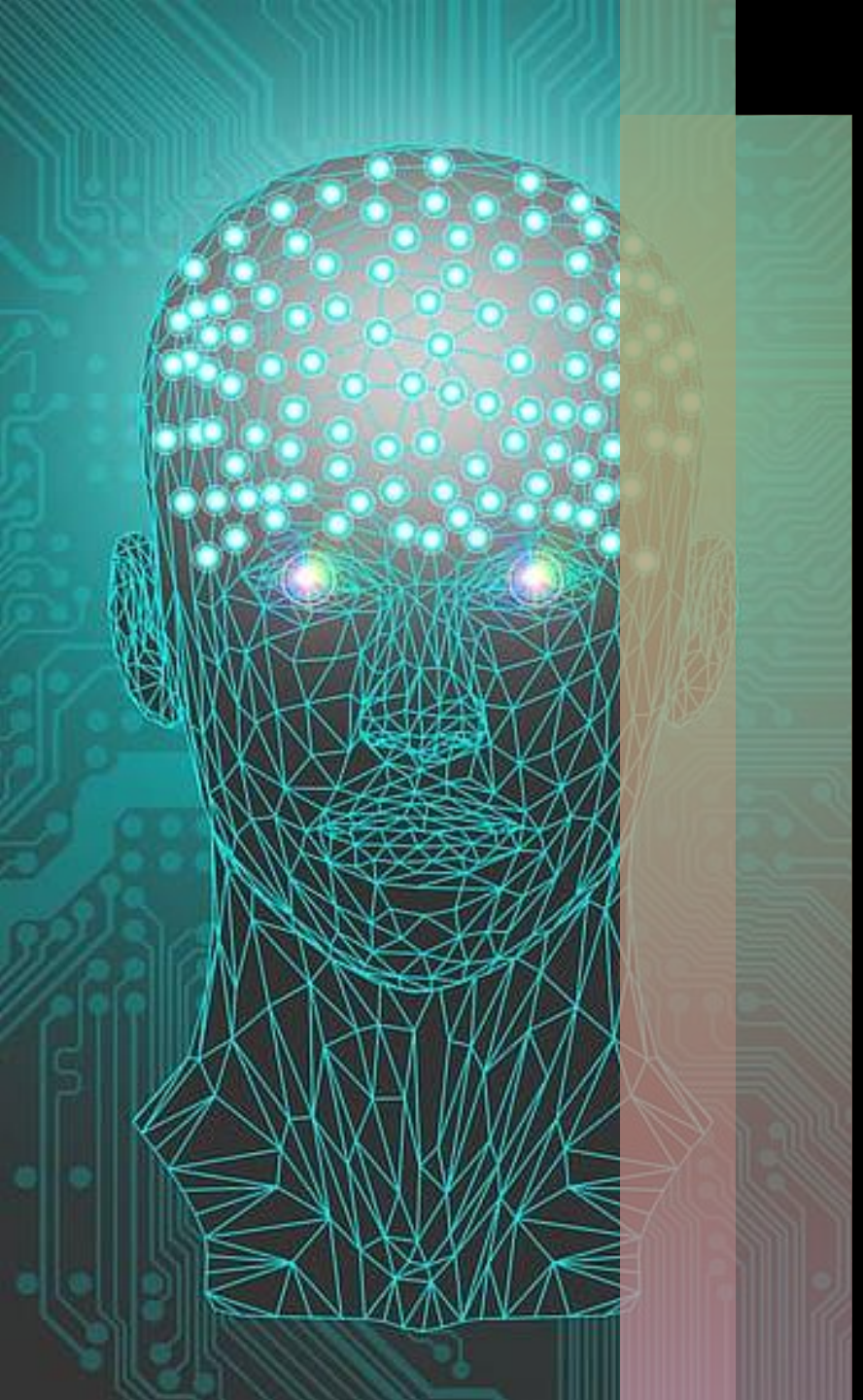
COM4013
Introduction to Software Development

