

Review – True/False

1. The size of a pointer variable depends on the data type it points to (for example, an `int*` is larger than a `char*`) False
2. Dereferencing a NULL pointer is safe and simply returns 0. False
3. When you perform pointer arithmetic, such as `ptr++`, the memory address increases by the size of the data type the pointer points to. True
4. An uninitialized pointer contains a 'garbage' value that can point to arbitrary locations in memory True
5. In C, the name of an array can be used as a pointer to its first element. True

Review – True/False

1. **FALSE.** The size of a pointer depends on the system architecture (e.g., 32-bit vs. 64-bit), not the type of data it points to. A pointer simply stores a memory address. Whether that address points to a single character or a massive structure, the address itself takes up the same amount of space in memory.
2. **FALSE.** Attempting to dereference a NULL pointer (e.g., `int x = *ptr;` where `ptr` is NULL) is a critical error. It will almost always cause your program to crash with a Segmentation Fault (or undefined behaviour), because you are trying to access a memory location that you do not have permission to use.
3. **TRUE.** If `ptr` is an `int*` and if an integer is 4 bytes, `ptr++` actually adds 4 to the memory address. This ensures that the pointer always lands exactly at the start of the *next* element in an array.
4. **TRUE.** As we discussed earlier, if you declare `int *ptr;` without assigning it an address, it contains "rubbish" (whatever random bits were left in that memory slot). If you try to use it, you are accessing a random, "arbitrary" location, which is highly dangerous.
5. **TRUE.** In most contexts, the name of an array (e.g., `myArray`) acts as a constant pointer to the address of the first element (`&myArray[0]`). This is why you can pass an array to a function by simply passing its name.

Review

1. What is the result of this code? Explain your answer

```
int main() {  
    int arr[] = {10, 20, 30, 40};  
    int *p = arr;  
    p++;  
    printf("%d", *p);  
    return 0;  
}
```

In C, the name of an array (arr) acts as a pointer to its first element (arr[0]).

So, p initially points to 10. When p++ is executed, the pointer moves to the next integer address in the array. Therefore, p now points to 20.

Review

2. What is the result of this code? Explain your answer

```
int main() {  
    int a = 10, b = 20;  
    int *p = &a;  
    int *q = &b;  
    *p = *q;  
    printf("%d, %d", a, b);  
    return 0;  
}
```

`*p = *q` means "Take the value that `q` points to (which is 20) and assign it to the location that `p` points to (which is `a`)."

After this, **a** becomes 20.

Crucially, **b** was never changed; it remains 20.

Review

3. What is the result of this code? Explain your answer

```
int main() {  
    int val = 15;  
    int *ptr = &val;  
    *ptr = *ptr + 5;  
  
    printf("%d", val);  
    return 0;  
}
```

ptr points to val.

The line `*ptr = *ptr + 5` dereferences the pointer to get the value of val (15), adds 5 to it (20), and stores that new value back into the memory location of val.

Therefore, val is now 20.

Review

4.a. What is the result of the code? Explain your answer

```
#include <stdio.h>

int main() {
    int a[] = {10, 20, 30, 40, 50};
    int len = sizeof(a) / sizeof(int);
    int sum = 0;

    for (int *ip = a; ip < a + len; ip++)
    {
        *ip = *ip + 5;      // Line 1
        sum = sum + *ip;    // Line 2
    }

    printf("%d, %d", a[2], sum);
    return 0;
}
```

Loop Execution:

- Iteration 1: ip points to a[0] (10).*
*ip = 10 + 5 -> a[0] becomes 15.
sum = 0 + 15 -> sum becomes 15.
- Iteration 2: ip points to a[1] (20).*
*ip = 20 + 5 -> a[1] becomes 25.
sum = 15 + 25 -> sum becomes 40.
- Iteration 3: ip points to a[2] (30).*
*ip = 30 + 5 -> a[2] becomes 35.
sum = 40 + 35 -> sum becomes 75.
- Iteration 4: ip points to a[3] (40).*
*ip = 40 + 5 -> a[3] becomes 45.
sum = 75 + 45 -> sum becomes 120.
- Iteration 5: ip points to a[4] (50).*
*ip = 50 + 5 -> a[4] becomes 55.
sum = 120 + 55 -> sum becomes 175.

