Pointer & Array

“Reference to a Location”

Prerequisite: Pointer Basic

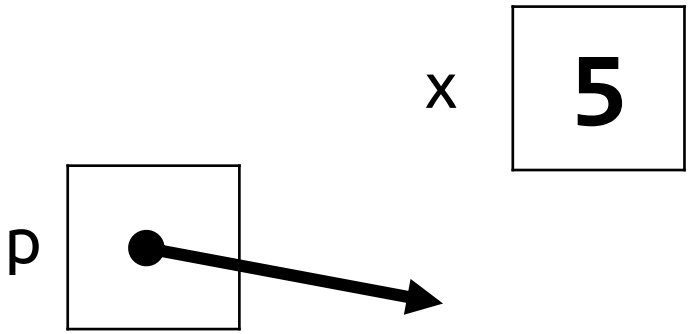
Find more contents at
https://sites.google.com/view/cse105june18/home

Md. Saidul Hoque Anik
onix.hoque.mist@gmail.com

Pointer

Recap

```
int x = 5;
int * p;
```

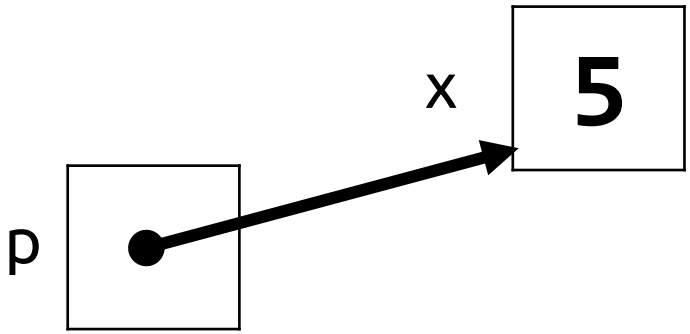


(Currently pointing
to unknown address)

Pointer

Recap

```
int x = 5;
int * p;

p = &x; //&x gives the 'address' of x
```

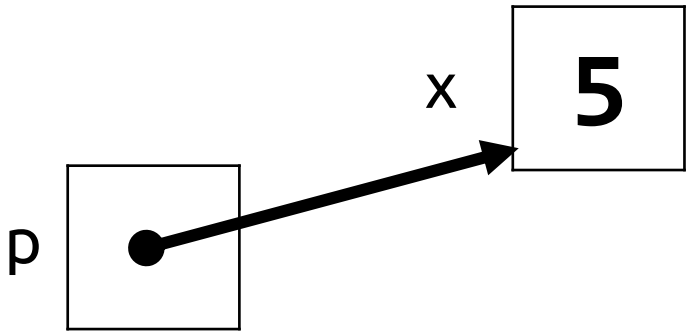


(Currently pointing to address of x)

Pointer

Recap

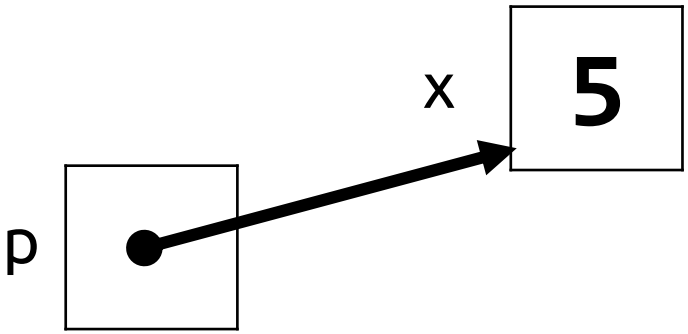
```c
int x = 5;
int * p;

p = &x; //&x gives the 'address' of x

printf("%d", *p);    //*p gives the content
                     // that p is pointing at
```



(Currently pointing to address of x)

Pointer

So, what is a pointer?

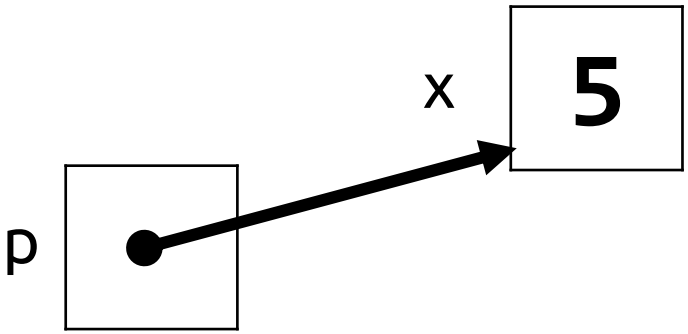


(Currently pointing
to address of x)

Pointer

So, what is a pointer?

A variable that `holds' the address of another variable



(Currently pointing to address of x)

Pointer

What is the size of it?

```c
int x = 5;
int * p;
p = &x;

printf("%d", sizeof x); // 4 byte
printf("%d", sizeof p); // ?
```

Pointer

What is the size of it?

```
int x = 5;
int * p;
p = &x;

printf("%d", sizeof x); // 4 byte
printf("%d", sizeof p); // ?
```

What does it hold?

Pointer

What is the size of it?

```
int x = 5;
int * p;
p = &x;

printf("%d", sizeof x); // 4 byte
printf("%d", sizeof p); // ?
```

What does it hold?

Normally,
A 4 GB RAM = 4 x 2^{30} Byte = 2^{32} Byte = 2^{32} Boxes

Pointer

What is the size of it?

```c
int x = 5;
int * p;
p = &x;

printf("%d", sizeof x); // 4 byte
printf("%d", sizeof p); // ?
```

What does it hold?

Normally,

A 4 GB RAM = 4 x 2^{30} Byte = 2^{32} Byte = 2^{32} Boxes

0	1	2	3	4	5	6	7

8 = 2^3 Boxes can be named using 3 bits.

```
000 => 0
001 => 1
010 => 2
011 => 3
100 => 4
101 => 5
110 => 6
111 => 7
```

Pointer

What is the size of it?

```c
int x = 5;
int * p;
p = &x;

printf("%d", sizeof x); // 4 byte
printf("%d", sizeof p); // ?
```

What does it hold?