- Week 15
- Searching
- Recursion
- Divide and Conquer

- *Umar Arif – u.arif@leedstrinity.ac.uk*



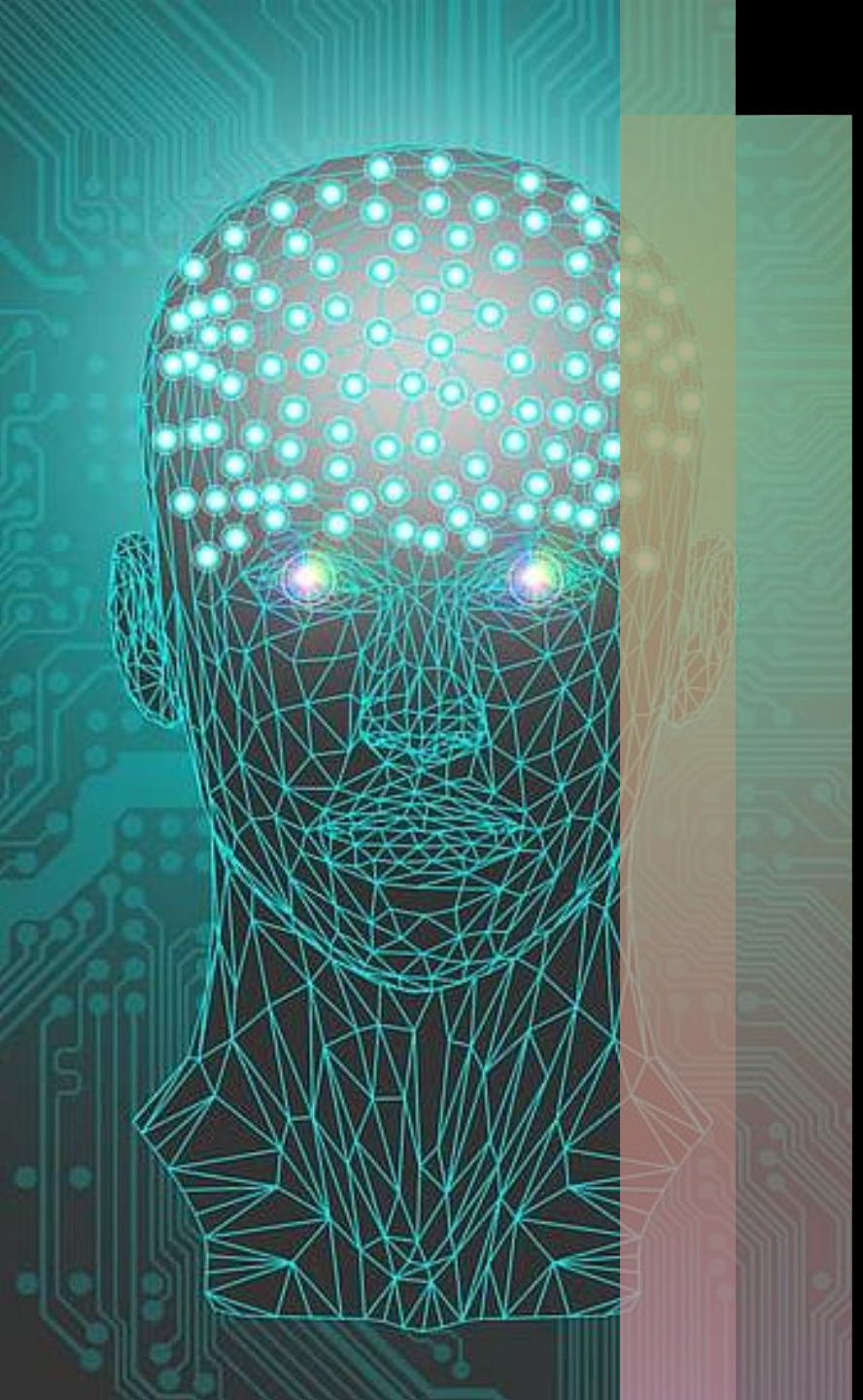
Searching

- We have formally covered arrays and structs
- Let’s consider the task of searching an array for a specific element
- To do this we need to **examine every element** of the array
- A Boolean could be used, to keep track of whether an element exists

```
bool found = false;
```

