

Lecture 4: Outline

I. Pointers

- A. Accessing data objects using pointers
- B. Type casting with pointers
- C. Difference with Java references
- D. Pointer pitfalls
- E. Use case

II. Arrays

- A. Representation in memory (difference with arrays in Java)
- B. Arrays and Pointers
- C. Fast data access with arrays
- D. Array pitfalls

III. Pointer arithmetic

IV. Strings

Pointers

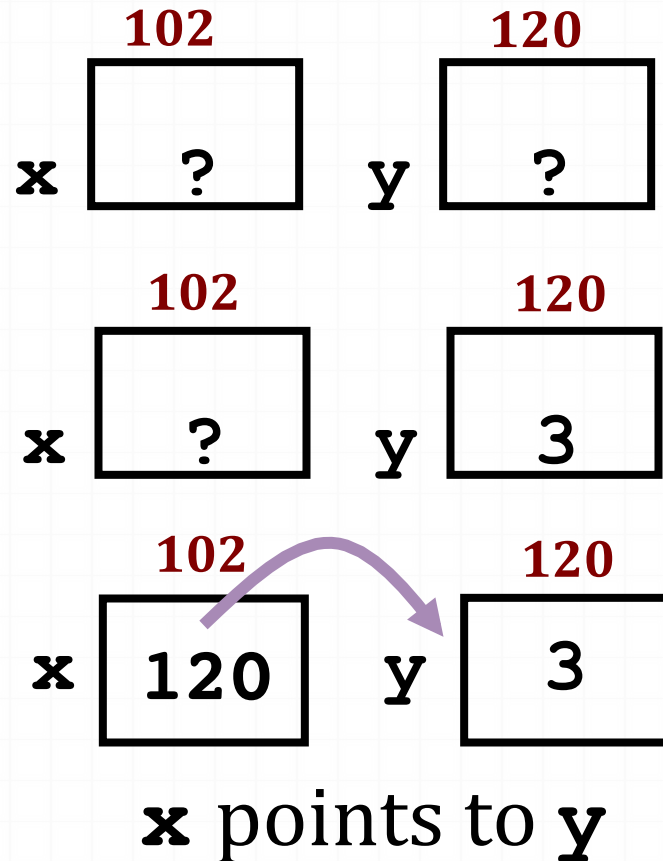
- **Pointer:** A variable that contains the address of a variable
- How to create a pointer:

```
int *x, y;
```

```
y = 3;
```

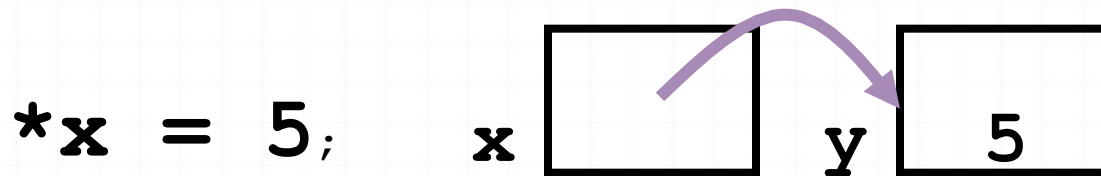
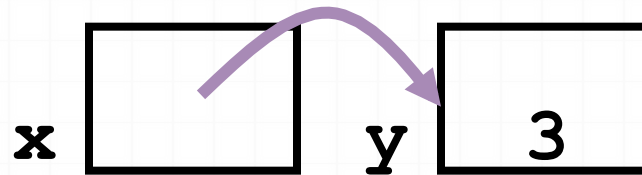
```
x = &y;
```

```
sizeof(x) = ?
```



How to change the value of a variable pointed to by a pointer?

Use dereference $*$ operator to left of pointer name



- Two ways of changing the value of any variable
- Why this is useful will be clear when we discuss functions and pointers

Pointer and Pointee data types

Q: This code gives a warning at compile time. Why?

```
main() {  
    int *p;  
    float y;  
    p = &y;  
    printf("Pointer points to value %d",*p);  
}
```

- A. The pointer 'p' is made to point to a variable of incompatible type
- B. The *printf* statement expects an integer but 'p' is pointing to a type *float*
- C. *p does not contain a valid value because y was not initialized

Pointer and Pointee data types

Q: This code gives a warning at compile time. Why?

```
main() {  
    int *p;  
    float y;  
    p = &y;  
    printf("Pointer points to value %d",*p);  
}
```

- A. The pointer 'p' is made to point to a variable of incompatible type
- B. The *printf* statement expects an integer but 'p' is pointing to a type *float*
- C. *p does not contain a valid value because y was not initialized

Type casting with pointers

- We can cast addresses that are assigned to pointers to a different type

```
char c = 0x04;
```

```
int *p;
```

```
p = (int *)&c; /* Treat &c as address of an int */
```

