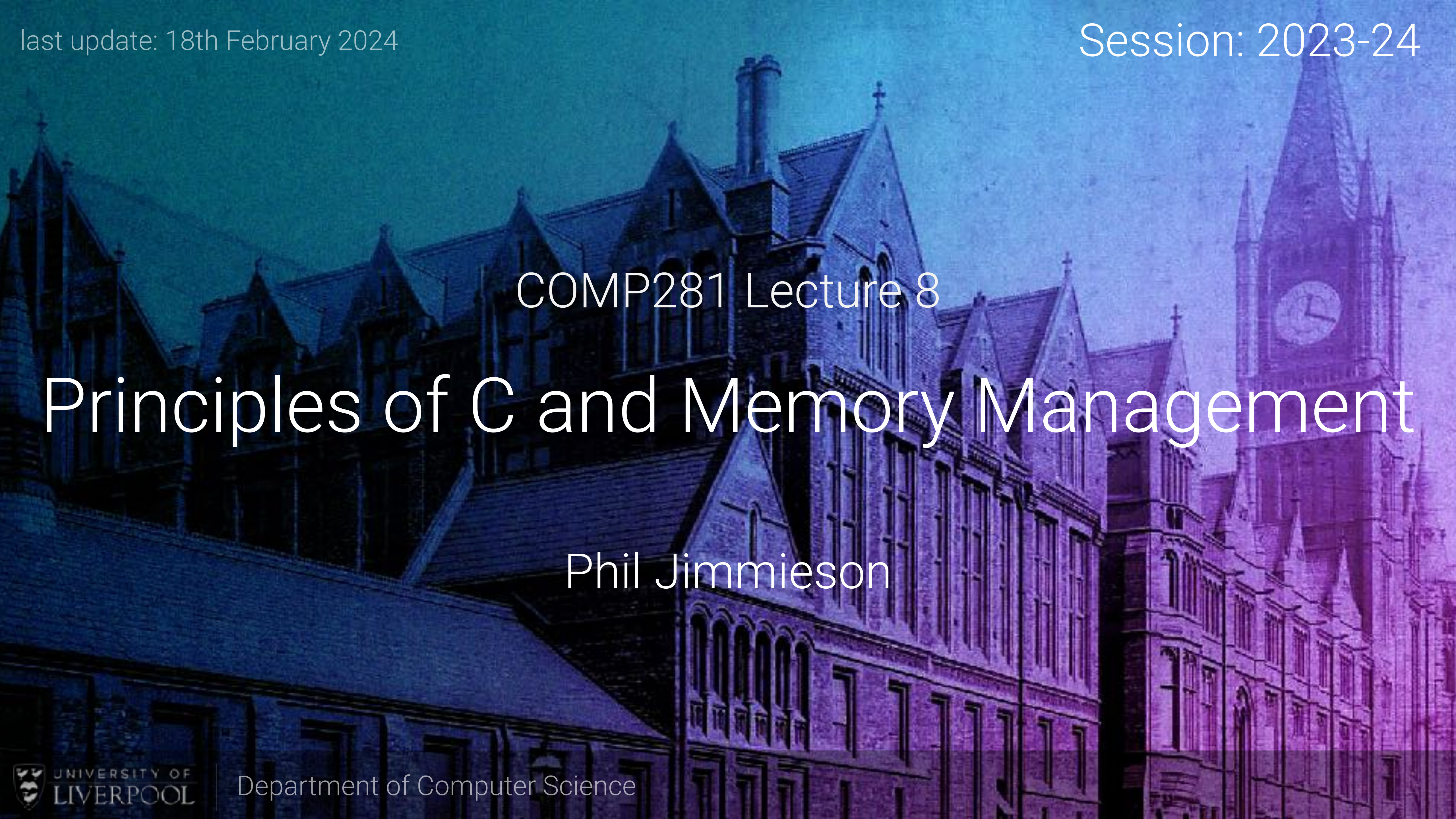


last update: 18th February 2024

Session: 2023-24

COMP281 Lecture 8

Principles of C and Memory Management

Phil Jimmieson

Last Lecture

- Pointer Basics

Previous Lectures

- Arrays, Arithmetic, Functions

Last Lecture

- Pointer Basics

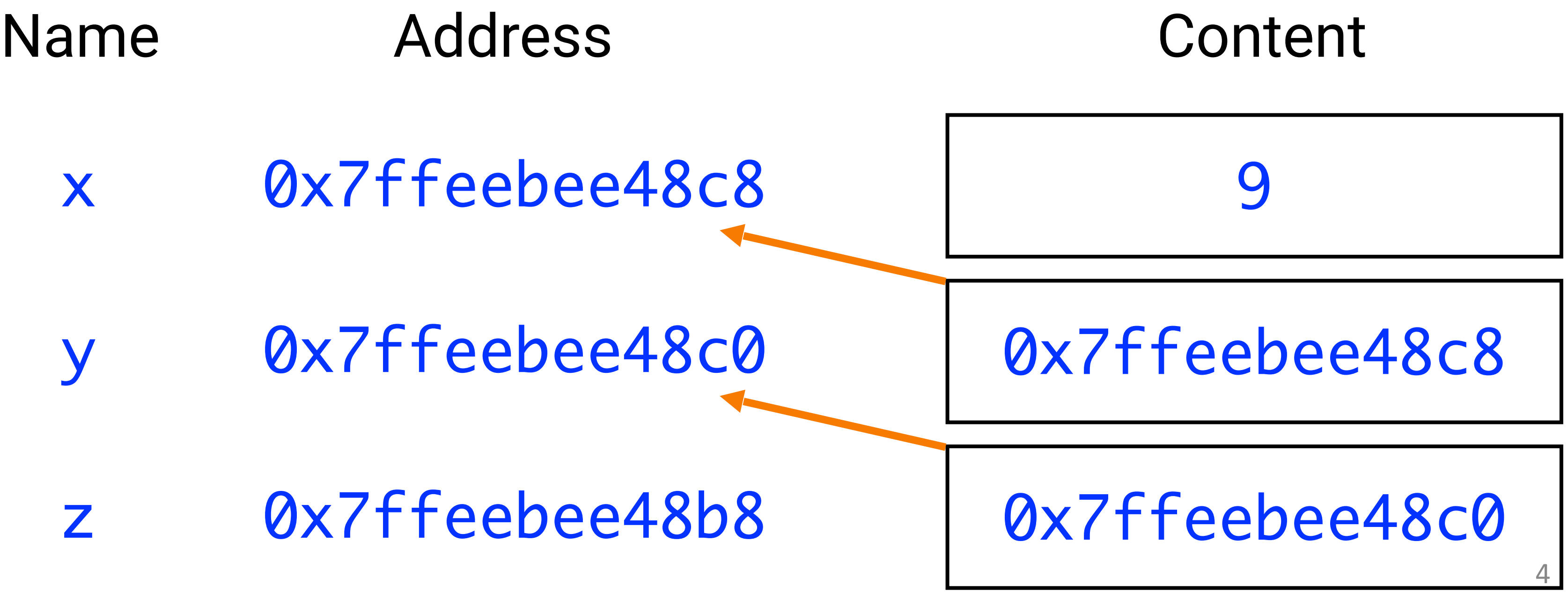
Previous Lectures

- Arrays, Arithmetic, Functions

Recap

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

```
int    x =  9;

int   *y = &x;

int  **z = &y;
```



Name	Address	Content
x	0x7ffeebee48c8	9
y	0x7ffeebee48c0	0x7ffeebee48c8
z	0x7ffeebee48b8	0x7ffeebee48c0

