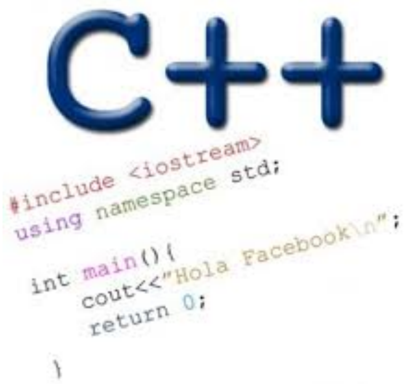
# C++ MEMORY MODEL
# C++ ARRAYS

Problem Solving with Computers-I



```
#include <iostream>
using namespace std;

int main(){
    cout<<"Hola Facebook\n";
    return 0;

}
```

## Memory and C++ programs

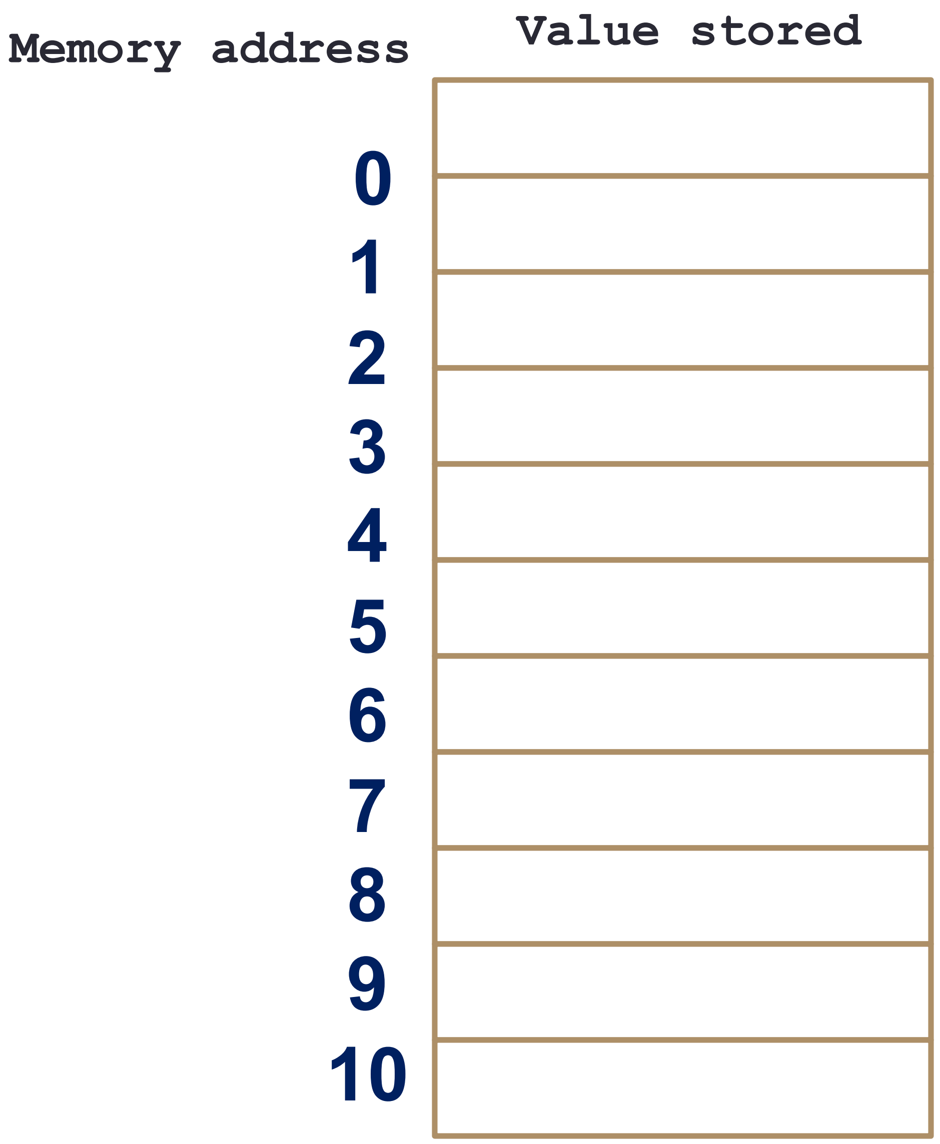
"*The overwhelming majority of program bugs and computer crashes stem from problems of memory access... Such memory-related problems are also notoriously difficult to debug. Yet the role that memory plays in C and C++ programming is a subject often overlooked…. Most professional programmers learn about memory entirely through experience of the trouble it causes.*"

