CS 262 Lecture 5: Arrays Part 1

GEORGE MASON UNIVERSITY

Overview of Lecture 5

Arrays

Introduction to what arrays are in C

Comparing C arrays with other familiar structures from Python/Java

Initialization, etc

Weird behavior

Basic array ‘operations’

Array Basics

An **array** is a collection of elements stored in a contiguous memory location, all of the same data type

Arrays

- Have a size that is fixed a declaration
- Can only store elements of the same type
- Have no built-in operations (must manually write searching, sorting, and so on)
- All arrays begin with index 0

C Arrays vs Python Lists vs Java Arrays

How do C arrays compare with structures we already know?

Feature	C Array	Java Array	Python List
Resizeability	Fixed	Fixed	Dynamic
Memory Management	Manual	Automatic (Garbage collection)	Automatic
Bounds Checking	No	Yes	Yes
Default Initialization	No (Garbage values)	Yes (0/null)	Yes (None)
Operations	None built-in	Some built-in	Many
Performance	Fastest	Moderate	Slowest

Declaring an Array

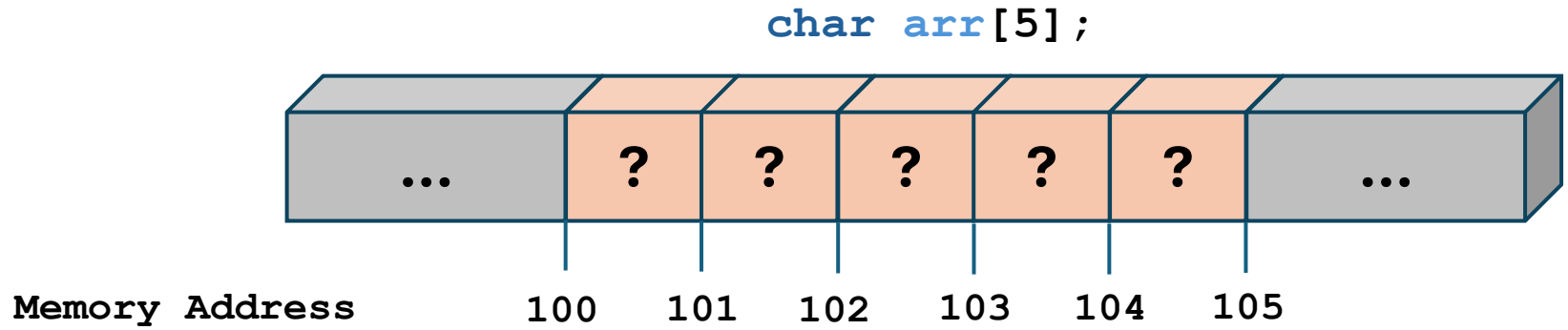
Syntax for declaring an array

```
type name[num_elements];
```

So if I want a character array of size 5 ..

```
char arr[5];
```

And let's see how this array look in memory



Initialization

Option 1

Initialization with specified size

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

Option 2

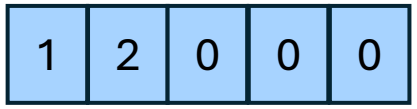
Initialization without size

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

Option 3

Partial Initialization

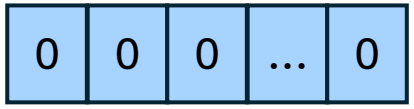
```
int arr[5] = {1, 2};
```



Option 4

Initialize to all 0s

```
int arr[100] = {0};
```



Array of ints

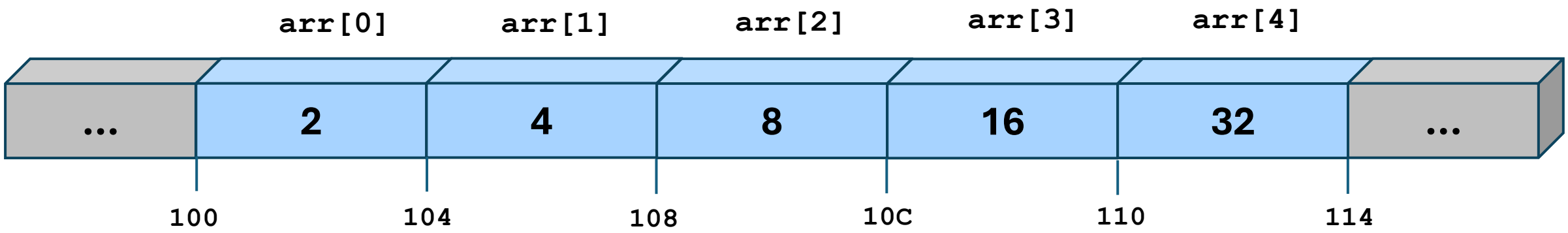
Syntax for declaring an array

```
type name[num_elements] = {var, var, ..., var};
```

