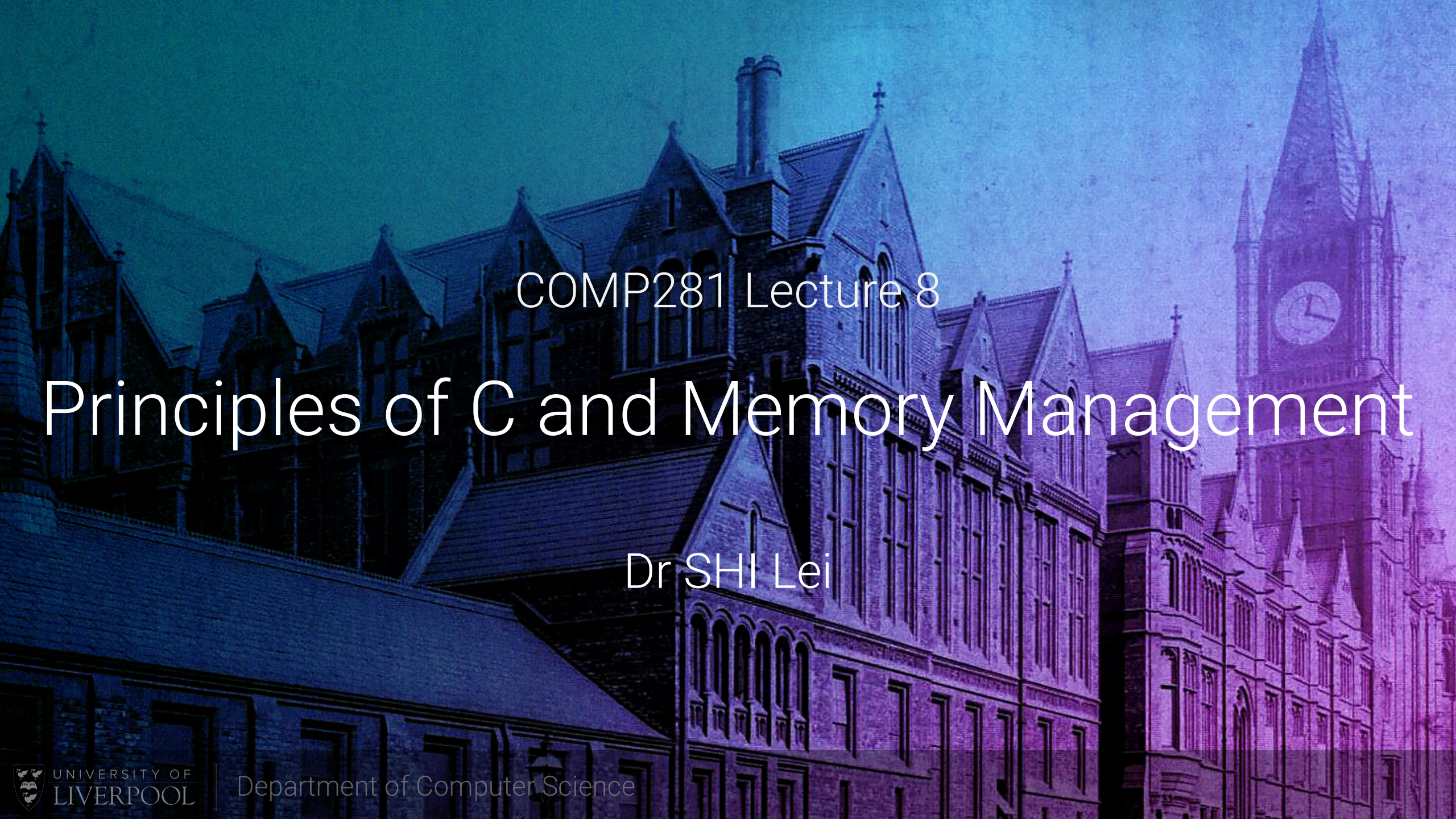


COMP281 Lecture 8

Principles of C and Memory Management

Dr SHI Lei

UNIVERSITY OF LIVERPOOL | Department of Computer Science

Last Lecture

- Pointer Basics

Previous Lectures

- Arrays, Arithmetic, Functions

Last Lecture

- Pointer Basics

Previous Lectures

- Arrays, Arithmetic, Functions

- Pointer Basics
  - Variable
  - Address `&`
  - Pointer `*`
  - Double Pointer `**`

```
int   x =  9;
int  *y = &x;
int **z = &y;
```

- Arrays
  - Declaring
  - Initialising
  - Accessing
  - 2D arrays

```
int n[10], i, j;
for(i=0; i<10; i++) {
   n[i] = i + 100;
}
for(j=0; j<10; j++) {
   printf("n[%d]=%d\n", j, n[j]);
}
```

```
for (i=0; i<4; i++) {
   for (j=0; j<3; j++) {
      arr[i][j]…;
   }
} /* 2D array */
```

- Functions
  - Declaring
  - Initialising
  - Accessing
  - Call-by-value vs call-by-reference

```
void incr(int x) {
    x++;
}
int x = 10;
incr(x);
```

```
void incr(int *x) {
    (*x)++;
}
int x = 10;
incr(&x);
```

