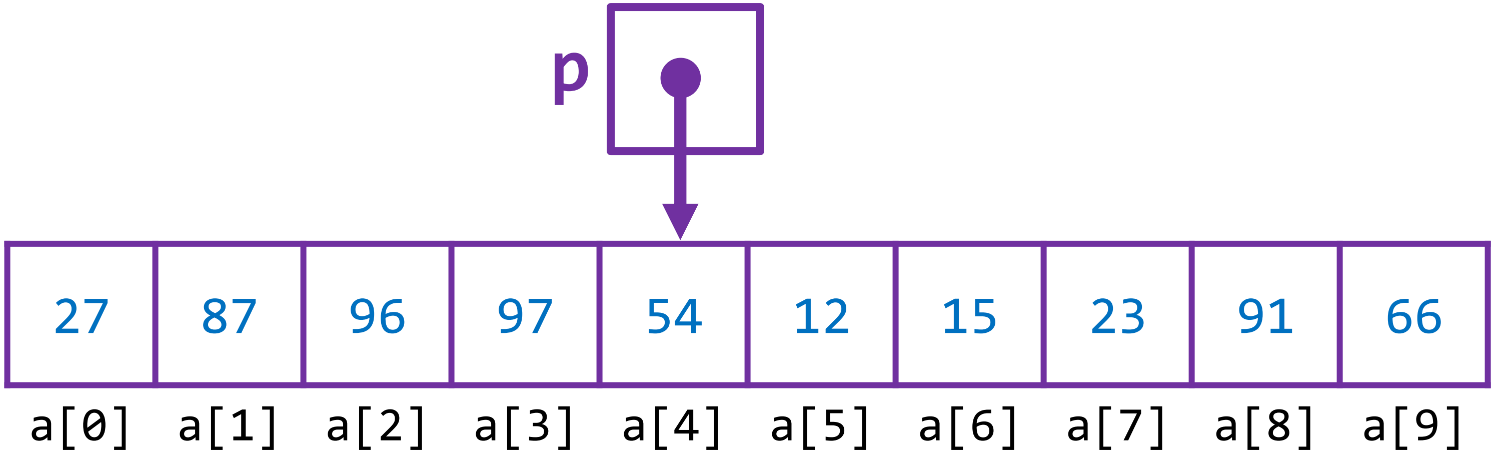
Using Pointer Arithmetic in Loops

- Pointer arithmetic can be very useful for processing and dealing with arrays when combined with loops (for loops in particular).

- **Example:**

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

…
```

Expression	Value
*p	54
sum	361



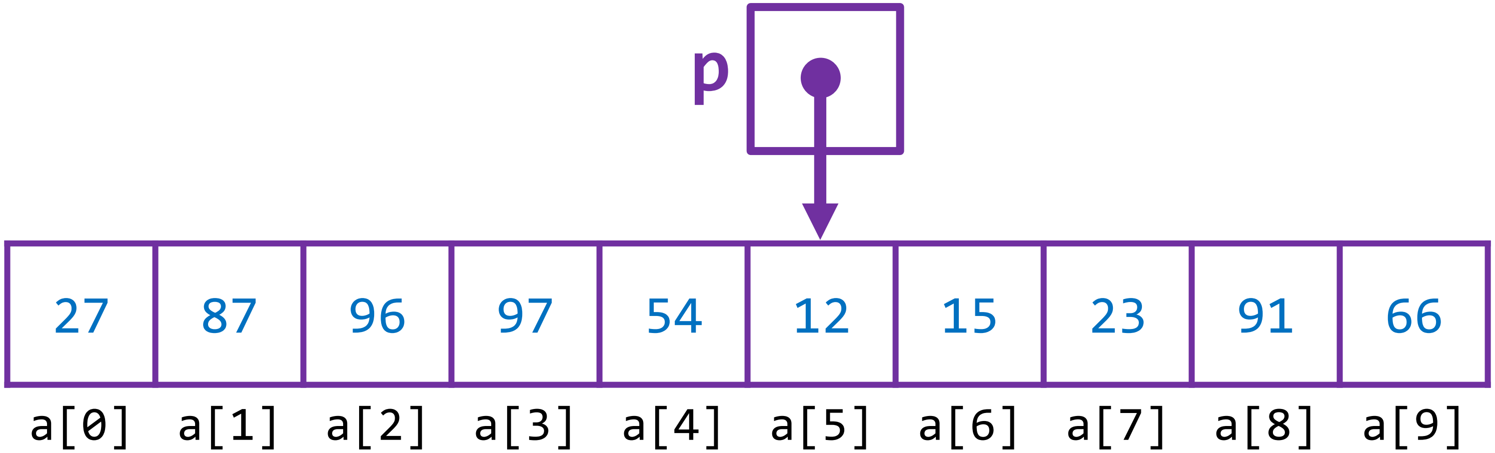
Using Pointer Arithmetic in Loops

- Pointer arithmetic can be very useful for processing and dealing with arrays when combined with loops (for loops in particular).

- **Example:**

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

…
```

Expression	Value
`*p`	12
`sum`	373



a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
27	87	96	97	54	12	15	23	91	66

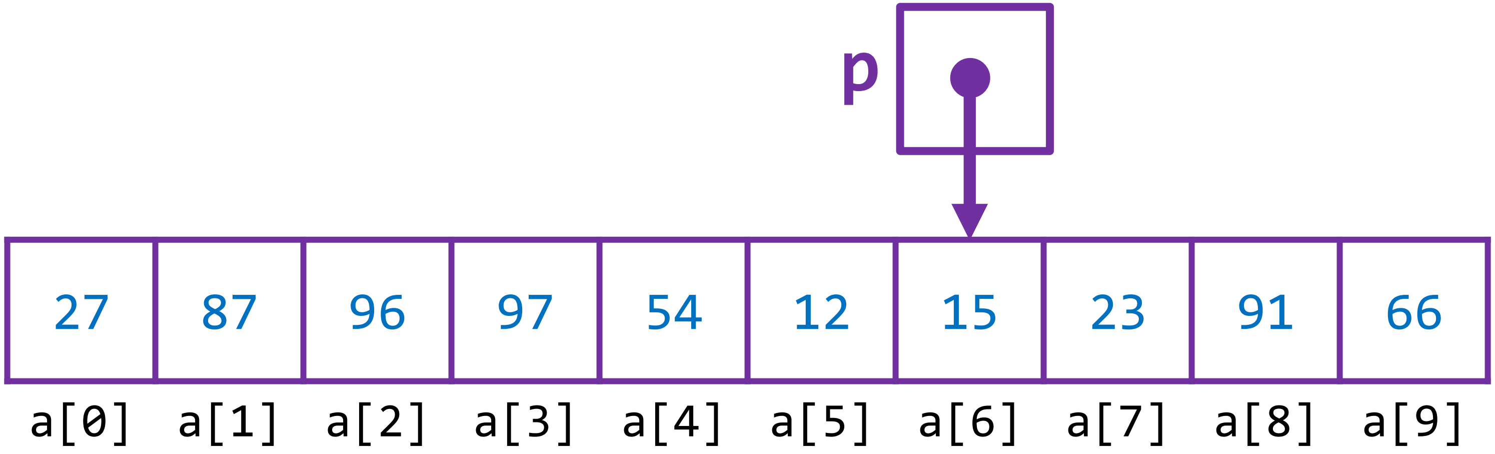
Using Pointer Arithmetic in Loops

- Pointer arithmetic can be very useful for processing and dealing with arrays when combined with loops (for loops in particular).

- **Example:**

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

Expression	Value
*p	15
sum	388



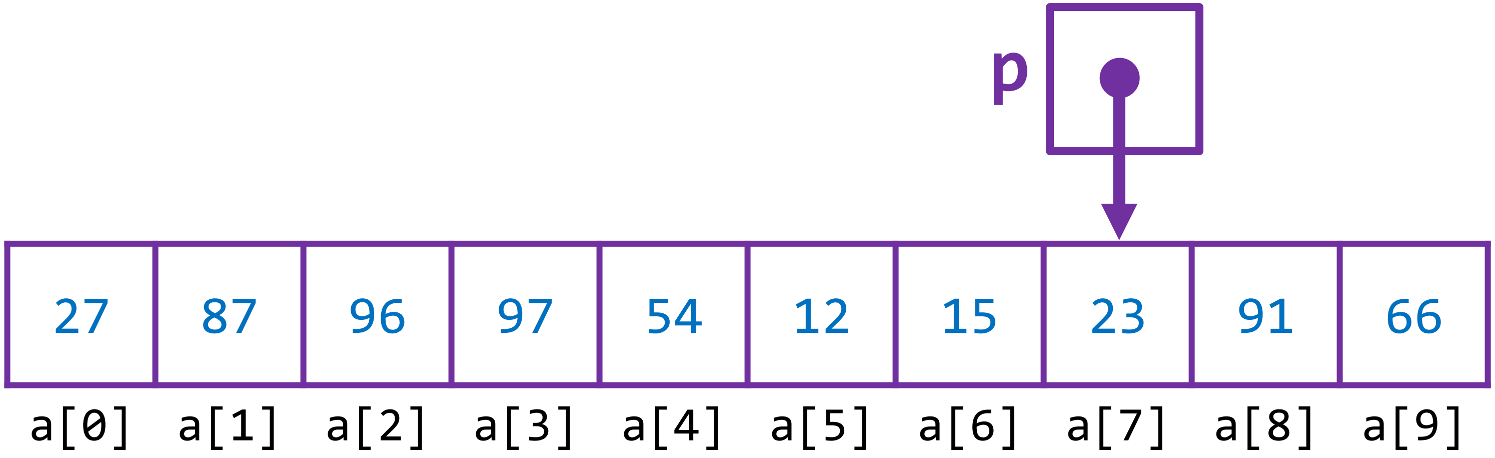
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
27	87	96	97	54	12	15	23	91	66

