

# C++ ARRAYS POINTERS

---

Problem Solving with Computers-I

C++

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

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



Are you willing to complete the homework electronically using a tablet to annotate or by editing the pdf?

**A. Yes**

**B. Yes, but I don't know how to**

**C. No**

# C++ Arrays

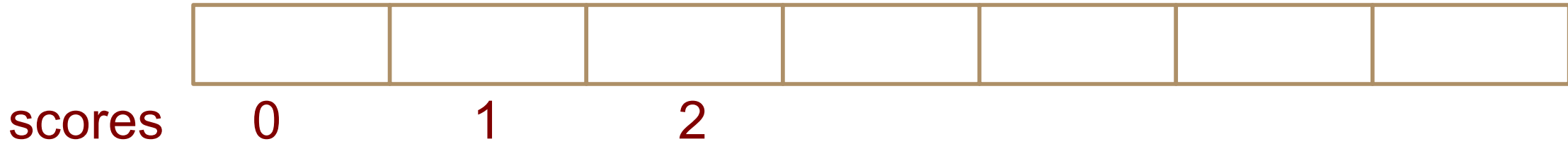
- **List of elements**
- All elements have the same data type
- The elements are located adjacent to each other in memory
- Like all variables in C++, you must declare an array before using it

# 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



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

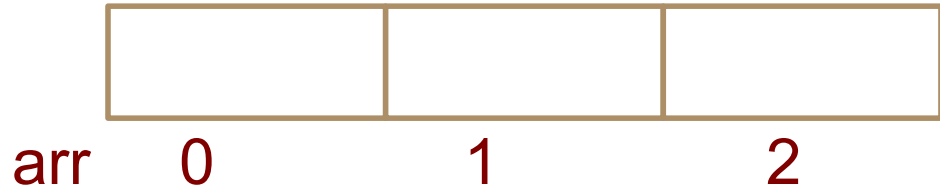
```
int scores[]={20,10,50};  
scores = scores + 10;  
for(int i=0; i<3; i++){  
    cout<<scores[i]<<"\t";  
}
```

**A. 30 20 60**

**B. 20 10 50**

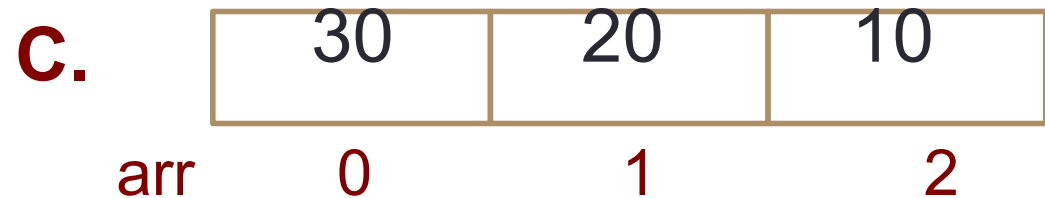
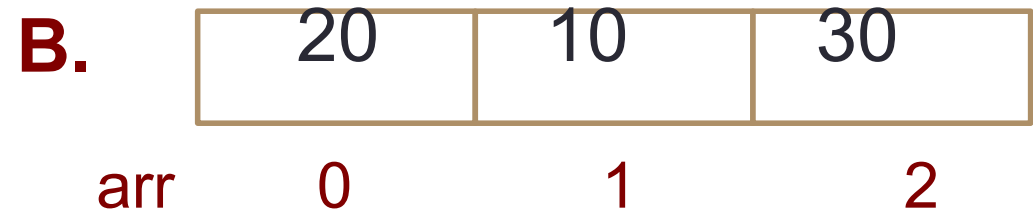
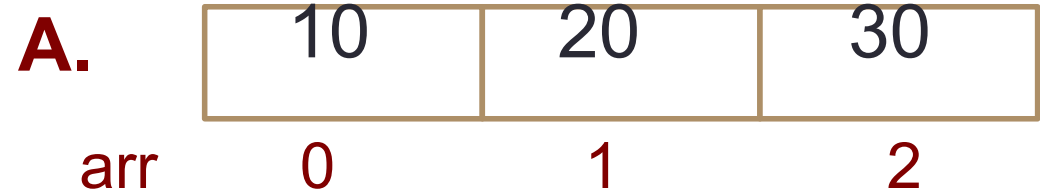
**C. Compiler error**