Last Lecture

- Pointer Basics

Previous Lectures

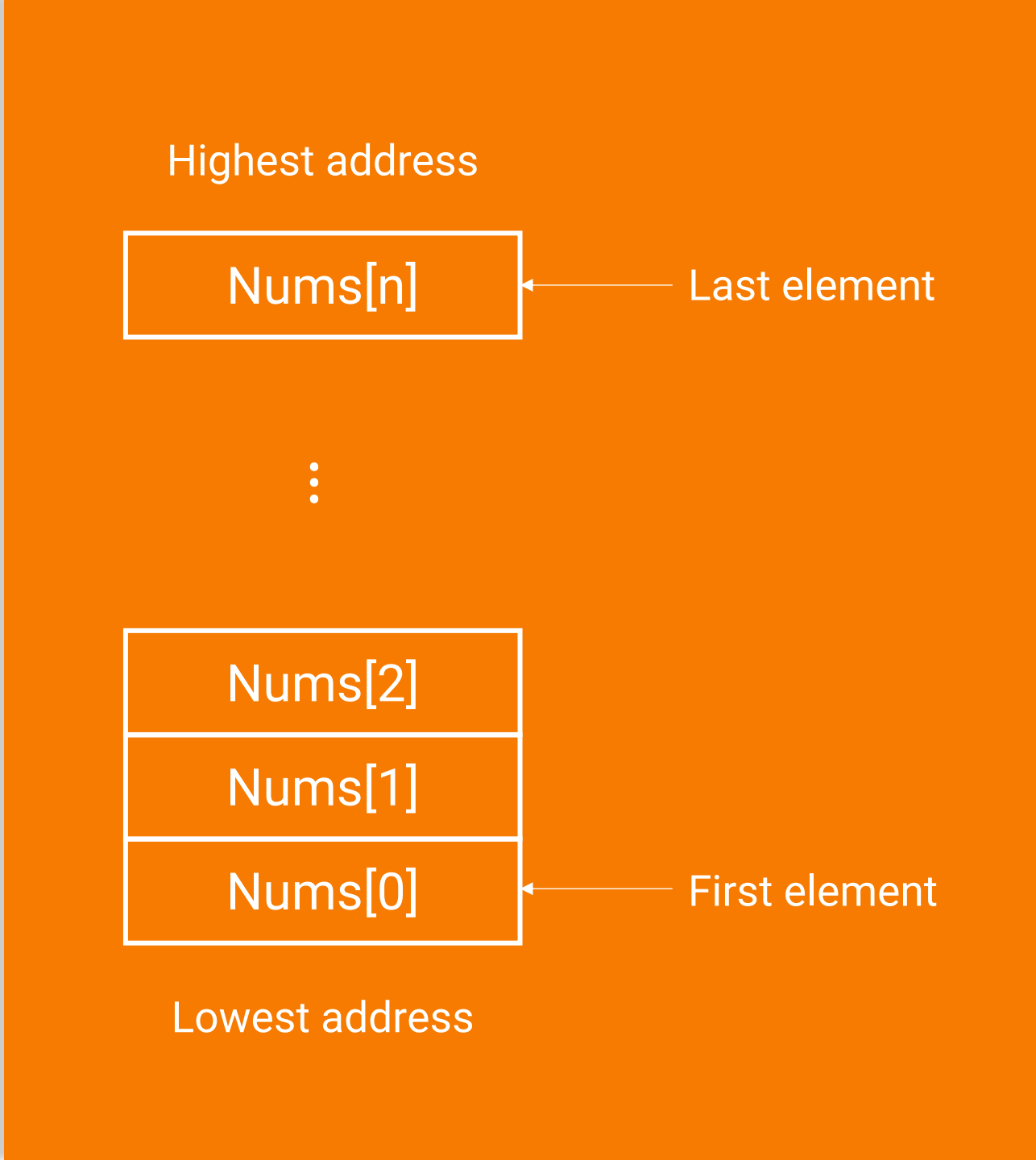
- Arrays, Arithmetic, Functions

Today

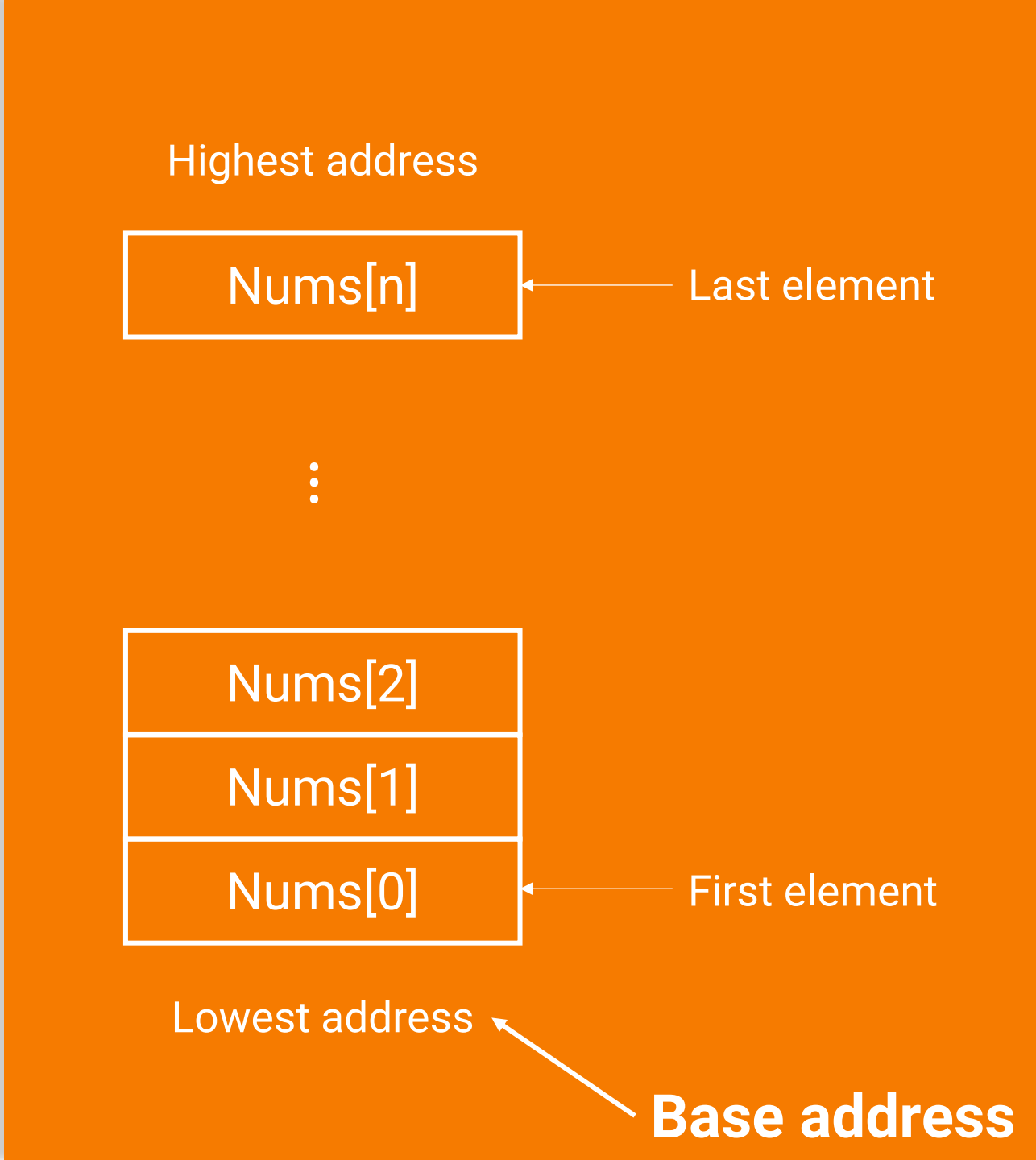
- Pointer to Array
- Pointer Arithmetic
- Pointer with Functions

Pointer to Array

- An array consist of contiguous memory locations.
- The highest address corresponds to the last element.
- The lowest address corresponds to the first element.



- When an array is declared, the Compiler allocates sufficient amount of memory to contain all the elements of the array.
- Base address i.e. address of the first element of the array is also allocated by the Compiler



Address vs pointer

Let `int arr[5] = {1,2,3,4,5};`

Assuming the base address of `arr` is 1000, and each integer requires 4 bytes, the 5 elements will be stored as:

element	`arr[0]`	`arr[1]`	`arr[2]`	`arr[3]`	`arr[4]`
address	**1000**	**1004**	**1008**	**1012**	**1016**

Address vs pointer

Let `int arr[5] = {1,2,3,4,5};`

Assuming the base address of `arr` is 1000, and each integer requires 4 bytes, the 5 elements will be stored as:

	1	2	3	4	5
element	`arr[0]`	`arr[1]`	`arr[2]`	`arr[3]`	`arr[4]`
address	**1000**	**1004**	**1008**	**1012**	**1016**

you can not change this once you define the program