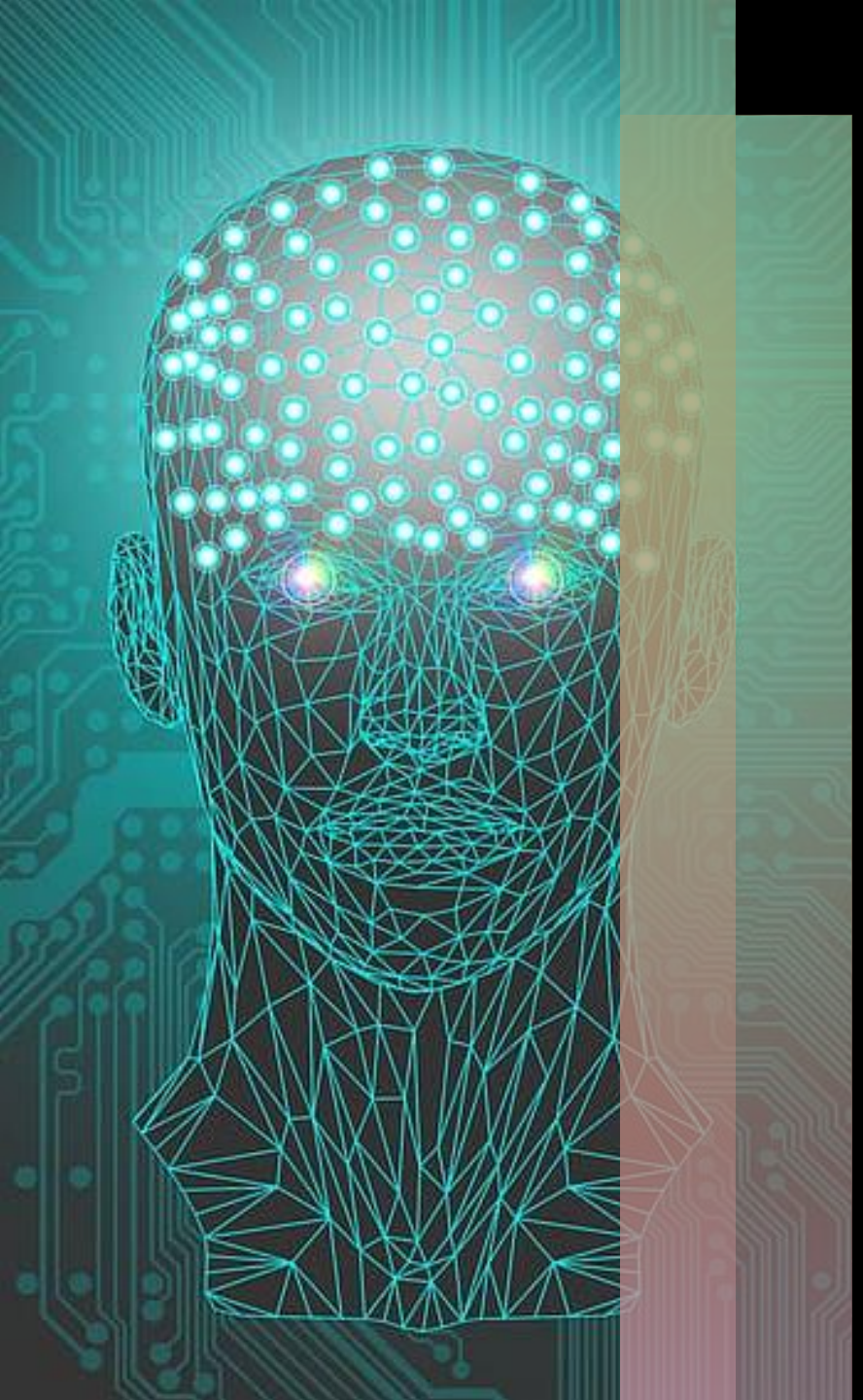
- This would be set to false on entry, and then be set to true if the element is found
- Otherwise, it will remain false telling us that the value has not been found.

- Okay so let's write a function for this...



