National University of Computer & Emerging Sciences
(NUCES) Islamabad
School of Computing

# DATA STRUCTURES – FALL 2023

## Cyber Security Department

## LAB 01

# Learning Outcomes

In this lab you are expected to learn the following:

- Abstract Data Types
- Templates in C++
- 2D Arrays

# Templates in C++

Templates are powerful features of C++ which allow us to write generic programs. We can create a single function to work with different data types by using a template.

**The simple idea** is to pass the data type as a parameter so that we don't need to write the same code for different data types.

**For example:** a software company may need to sort() for different data types. Rather than writing and maintaining multiple codes, we can write one sort() and pass the datatype as a parameter.

Templates are defined using the keyword "template" and angle brackets (e.g., template <typename T>). The data type that the template will work with is specified as a parameter within the angle brackets.

| Function Templates | Class Templates |
|---|---|
| C++ template functions are a type of feature that allows the creation of generic functions. These functions can operate on multiple data types rather than being limited to a specific type. | A class template in C++ is similar to a function template, but it is used to create generic classes |
| Syntax:<br>`template <typename T>`<br>`T findMax(T arr[], int size) {`<br>`// Function definition here`<br>`};` | Syntax:<br>`template <typename T>`<br>`class MyClass {`<br>`// Class definition here`<br>`};` |

In both examples, the "typename T" is the template parameter. It can be named anything, but "T" is a common convention. When the template is instantiated, the type passed as the template argument will replace T.

# Function Templates

## Task 1:

Let A and B be two sets: (Data of sets can be integer, double or float)

A = {10.43,4.3,5.61,6.90,11.57,12.11,3.8,2.4,9.5}

B = {11.01,12.34,16.5,3.8,8.1,2.4,9.11,12.11,6.75,10.43,20.2,2.1,4.3}

Write the following functions:

**a. Print**

Prints all the elements of the array.

**b. Union**

Find the union of A and B. The union of A and B is the set that contains those elements that are either in A or in B, or in both

**c. Intersection**

Find the intersection of A and B. The intersection of A and B is the set that contains those elements that are in both A and B.

e. **Disjoint**

Return true if A and B are disjoint sets (sets having no common elements).

**f. Find Element**

Return True if the Element exists in the set

# Class Templates

**Task 2:** Write the following codes using templates in C++

```
#include <iostream>
#include <string>
using namespace std;
class Numbers{
private:
        float A, int B;
public:
        Numbers(float A1, int B1)
        {
                A=A1;
                B=B1;
        }
        float Asquare()
        {
           float result=A*A;
           return result;
        }
        float division()
        {
          if(B!=0)
                return A/B ;
          else
                return 0;
        }
};
int main()
{
 Numbers num(10.9,3);
 cout <<" A square : "<< num.Asquare()<<endl;
 cout <<" A/B : "<< num.division()<<endl;
}
```

## Task 3: (2D Arrays)

### a. Linear Search:

Write a program to perform linear search in 2D array.
Let arr = [[12,43,66,78], [34,21,72,81], [7,13,2,59]]
Find Element=2

### b. Binary search:

Write a program to perform binary search in 2D array.
Let arr = [[10,20,30,40], [15,25,35,45], [27,29,37,48], [32,33,39,50]]
Find Element=29

## Task 4:

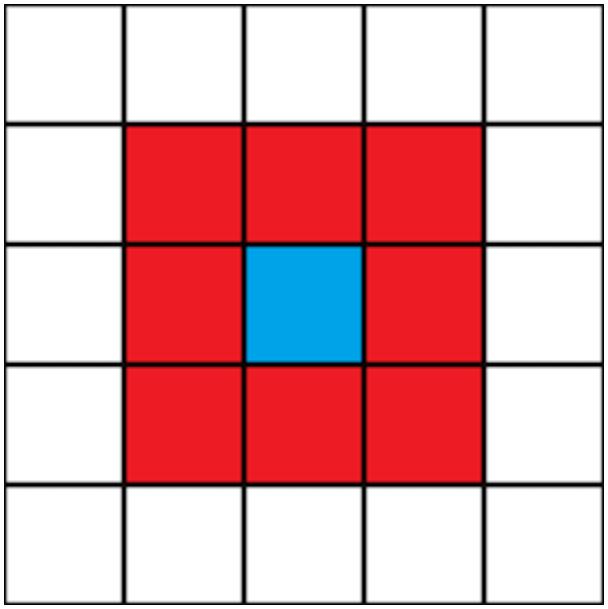
# Game of Life

### Introduction to Life

Game of Life (or just "Life") is not really a game. There's no winning or losing or destroying your opponent mentally and spiritually. Life is a "cellular automaton" - a system of cells that live on a grid, where they live, die and evolve according to the rules that govern their world.

Life's simple, elegant rules give rise to astonishingly complex emergent behavior. It is played on a 2-D grid Each square in the grid contains a cell, and each cell starts the game as either "alive" or "dead". Play proceeds in rounds. During each round, each cell looks at its 8 immediate neighbors and counts up the number of them that are currently alive.

Make a type char 30 x 30 2D grid. Randomly assign active and dead cells. Active cells will have value '*' and dead cell will have value ' '.



In Above diagram Blue cell is the current cell whereas Red cells are its neighboring cells

The cell then updates its own liveness according to 4 rules:

1. Any live cell with 0 or 1 live neighbors becomes dead, because of underpopulation
2. Any live cell with 2 or 3 live neighbors stays alive, because its neighborhood is just right
3. Any live cell with more than 3 live neighbors becomes dead, because of overpopulation
4. Any dead cell with exactly 3 live neighbors becomes alive, by reproduction

Run your code for infinite rounds and observe the pattern changing

And that's all there is to Life. These 4 rules give rise to some unbelievably complex and beautiful patterns, and an equally unbelievable quantity of analysis by Life devotees intent on discovering new ones.

## Useful links:

https://docs.microsoft.com/en-us/cpp/cpp/templates-cpp

https://docs.microsoft.com/en-us/cpp/cpp/class-templates?view=vs-2019