Code notes (c++ 19-11-2025)

---

tags:

- cpp
- 2D Arrays
- pointers
- memory
- structs
- scope

C++ Lecture Code Snippets

1. Returning Multiple Values

Method 1: Global Variables & Method 2: Static Local

```
#include<stdio.h>

// --- Method 1: Global Variable ---
int arr[2]={0,0}; // Scope: Global, Lifetime: App

void mult_sum(int x, int y){
    arr[0]=x+y;
    arr[1]=x*y;
}

// --- Method 2: Static Local Variable ---
int * mult_sum2(int x, int y){
    static int arr_local[2]={0,0}; // Lifetime: Persistent
    arr_local[0]=x+y;
    arr_local[1]=x*y;
    return arr_local; // Returns address of static array
}

int main() {
    // Using Static Method
    int * ptr = mult_sum2(3,5);
    printf("%d", ptr[0]);
    printf("%d", ptr[1]);
```

```
    return 0;
}
```

Method 3: Pass by Address (Pointers)

```
void mult_sum3(int x, int y, int *s , int *m){
    *s = x + y;
    *m = x * y;
}

void mult_sum4(int x, int y, int * ptr){
    ptr[0] = x + y;
    ptr[1] = x * y;
}

int main() {
    int sum=0, mult=0;
    int arr[2];

    mult_sum3(5, 4, &sum, &mult);
    mult_sum4(5, 4, arr);
    return 0;
}
```

---

2. Pass by Value vs. Pass by Address (Swap)

```
// Pass by Address (Works)
void swap(int * x, int *y){
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}

// Pass by Value (Does not work for swapping outside)
void swap2(int x, int y){
    int temp;
    temp = x;
    x = y;
    y = temp;
}

int main() {
    int x=5;
    int y=10;
```

```
    swap2(x,y);   // No change in x, y
    swap(&x,&y);  // Changes x, y
    return 0;
}
```

---

3. Scope, Lifetime & Memory Casting

Variable Scope & Shadowing

```
#include <iostream>
using namespace std;

int data_print=20; // Global Scope

int main() {
    int x=10;
    {
        int x=5; // Shadowing / Redeclaration
        cout << "x=" << x; // Prints 5
    }
    cout << "x=" << x; // Prints 10
    return 0;
}
```

Pointers & Endianness Check

```
int main() {
    unsigned int xx = 0x000000FF; // 4 Bytes
    unsigned int * ptr1 = &xx;

    // Casting int pointer to char pointer (byte access)
    unsigned char * ptr2 = (unsigned char*)&xx;

    if(ptr2 != NULL && ptr1 != NULL){
        for(int i=0; i<sizeof(int); i++){
            printf("%d\n ", *(ptr2+i)); // Access individual bytes
            printf("============\n");
        }
        printf("%d %d", sizeof(ptr1), sizeof(ptr2));
    }
    return 0;
}
```

---

4. Strings & Const Correctness

String Literal vs Array

```c
int main() {
    // 1. Pointer to String Literal (Read-Only)
    char * ptr = "mina";
    // ptr[0]='a'; // INVALID: Runtime Error
    ptr = "ali";   // VALID: Changing pointer address

    // 2. Array of Characters (Stack Memory)
    char arr[10] = "mina";
    arr[0] = 'a';  // VALID
    // arr = "ali"; // INVALID: Cannot reassign array address
    return 0;
}
```

Custom StrLen & Const Pointers

```c
// Const correctness
int strLen(const char * const string){
    int i=0;
    while(string[i]!='\0'){
        i++;
    }
    return i;
}

/* Const Pointer Rules:
   int * const ptr = &value;      // Constant Pointer (Cannot change address)
   const int * ptr = &value;      // Pointer to Constant (Cannot change value)
   const int * const ptr = &value;// Constant Pointer to Constant
*/
```

