CSE 30: Computer Organization and Systems Programming

Lecture 7:
Pointers and arrays
Pointer Arithmetic

Diba Mirza
University of California, San Diego
Q: What happens when we execute the following code?

```c
int *p;
*p = 5;
```

A. \[ \begin{array}{c}
p \\ 5 \end{array} \]  
B. \[ \begin{array}{c}
p \\ 5 \end{array} \]  

C. Compile time error  
D. Runtime error
Segmentation faults (aka segfault)

• Indicates that your program has crashed!
• What caused the crash?
  – Segfaults occur if your program is trying to read or write an illegal memory location.
Q: What is the output of this code?

```c
int *p, x = 5;
p = &x;
printf("%d",(*p)++);
```

A. The value pointed to by p, which is 5
B. The value pointed to by p plus one, which is 6
C. Undefined
D. Compiler error
E. Segmentation fault
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
   ➢ Null check before dereferencing
Array Basics

```
int ar[5]; // declares a 5-element integer array
```

![Array representation with values 100, 104, 108, 112, 116]
Array Basics

int ar[5];  // declares a 5-element integer array
int ar[] = {795, 635};  // declares and fills a 2-element integer array.
Array Basics

- Accessing elements:
  \[\text{ar}[i]\]; \text{// returns the } i^{\text{th}} \text{ element}

- How are arrays in C different from Java?
Arrays and Pointers

- \( ar \) is a pointer to the first element
- \( ar[0] \) is the same as \( \ast ar \)
- \( ar[2] \) is the same as \( \ast (ar+2) \)
- Use pointers to pass arrays to functions
- Use pointer arithmetic to access arrays more conveniently
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.
What is printed by the code below:
printf("%d, %d, %d\n", *ptr+1, *ptr++, *(ptr+1));

A. 21, 20, 60
B. 21, 21, 40
C. 21, 40, 40

Post/pre increment (++) has a higher precedence over indirection (*).
Q: Which of the assignment statements produces an error at compilation. Why?

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

i) p=ar+5;
ii) 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: What happens when the following code is executed?

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

A. Always results in a segmentation fault because a pointer cannot be used to change the value of an array element

B. Always results in a segmentation fault because the array element being accessed is out of bounds

C. 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.
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

- What if we have an array of large structs (objects)?
  - C takes care of it: In reality, \( p + 1 \) 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.