Normally,
A 4 GB RAM = 4 x 2^{30} Byte = 2^{32} Byte = 2^{32} Boxes

0	1	2	3	4	5	6	7

$8 = 2^3$ Boxes can be named using 3 bits.
Therefore, 2^{32} Boxes can be addressed using 32 bits
= 4 Byte

```
000 => 0
001 => 1
010 => 2
011 => 3
100 => 4
101 => 5
110 => 6
111 => 7
```

Pointer

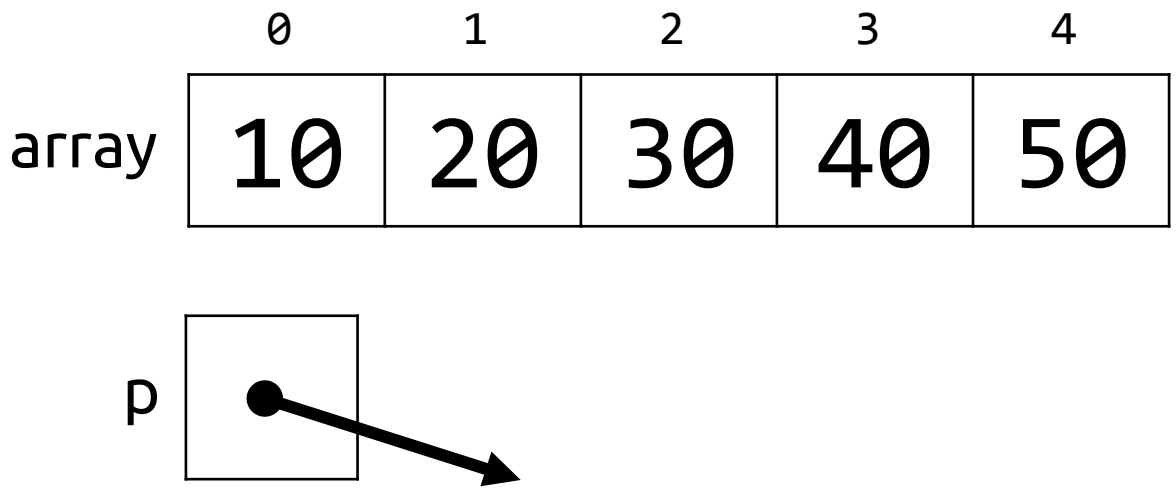
What is the size of it?

```c
int x = 5;
int * p;
p = &x;

printf("%d", sizeof x); // 4 byte
printf("%d", sizeof p); // ?
```

Depending on the architecture of the system, the size can be 4 Byte or 8 Byte

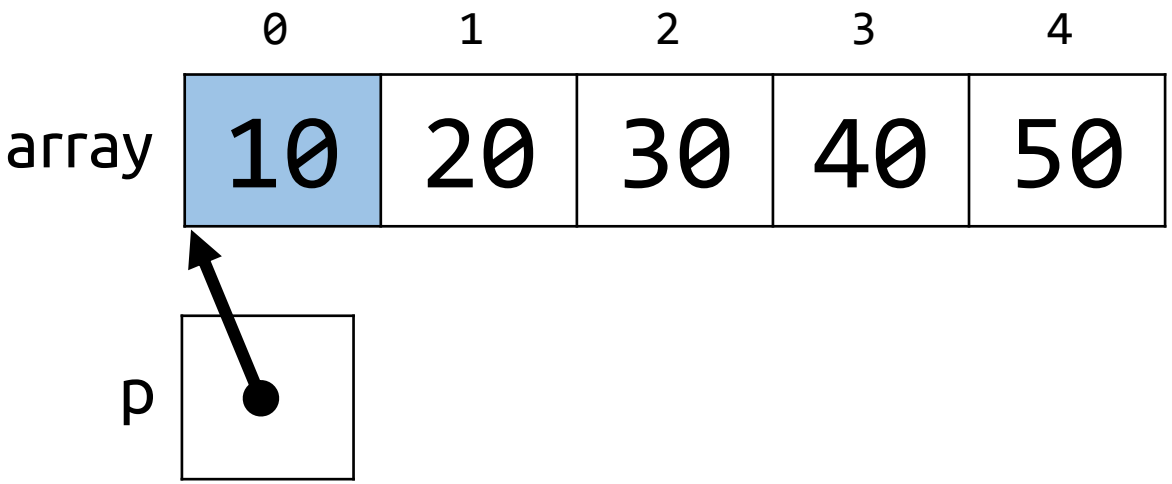
Pointing to Array

```
int array[] = {10, 20, 30, 40, 50};
int * p;
```



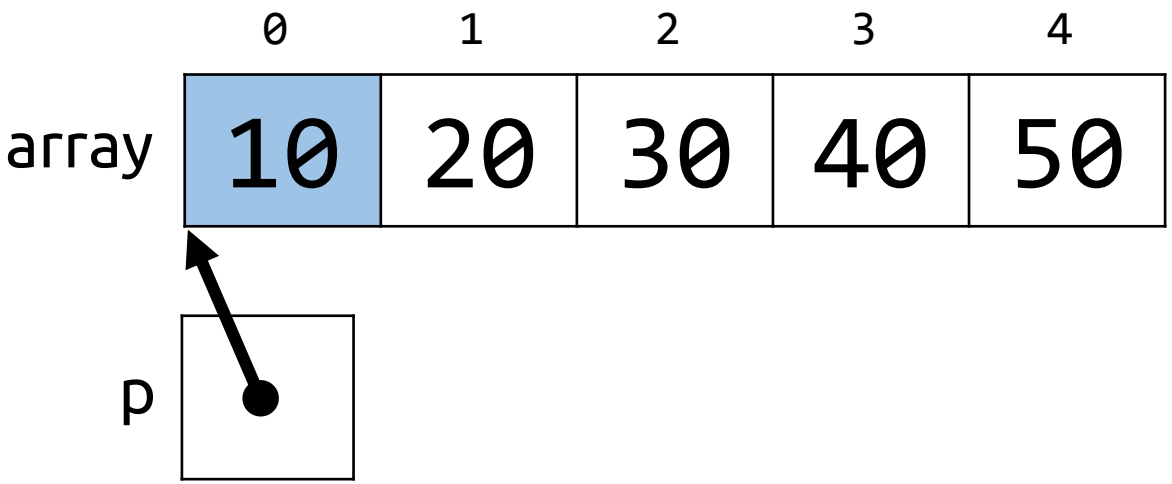
Pointing to Array

```
int array[] = {10, 20, 30, 40, 50};
int * p;

p = &array[0];
```



