DISCUSSION SESSION
WEEK 7

EXAM-02 REVIEW

1-DIMENSIONAL ARRAYS

- Arrays store data (elements) of the **same type** (that is, an array cannot store both integers and doubles, and so on..). Elements are stored in a sequence.
- Arrays are indexed from 0…size - 1
- Each element in an array can be accessed using its index (inside square brackets **[ ]**)
- The size of an array must be **determined at compile-time,** so the compiler knows how much memory to allocate for the array elements. **The size of an array cannot be changed!**

ARRAY DECLARATION & ACCESSING

```
int arr[5];
int arr[ ] = {2, 3, 4, 2};

int n = 8;
int arr[n];

int myArr[6] = {3, 4, 5, 6, 5, 2};
std::cout << myArr[2] << std::endl;
myArr[4] = 8;
```

Passing Array into Function

- Arrays are automatically passed into functions by reference. So, any changes made to the array within the function will be reflected in the original array.
- In the function parameters, it's best to use empty brackets and pass in the array size separately as another parameter.

```
void func(int myArr[ ], int arrSize) {
        // some code..
}
void func(int myArr[5], int arrSize) {
        // some code..
}
```

2-DIMENSIONAL ARRAYS

A 2D array is *just* an array of arrays. We use multidimensional arrays to store a grid of items, like a chessboard or a spreadsheet.

2-D ARRAY DECLARATION

```
int myGrid[3][5]; // creates a 2-D array with 3 rows, 5 cols
int myGrid[2][3] = {0}; // creates a 2-D array with 2 rows,
                        // 3 cols, all elements initialized to 0
```

2-D ARRAY AS FUNCTION PARAMETER

```
void func(char myArr[ ][5], int rows, int cols) {
    // some code..
}
```

- The second square brackets of array declaration parameter **cannot** be empty

POINTERS

A pointer is a special type of variable representing a memory address. A pointer should always be set equal to the memory address of another variable.

Note: If the ampersand is not included in the declaration of a variable, it is considered a get-address operator.

```
int a = 5;
std::cout << &a;  // prints out the memory address of a
int* ptr = &a; // pointer holding the address of a
```

```
int a = 5;
int* b = &a;   // b is holding the memory address of a
```

If the asterisk is not included in the declaration of a variable, it is considered a dereference operator.

Dereferencing a pointer means accessing the data stored at the memory address that the pointer is pointing to.

```
std::cout << *b;    // prints 5
std::cout << *a;    // Invalid because a is not a pointer and
                    // therefore cannot be dereferenced!
```

Passing Pointer into Function

- ✓ Function parameter is a pointer variable; argument is a memory address

```
double func(double a, double* b) {
        return a + *b;
}
int main( ) {
        double a = 3.5, b = 8.5;
        double sum = func(a, &b); // sets sum to 12
}
```

Arrays and Pointers

Arrays are pointers. Array names are constant pointers that point to the base address of the array. The base address of an array is the memory address of the first element of the array.

```
int arr[ ] = {3, 5, 2, 7};
std::cout << arr << std::endl;
std::cout << &arr[0] << std::endl;
```

These two print statements will output the same memory address.

Pointer Arithmetic

Reminder: Elements in arrays are laid out in memory sequentially/contiguously, one right after the other. So, given the base address of an array, we can iterate through the elements of the array.

```
char arr[ ] = {'f', 'y', 'p', 'm', 't'};
std::cout << arr; // prints out base address of array
std::cout << *arr; // dereferences address & prints out f
```

If we add 1 to the base address and dereference it, we can get the next element, and so on:

```
std::cout << *(arr + 1); // prints out y
```

Note: Due to precedence, you must use parentheses.

Arrays as Pointers in Functions

Provided that arrays are pointers, we can explicitly declare our array parameter as a pointer to the base address of the array, and still iterate through the array by indexing or by pointer arithmetic:

```
void func(int* arr, int size) {
    for(int i = 0; i < size; i++) {
        std::cout << arr[i] << std::endl;
    }
}
```

```
void func(int* arr, int size) {
    for(int i = 0; i < size; i++) {
        std::cout << *(arr + i) << std::endl;
    }
}
```

True or False?

1. Arrays in C++ can dynamically change their size after they have been declared**. False.**

2. The name of the array represent the address of the first element of the array. **True.**

3. Pointers in C++ must be initialized at the time of declaration. **False.**

4. Arrays automatically initialize all elements to zero if no initial values are provided. **False.**

5. The size of an array must be known at compile-time. **True.**

True or False: What's the Output?

```
int arr[3] = {3, 2, 1};
int grid[3] = {3, 2, 1};
if(arr == grid) {
        std::cout << "True";
} else {
        std::cout << "False";
}
```

Output: False

True or False?

6. Elements in 1D array are stored contiguously or sequentially, but elements in 2D array are not. **False.**

7. It is possible to return an array from a function. **False.**

8. The sizeof( ) operator returns the number of elements in an array. **False.**

9. A recursive function may not generally have a base case.

False.

10. Multidimensional arrays are stored in as 1-D arrays in row-major order. **True.**

11. In C++, arrays can be passed by value to functions. **False.**

True or False?

12. The size of an array can be determined using the built-in **size( )** function. **False.**

13. A recursive function that does not have a base case will always result in a compilation error. **False.**

14. The address of a pointer is the same as the address the pointer holds. **False.**

15. Given an array `int arr[10];` the expression `arr == &arr[0];` evaluates to true. **True.**

This code crashes when called with func(6). Why?

```
bool func(int num) {
    if(num <= 2) {
        return false;
    }
    return func(num + 1);
}
```

This is an infinite recursive function because it never reaches the base case.

Sense or Nonsense? If sense, what's the output?

```
int size = 7;
int arr[ ] = {1, 3, 8, 11, 9, 2, 5};
```

1. `std::cout << *arr;` **1**
2. `std::cout << arr[size];` **Nonsense**
3. `int* p = arr;  std::cout << **&p;` **1**
4. `std::cout << *(&arr[4]);` **9**
5. `std::cout << *arr + 3;` **4**
6. `std::cout << *size;` **Nonsense**
7. `std::cout << arr[-1];` **Nonsense**

What's the Output?

```
int main( ) {
        int a = 5;
        int* ptr = &a;
        char letter = 'A';
        char& p = letter;
        p += a;
        *ptr += letter;
        std::cout << a << " " << letter;
}
```

Output: 75 F

What's the formula that this mystery function calculates?

```
int mysteryFunc(int n) {
    if(n == 0) {
        return 0;
    }
    return mysteryFunc(n- 1) + 3 * n * n - 3 * n + 1;
}
```

Test the function using different values for *n*

Answer: n^3