So if I want a character array of size 5 ..

```
int arr[5] = {2, 4, 8, 16, 32};
```

And let's see how this array looks in memory

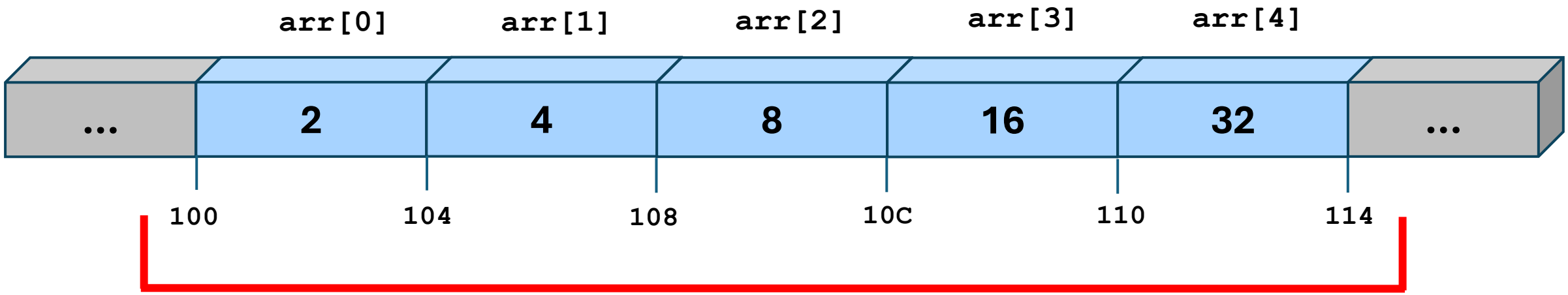


Size in Memory

sizeof() Function

Just like we used sizeof() on variable types to learn their size (in bytes), we can do the same for an array

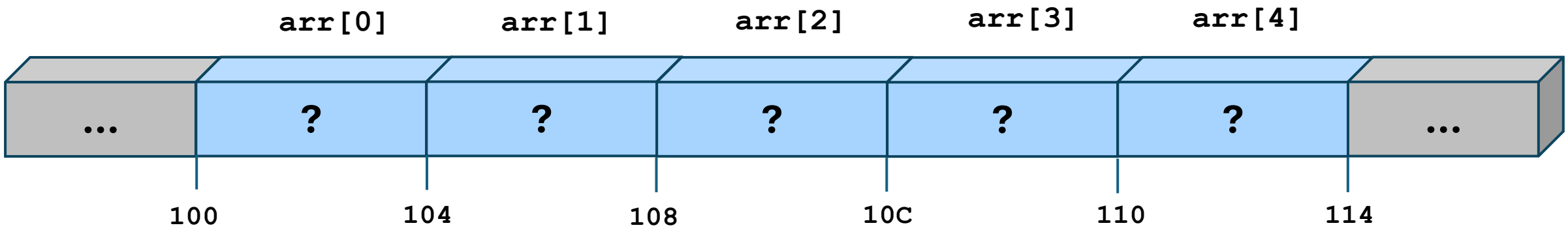
```
int arr[5] = {2, 4, 8, 16, 32};
printf("arr is %d bytes", sizeof(arr));
```