OR

```
short i = 0x0104;
```

```
char *p;
```

```
p = (char *)&i; /* Treat &i as address of a char */
```

What is the output of the following code?

```
char *p;  
short i = 0x0104;  
p = (char *)&i; /* Treat &i as address of a char */  
printf("Value in p is: %x", *p);
```

- A. It is always the lower byte 0x04
- B. It is always the higher byte 0x01
- C. Depends

What is the output of the following code?

```
char *p;  
short i = 0x0104;  
p = (char *)&i; /* Treat &i as address of a char */  
printf("Value in p is: %x", *p);
```

- A. It is always the lower byte 0x04
- B. It is always the higher byte 0x01
- C. Depends: Little endian 0x04, Big endian 0x01

Difference with Java references: Pointers in C have a more generic usage

Q: Does the following code give an error at run time? If yes, why? If no, what is the output?

```
main() {  
    int *p;  
    *p = 5;  
    printf("%d", (*p)++);  
}
```

- A. Yes, because the pointer does not point to a valid address in memory
- B. Yes, because there is a syntax error in the printf statement: `(*p)++` is not a valid operation on a pointer
- C. No, the output of the program is the value pointed to by p, which is 5
- D. No, the output of the program is the value pointed to by p, plus one, which is 6

Q: Does the following code give an error at run time? If yes, why? If no, what is the output?

```
main() {  
    int *p;  
    *p = 5;  
    printf("%d", (*p)++);  
}
```

- A. Yes, because the pointer does not point to a valid address in memory
- B. Yes, because there is a syntax error in the printf statement: `(*p)++` is not a valid operation on a pointer
- C. No, the output of the program is the value pointed to by p, which is 5
- D. No, the output of the program is the value pointed to by p, plus one, which is 6

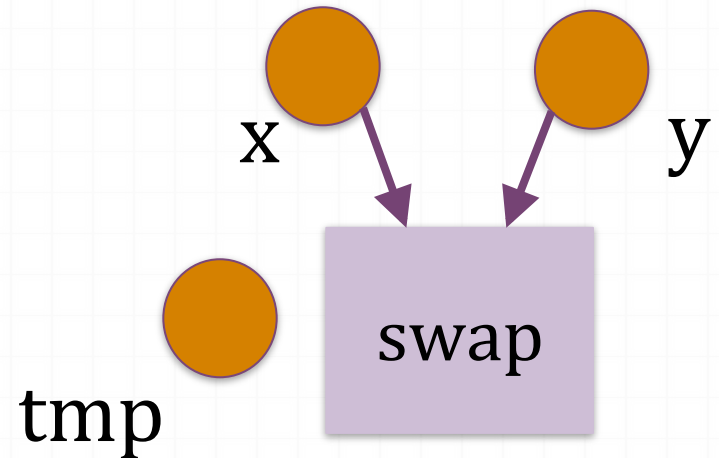
Two important facts about Pointers

- 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... (next lecture)

Consider the following function

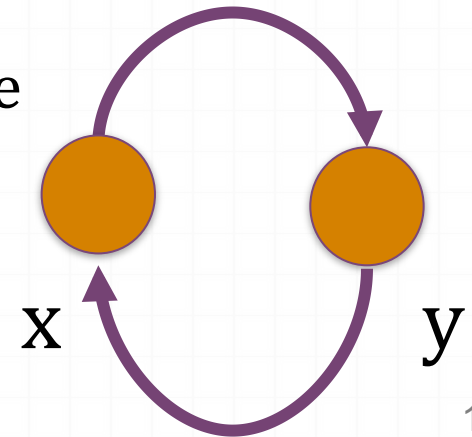
```
void swap(int x, int y); //Declaration
```

```
void swap(int x, int y) {  
    int tmp;  
    tmp= x;  
    x= y;  
    y= tmp; }  
}
```



The swap function intends to swap the value of its inputs. Is the logic correct?

- A. Yes
- B. No
- C. Depends



Q: Are the value of variables 'a' and 'b' interchanged after swap is called?

```
main() {  
    . . .  
    swap(a, b);  
    . . . }  
a 5 b 6
```

A. Yes, because

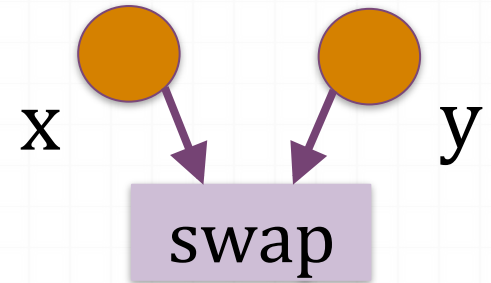
B. No, because.....

