C++ ARRAYS AND POINTERS

Problem Solving with Computers-I

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

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

C++ Arrays

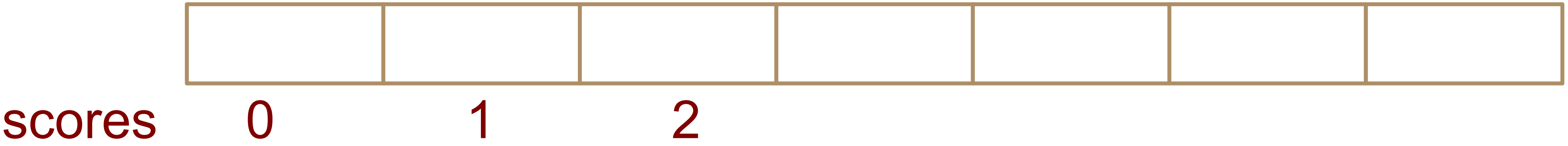
- **List of elements**
- All elements have the same data type
- The elements are located adjacent to each other in memory
- Like all variables in C++, you must declare an array before using it

Accessing elements of an array

`int scores[]={20,10,50};` // declare and initialize

- **`scores` is the starting memory location of the array**
  - **also called the base address**
  - **Base address (`scores`) cannot be modified**
- **Access array elements using their index**
- **Indices start at 0**
  - **scores[0]: 20**
  - **scores[1]: 10**
  - **scores[2]: 50**
  - **scores[3]: out of bound array access, undefined behavior**

Iterating through an array



scores 0 1 2

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

To iterate use:

- * **regular for loops**
- * **Or range based for loop (C++ 11 feature)**

Modifying the array

What is the output of this code?

```
int scores[]={20,10,50};
scores = scores + 10;
for(int i=0; i<3; i++){
    cout<<scores[i]<<“\t”;
}
```

A. 30 20 60

B. 20 10 50

C. Compiler error

Tracing code involving arrays

arr	0	1	2

```
int arr[]={10,20,30};
int tmp = arr[0];
arr[0] = arr[2];

arr[2] = tmp;
```

Choose the resulting array after the code is executed

A.

	10	20	30
arr	0	1	2

B.

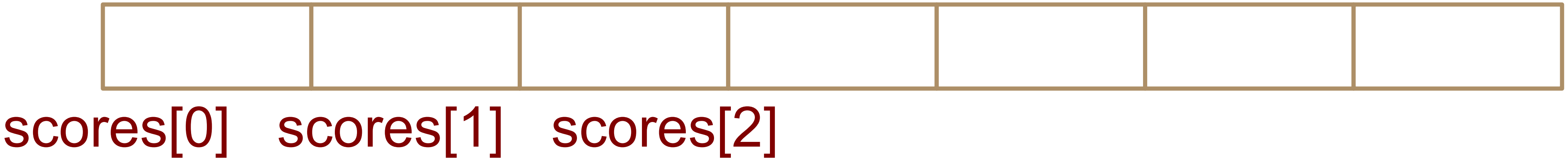
	20	10	30
arr	0	1	2

C.

	30	20	10
arr	0	1	2

D. None of the above

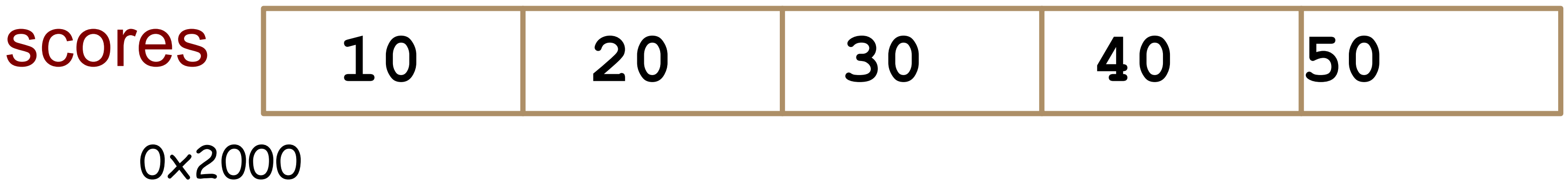
Most common array pitfall- out of bound access



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

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

Passing arrays to functions



scores

10	20	30	40	50

0x2000

```
int main(){
    int scores[]={10, 20, 30, 40, 50};
    foo(scores);
}
double foo(int sc[]){
    cout<<sc;
    return
}
```

What is the output?

A. 10
B. 10 20 30 40 50
C. 0x2000
D. None of the above

char arrays, C-strings

- How are ordinary arrays of characters and C-strings similar and how are they dissimilar?

What is the output of the code?

```
char s1[] = "Mark";
char s2[] = "Jill";
for (int i = 0; i <= 4; i++)
     s2[i] = s1[i];
if (s1 == s2) s1 = "Art";
cout<<s1<<" "<<s2<<endl;
```

A. Mark Jill

B. Mark Mark

C. Art Mark

D. Compiler error

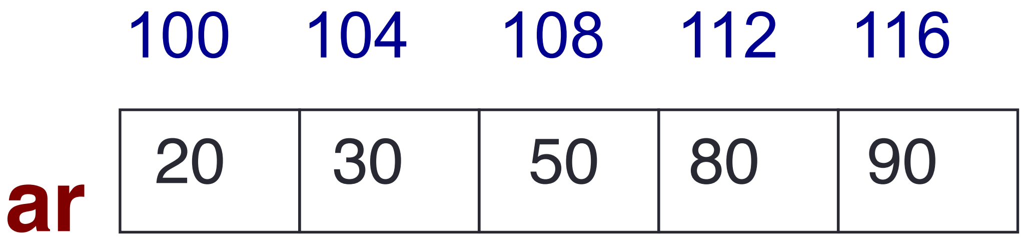
E. Run-time error

Pointers and references: Draw the diagram for this code

```
int a = 5;
int &b = a;
int *pt1 = &a;
```

What are three ways to change the value of ‘a’ to 42?