Review

4.a. What is the result of the code? Explain your answer

```
#include <stdio.h>

int main() {
    int a[] = {10, 20, 30, 40, 50};
    int len = sizeof(a) / sizeof(int);
    int sum = 0;

    for (int *ip = a; ip < a + len; ip++) {
        *ip = *ip + 5;    // Line 1
        sum = sum + *ip;  // Line 2
    }

    printf("%d, %d", a[2], sum);
    return 0;
}
```

Termination:

ip moves to a + 5.

The condition $ip < a + 5$ is now False.

Final Print:

- a[2] was modified to 35.
- sum is 175.
- Result: **35, 175.**

Review

4.b. Why is the condition `ip < a + len` used instead of `ip <= a + len`?

```
for (int *ip = a; ip < a + len; ip++) {...}
```

In C, if an array has `len` elements, the valid indices are 0 to `len - 1`.

- `a` is the address of `a[0]`.
- `a + len` is the address of `a[len]`.

Note: `a[len]` is out of bounds. It is the "empty space" right after the array ends.

- If you used `ip <= a + len`, the loop would run one extra time, trying to read or write to `a[len]`. This is a Buffer Overflow error and could cause a crash or "rubbish" data (the concept we discussed last week).

Review

4.b. Why is the condition `ip < a + len` used instead of `ip <= a + len`?

```
for (int *ip = a; ip < a + len; ip++) {...}
```

In C, if an array has `len` elements, the valid indices are 0 to `len - 1`.

- `a` is the address of `a[0]`.
- `a + len` is the address of `a[len]`.

Note: `a[len]` is out of bounds. It is the "empty space" right after the array ends.

- If you used `ip <= a + len`, the loop would run one extra time, trying to read or write to `a[len]`. This is a Buffer Overflow error and could cause a crash or "rubbish" data (the concept we discussed last week).

Review

5. Assume that an int occupies 4 bytes of memory

```
int x[5];
int *sp;
int **spp;

for (int i = 0; i < 5; i++) {
    x[i] = 100 - (i * 10);
}

sp = x;
spp = &sp;
```

1. What is the value of *sp?
2. What is the value of *(sp + 2)?
3. What is the value of **spp?
4. What is the value of *(*spp + 4)?

i=0: $100 - 0 = 100$

i=1: $100 - 10 = 90$

i=2: $100 - 20 = 80$

i=3: $100 - 30 = 70$

i=4: $100 - 40 = 60$

Array x is: {100, 90, 80, 70, 60}

1. 100

sp points to x[0]. *sp retrieves the value at x[0].

2. 80

sp + 2 points to x[2]. The value at x[2] is 80.

3. 100

spp points to sp. *spp is the address of x[0].
**spp is the value at x[0].

4. 60

*spp is sp. So *spp + 4 is sp + 4. This points to x[4].
The value at x[4] is 60.

Review

6. What is the result? Trace the execution of the code carefully. Assume standard C compiler behavior.

```
#include <stdio.h>
void processArray(int *arr, int n, int factor) {
    for (int i = 0; i < n; i++) {
        if (i % 2 == 0) {
            *(arr + i) += factor; // Add factor to even indices
        } else {
            *(arr + i) -= factor; // Subtract factor from odd indices
        }
    }
}

int main() {
    int myData[] = {10, 20, 30, 40, 50};
    int size = 5;
    int k = 5;
    processArray(myData, size, k);
    printf("%d, %d, %d", myData[0], myData[2], myData[4]);

    return 0;
}
```

Review

6. Trace the execution of the code carefully. Assume standard C compiler behavior.

15, 35, 55

Step-by-Step

1. Initial State in main:

`myData = {10, 20, 30, 40, 50}`

`size = 5`

`k = 5`

2. Function Call: The function `processArray` is called. The array `myData` is passed by its address. Crucially, `arr` inside the function points to the exact same memory locations as `myData` in main.

3. Inside the for loop in `processArray`:

- Iteration 1 (`i = 0`): 0 is even.
 - `*(arr + 0)` is `myData[0]`.
 - $10 + 5 = 15$.
 - `myData[0]` is now 15.
- Iteration 2 (`i = 1`): 1 is odd.
 - `*(arr + 1)` is `myData[1]`.
 - $20 - 5$