…. Frantisek Franek

(Memory as a programming concept)

## General model of memory

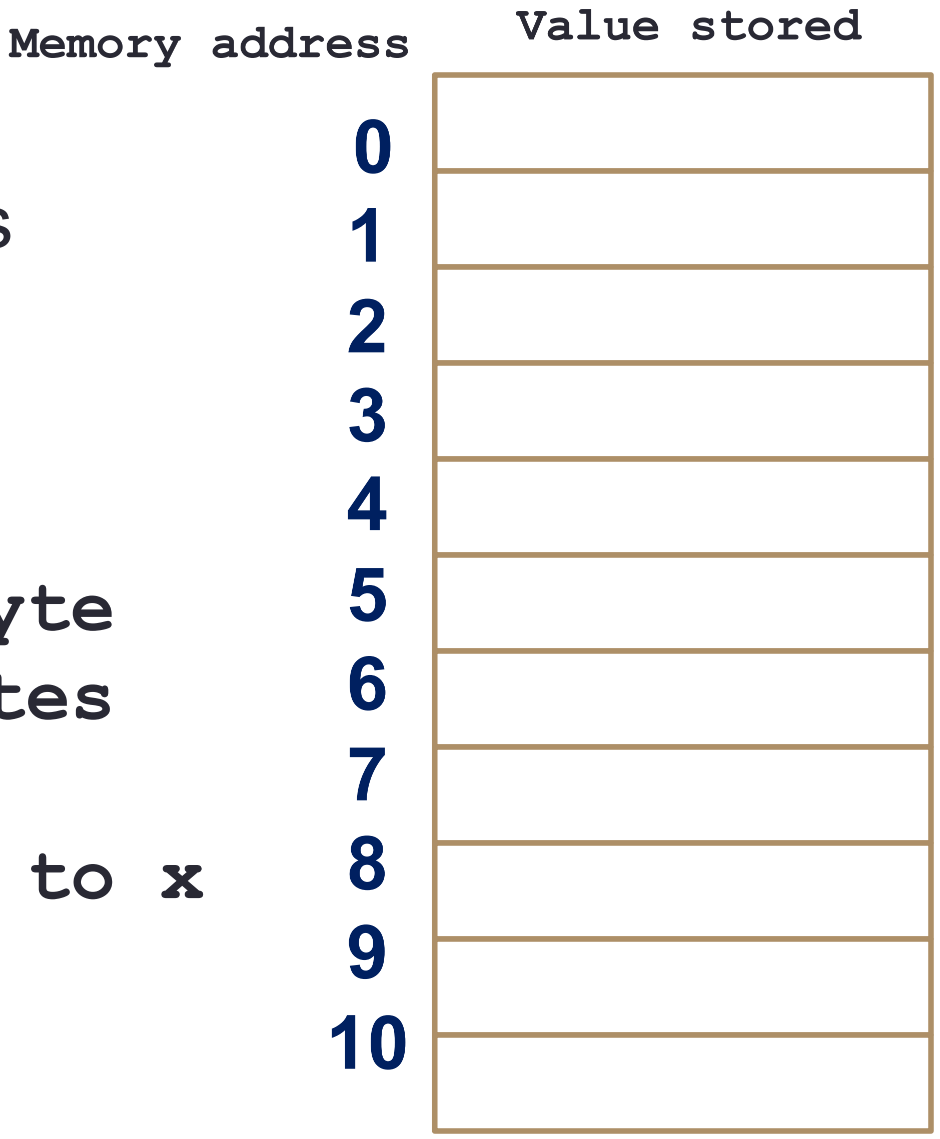
- Sequence of adjacent cells
- Each cell has 1-byte stored in it
- Each cell has an address (memory location)



# C++ data/variables

- When a variable is declared memory is allocated to store its value
- C understands the sizes of data types

```
char x = 1;         // x is 1 byte
int y = 0xFFFE;     //y is 4 bytes
char tmp = x;
x = y;     //value of y copied to x
y = tmp;
```