Arrays and pointers



	100	104	108	112	116
ar	20	30	50	80	90

- `ar` is like a pointer to the first element
- `ar[0]` is the same as `*ar`
- `ar[2]` is the same as `*(ar+2)`

- Use pointers to pass arrays in functions
- Use *pointer arithmetic* to access arrays more conveniently

Pointer Arithmetic

```
int ar[]={20, 30, 50, 80, 90};
int *p;
p = arr;
p = p + 1;
*p = *p + 1;
```

Draw the array ar after the above code is executed

Pointer Arithmetic

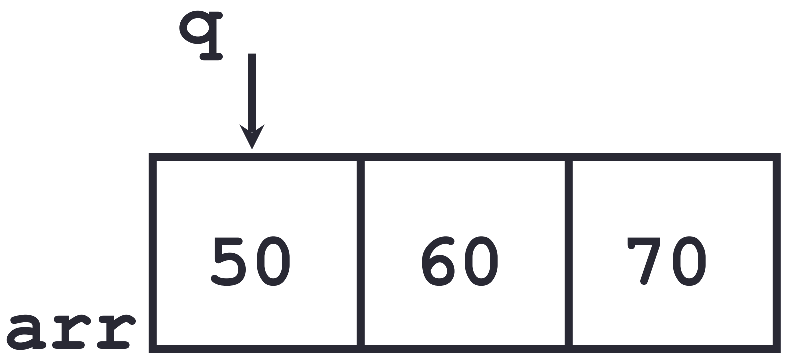
```
int ar[]={20, 30, 50, 80, 90};
```

How many of the following are invalid?

I. pointer + integer (ptr+1)
II. integer + pointer (1+ptr)
III. pointer + pointer (ptr + ptr)
IV. pointer – integer (ptr – 1)
V. integer – pointer (1 – ptr)
VI. pointer – pointer (ptr – ptr)
VII. compare pointer to pointer (ptr == ptr)
VIII. compare pointer to integer (1 == ptr)
IX. compare pointer to 0 (ptr == 0)
X. compare pointer to NULL (ptr == NULL)

```
#invalid
     A: 1
     B: 2
     C: 3
     D: 4
     E: 5
```

```
void IncrementPtr(int *p){
    p++;
}

int arr[3] = {50, 60, 70};
int *q = arr;
IncrementPtr(q);
```



Which of the following is true after **`IncrementPtr(q)`** is called in the above code:

A. ‘**q**’ points to the next element in the array with value 60

B. ‘**q**’ points to the first element in the array with value 50

How should we implement `IncrementPtr()`, so that ‘q’ points to 60 when the following code executes?

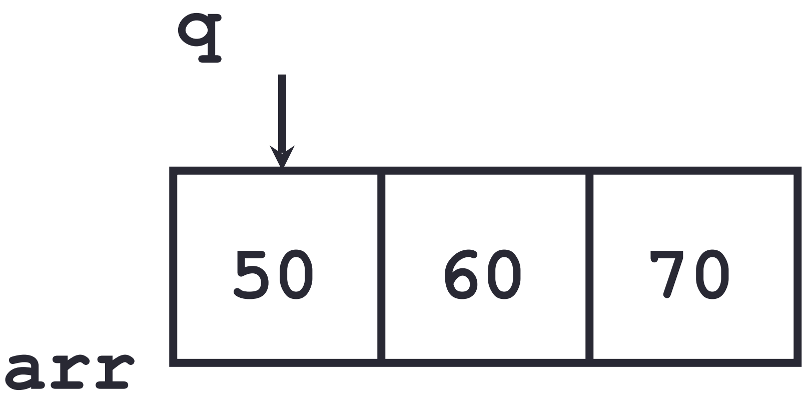
```
void IncrementPtr(int **p){
    p++;
}

int arr[3] = {50, 60, 70};
int *q = arr;
IncrementPtr(&q);
```

A. `p =  p + 1;`

B. `&p = &p + 1;`

C. `*p= *p + 1;`

D. `p= &p+1;`



Pointer pitfalls

- Dereferencing a pointer that does not point to anything results in undefined behavior.
- On most occasions your program will crash
- Segmentation faults: Program crashes because code tried to access memory location that either doesn’t exist or you don’t have access to

Two important facts about Pointers

1) A pointer can only point to one type –(basic or derived) such as `int`, `char`, a `struct`, another pointer, etc

2) After declaring a pointer: `int *ptr;`

   `ptr` doesn't actually point to anything yet.

   We can either:

   ➢ make it point to something that already exists, OR

   ➢ allocate room in memory for something new that it will point to

Pointer Arithmetic

- What if we have an array of large structs (objects)?
  - C++ takes care of it: In reality, `ptr+1` doesn't add `1` to the memory address, but rather adds the size of the array element.
  - C++ knows the size of the thing a pointer points to – every addition or subtraction moves that many bytes: 1 byte for a char, 4 bytes for an int, etc.

Next time

- Structs
- Arrays of structs