Using Pointer Arithmetic in Loops

- Pointer arithmetic can be very useful for processing and dealing with arrays when combined with loops (for loops in particular).
- **Example:**

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

Expression	Value
`*p`	23
`sum`	411



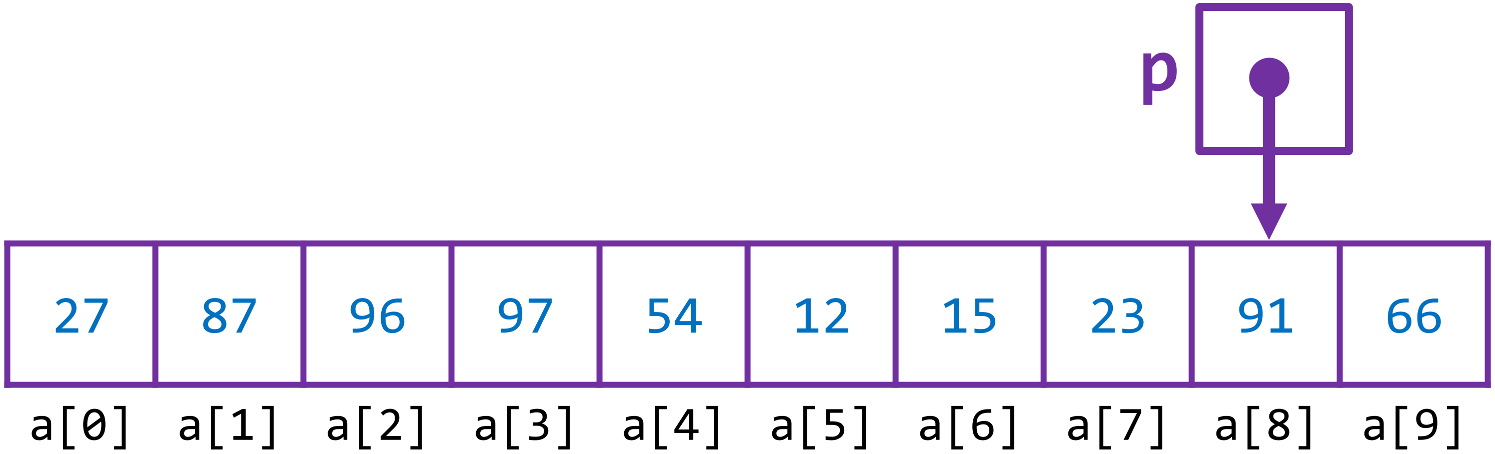
Using Pointer Arithmetic in Loops

- Pointer arithmetic can be very useful for processing and dealing with arrays when combined with loops (for loops in particular).

- **Example:**

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

…
```

Expression	Value
*p	91
sum	502



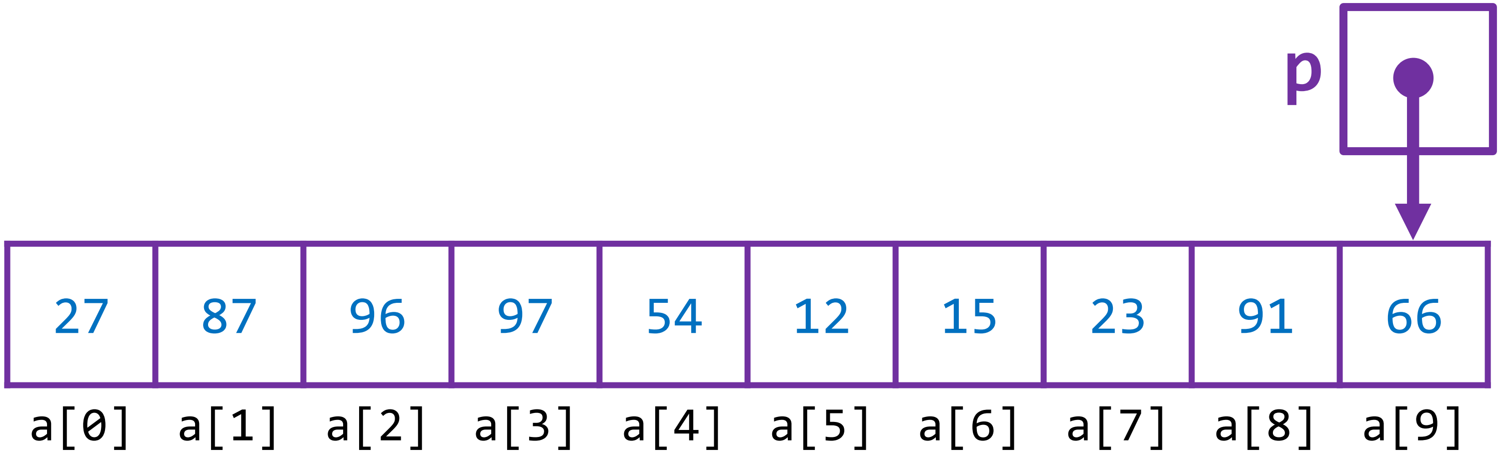
Using Pointer Arithmetic in Loops

- Pointer arithmetic can be very useful for processing and dealing with arrays when combined with loops (for loops in particular).

- **Example:**

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

Expression	Value
*p	66
sum	568



p

27	87	96	97	54	12	15	23	91	66
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

Using Pointer Arithmetic in Loops

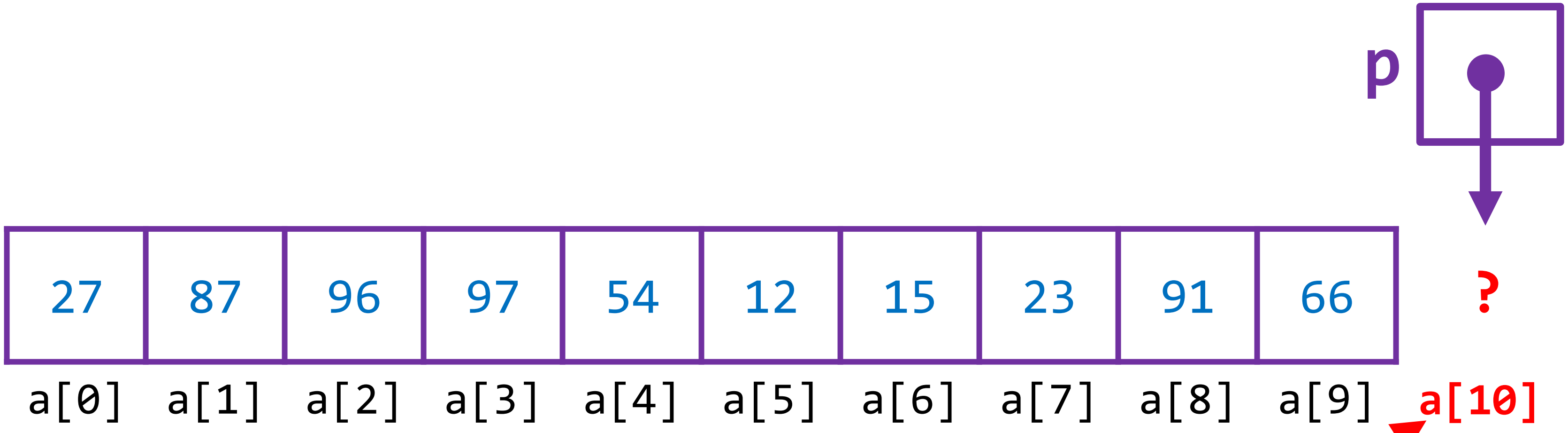
- Pointer arithmetic can be very useful for processing and dealing with arrays when combined with loops (for loops in particular).

- **Example:**

Loop is stopped before we dereference p again.

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

…
```

Expression	Value
*p	?
sum	568



a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]
27	87	96	97	54	12	15	23	91	66	?

Out of bounds.

Using Pointer Arithmetic in Loops

- Pointer arithmetic can be very useful for processing and dealing with arrays when combined with loops (for loops in particular).