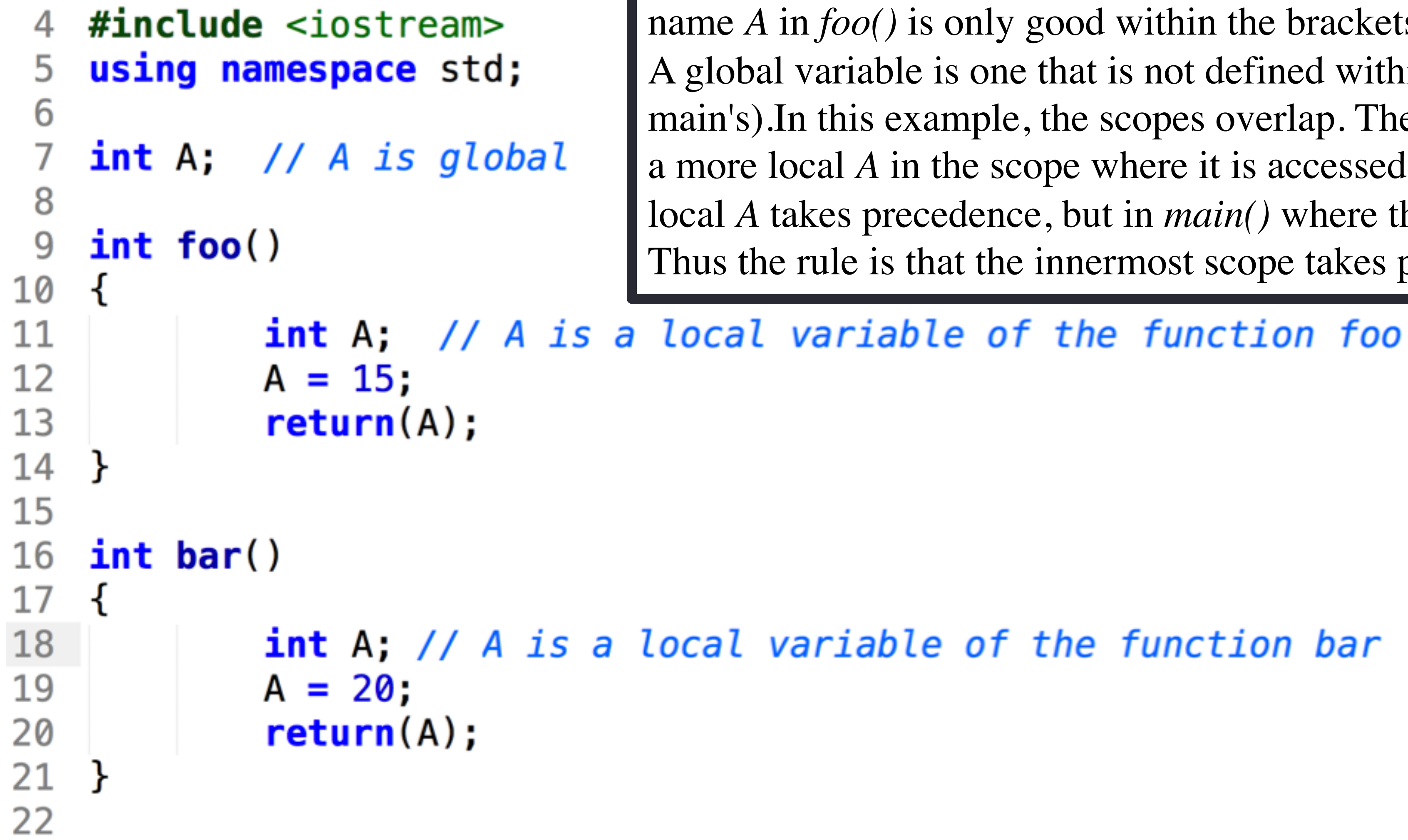
## C++ data/variables: the not so obvious facts

The not so obvious facts about data/variables in C++ are that there are:

- two scopes: local and global
- three different regions of memory: global data, heap, stack
- four variable types: local variable, global variables, dynamically allocated variables, and function parameters

## Variable: scope: Local vs global

The variable *A* appears in three places: in *foo()*, in *bar()*, and as a global variable. The scoping rules say that a variable declaration is fenced by its scope. Thus, the name *A* in *foo()* is only good within the brackets that define foo's function body.
A global variable is one that is not defined within any function body (including main's).In this example, the scopes overlap. The global *A* is valid as long as there is not a more local *A* in the scope where it is accessed. That is, in *foo()* and in *bar()* the local *A* takes precedence, but in *main()* where there is no local *A* the global *A* is used.
Thus the rule is that the innermost scope takes precedence.



```
#include <iostream>
using namespace std;

int A;   // A is global

int foo()
{
        int A;  // A is a local variable of the function foo
        A = 15;
        return(A);
}

int bar()
{
        int A; // A is a local variable of the function bar
        A = 20;
        return(A);
}