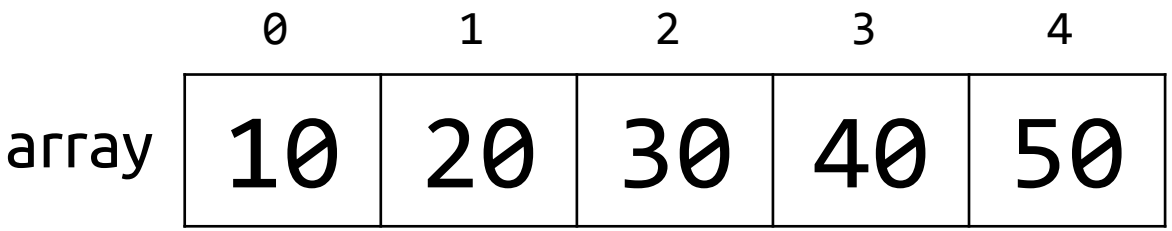
Pointing to Array

```
int array[] = {10, 20, 30, 40, 50};
int * p;

p = &array[0];

printf("%d", *p);    // What will be the output?
```



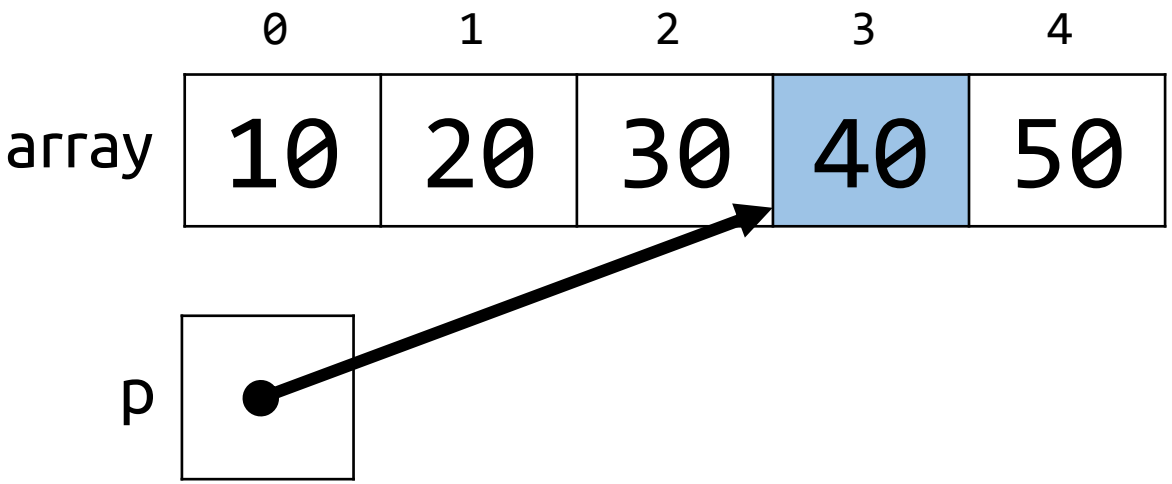
Pointing to Array

```
int array[] = {10, 20, 30, 40, 50};
int * p;

p = &array[3];

printf("%d", *p);    // What will be the output?
```



Pointing to Array

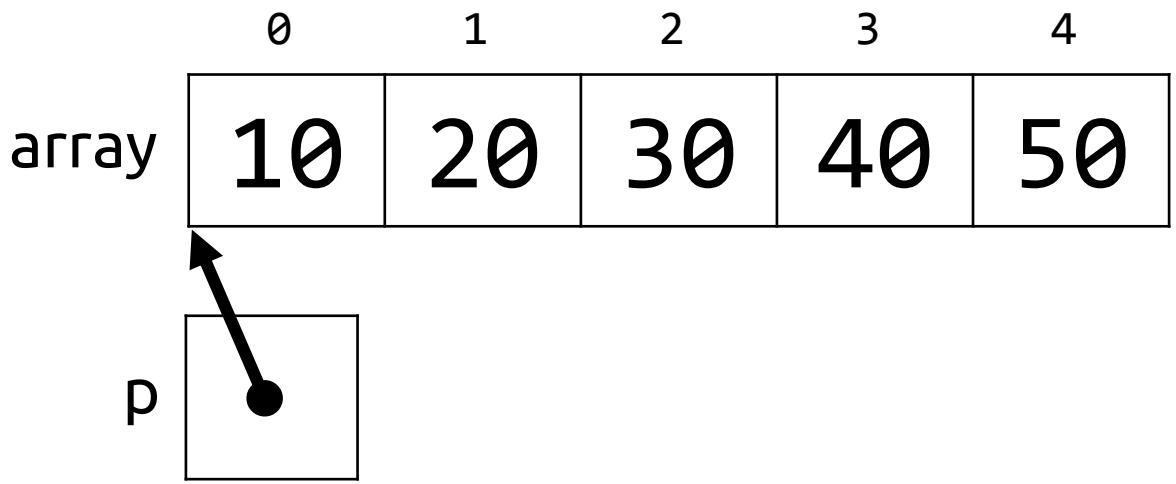
```
int array[] = {10, 20, 30, 40, 50};
int * p;

p = &array[3];

printf("%d", *p);    // What will be the output?
```