The variable `arr` gives the **Base Address**, a **Constant Pointer**, pointing to the 1st element of the array – `arr[0]`, so `arr` contains the address of `arr[0]` i.e. 1000.

`arr` has 2 purpose

- It is the name of the array
- It acts as a pointer pointing towards the 1st element in the array

`arr` is equal to `&arr[0]` by default

- We can declare a pointer of type `int` to point to the array `arr`

Example 1

```c
#include <stdio.h>
int main() {
    int arr[5] = {1,2,3,4,5};

    int *p = arr;
    printf("%i\n", *p);

    p++;
    printf("%i\n", *p);

    return 0;
}
```

Example 1

```c
#include <stdio.h>
int main() {
    int arr[5] = {1,2,3,4,5};

    int *p = arr;
    printf("%i\n", *p);

    p++;
    printf("%i\n", *p);

    return 0;
}
```

Output

```
1
2
```

Use a pointer to point to an array, then we can use the pointer to **access** the elements of the array.

Example 2

```c
#include <stdio.h>
int main() {
    int i;
    int arr[5] = {1, 2, 3, 4, 5};
    int *p = arr;
    for (i = 0; i < 5; i++) {
        printf("%d\n", *p);
        p++;
    }
    return 0;
}
```

Example 2

```c
#include <stdio.h>
int main() {
    int i;
    int arr[5] = {1, 2, 3, 4, 5};
    int *p = arr;
    for (i = 0; i < 5; i++) {
        printf("%d\n", *p);
        p++;
    }
    return 0;
}
```

Output

```
1
2
3
4
5
```

The pointer `*p` prints all the values stored in the array one by one.