```

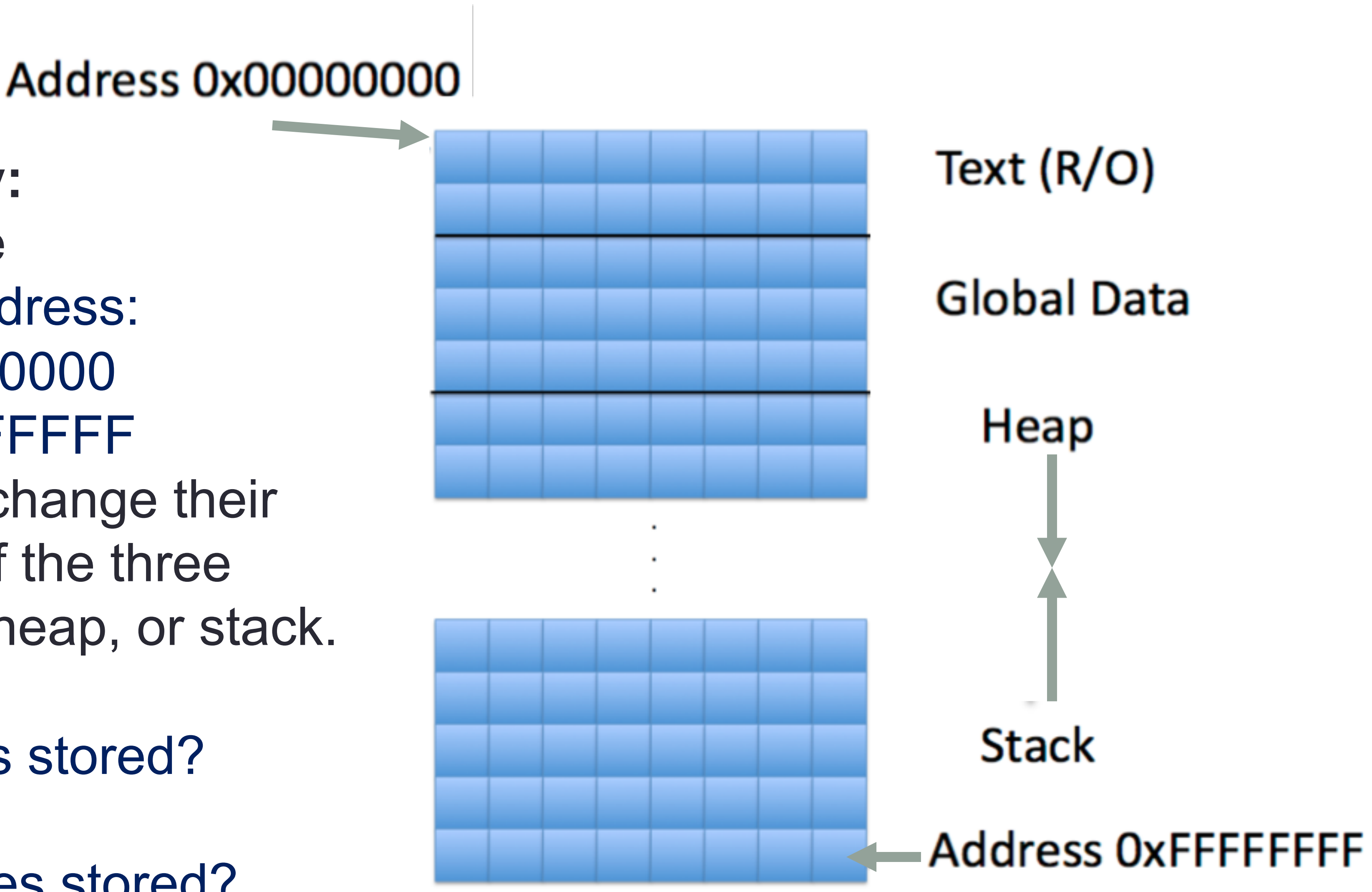
# C++ Memory Model

**Program layout in memory:**

- Each cell stores one byte
- Each cell has a 32-bit address:
  - Low address: 0x00000000
  - High address:0xFFFFFFFF
- All variables (which can change their values) must be in one of the three segments : Global data, heap, or stack.