Functions: Call by value

```
main() {  
    . . .  
    swap(a, b);  
    . . . }
```

a 5

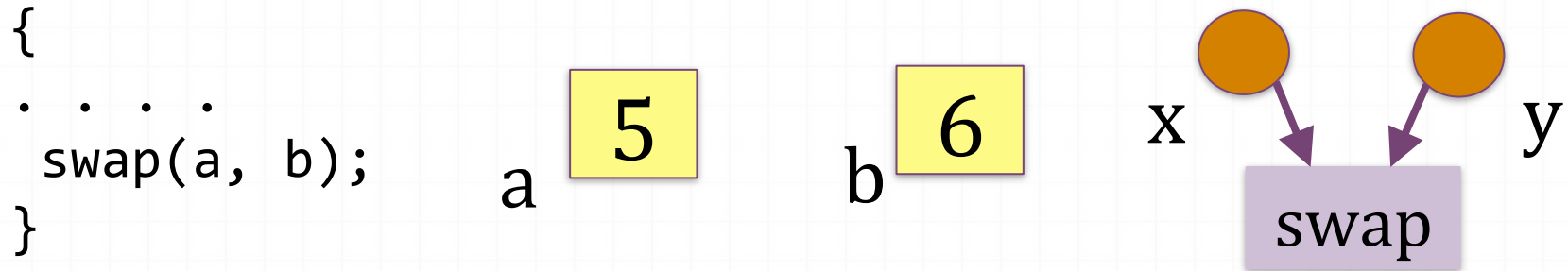
b 6



Q: Are the value of variables 'a' and 'b' interchanged after swap is called?

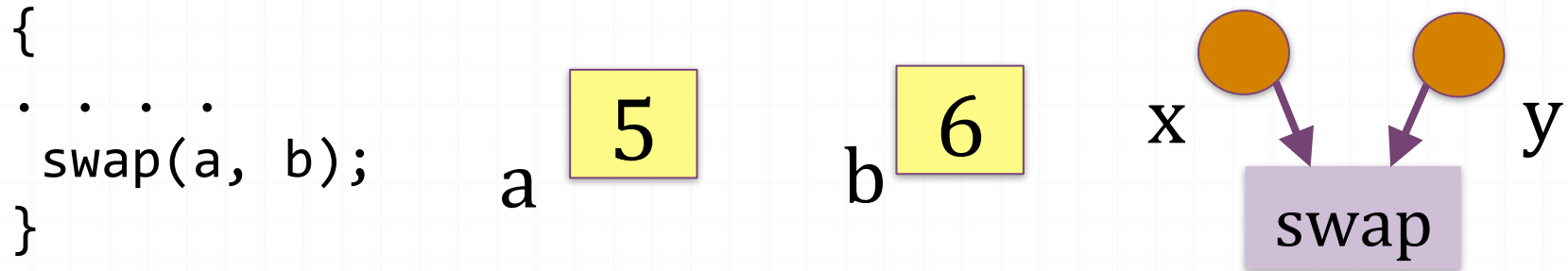
- A.** Yes, because that's what is implemented by the 'swap' routine
- B.** No, because the inputs to swap are only copies of 'a' and 'b'

Q: Which of the following changes are required to interchange the values in 'a' and 'b' when swap is called?



- A. In swap, return the values of 'x' and 'y' to the main function after swapping them
- B. Declare 'a' and 'b' as global variables, so that they become accessible to the swap routine
- C. Pass the address of 'a' and 'b' to swap instead of their value
- D. Move the implementation in swap to the main function

Q: Which of the following changes are required to interchange the values in 'a' and 'b' when swap is called?



- A. In swap, return the values of 'x' and 'y' to the main function after swapping them
- B. Declare 'a' and 'b' as global variables, so that they become accessible to the swap routine
- C. Pass the address of 'a' and 'b' to swap instead of their value
- D. Move the implementation in swap to the main function

Functions: Call by reference

```
void swap(int *x, int *y) {  
    . . .  
}
```

Q: What should the modified swap function do?

- A. Swap the addresses in 'x' and 'y'
- B. Swap the values pointed to by 'x' and 'y'
- C. Both the above operations are equivalent

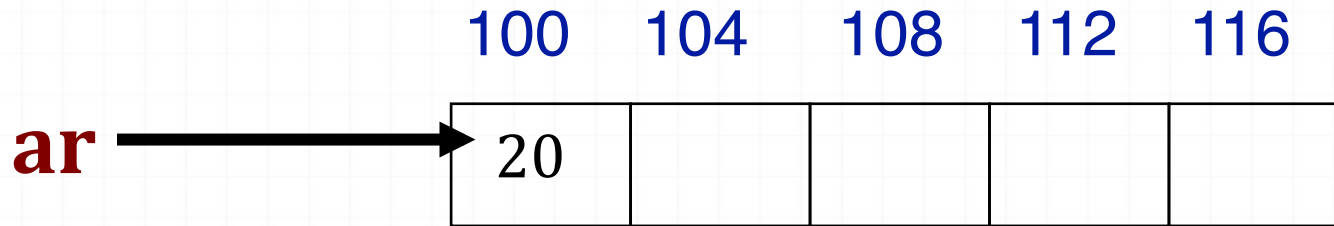
Functions: Call by reference

```
void swap(int *x, int *y) {  
    . . .  
}
```

Q: What should the modified swap function do?