Example 2

```c
#include <stdio.h>
int main() {
    int i;
    int arr[5] = {1, 2, 3, 4, 5};
    int *p = arr;
    for (i = 0; i < 5; i++) {
        printf("%d\n", *p);
        p++;
    }
    return 0;
}
```

What if we replace

```c
printf("%d\n", *p);
```

with

```c
printf("%d", arr[i]);
printf("%d", i[arr]);
printf("%d", arr+i );
printf("%d", *(arr+i));
printf("%d", *arr);
```

Example 2

```c
#include <stdio.h>
int main() {
    int i;
    int arr[5] = {1, 2, 3, 4, 5};
    int *p = arr;
    for (i = 0; i < 5; i++) {
        printf("%d\n", *p);
        p++;
    }
    return 0;
}
```

```c
printf("%d", arr[i]);
```

Prints all array elements

```c
printf("%d", i[arr]);
```

Also prints all elements of array

```c
printf("%d", arr+i);
```

Prints address of array elements

```c
printf("%d", *(arr+i));
```

Prints value of array elements

```c
printf("%d", *arr);
```

Prints value of a[0] only

Example 2

```c
#include <stdio.h>
int main() {
    int i;
    int arr[5] = {1, 2, 3, 4, 5};
    int *p = arr;
    for (i = 0; i < 5; i++) {
        printf("%d\n", *p);
        p++;
    }
    return 0;
}
```

What about

```c
arr++;
```

?

Example 2

```c
#include <stdio.h>
int main() {
    int i;
    int arr[5] = {1, 2, 3, 4, 5};
    int *p = arr;
    for (i = 0; i < 5; i++) {
        printf("%d\n", *p);
        p++;
    }
    return 0;
}
```

What about

```c
arr++;
```

?

Compiler time error -> cannot change **Base Address** of an array

(Constant Pointer)

Arrays are pointers in disguise.

Arrays: “syntactic sugar” for pointers.

```
int i = 0, arr[5] = {1, 2, 3, 4, 5};
printf("arr[i] = %d\n", arr[i]);
printf("arr[i] = %d\n", *(arr + i));
```

`arr[i]` and `*(arr + i)` are identical

`arr` is identical to `&arr[0]`

Pointer Arithmetic

It means we can do some calculation

Pointer arithmetic

Add/subtract integers to/from pointers

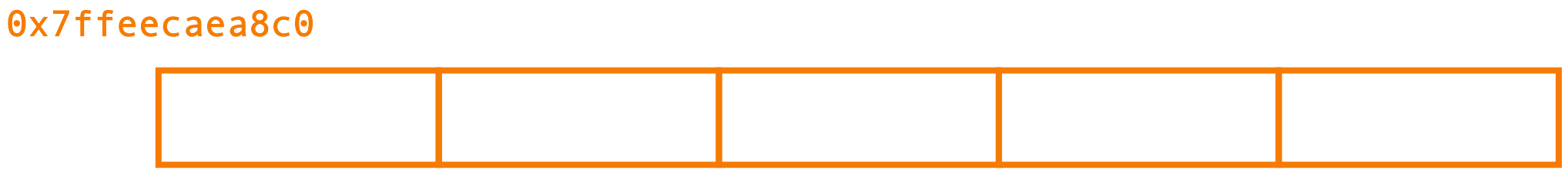
(assume 4 byte integers)

```
int arr[] = { 1, 2, 3, 4, 5 };

int *p = arr;          /* (*p) == ? */

p++;                   /* (*p) == ? */

p += 2;                /* (*p) == ? */

p -= 3;                /* (*p) == ? */
```

Pointer arithmetic

Add/subtract integers to/from pointers

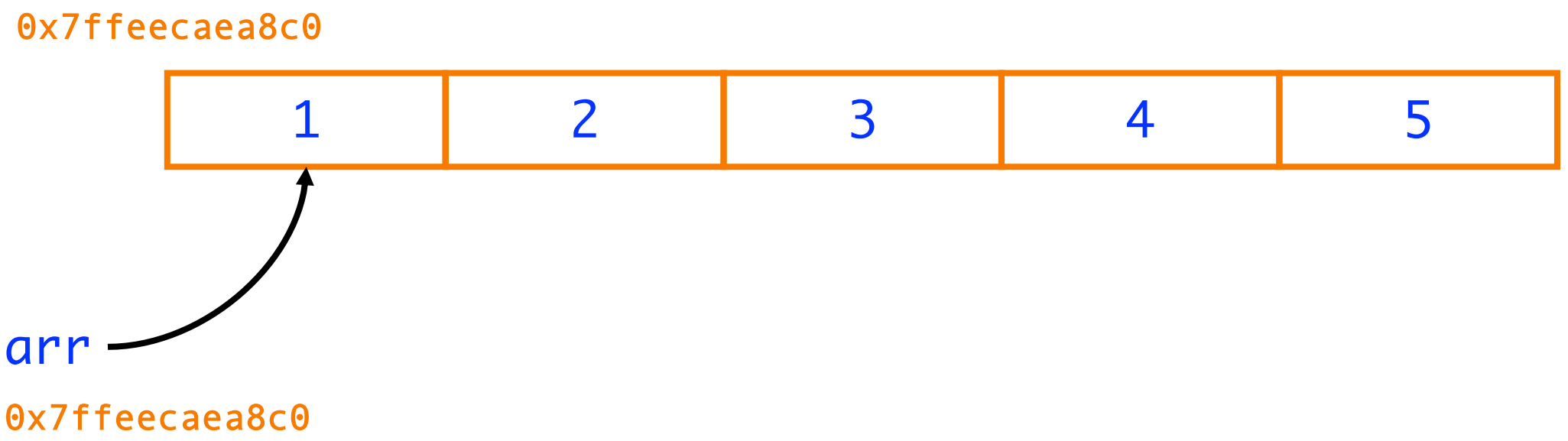
(assume 4 byte integers)



Pointer arithmetic

Add/subtract integers to/from pointers (assume 4 byte integers)