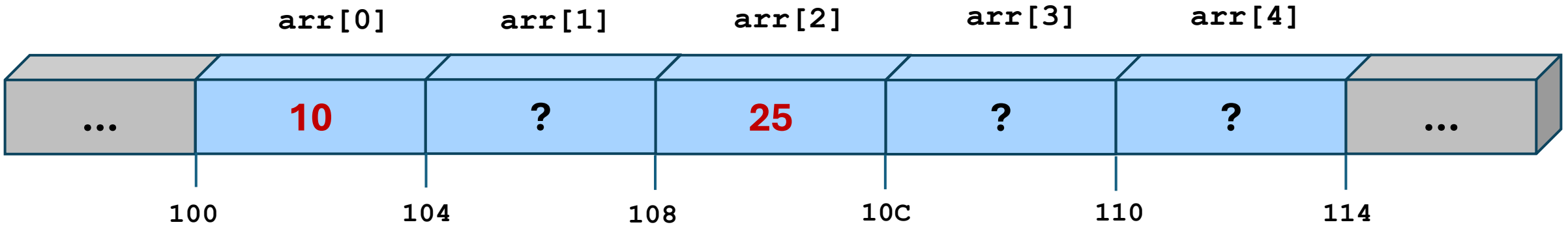
4 bytes * 5 elements = 20 bytes

Modifying and Accessing Elements

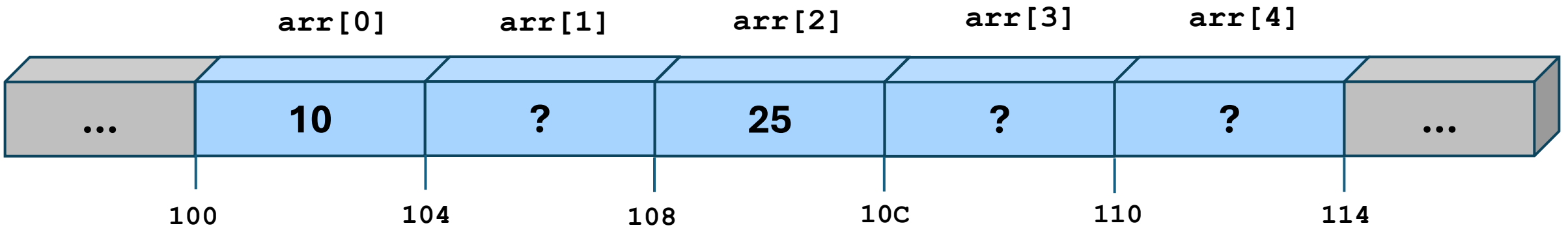
```
int arr[5];
```



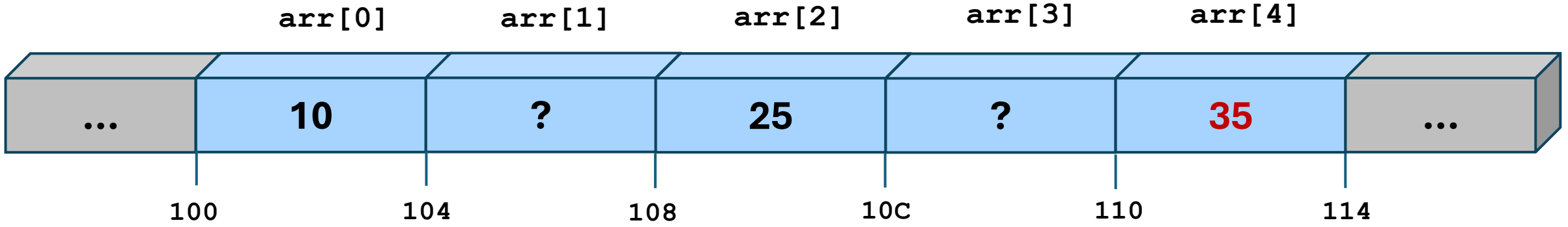

```
arr[0] = 10;
arr[2] = 25;
```



Modifying and Accessing Elements




```
arr[4] = arr[0] + arr[2]
```

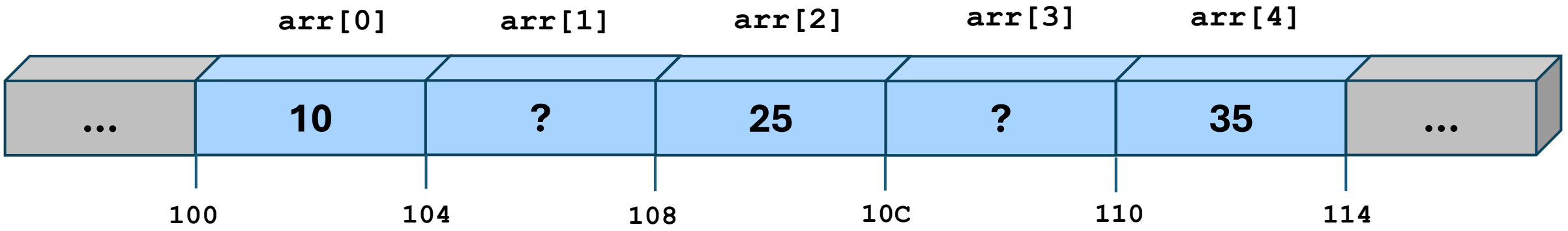


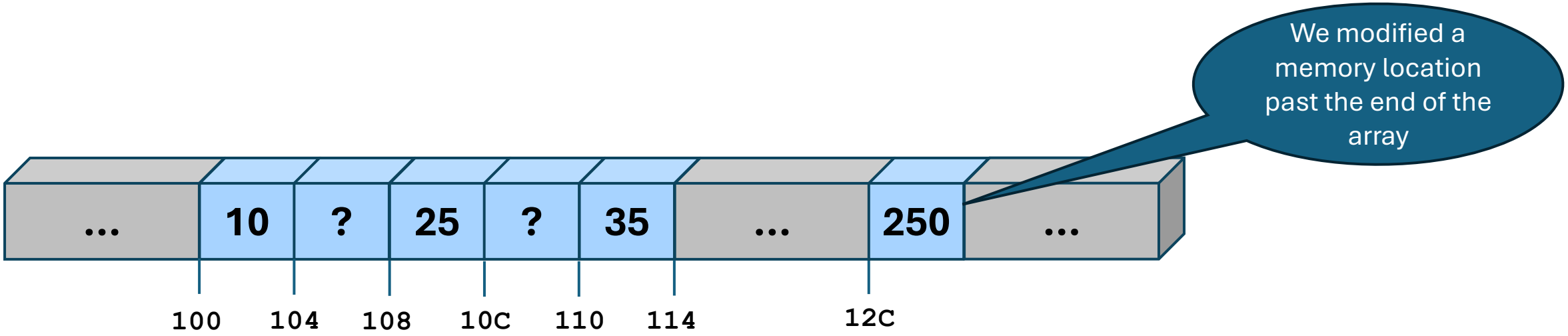
Out Of Bounds Reads and Writes