Searching Function

```
void ExistsInArray(int myArray[SIZE], int searchTerm)
{
    bool found = false;
    for( int i = 0; i < SIZE; i++ )
    {
        if ( myArray[i] == searchTerm )
        {
            found = true;
        }
    }
    if( found == true )
    {
        cout << searchTerm << " found" << endl;
    }
}
```



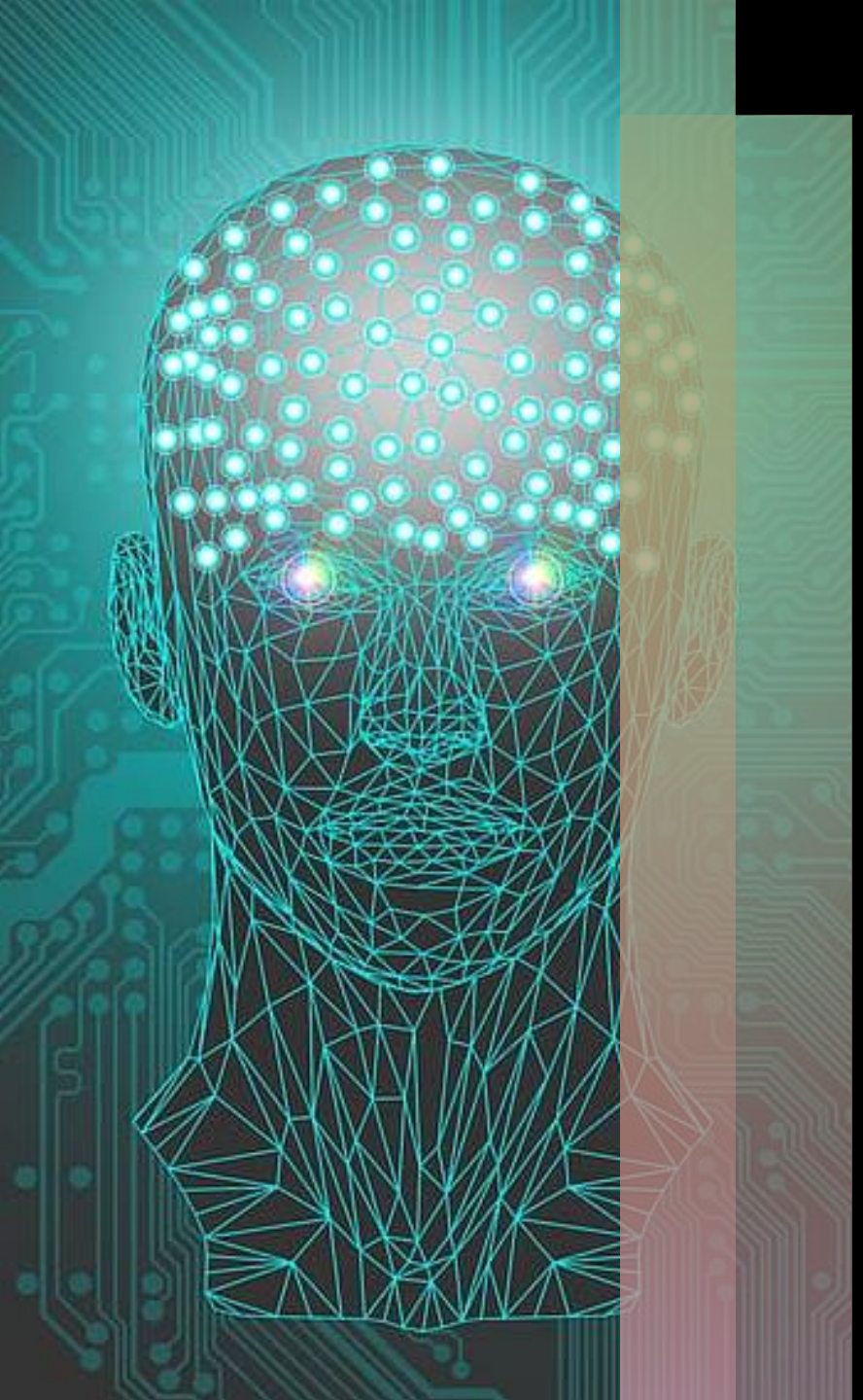
Function Discussion

- The function works.
- However, the search is not efficient.

- Q: What is inefficient about the function?
  - Continuing to search through the array even if the value has been found.