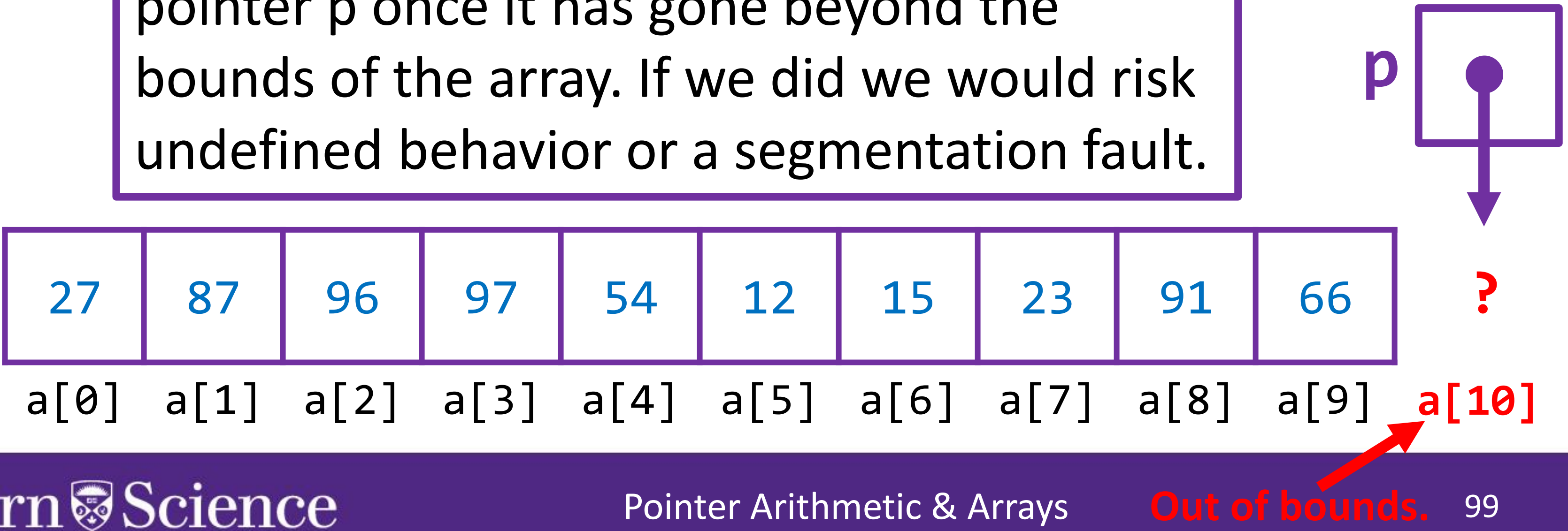
- **Example:**

Loop is stopped before we dereference p again.

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

Expression	Value
*p	?
sum	568

This is valid as we never dereference the pointer p once it has gone beyond the bounds of the array. If we did we would risk undefined behavior or a segmentation fault.



Out of bounds.

A Note on the -- and ++ Operators

- It is valid to use the -- and ++ postfix and prefix operators on pointers to arrays.
- This would increment or decrement the pointer as expected (moving up or down one element in the array).
- When combined with the indirection operator (*), it is important to pay attention to order of operations.
- **Examples:**

```
int x = *p++;
```

Returns the value p is pointing at and then increments the pointer p by 1 (moves it to the next array element).

Equivalent to `*(p++);`

```
int x = (*p)++;
```

Sets x to the value p is pointing at and then increments the value p is pointing at by 1.

A Note on the -- and ++ Operators

- It is valid to use the -- and ++ postfix and prefix operators on pointers to arrays.
- This would increment or decrement the pointer as expected (moving up or down one element in the array).

The ++ and -- operators have precedence over the indirection operator.

- **Examples:**

```
int x = *p++;
```

Returns the value p is pointing at and then increments the pointer p by 1 (moves it to the next array element).

Equivalent to `*(p++);`

```
int x = (*p)++;
```

Sets x to the value p is pointing at and then increments the value p is pointing at by 1.

A Note on the -- and ++ Operators

- We could use this to restructure our loops in ex8.c
- **Example:**

/cs2211/week10/ex8b.c

```
#include <stdio.h>
#define N 10

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

        printf("Input %d numbers:\n", N);
        p = &a[0];
        while(p < &a[N])
                scanf("%d", p++);

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

        printf("Sum is %d\n", sum);
        return 0;
}
```

Not necessarily a better solution but a different style of solution.

Array Names as Pointers

- Arrays are not equivalent to pointers, they contain extra information about the size of the array (this is important when using `sizeof`).
- However, we can use the name of an array as a pointer to the first element of that array.
- This can simplify some of the code we have seen so far.
- **Example:**

```
int a[5] = {17, 63, 9, 78, 12};
*a = 5;
*(a+1) = 32;
*(a+2) = *(a+4);
```

Array Names as Pointers

a[0]	a[1]	a[2]	a[3]	a[4]
17	63	9	78	12

- **Example:**

```
int a[5] = {17, 63, 9, 78, 12};
*a = 5;
*(a+1) = 32;
*(a+2) = *(a+4);
```

Array Names as Pointers

a[0]	a[1]	a[2]	a[3]	a[4]
5	63	9	78	12

Sets the first element of a equal to 5.

- **Example:**

```
int a[5] = {17, 63, 9, 78, 12};
*a = 5;
*(a+1) = 32;
*(a+2) = *(a+4);
```

Array Names as Pointers

a[0]	a[1]	a[2]	a[3]	a[4]
5	32	9	78	12

Sets the second element of a equal to 32.

- **Example:**

```
int a[5] = {17, 63, 9, 78, 12};
*a = 5;
*(a+1) = 32;
*(a+2) = *(a+4);
```

Array Names as Pointers

a[0]	a[1]	a[2]	a[3]	a[4]
5	32	12	78	12

Sets the third element of a equal to fifth element of a.

- **Example:**

```
int a[5] = {17, 63, 9, 78, 12};
*a = 5;
*(a+1) = 32;
*(a+2) = *(a+4);
```

Array Names as Pointers

In general:

- `a` is equivalent to `&a[0]`
- `*a` is equivalent to `a[0]`
- `a+i` is equivalent to `&a[i]`
- `*(a+i)` is equivalent to `a[i]`

- **Example:**

```
int a[5] = {17, 63, 9, 78, 12};
*a = 5;
*(a+1) = 32;
*(a+2) = *(a+4);
```

Array Names as Pointers

- We can use this fact to once again update and simplify ex8.c

/cs2211/week10/ex8c.c

```c
#include <stdio.h>
#define N 10

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

        printf("Input %d numbers:\n", N);
        for(p = a; p < a + N; p++)
                scanf("%d", p);

        for(p = a; p < a + N; p++)
                sum += *p;

        printf("Sum is %d\n", sum);
        return 0;
}
```

Array Names as Pointers

- We can use this fact to once again update and simplify ex8.c

/cs2211/week10/ex8c.c

```
#include <stdio.h>
#define N 10

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

        printf("Input %d numbers:\n", N);
        for(p = a; p < a + N; p++)
                scanf("%d", p);

        for(p = a; p < a + N; p++)
                sum += *p;

        printf("Sum is %d\n", sum);
        return 0;
}
```

Sets pointer p to the address of the first element of array a.

Short from for p = &a[0]

Array Names as Pointers

- We can use this fact to once again update and simplify ex8.c

/cs2211/week10/ex8c.c

```
#include <stdio.h>
#define N 10

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

        printf("Input %d numbers:\n", N);
        for(p = a; p < a + N; p++)
                scanf("%d", p);

        for(p = a; p < a + N; p++)
                sum += *p;

        printf("Sum is %d\n", sum);
        return 0;
}
```

Checks that pointer p is less than the first address outside of array a.

Short from for p < &a[N]

Array Names as Pointers

- Well we can use the name of an array as a pointer, we can not assign it a new value (change where it points).
- **Bad Example 1:**

```
int a[5] = {17, 63, 9, 78, 12};
a = a + 1;
```

- **Bad Example 2:**

```
int a[5] = {17, 63, 9, 78, 12};
while(*a != 0)
   a++;
```

Will result in an error when compiling.

Array as Arguments Revisited

- The reason we can use `sizeof` to find the size of a local array and not an array passed to a function is that an array passed to a function is treated as a pointer and not an array.
- Using `sizeof` on a pointer returns the size of the pointer variable (the variable that holds the memory address) and not the variable the pointer points to.
- **Example 1:**

```
int i;
int *p = &I;
printf("%d %d\n", sizeof(p), sizeof(*p));
```

Output: 8 4

Size of the pointer p

Size of the integer i

Array as Arguments Revisited

- The reason we can use `sizeof` to find the size of a local array and not an array passed to a function is that an array passed to a function is treated as a pointer and not an array.
- Using `sizeof` on a pointer returns the size of the pointer variable (the variable that holds the memory address) and not the variable the pointer points to.
- **Example 2:**

```
void foo(int a[], int n) {
   printf("%d %d\n", sizeof(a), sizeof(*a));
}
```

Output: 8 4

Size of the pointer a

Size of the first element of a

Neither value is the size of the array.

