Lecture 22

Pointers-4

Pointers and Functions

Pass By Value

```
void increment(int x)
{   x++;
}

int main() {
    int a = 5;
    increment(a);
    // a is still 5
    return 0;
}
```

Pass By Reference

```
void increment(int *x)
{   (*x)++;
}

int main() {
    int a = 5;
    increment(&a);
    // a is now 6
    return 0;
}
```

Pass-by-value v Pass-by-ref

Use pass-by-value when you don’t need to modify the original variable.	Use pass-by-reference (passing pointers) when you need to modify the original variable or for efficiency with large data structures.

Quiz

```c
#include <stdio.h>

int main() {
    int a = 5;
    int b = 5;
    int c = 5;
    int d = 5;
    int e = 5;
    int f = 5;

    // Pass-by-Value
    incrementByValue(a);
    printf("Pass-by-Value:\nValue of a: %d\n\n", a);

    // Pass-by-Reference
    incrementByReference(&b);
    printf("Pass-by-Reference:\nValue of b: %d\n\n", b);

    // Returning an Integer Pre
    int result = incrementAndReturnPre(c);
    printf("Returning an Integer:\nValue of c: %d\nResult: %d\n\n", c,
result);
    // Returning an Integer Post
    int result = incrementAndReturnPost(d);
    printf("Returning an Integer:\nValue of c: %d\nResult: %d\n\n", c,
result);
    // Post-Increment
    postIncrement(&e);
    printf("Post-Increment:\nValue of d after post-increment: %d\n\n", d);

    // Pre-Increment
    preIncrement(&f);
    printf("Pre-Increment:\nValue of e after pre-increment: %d\n", e);

    return 0;
}
```

```c
// Pass-by-Value
void incrementByValue(int x) {
    x++;
}

// Pass-by-Reference
void incrementByReference(int *x)
{   (*x)++;
}

// Returning an Integer Pre
int incrementAndReturnPre(int x) {
    return ++x;
}

// Returning an Integer Post
int incrementAndReturnPost(int x)
{   return x++;
}

// Post-Increment
void postIncrement(int *x) {
    (*x)++;
}

// Pre-Increment
void preIncrement(int *x) {
    ++(*x);
}
```

Returning Pointer from function

```
int* allocateMemory() {
    int *ptr = (int *)malloc(sizeof(int));
    *ptr = 30;
    return ptr;
}

int main() {
    int *p = allocateMemory();
    // use p
    free(p); // Don't forget to free the allocated
memoreturn 0;
}
```

Returning Pointer from function

```
int* dangerousFunction() {
    int x = 10;
    return &x; // Dangerous! x will be out of scope after function
}returns

int main() {
    int *ptr = dangerousFunction();
    // ptr now points to an invalid location
    return 0;
}
```

Multiple Return from a function

String Pointers

```c
#include <stdio.h>

int main() {
    char str[] = "Hello, World!";
    char *ptr = str;

    // Print the string using the pointer
    while (*ptr != '\0') {
        printf("%c", *ptr);
        ptr++;
    }
    printf("\n");

    return 0;
}
```

String Pointers

```
char str[] = "Hello";
str[0] = 'h'; // Allowed

char *str = "Hello";
str[0] = 'h'; // Undefined behavior, as string literals are read-
only
```

Array of pointers

```c
#include <stdio.h>

int main() {
    char *fruits[] = {"Apple", "Banana",
"Cherry"};
    for (int i = 0; i < 3; i++) {
        printf("%s\n", fruits[i]);
    }

    return 0;
}
```

Function Pointer

```c
#include <stdio.h>
// Function prototypes
int add(int a, int b);
int subtract(int a, int b);
int main() {
    // Function pointer declaration
    int (*operation)(int, int);
    // Variables
    int a = 10;
    int b = 5;
    int result;
    // Pointing to the add function
    operation = add;
    result = operation(a, b);
    printf("Addition Result: %d\n", result);
    // Pointing to the subtract function
    operation = subtract;
    result = operation(a, b);
    printf("Subtraction Result: %d\n",
result);
    return 0;
}
// Function definitions
int add(int a, int b) {
    return a + b;
}
int subtract(int a, int b) {
    return a - b;
}
```

Function as parameter

```c
#include <stdio.h>

// Function prototypes
int add(int a, int b);
int subtract(int a, int b);
void executeOperation(int (*operation)(int, int), int a, int b);

int main() {
    // Variables
    int a = 10;
    int b = 5;

    // Execute add operation
    printf("Executing Add Operation:\n");
    executeOperation(add, a, b);

    // Execute subtract operation
    printf("Executing Subtract Operation:\n");
    executeOperation(subtract, a, b);

    return 0;
}

// Function definitions
int add(int a, int b) {
    return a + b;
}

int subtract(int a, int b) {
    return a - b;
}

void executeOperation(int (*operation)(int, int), int a, int b)
{   int result = operation(a, b);
    printf("Result: %d\n", result);
}
```

Problem

```
Write a program in C to swap elements using call by
reference
```

Solution

```c
#include <stdio.h>
// Function prototype
void swap(int *a, int *b);
int main() {
    int num1, num2;
    // Input numbers
    printf("Enter first number: ");
    scanf("%d", &num1);
    printf("Enter second number: ");
    scanf("%d", &num2);
    // Display numbers before swapping
    printf("\nBefore swapping:\n");
    printf("First number = %d\n", num1);
    printf("Second number = %d\n", num2);
    // Call swap function
    swap(&num1, &num2);
    // Display numbers after swapping
    printf("\nAfter swapping:\n");
    printf("First number = %d\n", num1);
    printf("Second number = %d\n", num2);
    return 0;
}
// Function definition to swap values using call by reference
void swap(int *a, int *b) {
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

BUET TF 2021-22

4. (a) Re-declare the following using pointer and malloc() function. Do not use [] operator. **(5)**

```
double A[10][20][30];
```

(b) Consider the following declaration: **(4+6=10)**

```
int p[2][3][2] ={ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

If memory location p contains a value of B040 (in hexadecimal) and an integer in represented by 2 bytes, then show the addresses of each of the integers in the list.

Also write down the values of the following expressions:

(i) **p (ii) *(*(*p+1)+2) (iii) *(**(p+2)) (iv) *(*p+1)+1