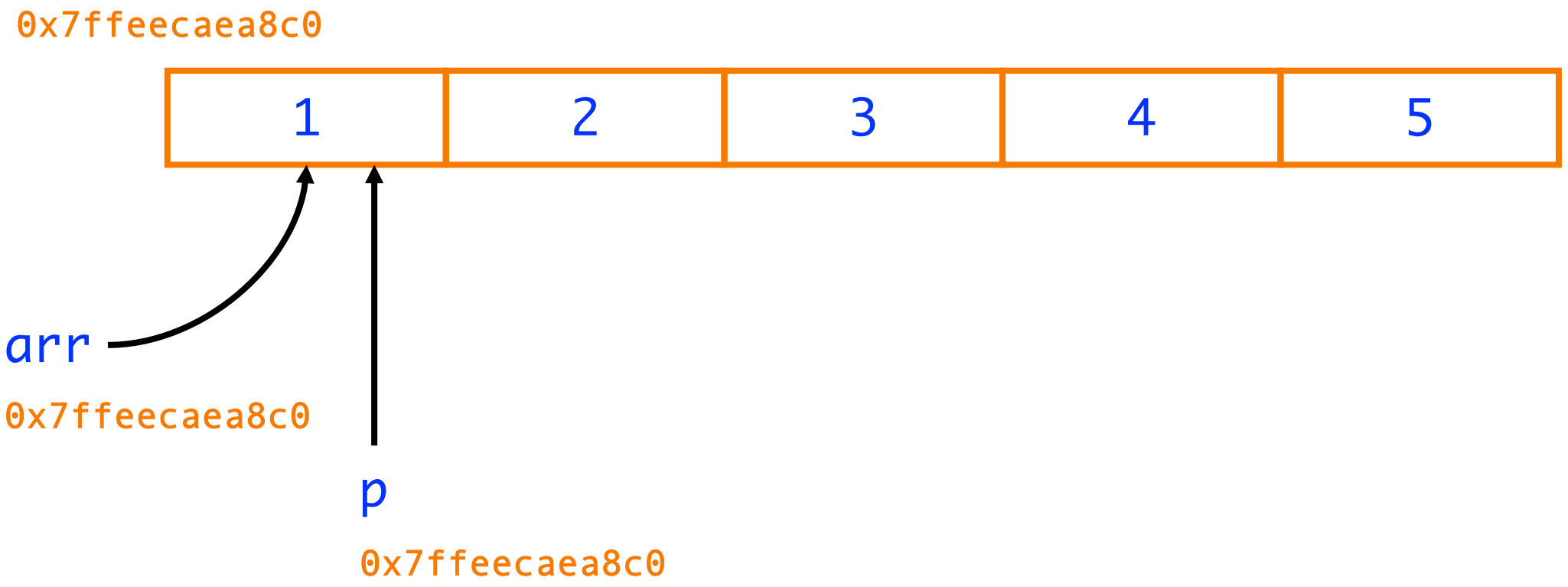
```
int arr[] = { 1, 2, 3, 4, 5 };
```



Pointer arithmetic

Add/subtract integers to/from pointers (assume 4 byte integers)

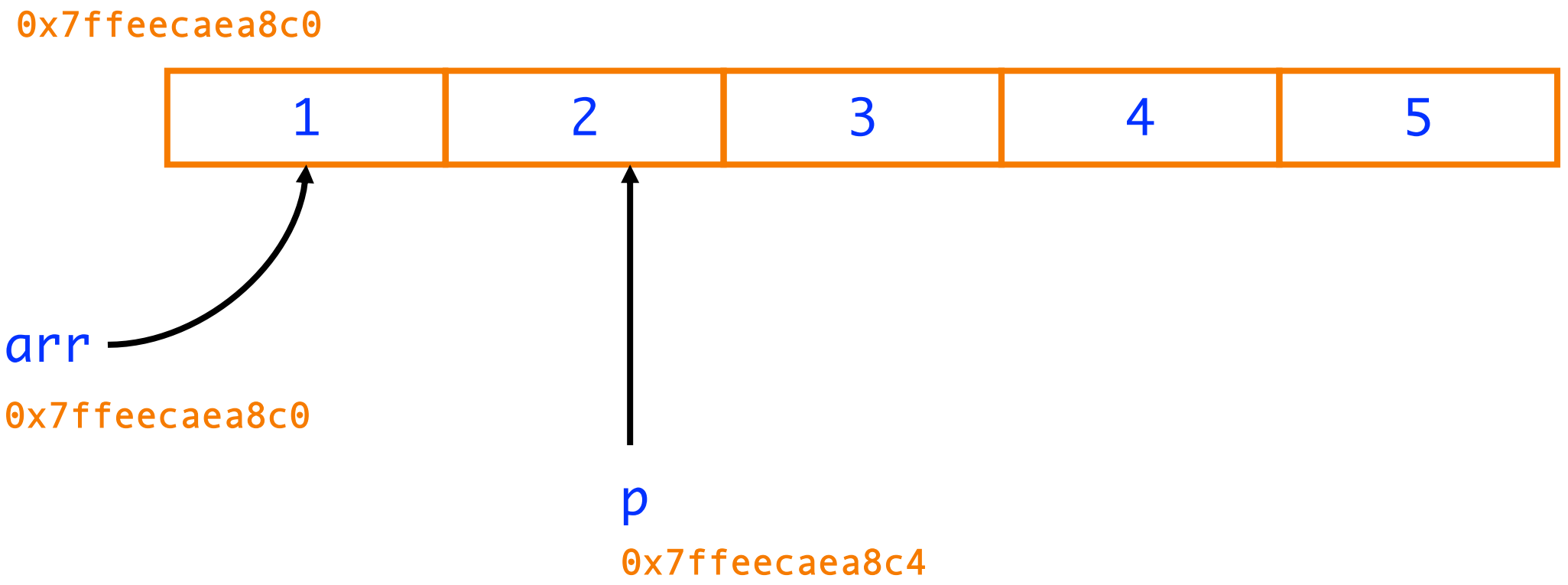
```
int *p = arr; /* (*p)==? */
```



Pointer arithmetic

Add/subtract integers to/from pointers (assume 4 byte integers)