Array as Arguments Revisited

- This means we are allowed to do something like this:
- **Example:**

```
void bar(int a[], int n) {
     int *p = a;
     while(a - p < n)
             *a++ = 0;
}
```

- This function sets all elements in the array to 0.
- We are allowed to do a++ in this case as a is treated as a pointer and not an array.
- This also means that the size of an array does not have a speed or efficiency impact when sending large arrays to a function (only sending a memory address not copying the array).

Array as Arguments Revisited

- Using a integer pointer parameter is equivalent to having an array parameter.
- **Examples:**

```
void foo(int a[], int n) {
    …
}
```

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

- Functions `foo` and `bar` are equivalent.
- Both can take and reference an array in the same manner.
- Declaring a to be a pointer is the same as declaring it to be an array in the case of parameters.

Array as Arguments Revisited

- Note that same is not true of variables.
- **Examples:**

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

Declares 10 integers sequentially in memory.

```
int *a;
```

Only declares one integer pointer (only enough space to store a memory address).

Array as Arguments Revisited

- Having arrays treated as pointers in function calls also allows us to pass only ***part*** of an array.
- **Example:**

```
int *p, a[7] = {77, 89, 90, 38, 74, 10, 12};
p = max_a(a+3, 4);
printf("max is %d\n", *p);
```

Output: `max is 74`

- Here `max_a` is the `max_a` function from ex7.c (returns a pointer to the largest element in the given array).
- Available as ex9.c: */cs2211/week10/ex9.c*

Array as Arguments Revisited

- Having arrays treated as pointers in function calls also allows us to pass only ***part*** of an array.
- **Example:**

```
int *p, a[7] = {77, 89, 90, 38, 74, 10, 12};
p = max_a(a+3, 4);
printf("max is %d\n", *p);
```

Output: `max is 74`

Sends a pointer to the 4th element of array a to function `max_a`.

This causes `max_a` to get a subset of the array, only the values 38, 74, 10 and 12.

Subscript Notation with Pointers

- C allows us to subscript a pointer as though it were an array name.
- **Example:**

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

- `p[i]` is equivalent to `*(p+i)`

Multidimensional Arrays & Pointers

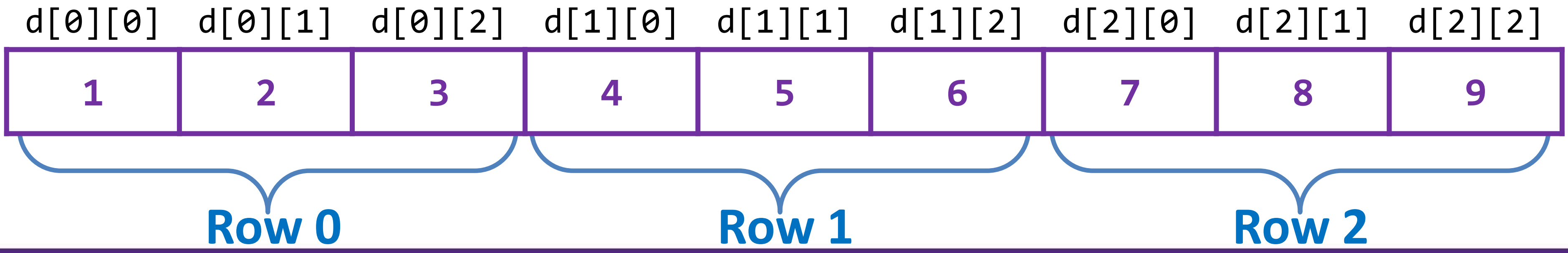
- We can create pointers to multidimensional arrays just like we can one dimensional arrays.

- Recall from last lecture that multidimensional arrays are stored in memory in **row-major order**. That is with row 0 first, then row 2, and so forth in a linear manner.

- **Example:**

```
int d[3][3] = {{1,2,3},
               {4,5,6},
               {7,8,9}};
```

In memory:

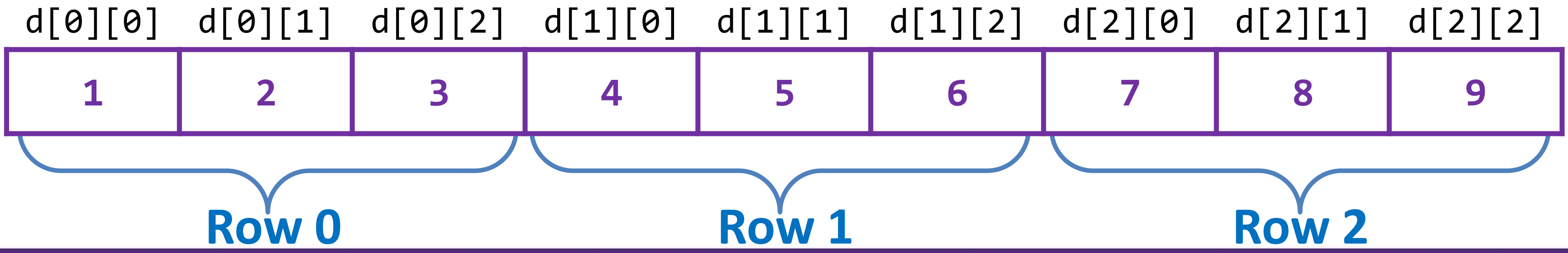


d[0][0]	d[0][1]	d[0][2]	d[1][0]	d[1][1]	d[1][2]	d[2][0]	d[2][1]	d[2][2]
1	2	3	4	5	6	7	8	9

Multidimensional Arrays & Pointers

- If we create a pointer to a multidimensional array and increment it past the end of a row, it will wrap around to the next row.
- **Example:**

```
int *p, d[3][3] = {{1,2,3},
                   {4,5,6},
                   {7,8,9}};

for(p = &d[0][0]; p <= d[2][2]; p++)
   printf("%d ", *p);
```

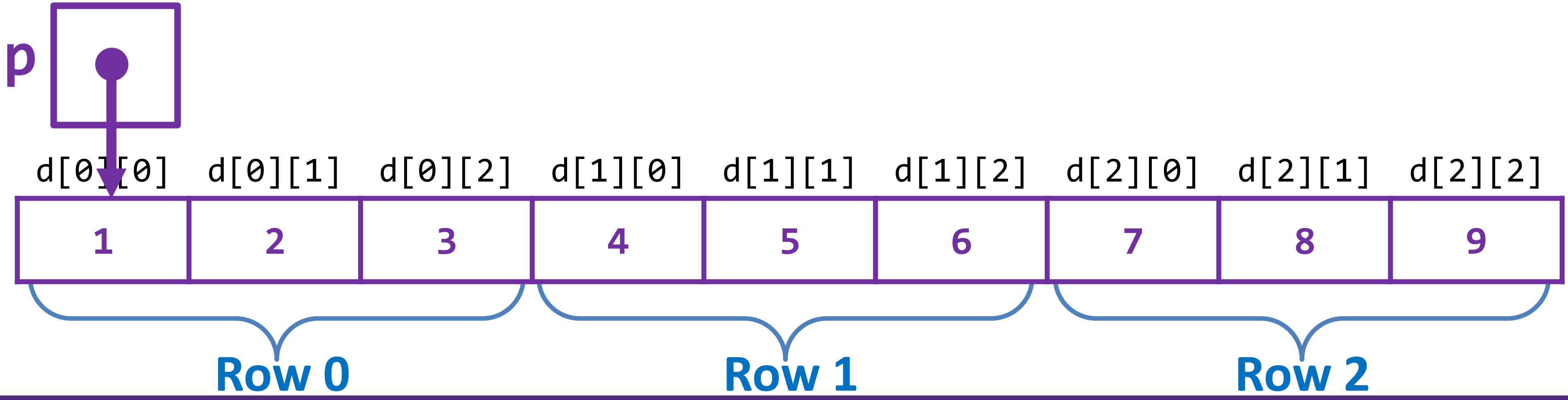


d[0][0]	d[0][1]	d[0][2]	d[1][0]	d[1][1]	d[1][2]	d[2][0]	d[2][1]	d[2][2]
1	2	3	4	5	6	7	8	9

Multidimensional Arrays & Pointers

- If we create a pointer to a multidimensional array and increment it past the end of a row, it will wrap around to the next row.
- **Example:** **Output:** 1

```
int *p, d[3][3] = {{1,2,3},
                   {4,5,6},
                   {7,8,9}};

for(p = &d[0][0]; p <= d[2][2]; p++)
    printf("%d ", *p);
```