Pointer arithmetic

Let's go back to the previous example

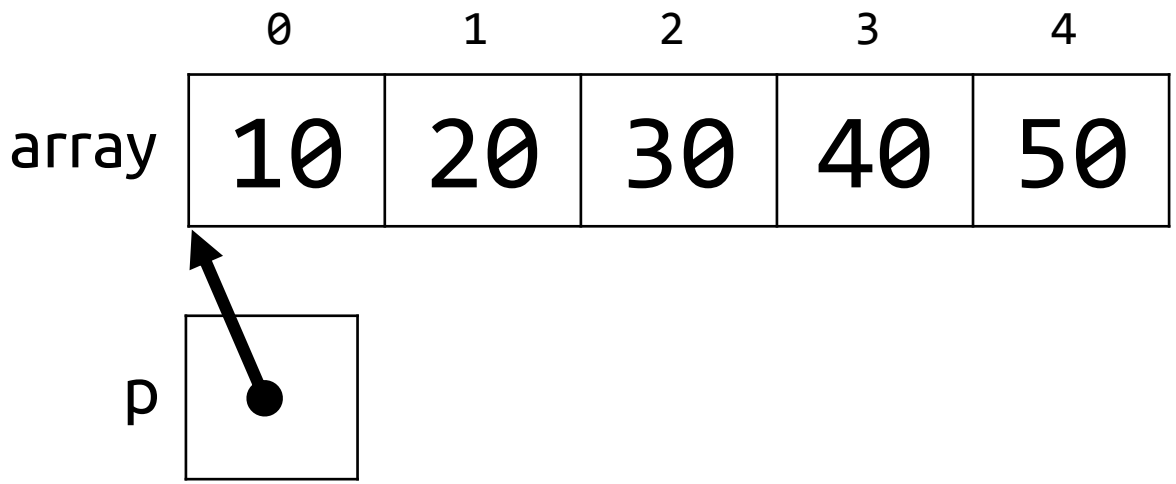
```
int array[] = {10, 20, 30, 40, 50};
int * p;

p = &array[0];
```



Pointer arithmetic

What happens if we increment p?

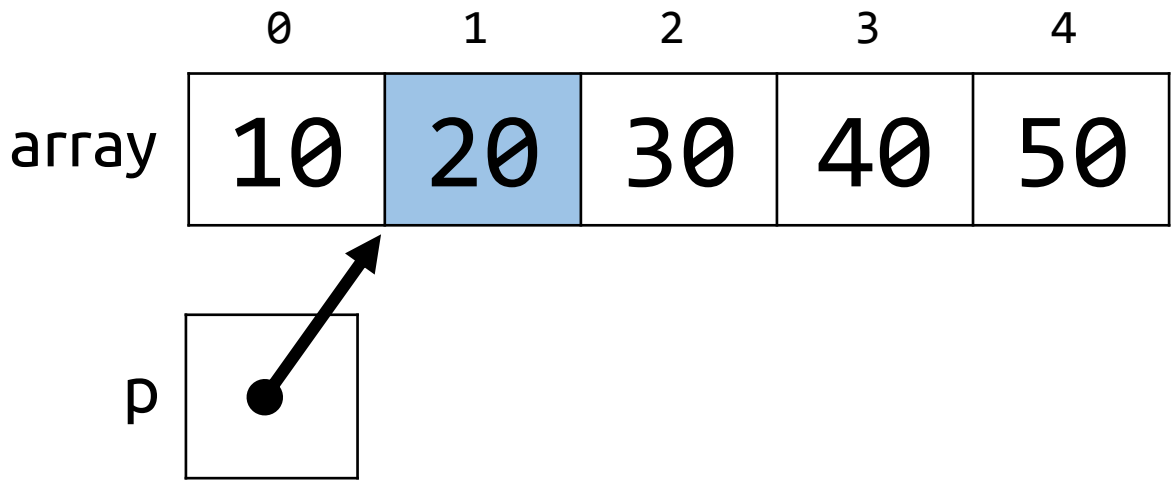
```
int array[] = {10, 20, 30, 40, 50};
int * p;

p = &array[0];

p = p + 1;
```



Pointer arithmetic

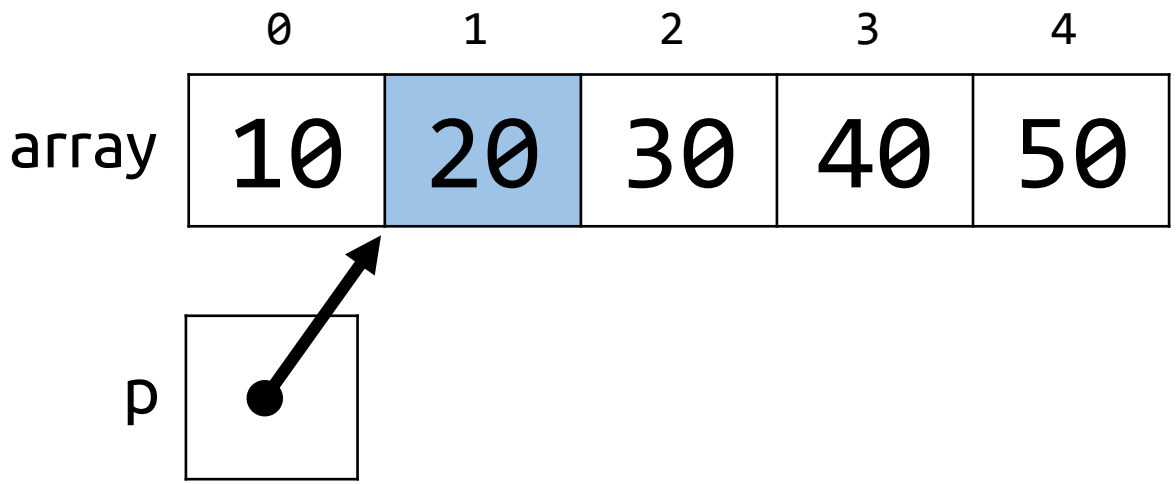
p will 'advance' one `int` size (because it's type is `int`)

```
int array[] = {10, 20, 30, 40, 50};
int * p;

p = &array[0];

p = p + 1;
```