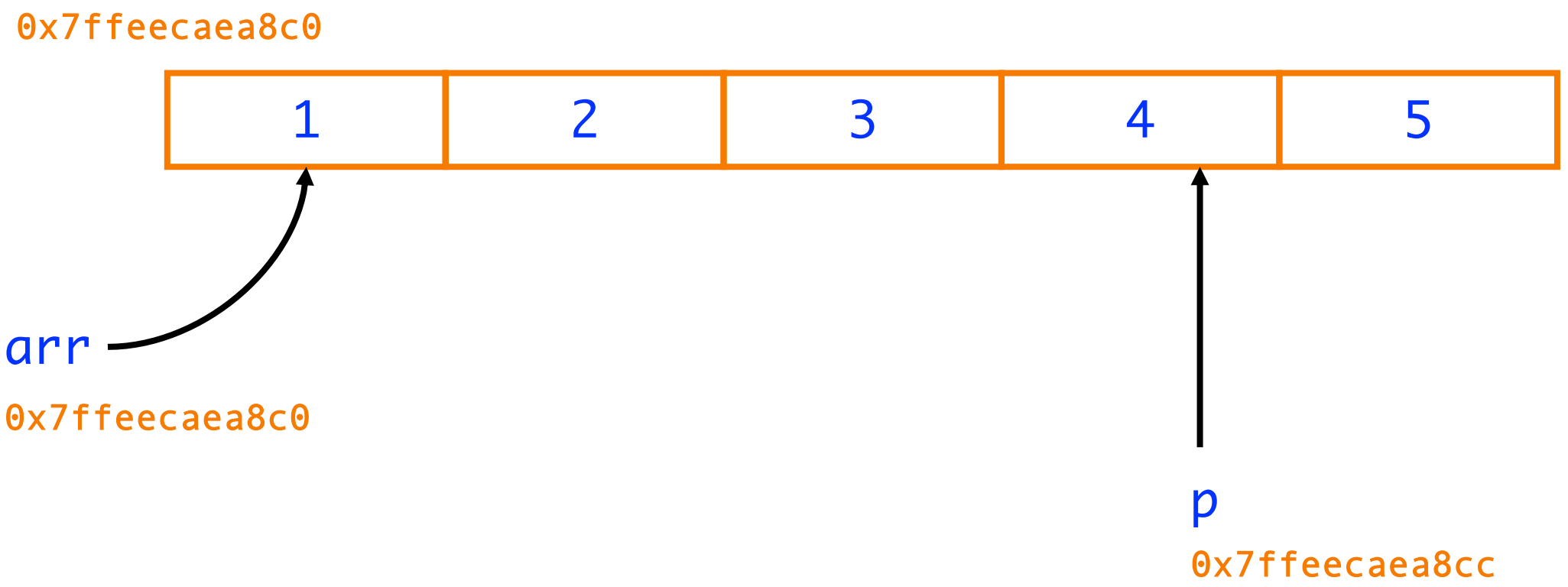
```
p++;                /* (*p)==? */
```



Pointer arithmetic

Add/subtract integers to/from pointers (assume 4 byte integers)