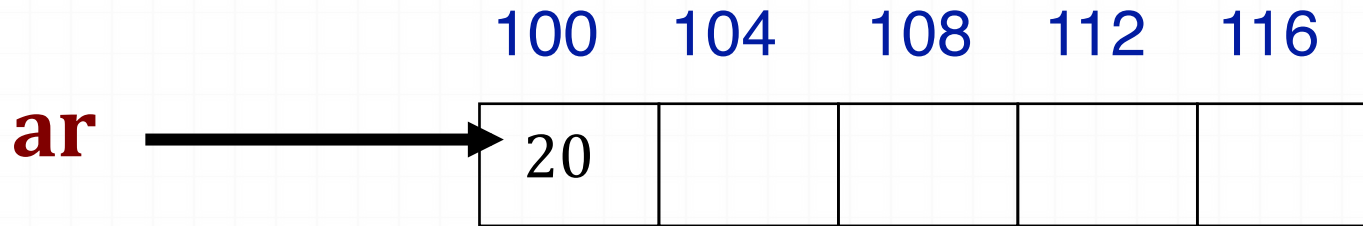
- A. Swap the addresses in 'x' and 'y'
- B. Swap the values pointed to by 'x' and 'y'
- C. Both the above operations are equivalent

Array Basics



- Stores group of elements of the same type
- Declaration:
 - `int ar[5];` // declares a 5-element integer array
 - `int ar[] = {795, 635};` //declares and fills a 2-element integer array.
- Accessing elements: `ar[i];` // returns the $(i+1)^{\text{th}}$ element
- How are arrays in C different from Java?
- Pointers and arrays are very similar

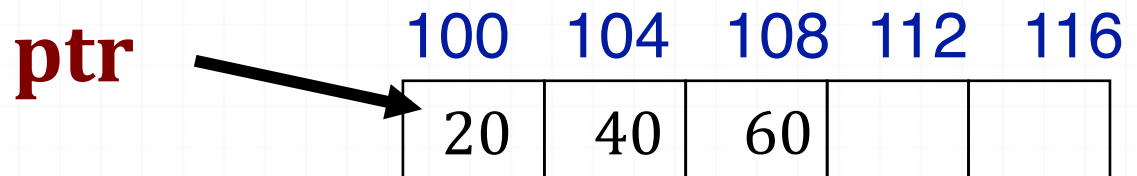
Arrays and Pointers



- `ar` is a pointer to the first element
- `ar[0]` is the same as `*ar`
- `ar[2]` is the same as `*(ar+2)`
- Use pointer arithmetic to access arrays more conveniently e.g. when passing arrays to functions

Pointer Arithmetic

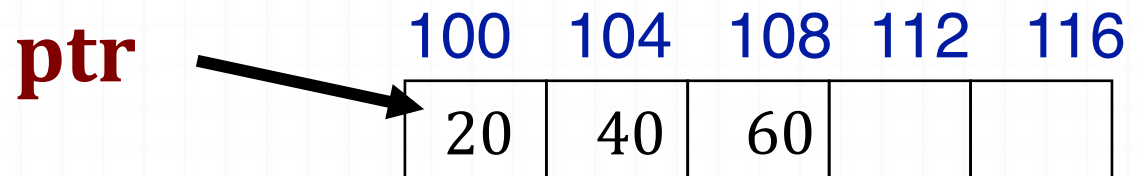
- Since a pointer is just a memory address, we can add to it to traverse an array.
- `ptr+1` will return a pointer to the next array element.