- 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

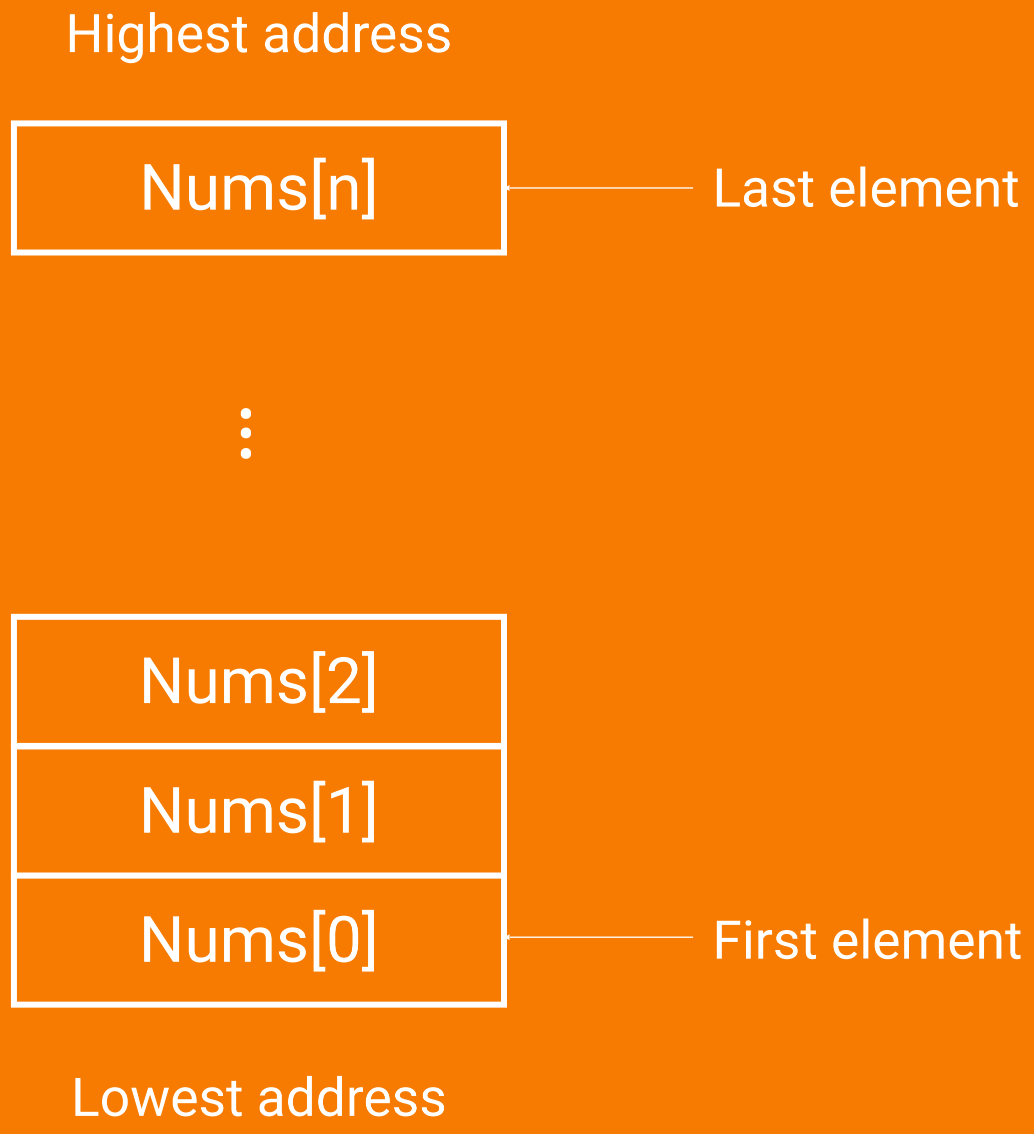
- Arrays, Arithmetic, Functions

Today

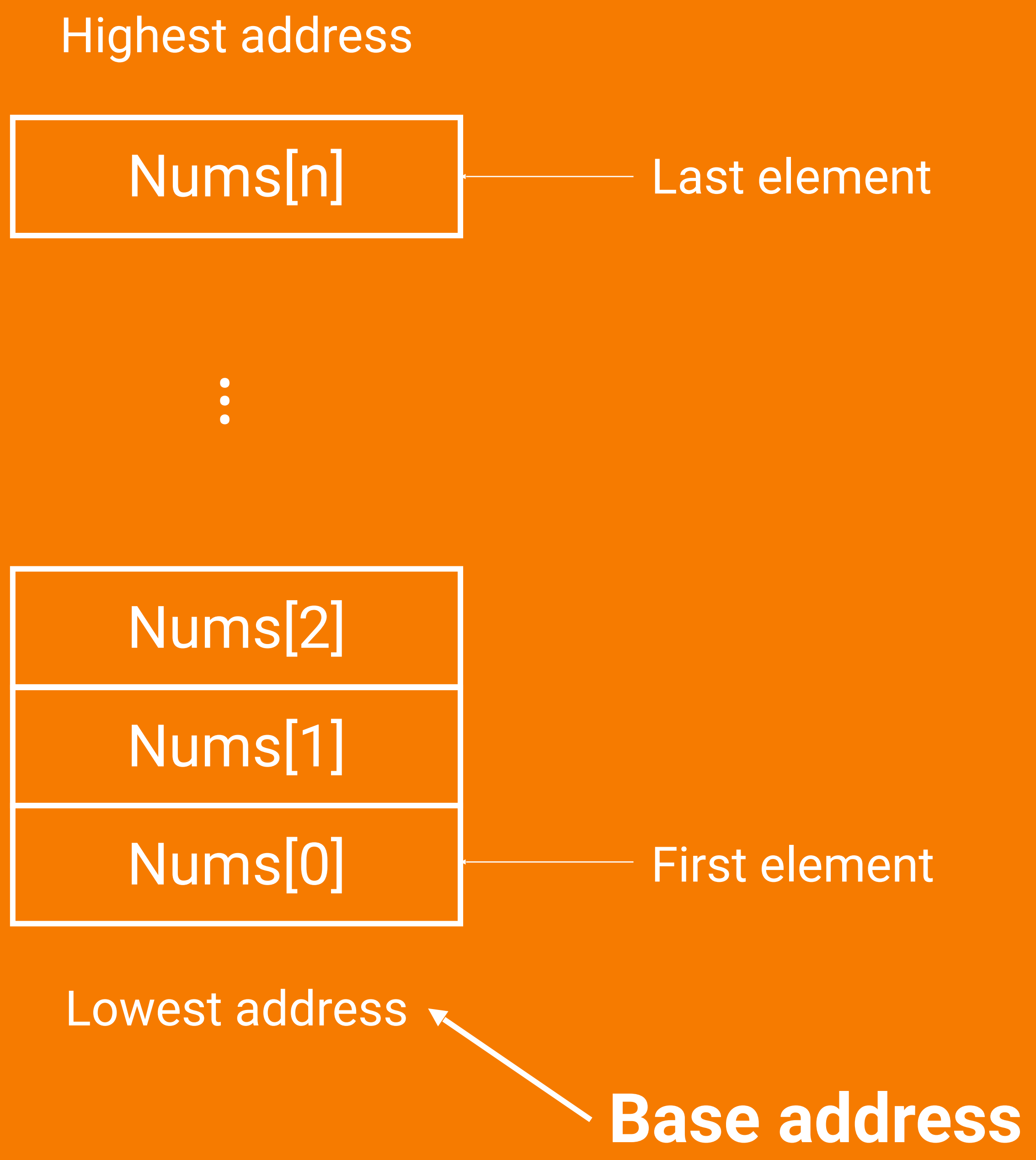
- Pointer to Array
- Pointer Arithmetic
- Pointer with Functions

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 a sufficient amount of memory to contain all the elements of the array.
- The *base address* i.e. address of the first element of the array is also allocated by the Compiler