Pointer arithmetic

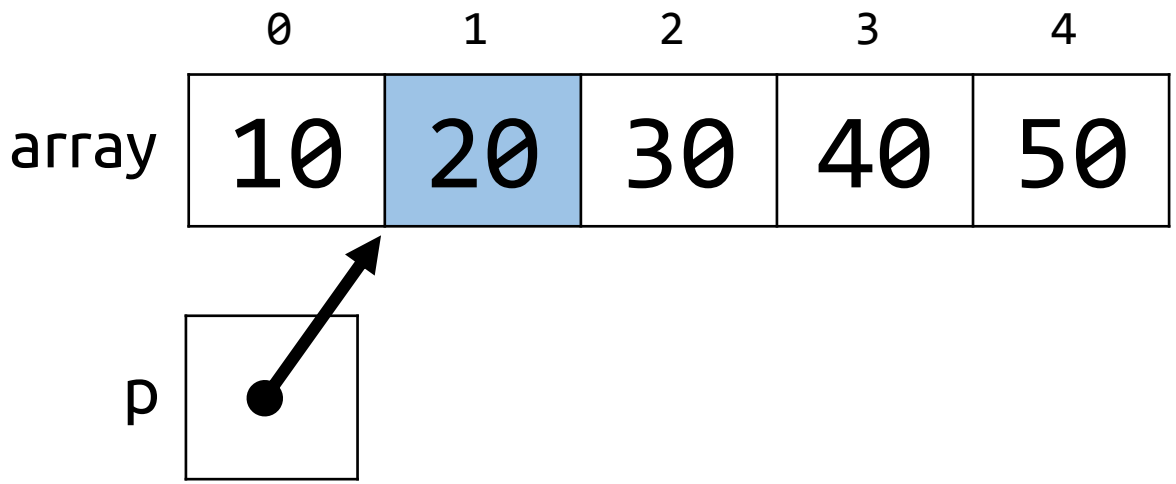
```
int array[] = {10, 20, 30, 40, 50};
int * p;

p = &array[0];

p = p + 1;

printf("%d", *p);    // What will be the output?
```



Pointer arithmetic

How can we go to index 4 to access 50 now?

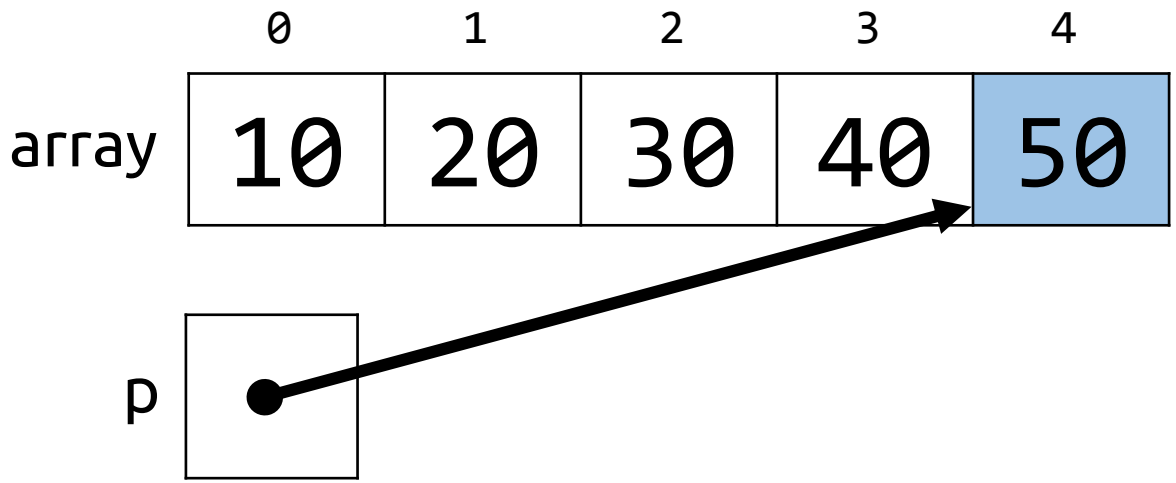
```
int array[] = {10, 20, 30, 40, 50};
int * p;

p = &array[0];

p = p + 1;

printf("%d", *p);    // What will be the output?
```



Pointer arithmetic

How can we go to index 4 to access 50 now?

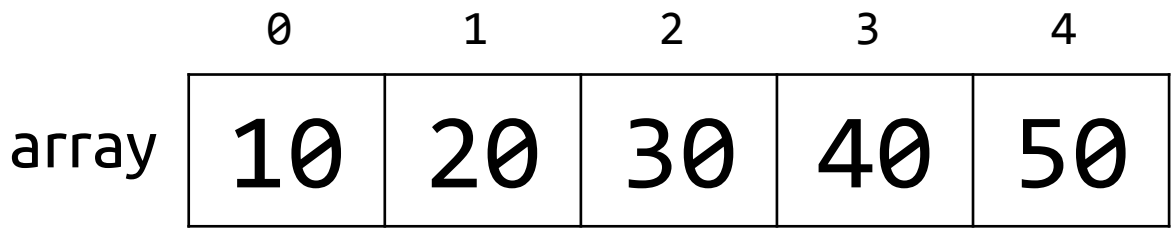
```
int array[] = {10, 20, 30, 40, 50};
int * p;

p = &array[0];

p = p + 1;

printf("%d", *p);    // What will be the output?

p = p + 3;
```