- `*ptr+1 = ?` A. 21, 20, 40
- `*ptr++ = ?` B. 21, 21, 40
- `*(ptr+1) = ?` C. 21, 40, 40

Pointer Arithmetic

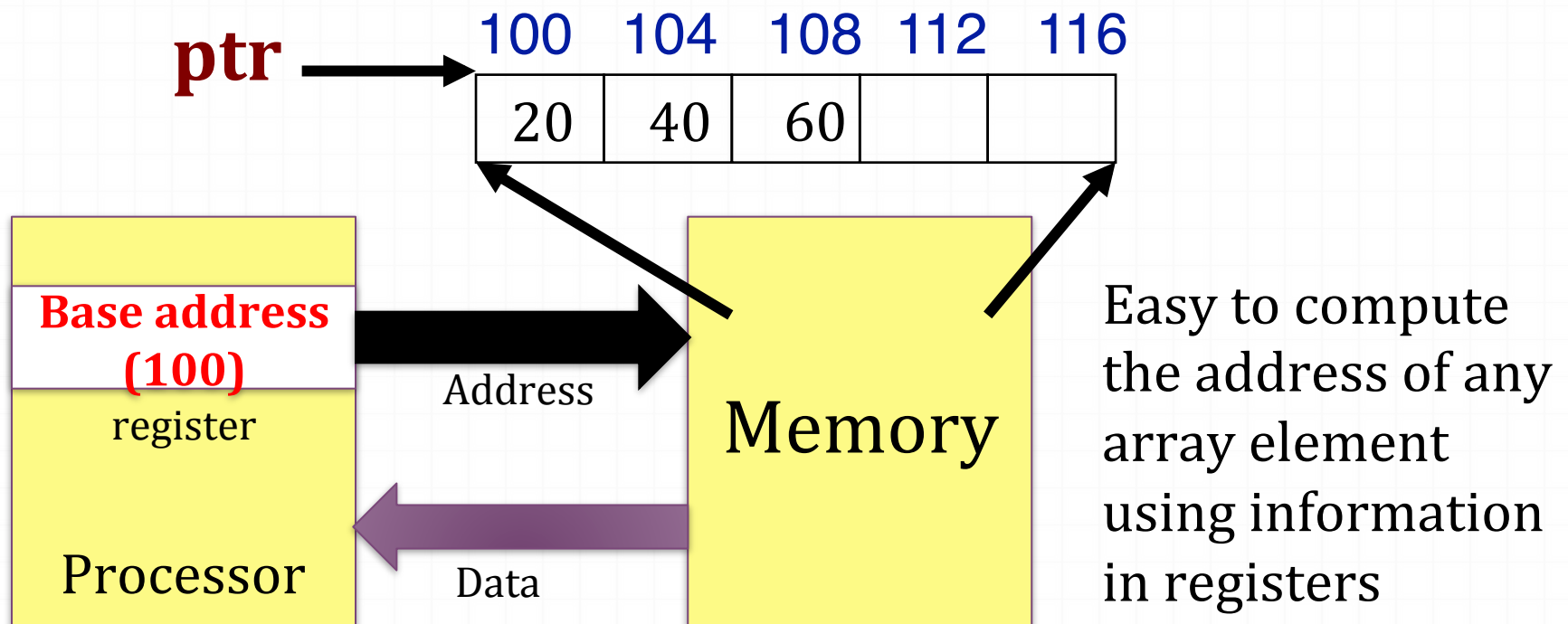
- Since a pointer is just a memory address, we can add to it to traverse an array.
- `ptr+1` will return a pointer to the next array element.



- `*ptr+1 = 21`
- `*ptr++ = 20`
- `*(ptr+1) = 40`

Arrays: Fast data access

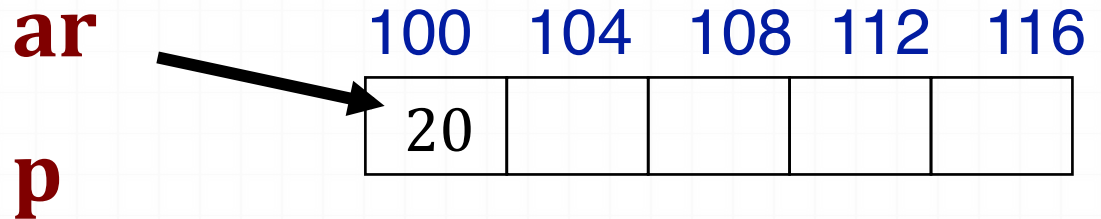
- Using pointer arithmetic, easy to compute the address of any array element in memory
- How are array elements accessed on an ARM?