---

5. Arrays & Pointers (2D)

```c
#define ROW 3
#define COL 4

int main() {
    char str[ROW][20] = {{'M','i','n','a','s','s'}, {"Ahmed"}};
```

```
    // Accessing Elements
    cout << str[0] << endl;
    printf("%c \n", *((*(str+1))+1)); // Pointer arithmetic access

    // Array of Pointers (jagged-like access)
    char *ptrArr[ROW] = {&str[0][0], &str[1][0], &str[2][0]};

    // Array of Pointers to String Literals
    char *ptrArr2[ROW] = {"ahmed", "mark", "ali"};
    ptrArr2[1] = "Mina+Ahmed"; // Valid
    // ptrArr2[0][1] = 's';    // Invalid (Read-only memory)

    // Pointer to Array
    char(*ptrToArr)[COL] = (char(*)[4])str; // Type casting for demo

    return 0;
}
```

---

6. Structs

Struct Definition & Typedef

```
typedef struct date{
    int day;
    int month;
    int year;
} date;

typedef struct Student{
    char firstName[20];
    char lastName[20];
    int age;
    char gender;
    Student * s; // Pointer to self
    int degree;
} Student;

typedef struct Employee{
    char firstName[20];
    char lastName[20];
    date bod; // Nested Struct
} Employee;
```

Struct Functions (Pass by Address vs Value)

```
// Pass by Value (Copy)
Student enterDataStudent(){
    Student s1;
    cin >> s1.firstName;
    return s1;
}

// Pass by Address (Pointer - Efficient)
void enterDataStudent2(Student * s1){
    cin >> s1->firstName; // Arrow operator
    cin >> s1->degree;
}

void displayEmployees(Employee e[], int sizeEmployee){
    for(int i=0; i<sizeEmployee; i++){
        cout << e[i].firstName << " " << e[i].bod.year << endl;
    }
}

int main() {
    Employee m[3]; // Array of Structs
    // enterEmployees(m, 3);
    // displayEmployees(m, 3);
    return 0;
}
```

C++ Memory & Data Structures Map

1. Variables: Scope & Lifetime

Type	Keyword	Scope	Lifetime
Local	`auto`	Block `{ }`	Function Execution
Global	`extern`	File / App	Application Run
Static Local	`static`	Block `{ }`	Application Run
Static Global	`static`	File Only	Application Run

---

2. Function Arguments

- **Call By Value**
  - Copy Mechanism
- **Call By Address**
  - Pointer Mechanism
  - **Use Cases:**
    - Swap
    - Return Multiple Values
    - Passing Arrays

---

3. Pointers

A. Operations

- **Address of:** `&var`
- **Dereference:** `*ptr`
- **Arithmetic:** `ptr++`, `ptr + n`

B. Types & Casting

- **Char *to Int*:** `(int*)charPtr`
- **Void Pointer:** `void* ptr` (Generic / No Dereference without cast)

C. Pointer States

- **Null:** `ptr = NULL`
- **Wild:** `int *ptr;` (Uninitialized)
- **Dangling:** Points to freed memory

D. Const Correctness

Syntax	Pointer Modifiable?	Value Modifiable?
`const int * p`	✅ Yes	❌ No
`int * const p`	❌ No	✅ Yes
`const int * const p`	❌ No	❌ No

E. Strings

- **String Literal:** `char *p = "Text";` (Read-Only)
- **Char Array:** `char arr[] = "Text";` (Read-Write)

F. Pointers & 2D Arrays

- **Array of Pointers:** `int *arr[ROW];`
- **Pointer to Array:** `int (*ptr)[COL];`
- **Pointer to Pointer:** `int **ptr;` (Dynamic Arrays / Array of Pointers)
    - *Note: Cannot point directly to Static 2D Array*

---

4. Structs

A. Definition

```
typedef struct Student {
    char name[20];
    struct Student *next; // Self-Referential (Data Structure)
} Student; ```
```