Highest address

Nums[n]

Last element

⋮

Nums[2]

Nums[1]

Nums[0]

First element

Lowest address

Base address

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

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**

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 purposes

- 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

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

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

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

   return 0;
}
```

Example 1

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

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

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

    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

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

Example 2

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

Output

```
1
2
3
4
5
```

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

Example 2

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

What if we replace

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

with

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

  printf("%d\n", i[arr]);

  printf("%d\n", arr+i);

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

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

Example 2

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

What if we replace

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

with

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

Prints all array elements

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

Also prints all elements of array

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

Prints address of array elements

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

Prints value of array elements

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

Prints value of a[0] only

Example 2

What if we replace

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

with

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

Prints all array elements

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

Also prints all elements of array

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

Prints address of array elements

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

Prints value of array elements

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

Prints value of a[0] only

`-Wall` flag gives a warning:

```
warning: format specifies type 'int' but the argument has type 'int *'
```

Example 2

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

What about

```
arr++;
```

?

Example 2

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

What about

`arr++;`

?

Compile-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]`

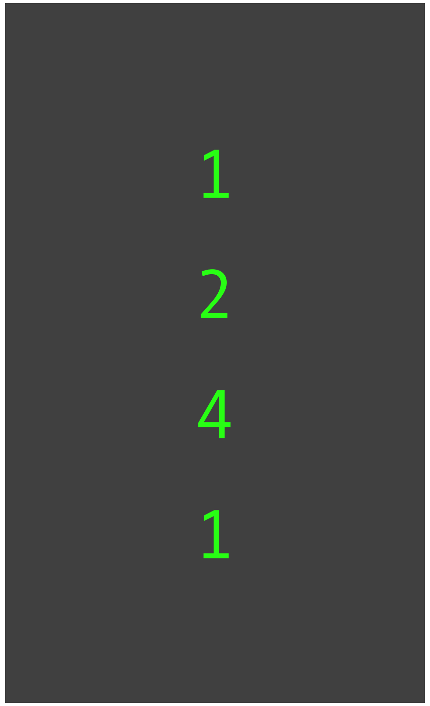
Pointer Arithmetic

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) == ? */
```



```
1
2
4
1
```

Pointer arithmetic

Add/subtract integers to/from pointers

(assume 4 byte integers)

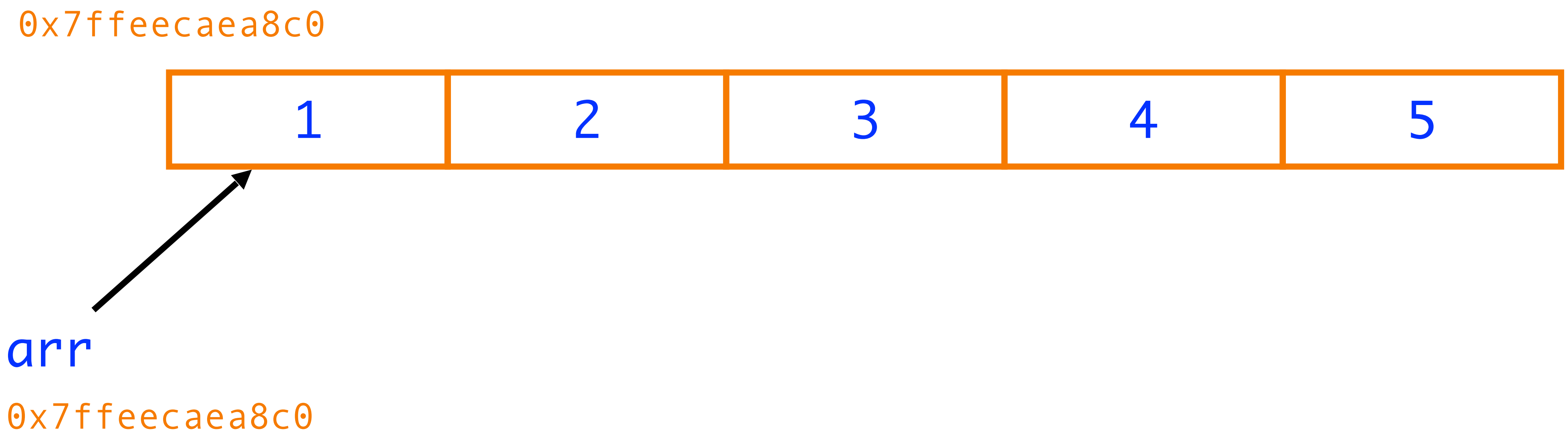
0x7ffeecaea8c0

Pointer arithmetic

Add/subtract integers to/from pointers

(assume 4 byte integers)