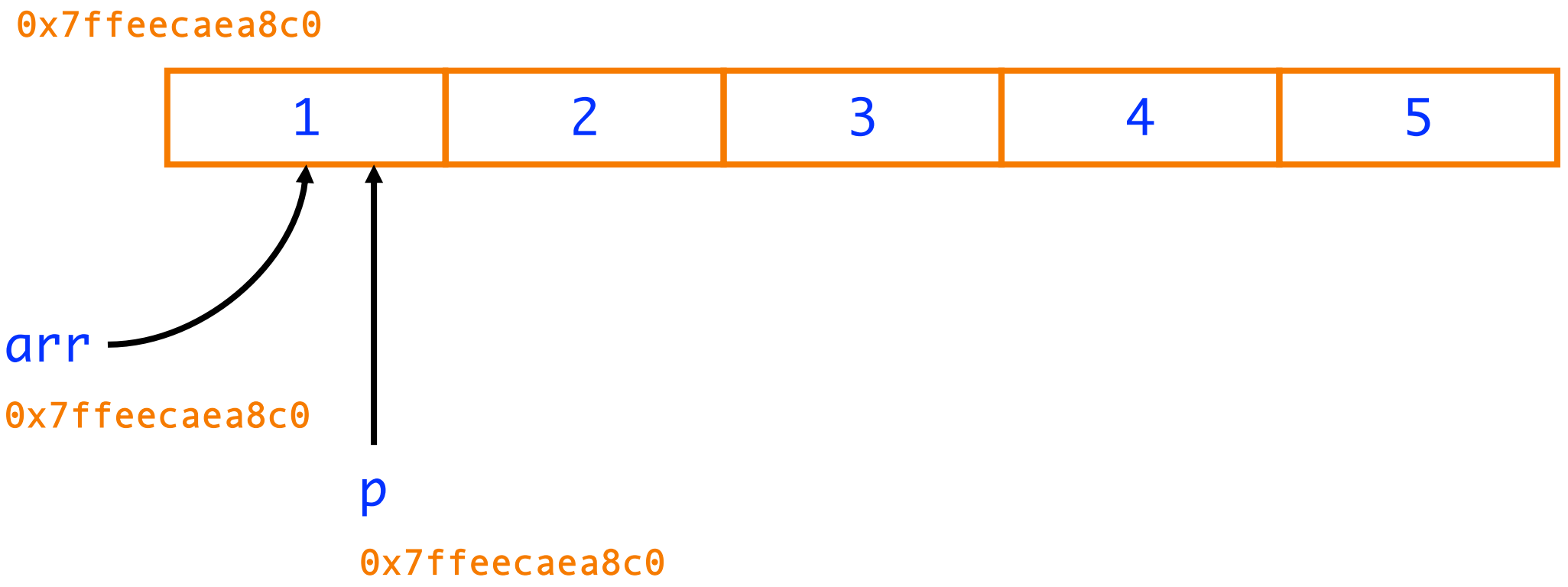
```
p += 2;          /* (*p)==? */
```



Pointer arithmetic

Add/subtract integers to/from pointers (assume 4 byte integers)

```
p -= 3;          /* (*p)==? */
```



Note:

Pointer arithmetic does NOT add/subtract address directly

but in multiplies of the size of type in bytes.

```
int arr[] = { 1, 2, 3, 4, 5 };
int *p = arr;
p++; /* means: p = p + sizeof(int); */
```

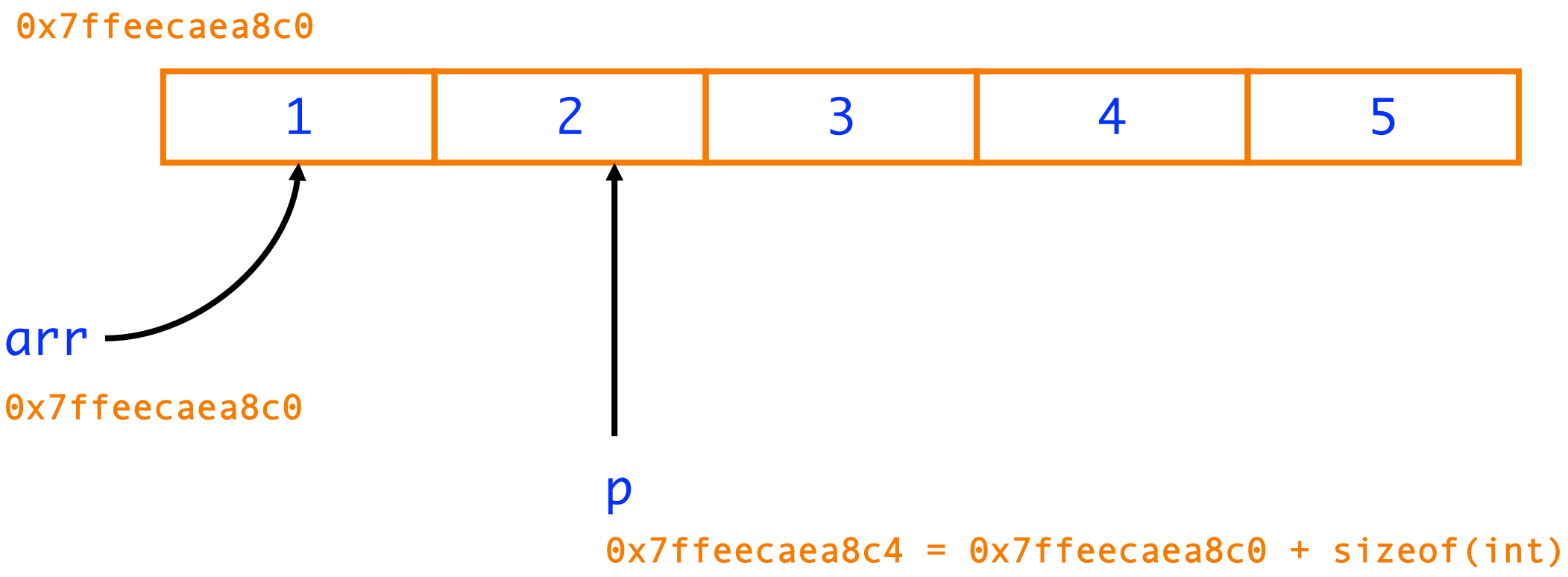
4

(assume 4 byte integers)

Pointer arithmetic

Add/subtract integers to/from pointers (assume 4 byte integers)

```
int *p = arr;
p++;
```



Note:

`sizeof()` is NOT a function

- takes a type name as an argument

Size of pointer

- On a 64 bit machine, size of all types of **pointer**, be it `int*`, `float*`, `chart*`, `double*` is always **8 bytes**.
- When performing arithmetic function, e.g. increment on a pointer, changes occur as per the size of their primitive data type.