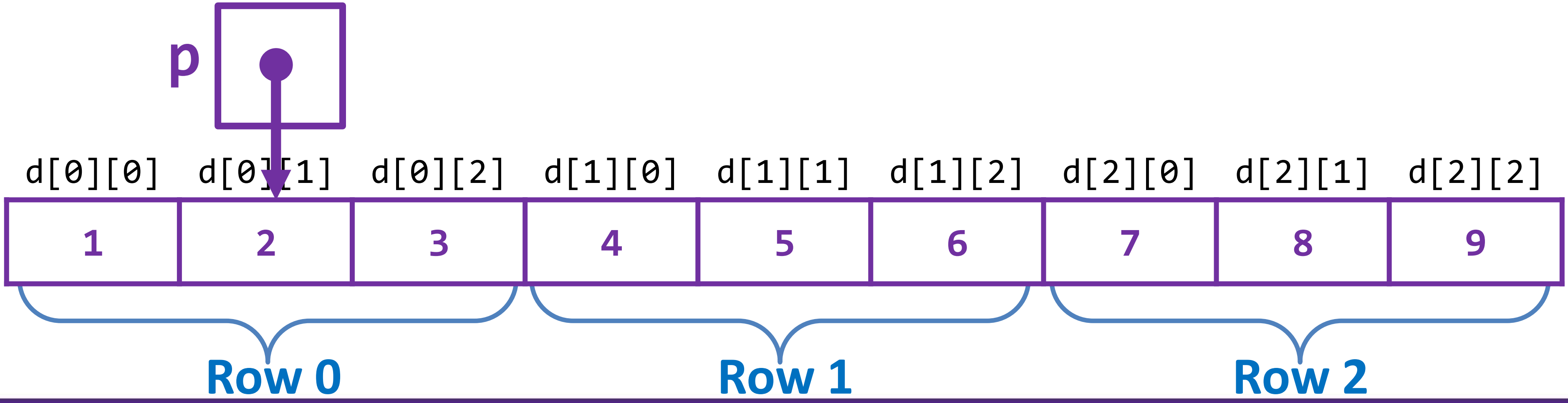


Multidimensional Arrays & Pointers

- If we create a pointer to a multidimensional array and increment it past the end of a row, it will wrap around to the next row.

- **Example:** **Output:** 1 2

```
int *p, d[3][3] = {{1,2,3},
                   {4,5,6},
                   {7,8,9}};

for(p = &d[0][0]; p <= d[2][2]; p++)
   printf("%d ", *p);
```



p

d[0][0]	d[0][1]	d[0][2]	d[1][0]	d[1][1]	d[1][2]	d[2][0]	d[2][1]	d[2][2]
1	2	3	4	5	6	7	8	9

Row 0

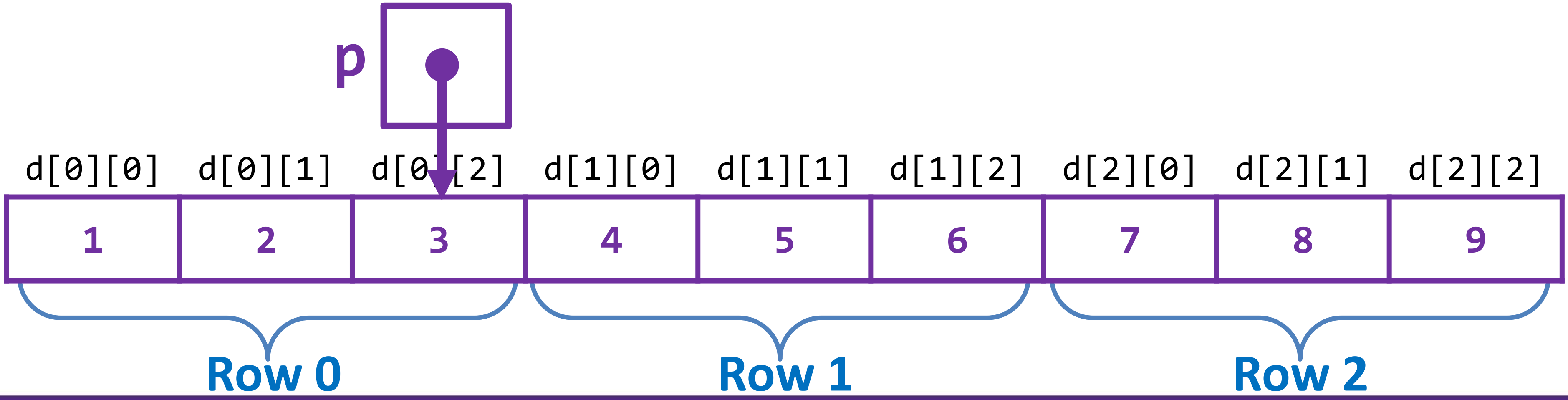
Row 1

Row 2

Multidimensional Arrays & Pointers

- If we create a pointer to a multidimensional array and increment it past the end of a row, it will wrap around to the next row.
- **Example:** **Output:** `1 2 3`

```
int *p, d[3][3] = {{1,2,3},
                   {4,5,6},
                   {7,8,9}};

for(p = &d[0][0]; p <= d[2][2]; p++)
    printf("%d ", *p);
```

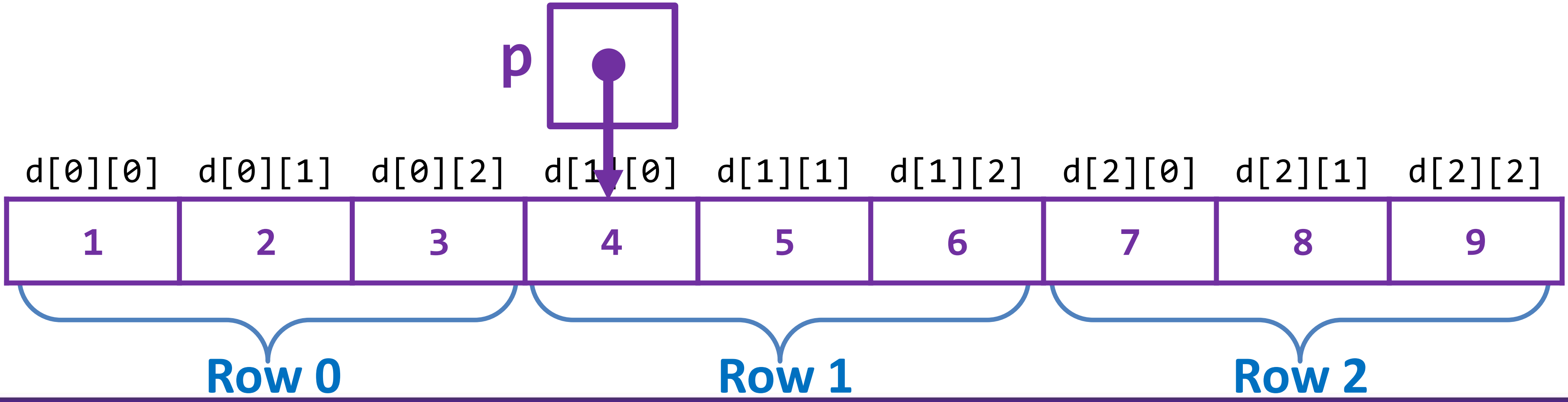


d[0][0]	d[0][1]	d[0][2]	d[1][0]	d[1][1]	d[1][2]	d[2][0]	d[2][1]	d[2][2]
1	2	3	4	5	6	7	8	9

Multidimensional Arrays & Pointers

- If we create a pointer to a multidimensional array and increment it past the end of a row, it will wrap around to the next row.
- **Example:**   **Output:** `1 2 3 4`

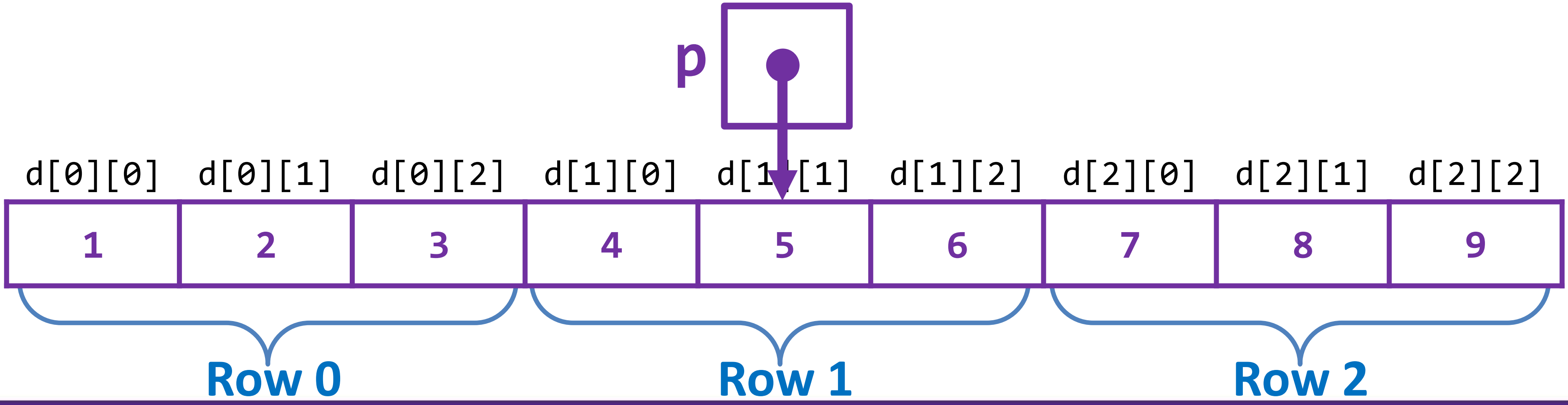
```
int *p, d[3][3] = {{1,2,3},
                   {4,5,6},
                   {7,8,9}};

for(p = &d[0][0]; p <= d[2][2]; p++)
   printf("%d ", *p);
```