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



0x7ffeecaea8c0

1	2	3	4	5

arr

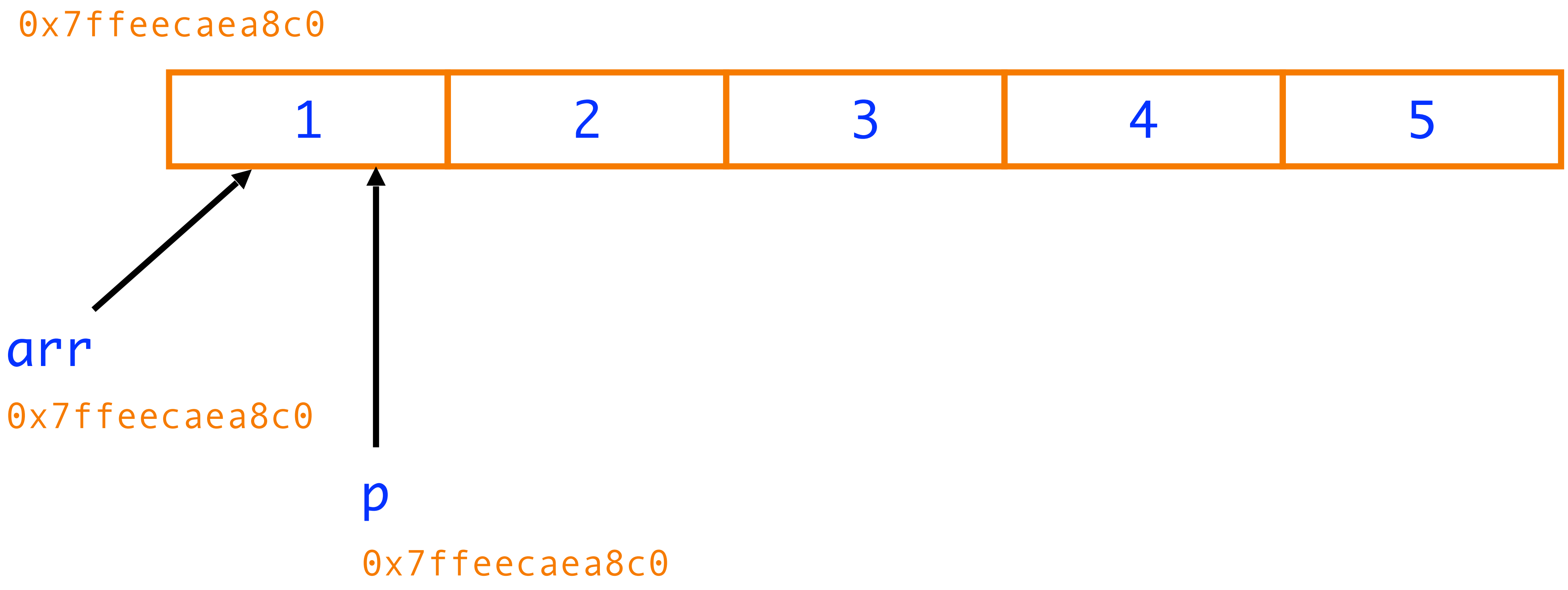
0x7ffeecaea8c0

Pointer arithmetic

Add/subtract integers to/from pointers

(assume 4 byte integers)

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

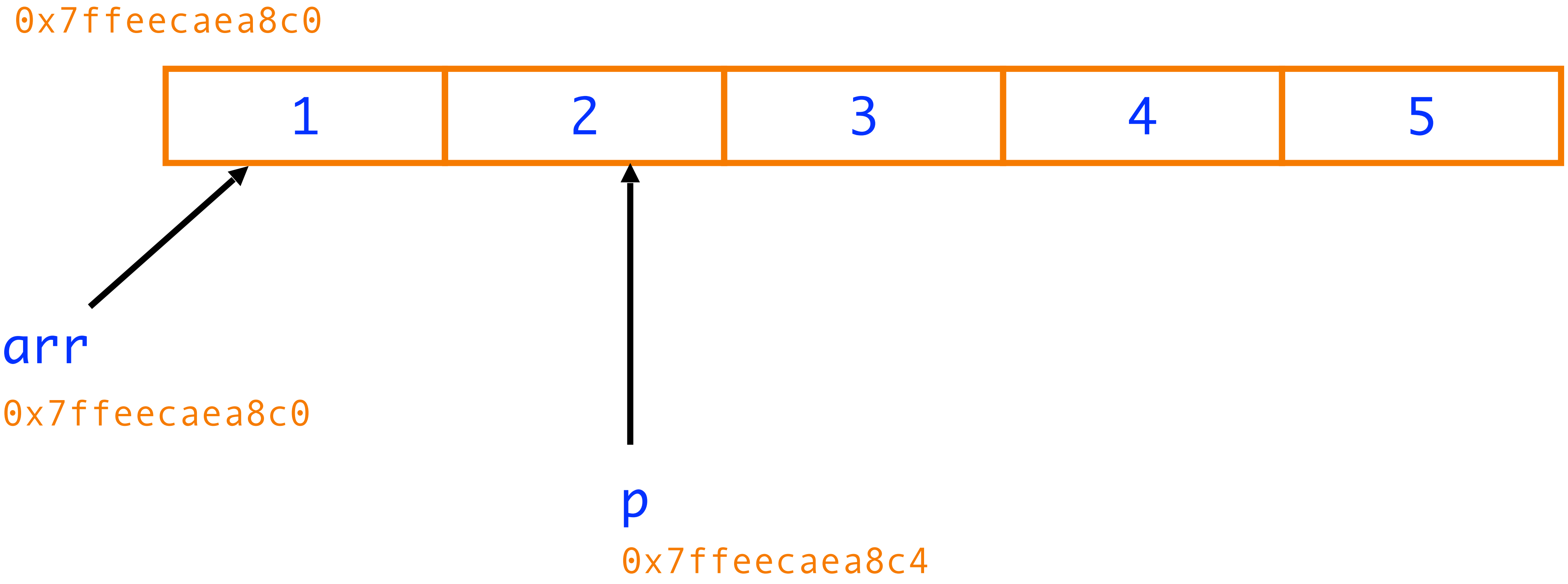


Pointer arithmetic

Add/subtract integers to/from pointers

(assume 4 byte integers)