??

```
arr[10] = 250;
```

??





Out of Bounds Reads and Writes

What Just Happened?

An array of size 5 was declared, meaning space for 5 integers was allocated contiguously in memory

When we tried to access `arr[10]`, we reached beyond the allocated block

C does not stop us from doing this despite it not being part of the declared array

Out of Bounds Reads and Writes

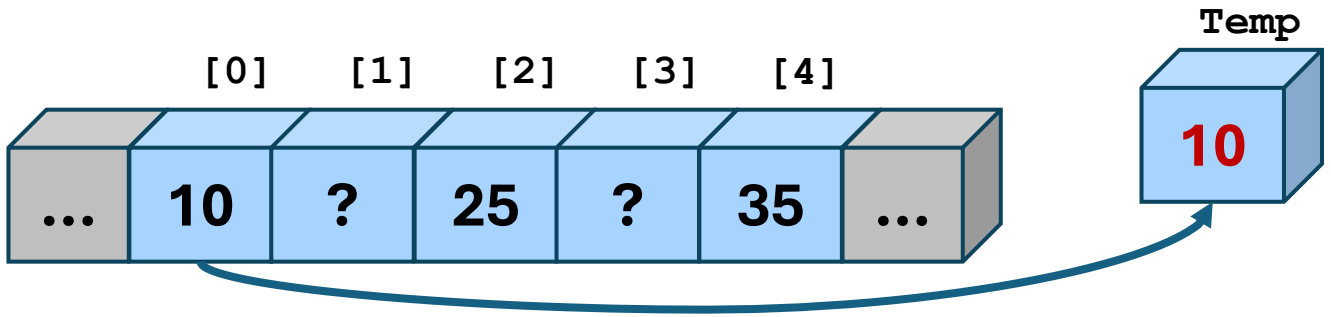
Is this ok to do?