Multidimensional Arrays & Pointers

- If we create a pointer to a multidimensional array and increment it past the end of a row, it will wrap around to the next row.
- **Example:** **Output:** 1 2 3 4 5

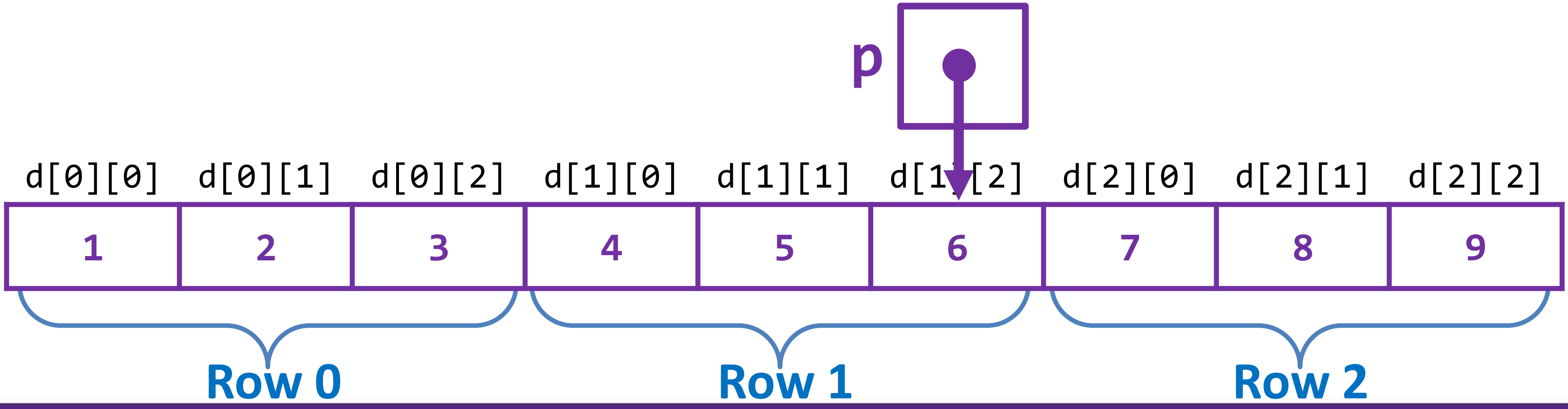
```
int *p, d[3][3] = {{1,2,3},
                   {4,5,6},
                   {7,8,9}};

for(p = &d[0][0]; p <= d[2][2]; p++)
   printf("%d ", *p);
```



Multidimensional Arrays & Pointers

- If we create a pointer to a multidimensional array and increment it past the end of a row, it will wrap around to the next row.
- **Example:** **Output:** 1 2 3 4 5 6

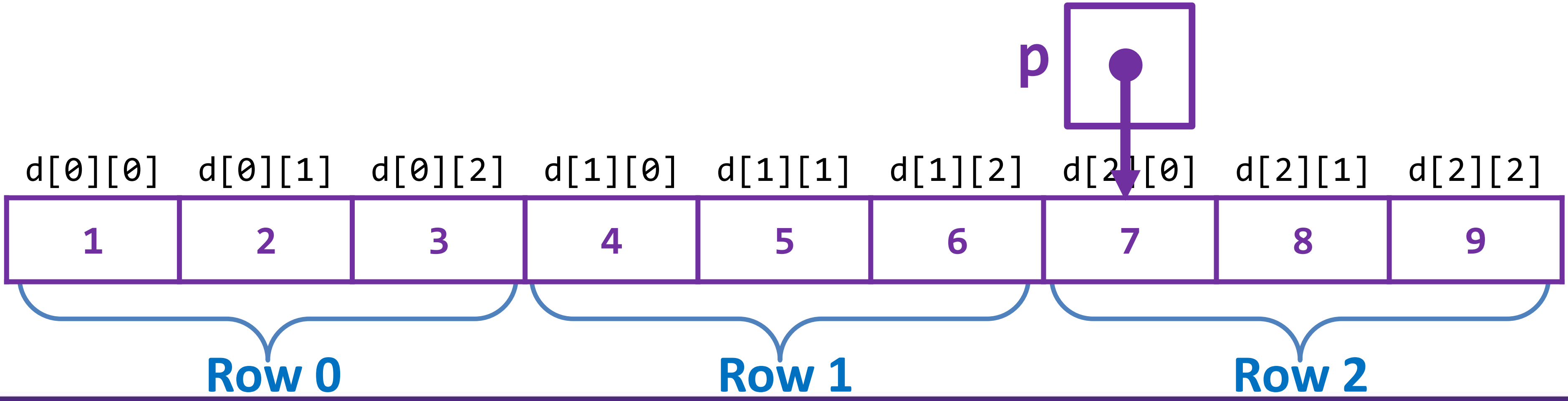
```
int *p, d[3][3] = {{1,2,3},
                   {4,5,6},
                   {7,8,9}};

for(p = &d[0][0]; p <= d[2][2]; p++)
    printf("%d ", *p);
```



Multidimensional Arrays & Pointers

- If we create a pointer to a multidimensional array and increment it past the end of a row, it will wrap around to the next row.
- **Example:** **Output:** 1 2 3 4 5 6 7

```
int *p, d[3][3] = {{1,2,3},
                   {4,5,6},
                   {7,8,9}};

for(p = &d[0][0]; p <= d[2][2]; p++)
    printf("%d ", *p);
```

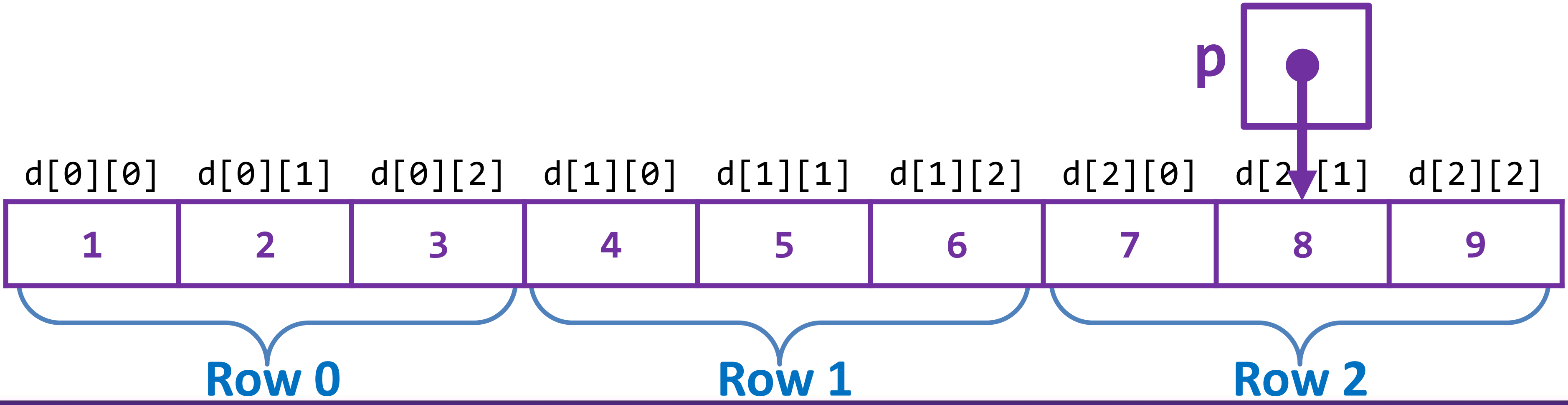


Multidimensional Arrays & Pointers

- If we create a pointer to a multidimensional array and increment it past the end of a row, it will wrap around to the next row.
- **Example:**