# Tracing code involving arrays



```
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



**D.** None of the above

# Most common array pitfall- out of bound access

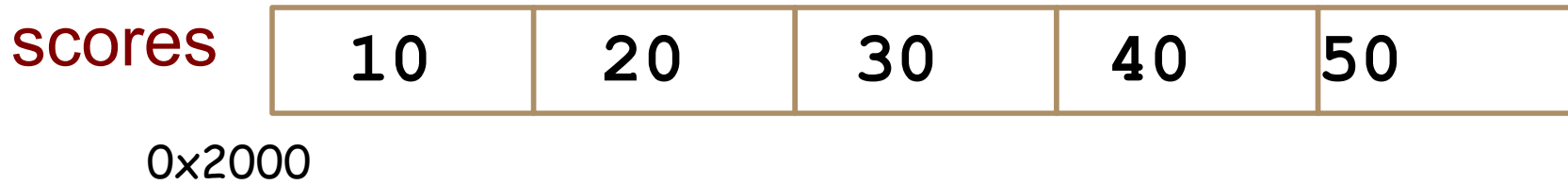


scores[0] scores[1] scores[2]

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



# Passing arrays to functions



```
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

# Pointers

- **Pointer:** A variable that contains the address of another variable
- Declaration: `type * pointer_name;`

```
int* p;
```



# How to make a pointer **point to** something

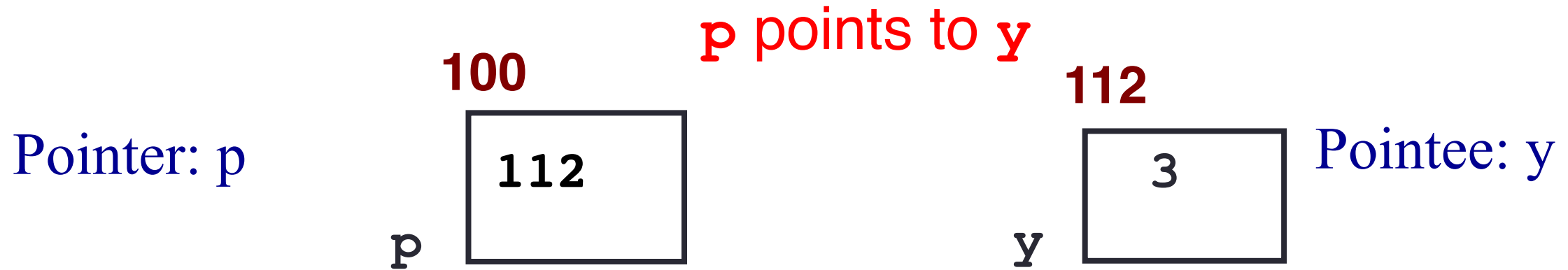
```
int *p;  
int y = 3;
```



To access the location of a variable, use the address operator '&'

## Pointer Diagrams:

Diagrams that show the relationship between pointers and pointees



You can change the value of a variable using a pointer !

```
int *p, y;
```

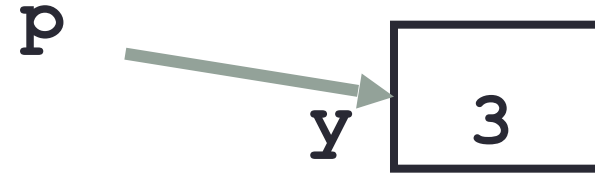
```
y = 3;
```

```
p = &y;
```

```
*p = 5;
```

# Two ways of changing the value of a variable

- Change the value of y directly:

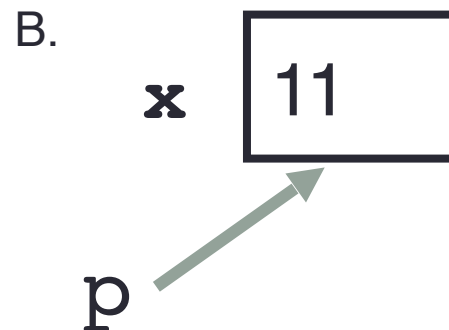
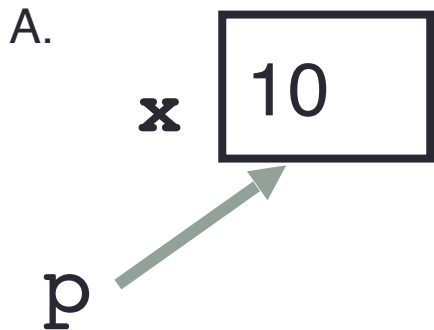


- Change the value of y indirectly (via pointer p):

# Tracing code involving pointers

```
int *p;  
int x=10;  
p = &x;  
*p = *p + 1;
```

Q: Which of the following pointer diagrams best represents the outcome of the above code?

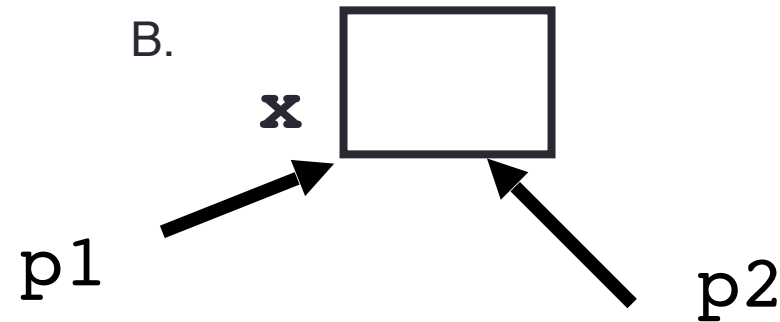
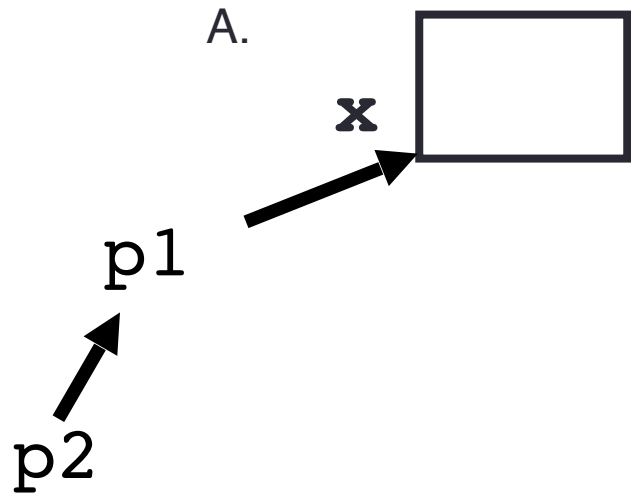


C. Neither, the code is incorrect

# Pointer assignment

```
int *p1, *p2, x;  
p1 = &x;  
p2 = p1;
```

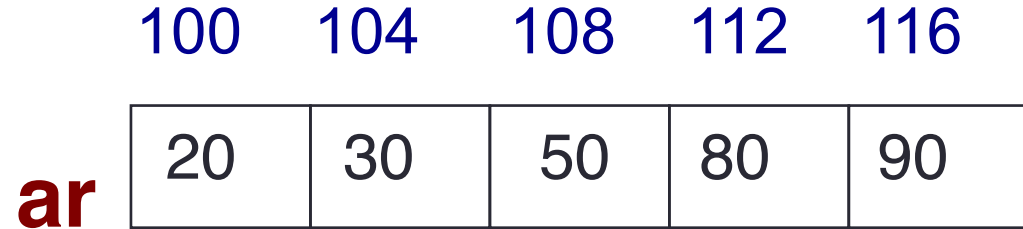
Q: Which of the following pointer diagrams best represents the outcome of the above code?



C. Neither, the code is incorrect



# Arrays and pointers



- `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};
```

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

# 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**

# 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

# 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

- References
- Call by value, call by reference and call by address