Pointer arithmetic

We can also use this shortcut

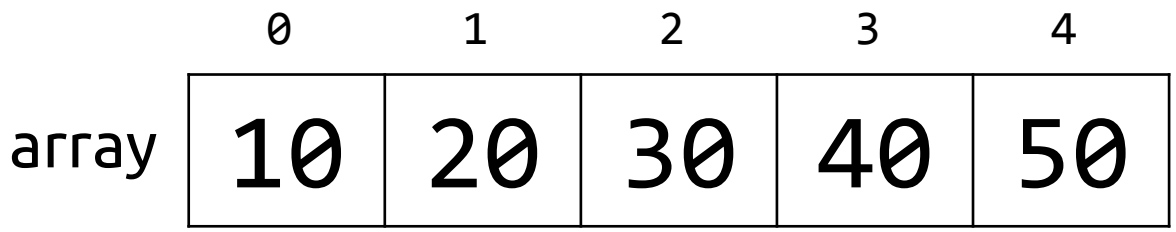
```
int array[] = {10, 20, 30, 40, 50};
int * p;

p = &array[0];

printf("%d", *(p + 4)); // 50
```



Pointer arithmetic

Where is p pointing now?

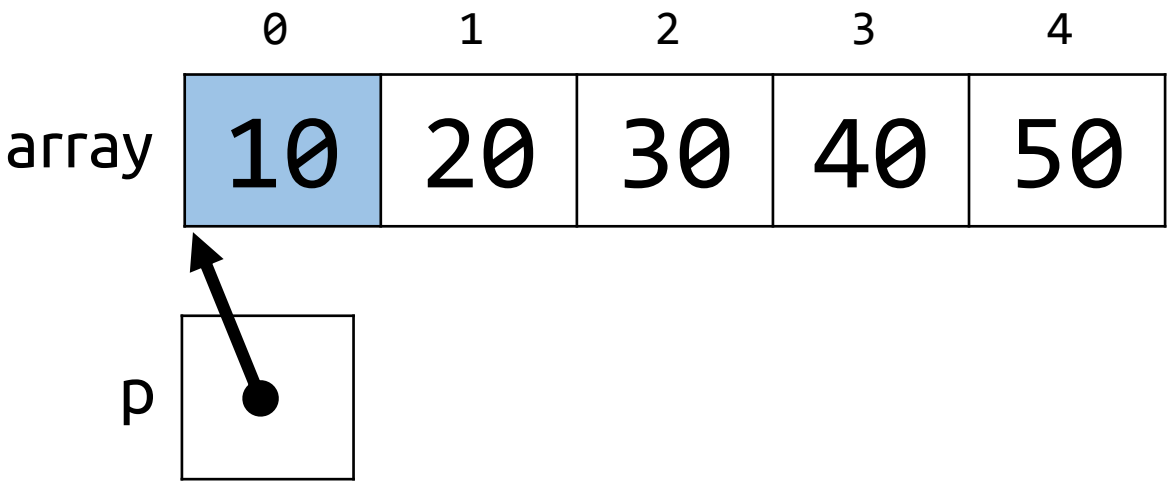
```
int array[] = {10, 20, 30, 40, 50};
int * p;

p = &array[0];

printf("%d", *(p + 4)); // 50
```



Pointing to Array

p is still pointing at array[0]

```
int array[] = {10, 20, 30, 40, 50};
int * p;

p = &array[0];

printf("%d", *(p + 4)); // 50
```



Array as function parameter

We want to access the array from fn

```
int main()
{
    int array[] = {10, 20, 30, 40, 50};

    return 0;
}

void fn()
{
    //What will be the parameter of fn?
}
```

Array as function parameter

We can use pointer as parameter

```
int main()
{
    int array[] = {10, 20, 30, 40, 50};

    return 0;
}

void fn(int * p)
{
    //we want to access the array from here
}
```

Array as function parameter

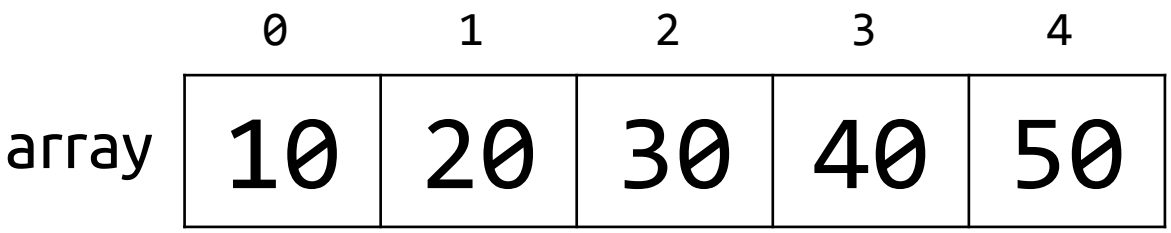
How do we pass the array in function?

```
int main()
{
    int array[] = {10, 20, 30, 40, 50};

    return 0;
}

void fn(int * p)
{
    //we want to access the array from here
}
```

Array as function parameter

We simple pass the 0th element's address

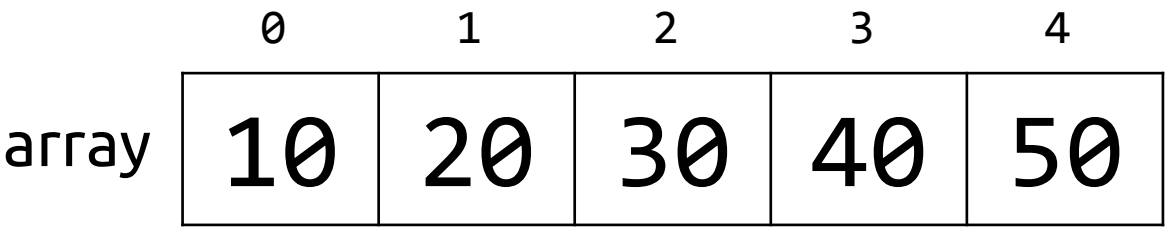
```c
int main()
{
    int array[] = {10, 20, 30, 40, 50};
    fn(&array[0]);
    return 0;
}

void fn(int * p)
{
    printf("%d\n", *p); //10
}
```



Array as function parameter

How can we print the element at index 3?