Output: 1 2 3 4 5 6 7 8

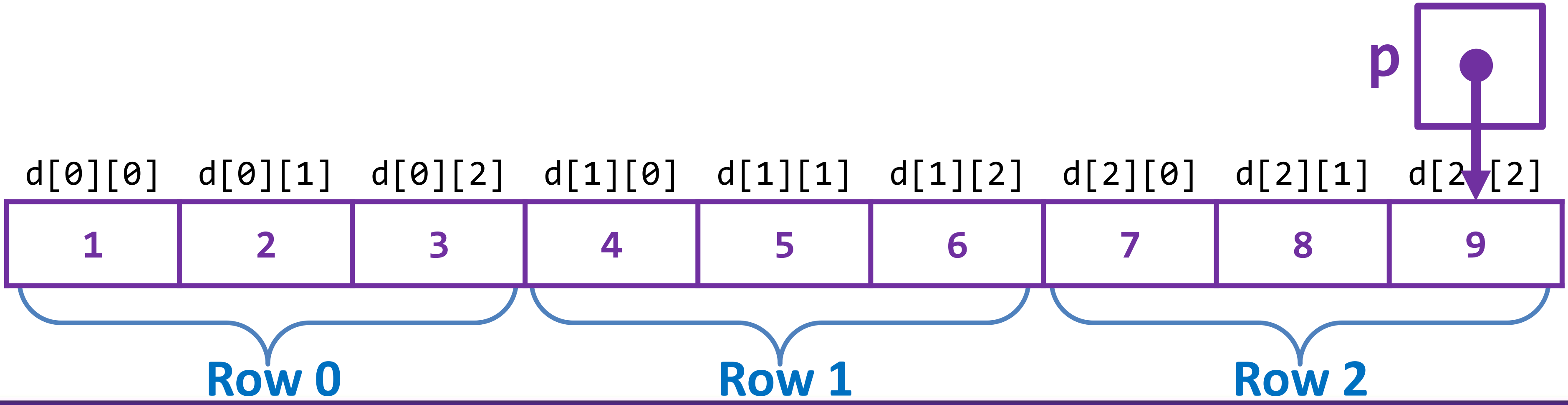
```
int *p, d[3][3] = {{1,2,3},
                   {4,5,6},
                   {7,8,9}};

for(p = &d[0][0]; p <= d[2][2]; p++)
    printf("%d ", *p);
```



Multidimensional Arrays & Pointers

- If we create a pointer to a multidimensional array and increment it past the end of a row, it will wrap around to the next row.
- **Example:** **Output:** 1 2 3 4 5 6 7 8 9

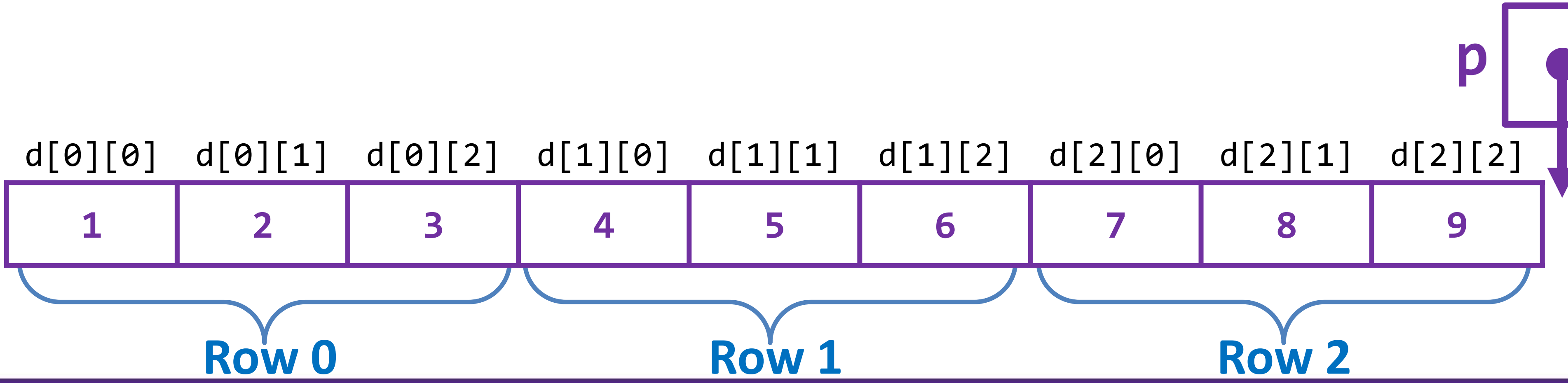
```
int *p, d[3][3] = {{1,2,3},
                   {4,5,6},
                   {7,8,9}};

for(p = &d[0][0]; p <= d[2][2]; p++)
    printf("%d ", *p);
```



Multidimensional Arrays & Pointers

- If we create a pointer to a multidimensional array and increment it past the end of a row, it will wrap around to the next row.
- **Example:** **Output:** 1 2 3 4 5 6 7 8 9

```
int *p, d[3][3] = {{1,2,3},
                   {4,5,6},
                   {7,8,9}};

for(p = &d[0][0]; p <= d[2][2]; p++)
    printf("%d ", *p);
```



Multidimensional Arrays & Pointers

- We can also use pointers to process just one row of a multidimensional array.
- To set a pointer to row `i` we could initialize the pointer to:

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

  or just:

  ```
  p = a[i];
  ```

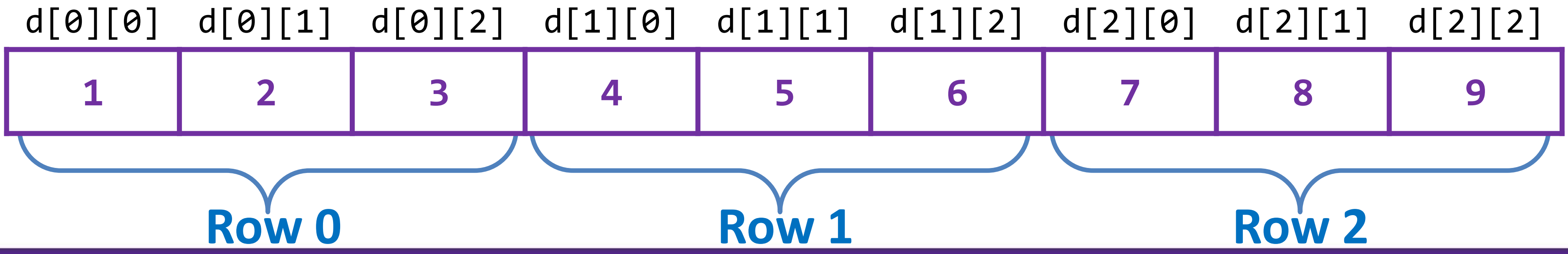
  Assuming a is a 2D array.
- For any 2D array `a[i]` is a pointer to first element in row `i`.
- Recall that `a[i]` is equivalent to `*(a+i)`, thus, `&a[i][0]` is equivalent to `&(*(a[i] + 0))` which equals `&*a[i]`, which is the same as `a[i]`.

Multidimensional Arrays & Pointers

Example:

Output:

```
int *p, d[3][3] = {{1,2,3},
                   {4,5,6},
                   {7,8,9}};

for(p = d[1]; p < d[2]; p++)
    printf("%d ", *p);
```



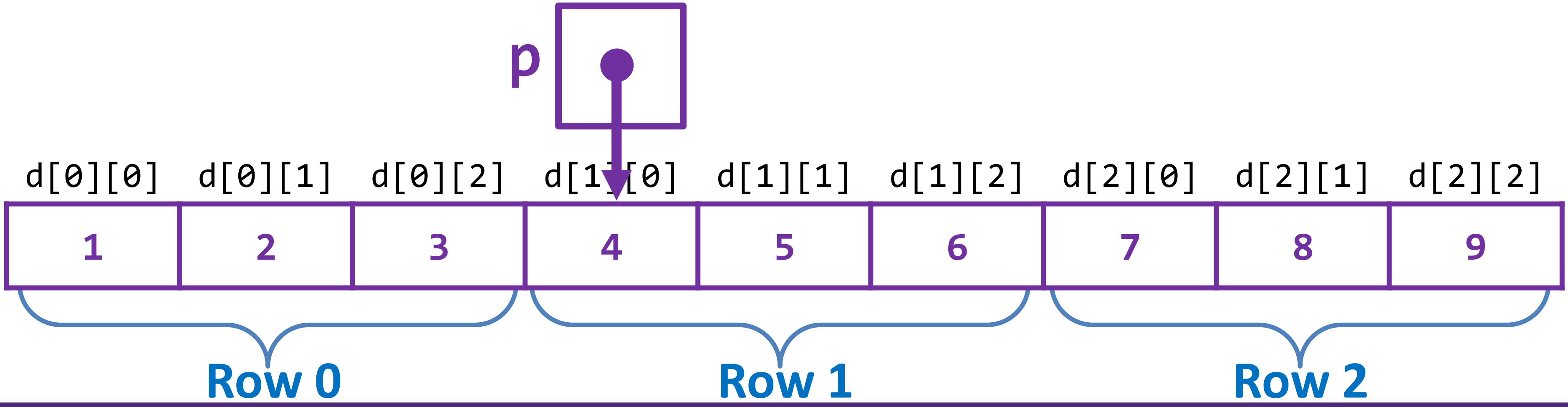
d[0][0]	d[0][1]	d[0][2]	d[1][0]	d[1][1]	d[1][2]	d[2][0]	d[2][1]	d[2][2]
1	2	3	4	5	6	7	8	9

Multidimensional Arrays & Pointers

Example:

Output: 4

```
int *p, d[3][3] = {{1,2,3},
                   {4,5,6},
                   {7,8,9}};

for(p = d[1]; p < d[2]; p++)
   printf("%d ", *p);
```



p

d[0][0]	d[0][1]	d[0][2]	d[1][0]	d[1][1]	d[1][2]	d[2][0]	d[2][1]	d[2][2]
1	2	3	4	5	6	7	8	9

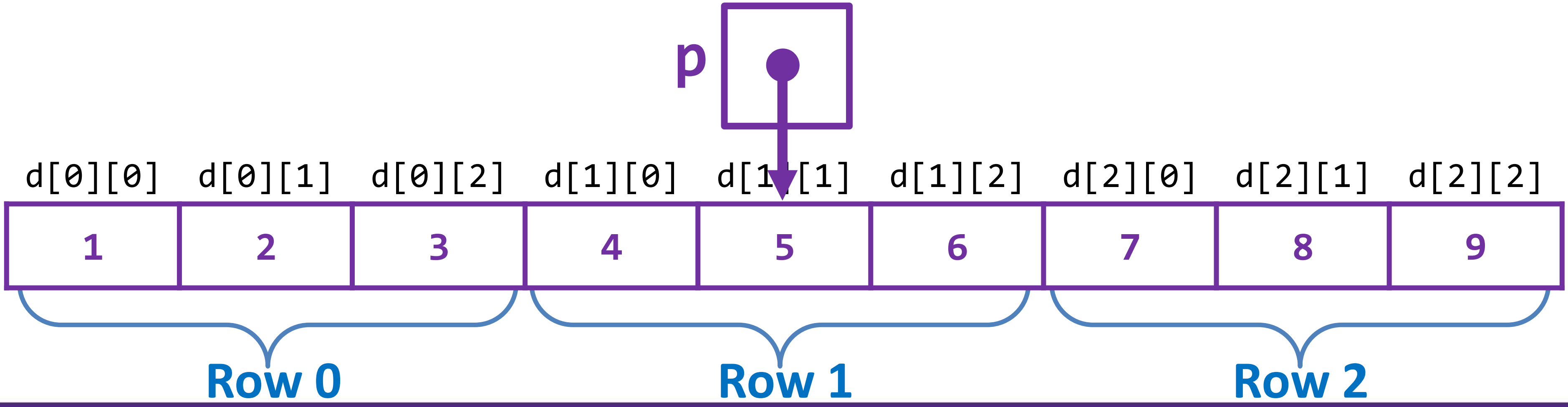
Row 0 Row 1 Row 2

Multidimensional Arrays & Pointers

Example:

Output: 4 5

```
int *p, d[3][3] = {{1,2,3},
                   {4,5,6},
                   {7,8,9}};

for(p = d[1]; p < d[2]; p++)
    printf("%d ", *p);
```



p

d[0][0]	d[0][1]	d[0][2]	d[1][0]	d[1][1]	d[1][2]	d[2][0]	d[2][1]	d[2][2]
1	2	3	4	5	6	7	8	9

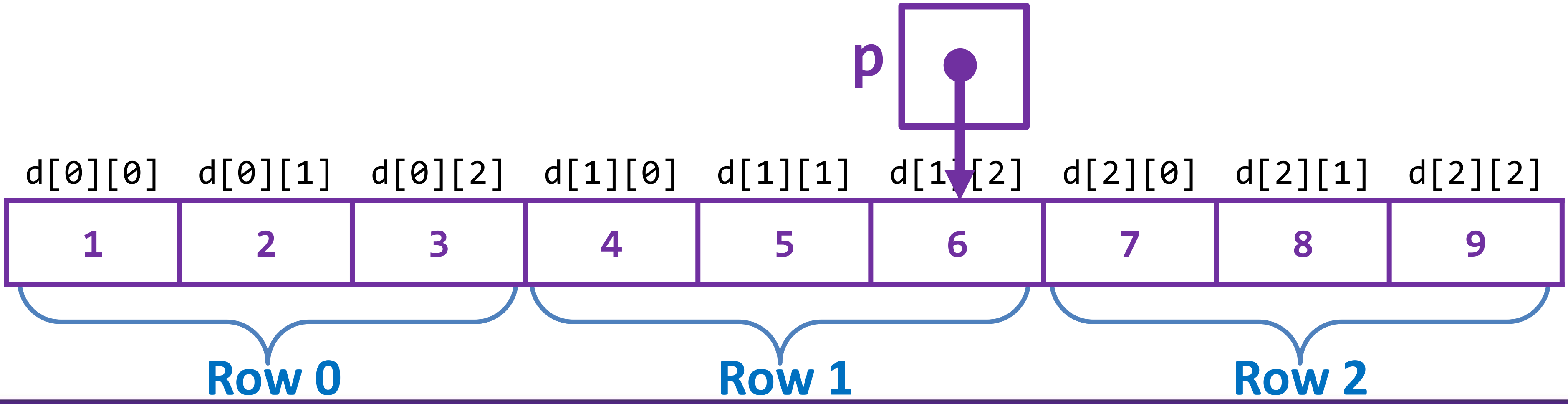
Row 0 Row 1 Row 2

Multidimensional Arrays & Pointers

Example:

Output: 4 5 6

```
int *p, d[3][3] = {{1,2,3},
                   {4,5,6},
                   {7,8,9}};

for(p = d[1]; p < d[2]; p++)
    printf("%d ", *p);
```

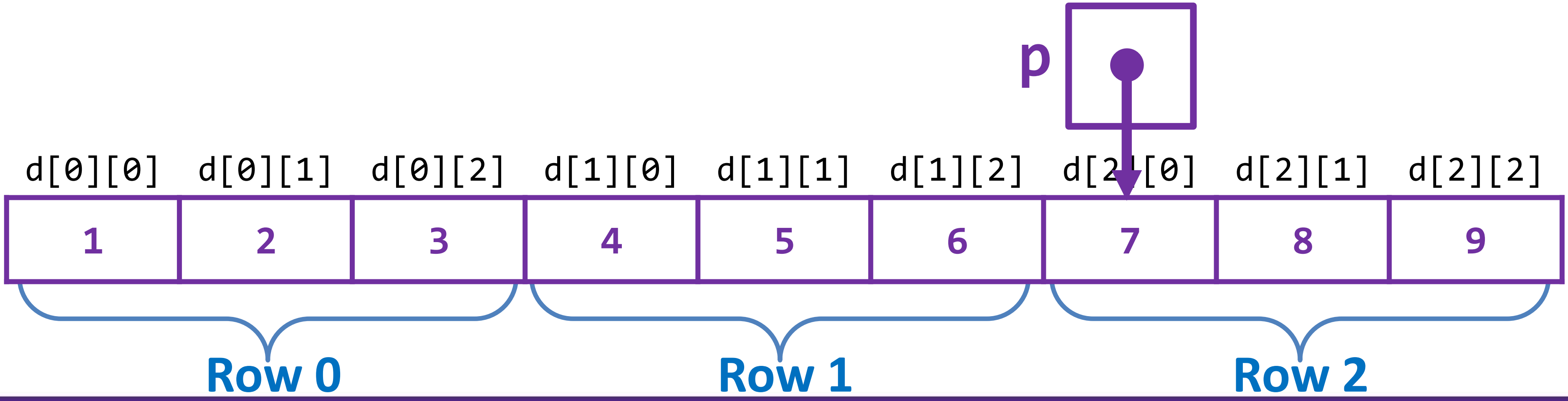


Multidimensional Arrays & Pointers

Example:

Output: 4 5 6

```
int *p, d[3][3] = {{1,2,3},
                   {4,5,6},
                   {7,8,9}};

for(p = d[1]; p < d[2]; p++)
    printf("%d ", *p);
```



Multidimensional Arrays & Pointers

- This means a function that is created to work with one dimensional arrays will work with rows from two dimensional arrays.

Available as ex10.c: */cs2211/week10/ex10.c*

- **Example:**

```
int *p, d[3][3] = {{1,2,3},
                   {4,5,6},
                   {7,8,9}};


p = max_a(d[1], 3);
printf("max is %d\n", *p);
```

Output: `max is 6`

max_a is the same max_a function as in ex7.c and ex9.c

Multidimensional Arrays Name as a Pointer

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

```
int a[5][10];
```

- 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 a one-dimensional arrays.
- When used as a pointer, a has type int (*)[10] (pointer to an integer array of length 10).

In-class Activity

Write a program that reads a message, then prints the reversal of the message:

```
Enter a message: Don't get mad, get even.
Reversal is: .neve teg ,dam teg t'noD
```

Read the message one character at a time using `getchar` and store the characters in an array. Stop reading when the array is full or '\n' is input.

Use a loop with a pointer to access the array (don't use array subscripts at all).

In-class Activity

```c
#include <stdio.h>
#define N 100

int main() {
        char a[N];
        char *p;

        printf("Enter a message: ");
        p = a;
        while(p < a + N && (*p++ = getchar()) != '\n');

        printf("Reversal is: ");
        p--;
        while(p-- >= a)
                putchar(*p);

        putchar('\n');
        return 0;
}
```