Q: Which of the assignment statements produces an error at compilation. Why?

```
int *p, ar[5]; //Declaration
```

```
p=ar+5;  
ar=p+1;
```

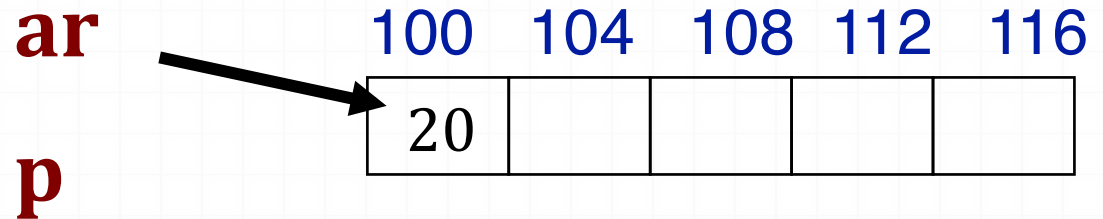


- A. `p=ar+5;`
- B. `ar=p+1;`
- C. Both statements result in error at compile time
- D. Neither results in a compilation error

Q: Which of the assignment statements produces an error at compilation. Why?

```
int *p, ar[5]; //Declaration
```

```
p=ar+5;  
ar=p+1;
```



- A. `p=ar+5;`
- B. `ar=p+1;` because the address stored in an array cannot be changed
- C. Both statements result in error at compile time
- D. Neither results in a compilation error

Q: What happens when the following code is executed?

```
int *p, ar[5]; //Declaration  
p=ar-5;  
*p=0;
```

- A. It always results in a segmentation fault because a pointer cannot be used to change the value of an array element
- B. It always results in a segmentation fault because the array element being accessed is out of bounds
- C. It is likely to result in a segmentation fault because the memory location being accessed may not be a valid address
- D. It results in a compilation error

Q: What happens when the following code is executed?

```
int *p, ar[5]; //Declaration  
p=ar-5;  
*p=0;
```

- A. It always results in a segmentation fault because a pointer cannot be used to change the value of an array element
- B. It always results in a segmentation fault because the array element being accessed is out of bounds
- C. It is likely to result in a segmentation fault because the memory location being accessed may not be a valid address
- D. It results in a compilation error

Arrays

- Pitfall: An array in C does not know its own length, & bounds not checked!
 - Consequence: We can accidentally access off the end of an array.
 - Consequence: We must pass the array and its size to a procedure which is going to traverse it.
- Segmentation faults and bus errors:
 - These are VERY difficult to find, so be careful.

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.

Pointer Arithmetic Question

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 == NULL)
- X. compare pointer to NULL (ptr == NULL)

#invalid

A: 1

B: 2

C: 3

D: 4

E: 5

Pointer Arithmetic Question

How many of the following are invalid?

- I. pointer + integer ($\text{ptr} + 1$)
- II. integer + pointer ($1 + \text{ptr}$)
- III. pointer + pointer ($\text{ptr} + \text{ptr}$)
- IV. pointer - integer ($\text{ptr} - 1$)
- V. integer - pointer ($1 - \text{ptr}$)
- VI. pointer - pointer ($\text{ptr} - \text{ptr}$)
- VII. compare pointer to pointer ($\text{ptr} == \text{ptr}$)
- VIII. compare pointer to integer ($1 == \text{ptr}$)
- IX. compare pointer to 0 ($\text{ptr} == \text{NULL}$)
- X. compare pointer to NULL ($\text{ptr} == \text{NULL}$)