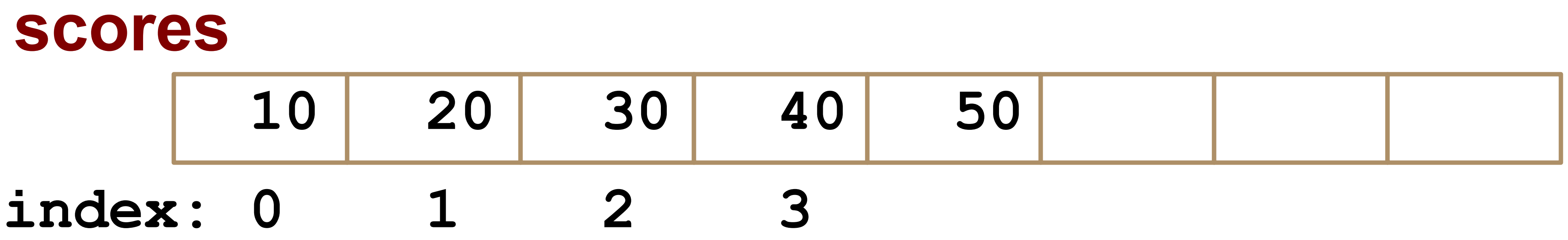
- Where are local variables stored?

- Where are global variables stored?



## C++ Arrays

A C++ array is a **list of elements** that share the same name, have the same data type and are located adjacent to each other in memory



```
int scores[5]; //Array declaration

//Declare and initialization as follows:

int scores[]={10, 20, 30, 40, 50};
```

## What is the memory location of each element?

| | | | | | |
|---|---|---|---|---|---|
| scores | 10 | 20 | 30 | 40 | 50 |

```
int scores[]={10, 20, 30, 40, 50};
```

If the starting location of the array is 0x200, what is memory location of element at index 2?

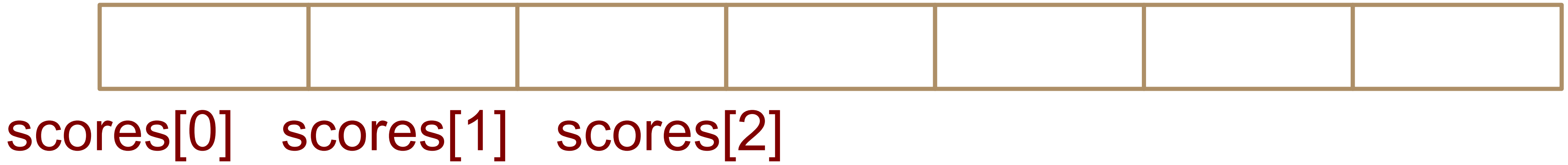
A. 0x201

B. 0x202

C. 0x204

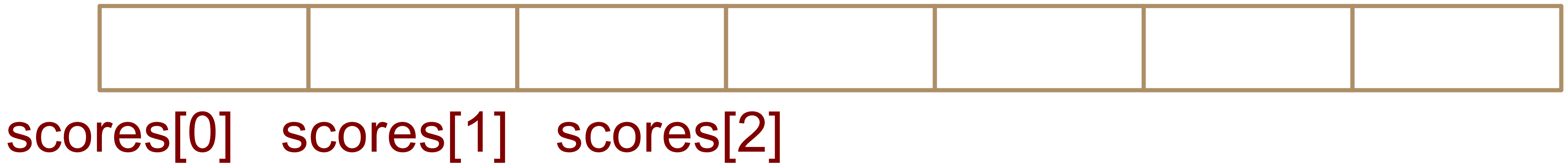
D. 0x208

## Exercise: Reassign each value to 60



```
int scores[]={20,10,50}; // declare an initialize
//Access each element and reassign its value to 60
```

## Exercise: Increment each element by 10



```
int scores[]={20,10,50}; // declare an initialize
//Increment each element by 10
```

## Most common array pitfall- out of bound access

scores[0] scores[1] scores[2]

```
int arr[]={20,10,50}; // declare an initialize
for(int i=0; i<=3; i++)
    scores[i] = scores[i]+10;
```

**Demo: Passing arrays to functions**

## Tracing code involving arrays

|  |  |  |
|---|---|---|
|  |  |  |
| arr[0] | arr[1] | arr[2] |

```
int arr[]={1,2,3};
int tmp = arr[0];
arr[0] = arr[2];
arr[2] = tmp;
```

Choose the resulting array after the code is executed

**A.**

|  |  |  |
|---|---|---|
| 1 | 2 | 3 |
| arr[0] | arr[1] | arr[2] |

**B.**

|  |  |  |
|---|---|---|
| 2 | 1 | 3 |
| arr[0] | arr[1] | arr[2] |

**C.**

|  |  |  |
|---|---|---|
| 3 | 2 | 1 |
| arr[0] | arr[1] | arr[2] |

**D.** None of the above

## Next time

- Pointers
- Mechanics of function calls – call by value and call by reference