Lecture 8: 
Pointer Arithmetic (review) 
Endianness 
Functions and pointers

Diba Mirza 
University of California, San Diego
Q: Which of the assignment statements produces an error at compilation. Why?

```
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)
Promotion and truncation

• Often arithmetic expressions contain mixed integral types
• Promotion: In an expression all variables are ‘promoted’ to the data type of the largest size

```c
char c=4;
int i= c+ 10; /* c is promoted to an int */
/*What is the value in i? */
```

• Truncation: If the value of a larger data type is assigned to a smaller one, the compiler drops the most significant bits

```c
short i= 0x0104;
char c =i; /*The value in i is truncated*/
/* c =0x04 – Least significant byte is assigned */
```
Type casting

• If we want the variables in an expression to be interpreted differently, use casting

• Example:
  int i=10;
  float f = i/3;

  – In this case, f=3.0
  – If we don’t want the result of the division to get truncated to an int, use explicit casting

  int i=10;
  float f = (float)i/3; /* Explicit cast of int to float */
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*/
```
Byte ordering in memory

Consider the 4 byte number 0x0a0b0c0d:

- **Big Endian**: Most significant byte in lowest memory address
- **Little Endian**: Least significant byte in lowest memory address

```plaintext
0a 0b 0c 0d
```

Increasing memory addresses

```plaintext
0a 0b 0c 0d
```

Increasing memory addresses
What is the output of the following code?

```c
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;
char arr[2] = {0x01, 0x04};
p = (char *) &arr;
printf("Value in p is: %x", *p);
```

A. 0x04  
B. 0x01  
C. Depends
Consider the following function

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

void swap(int x, int y) {
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```
int main()
{
    int a=10, b=20;
    swap(a, b);
}

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

Q: Which of the following changes would you make 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 and modify the swap function
D. Move the implementation in swap to the main function
Functions: Pass by reference

void swap(int *x, int *y) {

}

Q: What should the modified swap function do?

A. Swap the values in ‘x’ and ‘y’

B. Swap the values pointed to by ‘x’ and ‘y’

C. Both the above operations are equivalent
Q: What happens when IncrementPtr(q) is called in the following code:

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

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

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

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

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

A. `p = p + 1;`
B. `&p = &p + 1;`
C. `*p = *p + 1;`
D. `p = &p + 1;`
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.