If there is extra unused memory, the program might *seem* to work fine

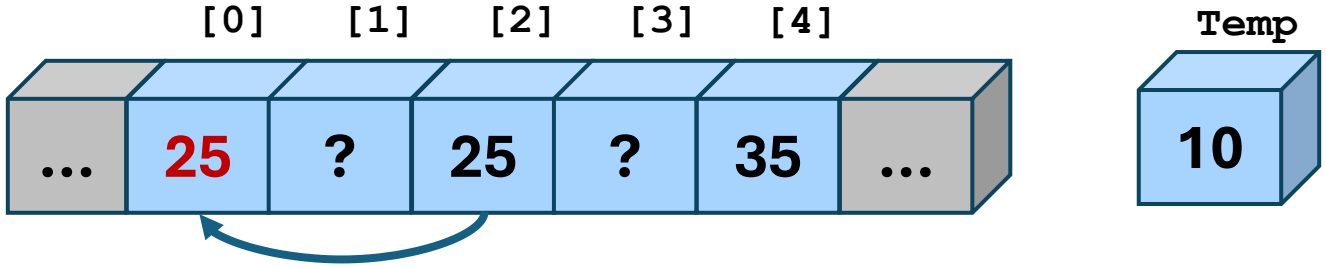
This memory could be in use by another variable or function call

Even if it appears to work fine (no crash), it could lead to silent corruption (data being corrupted elsewhere in the program)

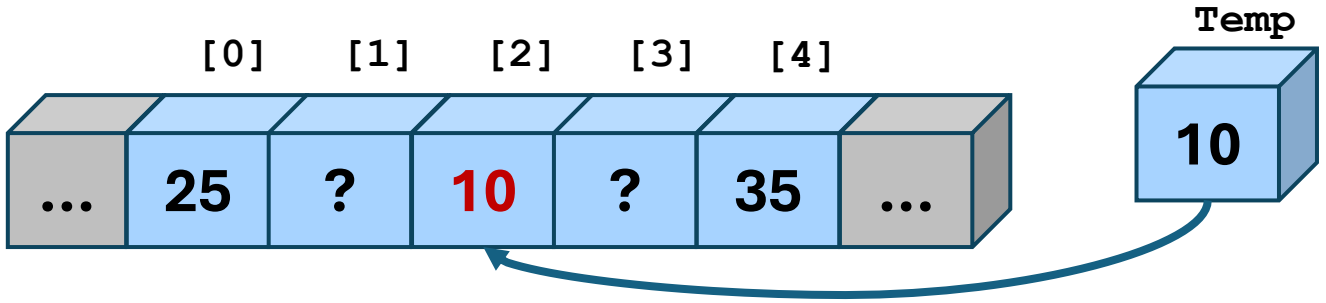
Swapping Values




```
// swap arr[0] with arr[2]
int temp = arr[0];
arr[0] = arr[2];
arr[2] = temp;
```




```
// swap arr[0] with arr[2]
int temp = arr[0];
arr[0] = arr[2];
arr[2] = temp;
```




```
// swap arr[0] with arr[2]
int temp = arr[0];
arr[0] = arr[2];
arr[2] = temp;
```

Printing An Array

There's no built-in way to do this, so we just use a familiar for loop

```
for(int i=0; i<5; i++) {
   printf("%d\n", arr[i]);
}
```

Strings

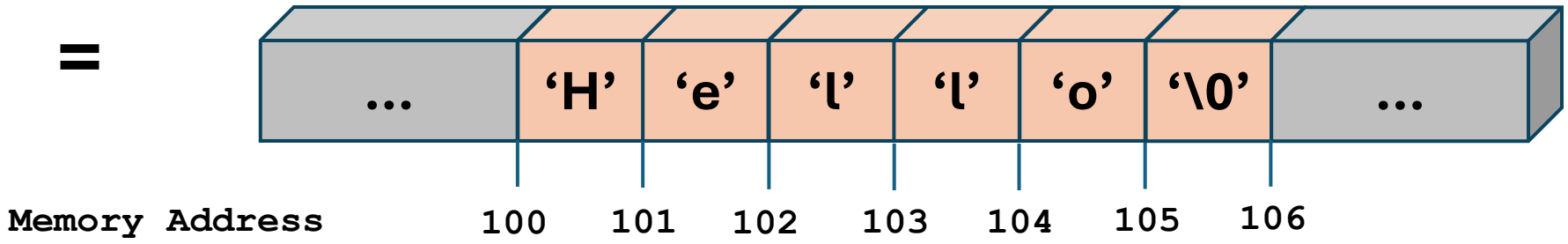
In C, strings are null-terminated arrays of chars

```
char arr[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

=

```
char arr[6] = "Hello";
```

=



Strings

What is this ‘\0’ character?

C does not know where a string is supposed to end. It just keeps reading memory until it finds ‘\0’ or until it crashes

```
char str[5] = {'H', 'e', 'l', 'l', 'o'};
printf("%s", str); // Keeps printing until it sees '\0;
```