#invalid

A: 1

B: 2

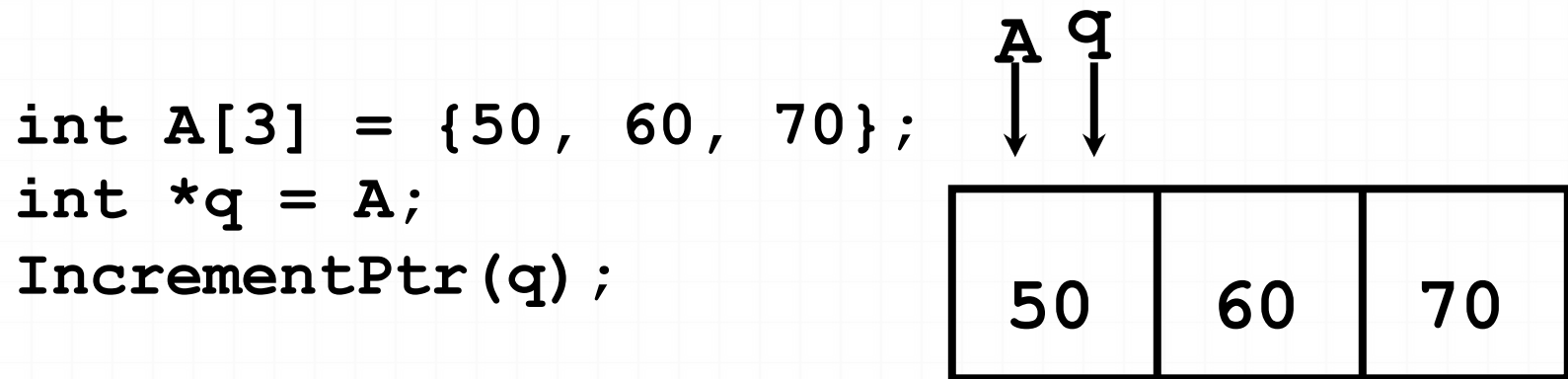
C: 3

D: 4

E: 5

```
void IncrementPtr(int *p) {  
    p = p + 1; }  
}
```

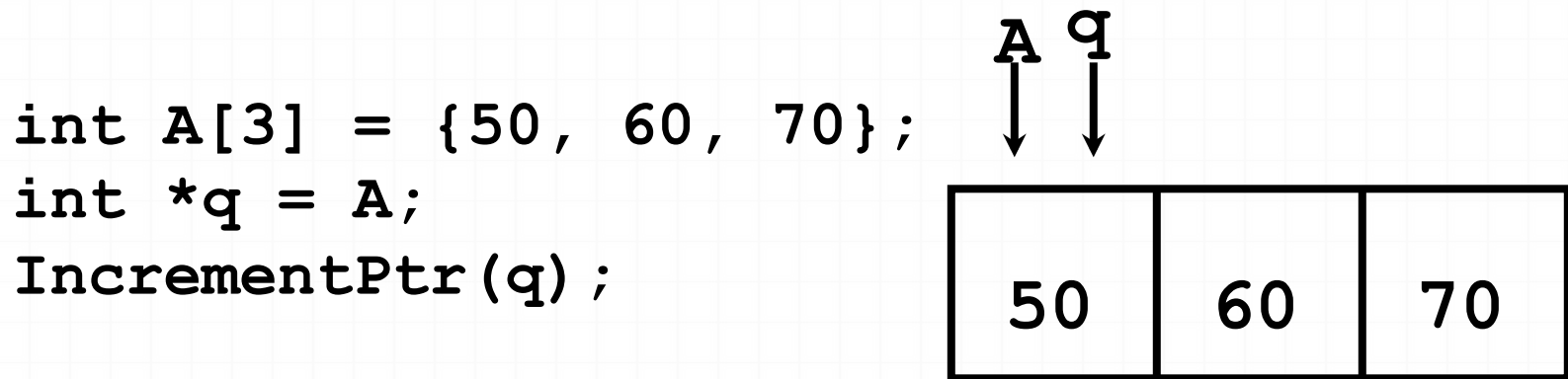
Q: What happens when **IncrementPtr(q)** is called in the following code:



- A. The pointer 'q' points to the next element in the array with value 60
- B. The pointer 'q' points to the first element in the array with value 50

```
void IncrementPtr(int *p) {  
    p = p + 1; }  
}
```

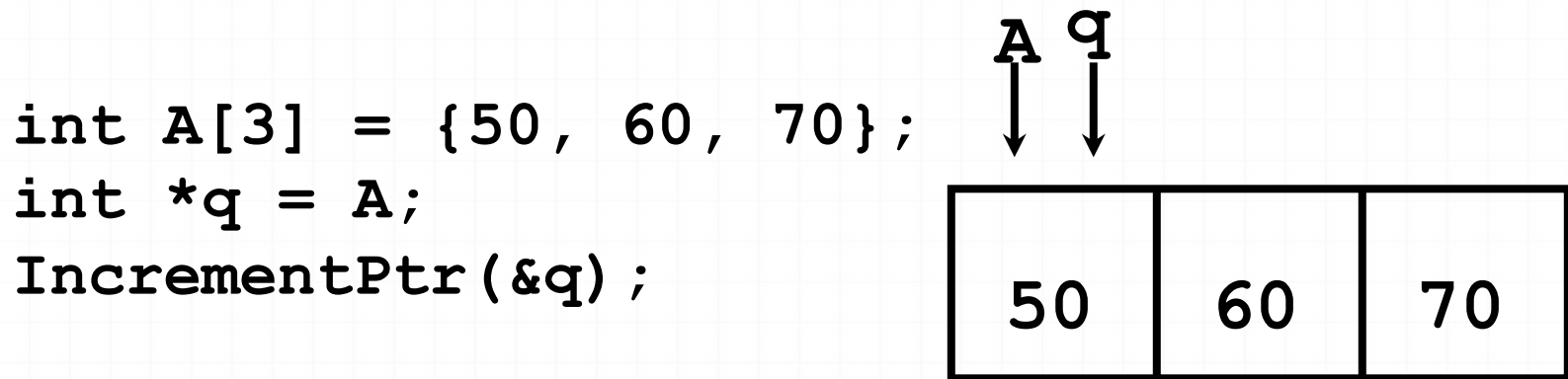
Q: What happens when **IncrementPtr(q)** is called in the following code:



- A. The pointer 'q' points to the next element in the array with value 60
- B. The pointer 'q' points to the first element in the array with value 50

```
void IncrementPtr(int **p) {  
    p = p + 1; }  
}
```