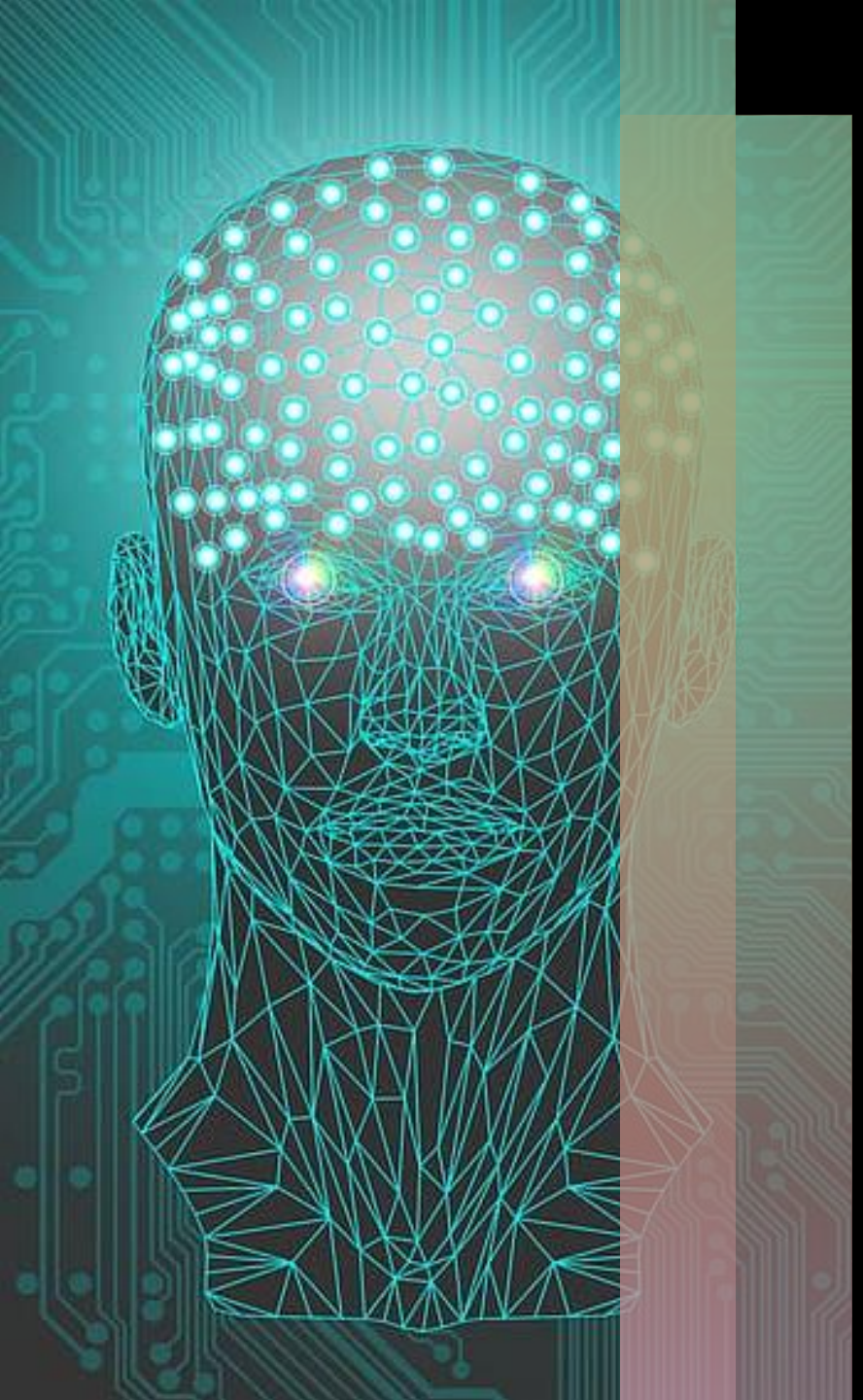
- Instead, we want to end the search as soon as the value has been found
- This will make the function more efficient.

- A "flag" in a program signals its state. Currently, a Boolean is used to indicate whether an element has been found or not
- To enhance efficiency, we can integrate this Boolean as a flag directly into the loop condition, allowing the loop to conclude when the specified value is found. This way, the Boolean acts as a flag, indicating when the loop can terminate.