Size of pointer

long unsigned decimal integer

```
printf("sizeof(int) is %lu\n", sizeof(int));
printf("sizeof(char) is %lu\n", sizeof(char));
printf("sizeof(float) is %lu\n", sizeof(float));
printf("sizeof(double) is %lu\n", sizeof(double));
Printf("==========");
printf("sizeof(int*) is %lu\n", sizeof(int*));
printf("sizeof(char*) is %lu\n", sizeof(char*));
printf("sizeof(float*) is %lu\n", sizeof(float*));
printf("sizeof(double*) is %lu\n", sizeof(double*));
```

Outpout

```
sizeof(int) is 4
sizeof(char) is 1
sizeof(float) is 4
sizeof(double) is 8
==========
sizeof(int*) is 8
sizeof(char*) is 8
sizeof(float*) is 8
sizeof(double*) is 8
```

Size of pointer

```
int* p1;
printf("%p\n", p1);
p1++;
printf("%p\n", p1);
```

```
0x7ffee46608f0
0x7ffee46608f4
```

```
char* p2;
printf("%p\n", p2);
p2++;
printf("%p\n", p2);
```

```
0x7ffee240c8f0
0x7ffee240c8f1
```

```
double* p3;
printf("%p\n", p3);
p3++;
printf("%p\n", p3);
```

```
0x7ffeebfe08f0
0x7ffeebfe08f8
```

Pointer with Functions

Pointers as function arguments

- Pointer **as** a function parameter is used to hold addresses of arguments passed during function call, known as **call-by-reference**.
- When a function is called by reference any change made to the reference variable will effect the original variable.

Example 3

```c
#include <stdio.h>
void swap(int *a, int *b);
int main() {
    int m = 66, n = 99;
    printf("m = %d\n", m);
    printf("n = %d\n\n", n);
    swap(&m, &n);
    printf("After swapping:\n\n");
    printf("m = %d\n", m);
    printf("n = %d\n", n);
    return 0;
}
```

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

Example 3

```c
#include <stdio.h>
void swap(int *a, int *b);
int main() {
    int m = 66, n = 99;
    printf("m = %d\n", m);
    printf("n = %d\n\n", n);
    swap(&m, &n);
    printf("After swapping:\n\n");
    printf("m = %d\n", m);
    printf("n = %d\n", n);
    return 0;
}
```

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

Outpout

```
m = 66
n = 99

After Swapping:

m = 99
n = 66
```

Functions returning pointer variables

- A function can **return** a pointer to the calling function.
- **Be careful!**
  - Local variables of function doesn’t live outside of the function
  - If returning a pointer pointing to a local variable, that pointer will be pointing to nothing when the function ends.

Example 4

```c
#include <stdio.h>
int* larger(int *a, int *b);
int main() {
    int m = 66, n = 99;
    int *p;

    p = larger(&m, &n);

    printf("%d is larger.\n",*p);

    return 0;
}
```

```c
int* larger(int *a, int *b) {
    if(*a > *b)
        return a;
    else
        return b;
}
```

return the pointer

Example 4

```c
#include <stdio.h>
int* larger(int *a, int *b);
int main() {
    int m = 66, n = 99;
    int *p;

    p = larger(&m, &n);

    printf("%d is larger.\n",*p);

    return 0;
}
```

```c
int* larger(int *a, int *b) {
    if(*a > *b)
        return a;
    else
        return b;
}
```

Outpout

```
99 is larger.
```

Pointer to functions

- A pointer pointing to a function can be used as an argument in another function.
  - to declare a pointer to a function:

    ```
    type (*pointer-name)(parameter);
    ```

  - an example

    ```
    int (*sum)();
    ```

Pointer to functions

- A function pointer can point to a specific function when it is assigned the name of that function

```
int sum(int, int);
int (*s)(int, int);
s = sum;
```

  - `s` is a pointer to a function sum. Now sum can be called using **function pointer** `s` with required argument values.

```
s(10, 20);
```

Example 5

```c
#include <stdio.h>
int sum(int x, int y) {
    return x + y;
}
int main() {
    int (*fp)(int, int);
    fp = &sum;
    printf("Sum is %d.\n", (*fp)(6, 9));
    return 0;
}
```

Outpout

```
Sum is 165.
```

Example 5

```c
#include <stdio.h>
int sum(int x, int y) {
    return x + y;
}
int main() {
    int (*fp)(int, int);
    fp = &sum;
    printf("Sum is %d.\n", (*fp)(6, 9));
    return 0;
}
```


Outpout

```
Sum is 165.
```

Example 5

```c
#include <stdio.h>
int sum(int x, int y) {
    return x + y;
}
int main() {
    int (*fp)(int, int);
    fp = &sum;
    printf("Sum is %d.\n", (*fp)(6, 9));
    return 0;
}
```

fp = ∑ → fp = sum;

(*fp)(6, 9); → fp(6, 9);

Outpout

```
Sum is 165.
```

Example 6 Passing the pointer to another function

```
#include <stdio.h>
int sum(int x, int y) {
    return x + y;
}
int sum6_9(int (*fp)(int,int)){
     return (*fp)(6, 9);
}
int main(){
    int (*fp)(int, int);
    fp = sum;
    printf("Sum is %d.\n", sum6_9(fp));
    return 0;
```

Example 7 Using function pointers in return values

```
#include <stdio.h>
int sum(int x, int y) {
    return x + y;
}
int (*functionFactory(int z))(int, int) {
    printf("Got parameter %d.\n", z);
    int (*fp)(int,int) = sum;
    return fp;
}
int main() {
    printf("Sum is %d.\n", functionFactory(3)(6,9));
    return 0;
```

Summary

Today

- Pointer to Array
- Pointer Arithmetic
- Pointer with Functions

Next

- Structure
- Union
- Typedef
- String