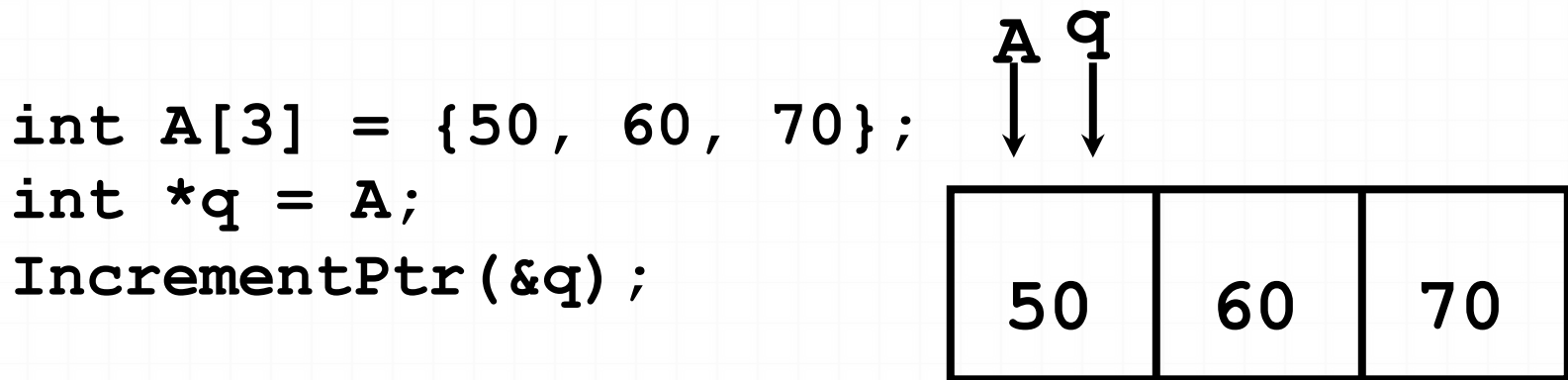
Q: How should we implement `IncrementPtr()`, so that 'q' moves by one element when the following code executes?



- A.** `p = p + 1;` //The current one is correct
- B.** `&p = &p + 1;`
- C.** `*p = *p + 1;`
- D.** `*p++;`
- E.** `p = &p+1;`

```
void IncrementPtr(int **p) {  
    p = p + 1; }  
}
```

Q: How should we implement `IncrementPtr()`, so that 'q' moves by one element when the following code executes?



- A. `p = p + 1;` //The current one is correct
- B. `&p = &p + 1;`
- C. `*p = *p + 1;`
- D. `*p++;`
- E. `p = &p+1;`

C Strings

- A string in C is just an array of characters.

```
char string[] = "abc" ;  
char string[20];
```

- How do you tell how long a string is?
 - Last character is followed by a 0 byte (null terminator)

```
int strlen(char s[])  
{  
    int n = 0;  
    while (s[n] != 0) n++;  
    return n;  
}
```

Q: Why can we not copy strings using the function below?

```
void copy (char sTo[], char sFrom[]) {  
    sTo = sFrom;  
}
```

- A.** **'sTo'** is an array, therefore its value cannot be changed
- B.** We can use the assignment statement but we have to allocate space for **'sTo'** prior to the assignment
- C.** The change in the value of **'sTo'** is not reflected in the calling function

Q: Why can we not copy strings using the function below?

```
void copy (char sTo[], char sFrom[]) {  
    sTo = sFrom;  
}
```

- A.** 'sTo' is an array, therefore its value cannot be changed
- B.** We can use the assignment statement but we have to allocate space for 'sTo' prior to the assignment
- C.** The change in the value of 'sTo' is not reflected in the calling function

Q: Why can we not compare strings using the function below?

```
int compare(char sT1[], char sT2[]) {  
    return(sT1 == sT2);  
}
```

- A. We cannot have an arithmetic expression in the return statement
- B. The function only compares the base address of the strings, not their value
- C. The function only compares if the first element of the two strings are the same

Q: Why can we not compare strings using the function below?

```
int compare(char sT1[], char sT2[]) {  
    return(sT1 == sT2);  
}
```

- A. We cannot have an arithmetic expression in the return statement
- B. The function only compares the base address of the strings, not their value
- C. The function only compares if the first element of the two strings are the same

C String Standard Functions

- `int strlen(char *string);`
 - compute the length of `string`
- `int strcmp(char *str1, char *str2);`
 - return 0 if `str1` and `str2` are identical (how is this different from `str1 == str2`?)
- `int strcpy(char *dst, char *src);`
 - copy the contents of string `src` to the memory at `dst`. The caller must ensure that `dst` has enough memory to hold the data to be copied.
- Defined in the header file `string.h`