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

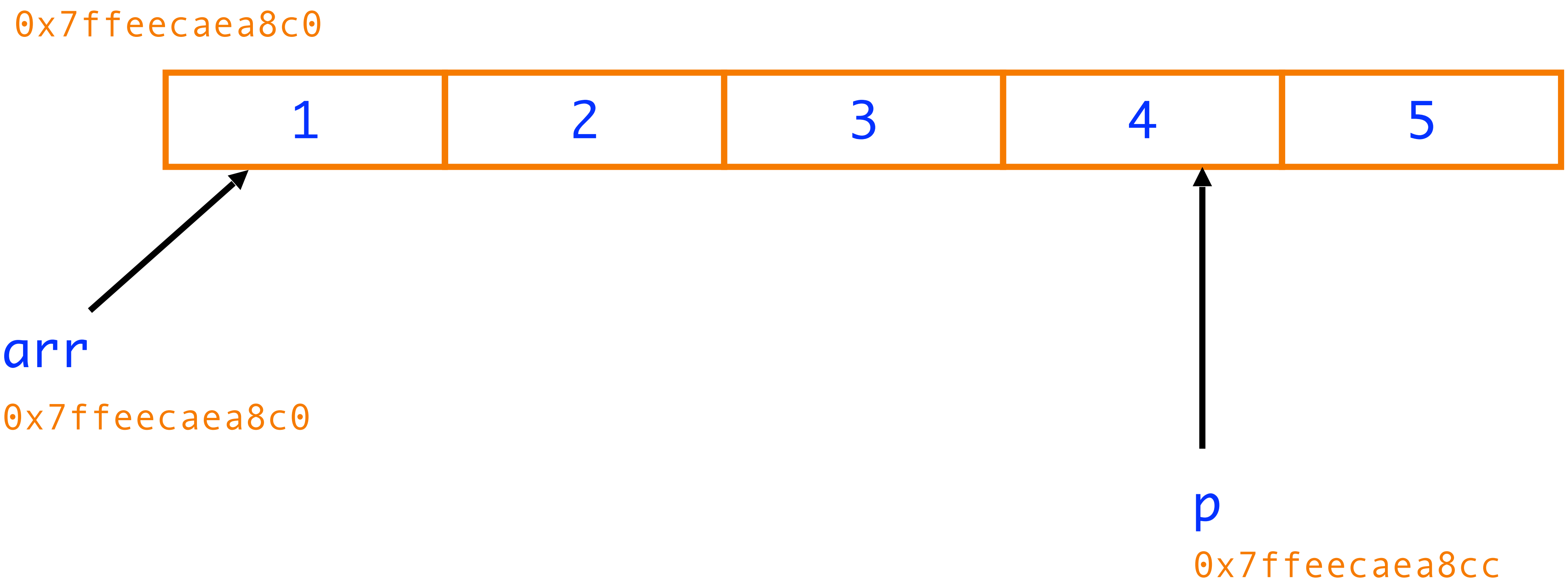


Pointer arithmetic

Add/subtract integers to/from pointers

(assume 4 byte integers)