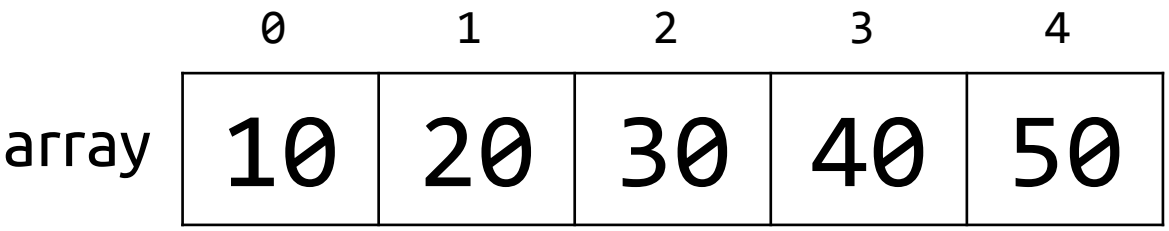
```c
int main()
{
    int array[] = {10, 20, 30, 40, 50};
    fn(&array[0]);
    return 0;
}

void fn(int * p)
{
    printf("%d\n", *p); //10
}
```



Array as function parameter

Pointer arithmetic

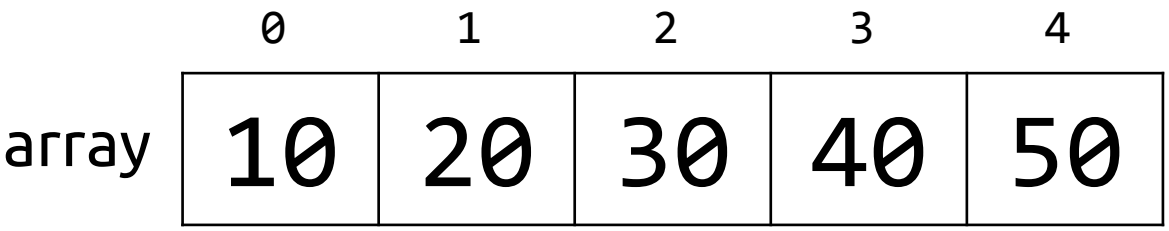
```c
int main()
{
    int array[] = {10, 20, 30, 40, 50};
    fn(&array[0]);
    return 0;
}

void fn(int * p)
{
    printf("%d\n", *(p+3)); //40
}
```



Array as function parameter

We can also write it like this

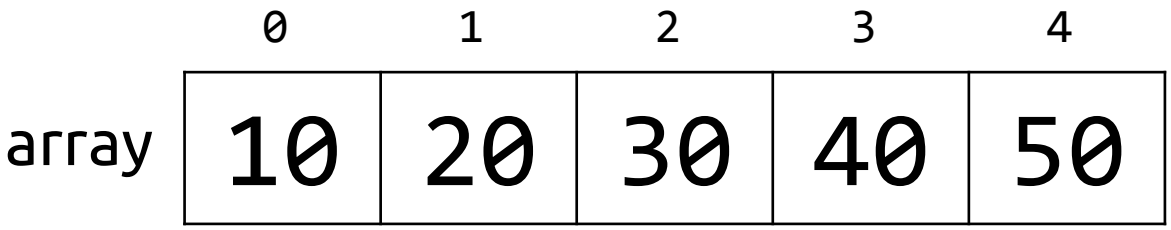
```c
int main()
{
    int array[] = {10, 20, 30, 40, 50};
    fn(&array[0]);
    return 0;
}

void fn(int * p)
{
    printf("%d\n", p[3]);   //40, same as *(p+3)
}
```



Array as function parameter

Write a function to print all the contents of an array

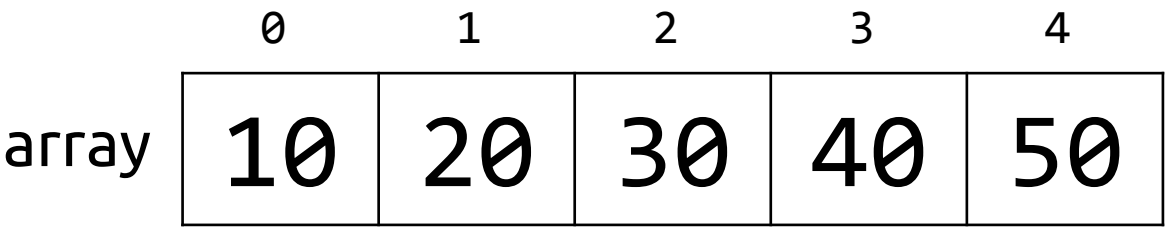
```
int main()
{
    int array[] = {10, 20, 30, 40, 50};
    print_array(              );
    return 0;
}

void print_array(                  )
{




}
```



Array as function parameter

Write a function to print all the contents of an array

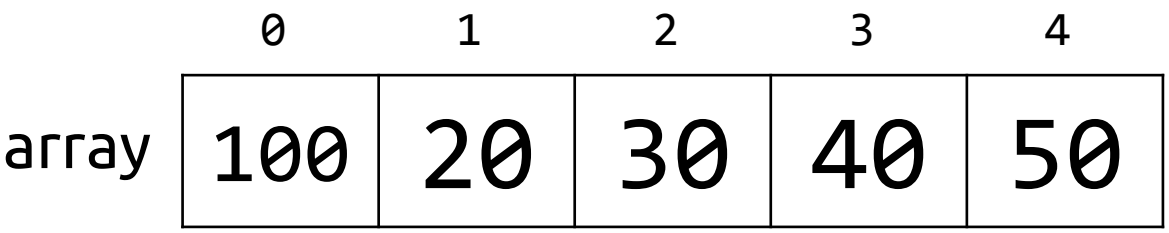
```c
int main()
{
    int array[] = {10, 20, 30, 40, 50};
    print_array(&array[0], 5);
    return 0;
}

void print_array(int * p, int len)
{
    int i;
    for (i = 0; i<len; i++)
        printf("%d\n", p[i]);
}
```



