CSE 30: Computer Organization and Systems Programming

Lecture 9:
Functions and pointers
Dynamic memory allocation

Diba Mirza
University of California, San Diego
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;
}
```

```c
main() {
  int a = 10, b = 20;
  swap(a, b);
  3
  The values stored in 'a' & 'b' remain unchanged after the call to swap
}
```
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

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

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 \texttt{IncrementPtr} (\texttt{q}) is called in the following code:

\begin{verbatim}
void IncrementPtr(int *p){
    p = p + 1; }
\end{verbatim}

\begin{verbatim}
int A[3] = {50, 60, 70};
int *q = A;
IncrementPtr(q);
\end{verbatim}

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 and ‘p’ points to the second element of the array before \texttt{IncrementPtr} returns. However, after the function returns ‘p’ is lost (reclaimed by stack).
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;
}

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;
```
Dynamic memory allocation

void* malloc (size_t s);

- Allocates a block of ‘s’ bytes on the heap and returns the starting address of the block
- If memory cannot be successfully allocated, returns a NULL.

/* Allocate memory for an integer array of size 5 */

int *arr = malloc (5 * sizeof (int));
Dynamic memory allocation

```c
void foo () {
    int * arr1 = malloc (8);
    int arr2[2];
}
```

What is the difference between the two statements above, if both appear within function scope?

`arr1` points to a block created on the heap.
However, the variable `arr1` is created on the stack.
`arr2` and all its elements reside on the stack.
Dynamic memory allocation

```c
void* calloc(size_t n, size_t s);
```

- Allocates a block of memory on the heap to store ‘n’ elements, each of ‘s’ bytes and returns the starting address of the block.
- If memory cannot be successfully allocated, returns a NULL
- Initializes memory to zero (not guaranteed)

/* Allocate memory for an integer array of size 5 */

```c
 int *arr = calloc(5, sizeof(int));
 int *arr = malloc(5 * sizeof(int));
```
Dynamic memory allocation

```c
void* realloc (void * p, size_t s);
```

- Resize an existing block of memory pointed to by ‘p’ to a total size of ‘s’ bytes.
- Returns NULL on failure
- Are the contents copied over to the new block?

```c
// Assume p initially points to a block of 5 bytes
int *q = realloc (p, 10);
/* If successful returns the address of a new block of size 10 bytes that contains the values stored in the old block */
/* If the new block is at a new memory location, the old block is automatically freed */
```
`void free(void * ptr);`

- Frees the heap block pointed to by `p`
- What happens in the following code?

A double free results in a runtime error.

```c
int *p = malloc(8);
free(p);
free(p);
```

// Does not cause a runtime error if `p` is `NULL`!