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

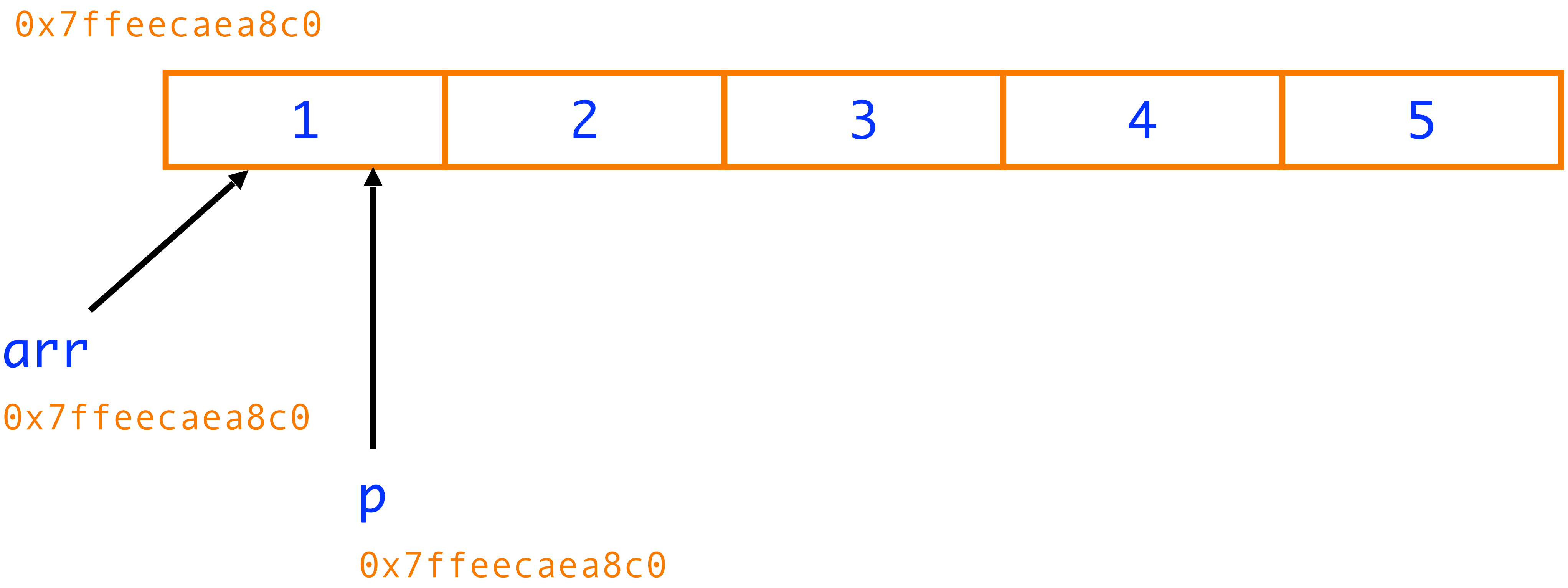


Pointer arithmetic

Add/subtract integers to/from pointers

(assume 4 byte integers)

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



0x7ffeecaea8c0

1	2	3	4	5

arr

0x7ffeecaea8c0

p

0x7ffeecaea8c0

Note:

Pointer arithmetic does NOT add/subtract the address directly, but in multiples of the size of the type in bytes.

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

4

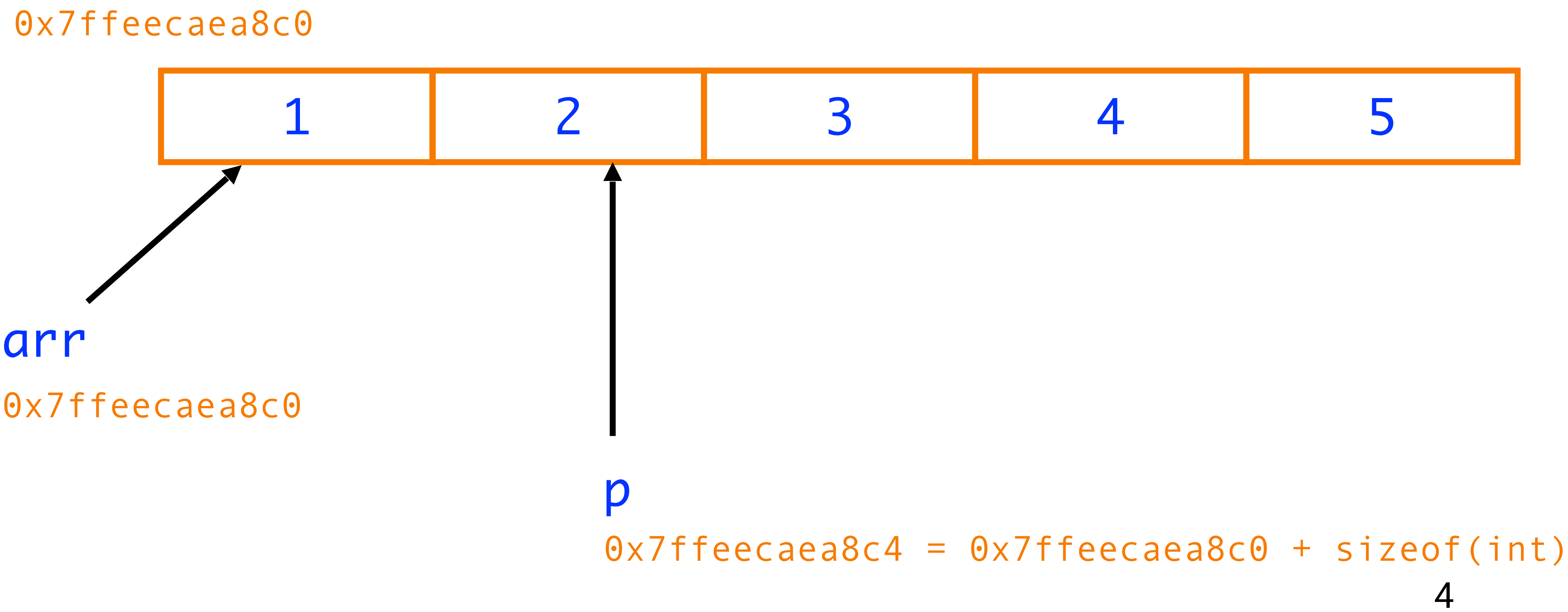
(assume 4 byte integers)

Pointer arithmetic

Add/subtract integers to/from pointers

(assume 4 byte integers)

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



0x7ffeecaea8c0

1 2 3 4 5

arr

0x7ffeecaea8c0

p

0x7ffeecaea8c4 = 0x7ffeecaea8c0 + sizeof(int)

4

Note:

`sizeof()` is NOT a function

- takes a type name as an argument

Size of pointer

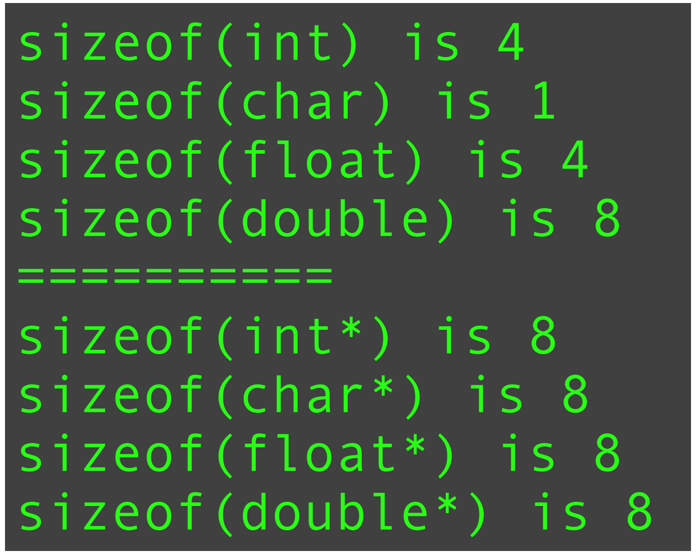
- On a 64 bit machine, the size of all types of **pointer**, be it `int*`, `float*`, `char*`, `double*` is always **8 bytes**.
- When performing arithmetic functions, e.g. increment on a pointer, changes occur as per the size of their primitive data type.

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*));
```

Output



```
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
```

4 bytes

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

```
0x7ffee240c8f0
0x7ffee240c8f1
```

1 byte

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

```
0x7ffeebfe08f0
0x7ffeebfe08f8
```

8 bytes

Pointer with Functions

Pointers as function arguments

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

Example 3

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

Output

```
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 a function don't have a "life" outside of the function
  - If you return a pointer, pointing to a local variable declared within that function, the pointer will be pointing to ???? when the function ends.

Example 4

```c
#include <stdio.h>
int* larger(int *a, int *b);
int main(void) {
   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;
}
```

Example 4

```c
#include <stdio.h>
int* larger(int *a, int *b);
int main(void) {
   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;
}
```

Output

```
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)(); //sum is a pointer to func with no params
                  //but which returns an int
    ```

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);//declared a func with 2 int params
  int (*s)(int, int);
  s = sum;
  ```

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

    ```
    s(10, 20);/* equivalent to the call sum(10,20) */
    ```

Example 5

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

Output

```
Sum is 15.
```

Example 5

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

```c
fp = sum;
```

Output

```
Sum is 15.
```

Example 5

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

Output

```
Sum is 15.
```

Example 6 Passing the pointer to another function

```
#include <stdio.h>
int sum(int x, int y) {
   return x + y;
}
int sum6_9(int (*fp2)(int,int)){
    return (*fp2)(6, 9);  →  return fp2(6, 9);
}
int main(void){
   int (*fp)(int, int);
   fp = sum;
   printf("Sum is %d.\n", sum6_9(fp));
   return 0;
}
```

Example 6 Passing the pointer to another function

```
#include <stdio.h>
int sum(int x, int y) {
   return x + y;
}
int sum6_9(int (*fp2)(int,int)){
    return fp2(6, 9);
}
int main(){
   int (*fp)(int, int);
   fp = sum;
   printf("Sum is %d.\n", sum6_9(fp));
   return 0;
}
```

Example 6 Passing the pointer to another function

```
#include <stdio.h>
int sum(int x, int y) {
   return x + y;
}
int sum6_9(int (*fp2)(int,int)){
    return fp2(6, 9);
}
int main(){
   int (*fp)(int, int);
   fp = sum;
   printf("Sum is %d.\n", sum6_9(fp));
   return 0;
}
```

Example 6 Passing the pointer to another function

```
#include <stdio.h>
int sum(int x, int y) {
   return x + y;
}
int sum6_9(int (*fp2)(int,int)){
    return fp2(6, 9);
}
int main(){
   int (*fp)(int, int);
   fp = sum;
   printf("Sum is %d.\n", sum6_9(fp));
   return 0;
}
```

Example 6 Passing the pointer to another function

```
#include <stdio.h>
int sum(int x, int y) {
   return x + y;
}
int sum6_9(int (*fp2)(int,int)){
    return fp2(6, 9);
}
int main(){
   int (*fp)(int, int);
   fp = sum;
   printf("Sum is %d.\n", sum6_9(fp));
   return 0;
}
```

Example 6 Passing the pointer to another function

```
#include <stdio.h>
int sum(int x, int y) {
   return x + y;
}
int sum6_9(int (*fp2)(int,int)){
    return fp2(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;
}
```

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;
}
```

Example 7 Using function pointers in return values

```c
#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;
}
```

Example 7 Using function pointers in return values

```c
#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;
}
```

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;
}
```

Example 7 Using function pointers in return values

```c
#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;
}
```

Output

```
Got parameter 3.
Sum is 15.
```

Summary

Today

- Pointer to Array
- Pointer Arithmetic
- Pointer with Functions

Next

- Structures
- Union
- Typedef
- String