Array as function parameter

Modifying the contents

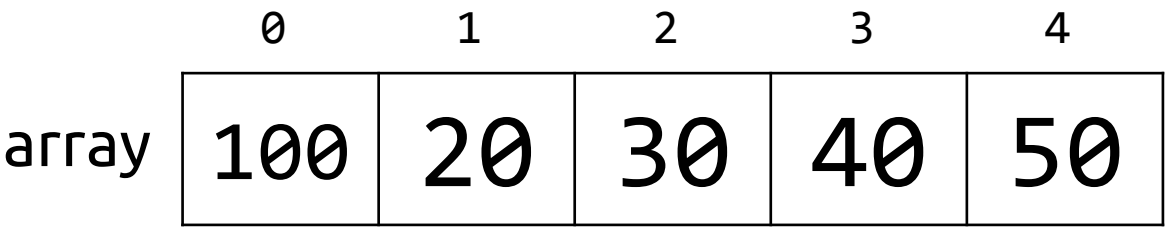
```c
int main()
{
    int array[] = {10, 20, 30, 40, 50};
    fn(&array[0]);

    return 0;
}

void fn(int * p)
{
    p[0] = 100;
}
```



Array as function parameter

Modifying the contents

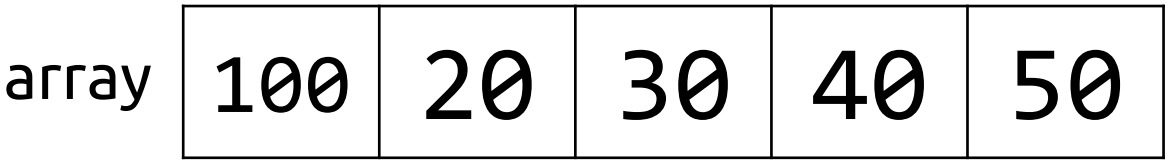
```c
int main()
{
    int array[] = {10, 20, 30, 40, 50};
    fn(&array[0]);
    printf("%d", array[0]); //100
    return 0;
}

void fn(int * p)
{
    p[0] = 100;
}
```



Array as function parameter

Write a function to add five with each elemt.

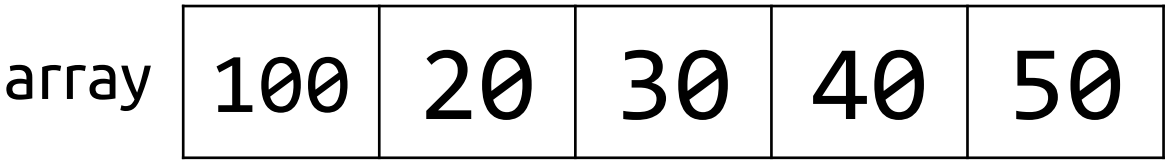
```
int main()
{
    int array[] = {10, 20, 30, 40, 50};
    add_five(&array[0], 5);
    print_array(&array[0], 5); //15 25 35 45 55
    return 0;
}

void add_five(int * p, int len)
{



}
```



Array as function parameter

Write a function to add five with each elemt.

```c
int main()
{
    int array[] = {10, 20, 30, 40, 50};
    add_five(&array[0], 5);
    print_array(&array[0], 5); //15 25 35 45 55
    return 0;
}

void add_five(int * p, int len)
{
    int i;
    for (i = 0; i<len; i++)
        p[i] = p[i] + 5;
}
```



Taking a closer look at the [ ] operator

`p[3]` is equivalent to `*(p+3)`

Taking a closer look at the [ ] operator

`p[3]` is equivalent to `*(p+3)`

```
int main()
{
    int array[] = {10, 20, 30, 40, 50};
    fn(&array[0]);
    return 0;
}

void fn(int * p)
{
    printf("%d\n", p[3]);    //40, same as *(p+3)
}
```

Taking a closer look at the [ ] operator

`p[3]` is equivalent to `*(p+3)`

```
int main()
{
    int array[] = {10, 20, 30, 40, 50};
    fn(&array[0]);
    return 0;
}

void fn(int * p)
{
    printf("%d\n", 3[p]);    //What will it print?
}
```

Another observation

`p[3]` is equivalent to `*(p+3)`

```
int main()
{
    int array[] = {10, 20, 30, 40, 50};
    int x = array[2]; //30
    return 0;
}
```

Another observation

We can write `*(array+2)` instead of `array[2]`

`p[3]` is equivalent to `*(p+3)`

```
int main()
{
    int array[] = {10, 20, 30, 40, 50};
    int x = *(array+2); //x = 30
    printf("%d", x);    //prints 30
    return 0;
}
```

Another observation

We can write `*(array+2)` instead of `array[2]`

`p[3]` is equivalent to `*(p+3)`

```
int main()
{
    int array[] = {10, 20, 30, 40, 50};
    int x = *(array+2); //x = 30
    printf("%d", x);    //prints 30
    return 0;
}
```

That means, the identifier `array` is sometimes treated as a pointer.

Exception: in sizeof, array is still an array

```
printf("%d", sizeof array); //20Byte, 4 Byte x 5
```

Another observation

So, assigning address to a pointer is now easier

```
int main()
{
    int array[] = {10, 20, 30, 40, 50};
    int * x = array;/instead of &array[0]
    printf("%d", *x);    //10
    return 0;
}
```

Another observation

So, assigning address to a pointer is now easier

```
int main()
{
    int array[] = {10, 20, 30, 40, 50};
    fn(array);
    return 0;
}

void fn(int * p)
{
    p[0] = 100;
}
```