Searching Function

```
void ExistsInArray(int myArray[SIZE], int searchTerm)
{
    bool found = false;
    for( int i = 0; i < SIZE && found == false; i++ )
    {
        if ( myArray[i] == searchTerm )
        {
            found = true;
        }
    }
    if( found == true )
    {
        cout << searchTerm << " found" << endl;
    }
}
```

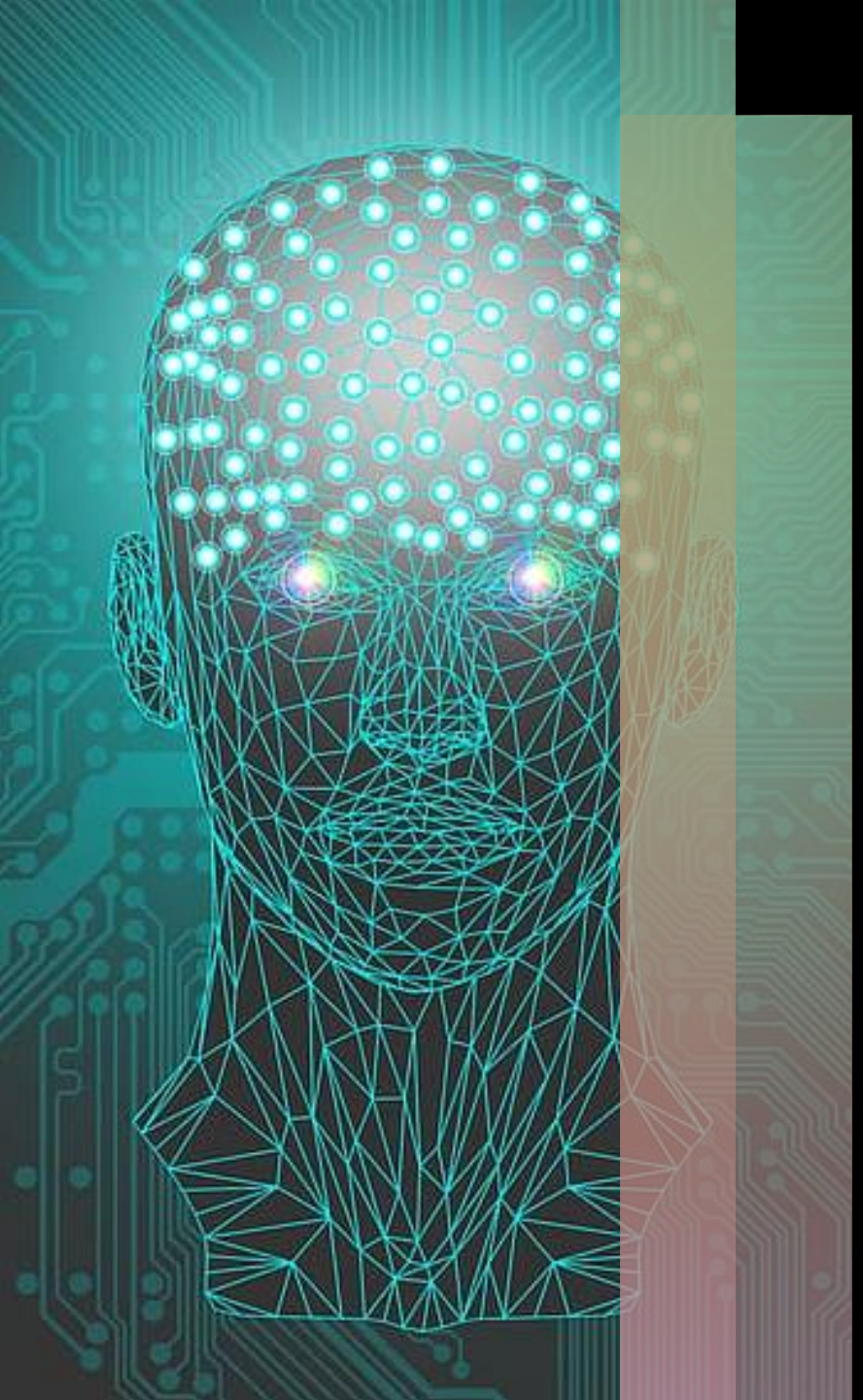


Break

- C++ has a keyword: **break**

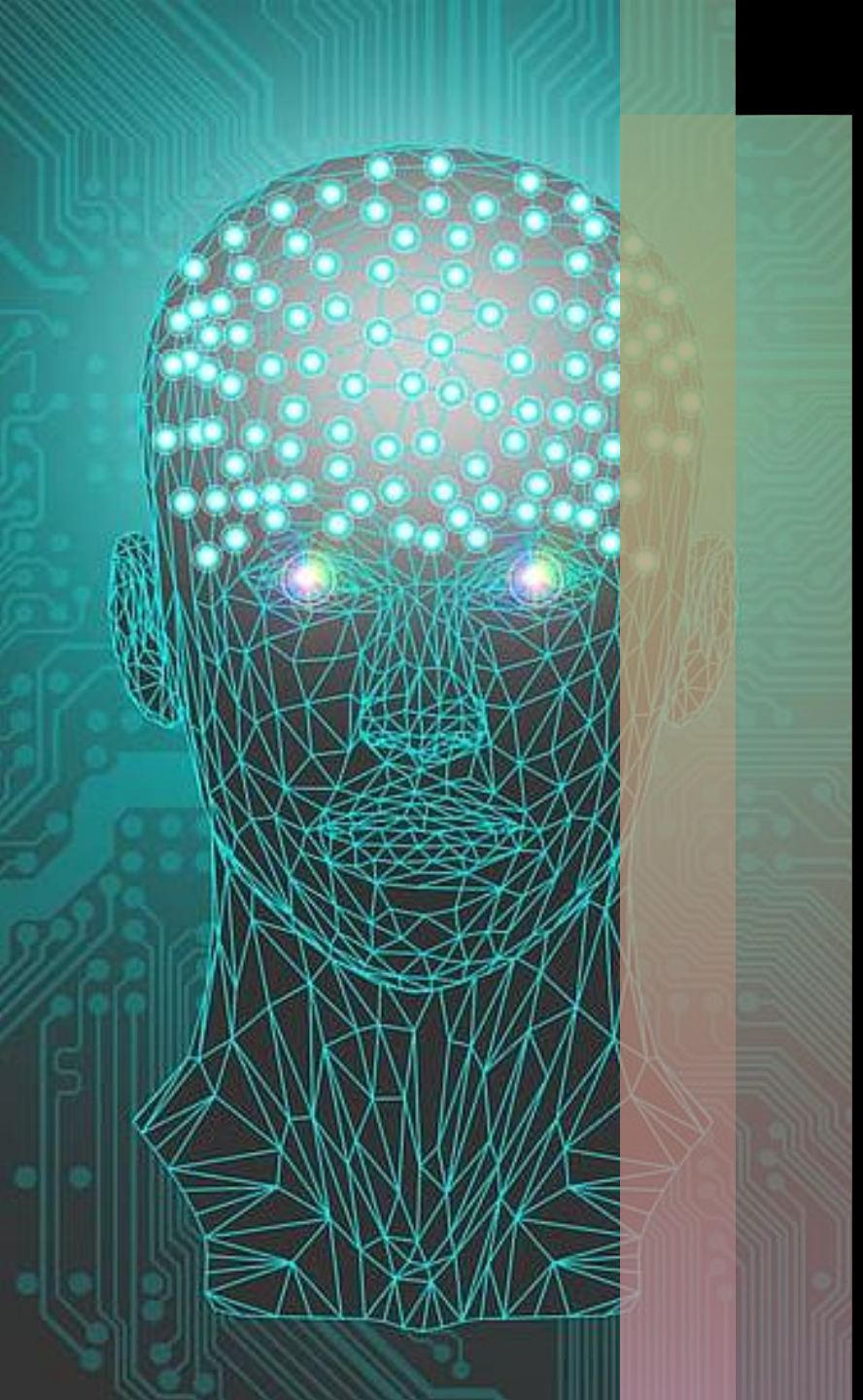
- You may see it being used in circumstances like this, so I'd better mention it here.
- The action of the keyword break is to terminate the nearest enclosing for, do, while or switch statement.
- It would work in this context.

- Note that it terminates **"the nearest enclosing..."**.
- It won't take you out of a nested loop, for example, only the inner loop.
- It would work but its limitations can lead to unintended problems if you forget its properties.



Searching Function

```
void ExistsInArray(int myArray[SIZE], int searchTerm)
{
    bool found = false;
    for( int i = 0; i < SIZE && found == false; i++ )
    {
        if ( myArray[i] == searchTerm )
        {
            found = true;
            break; // Terminate the loop
        }
    }
    if( found == true )
    {
        cout << searchTerm << " found" << endl;
    }
}
```

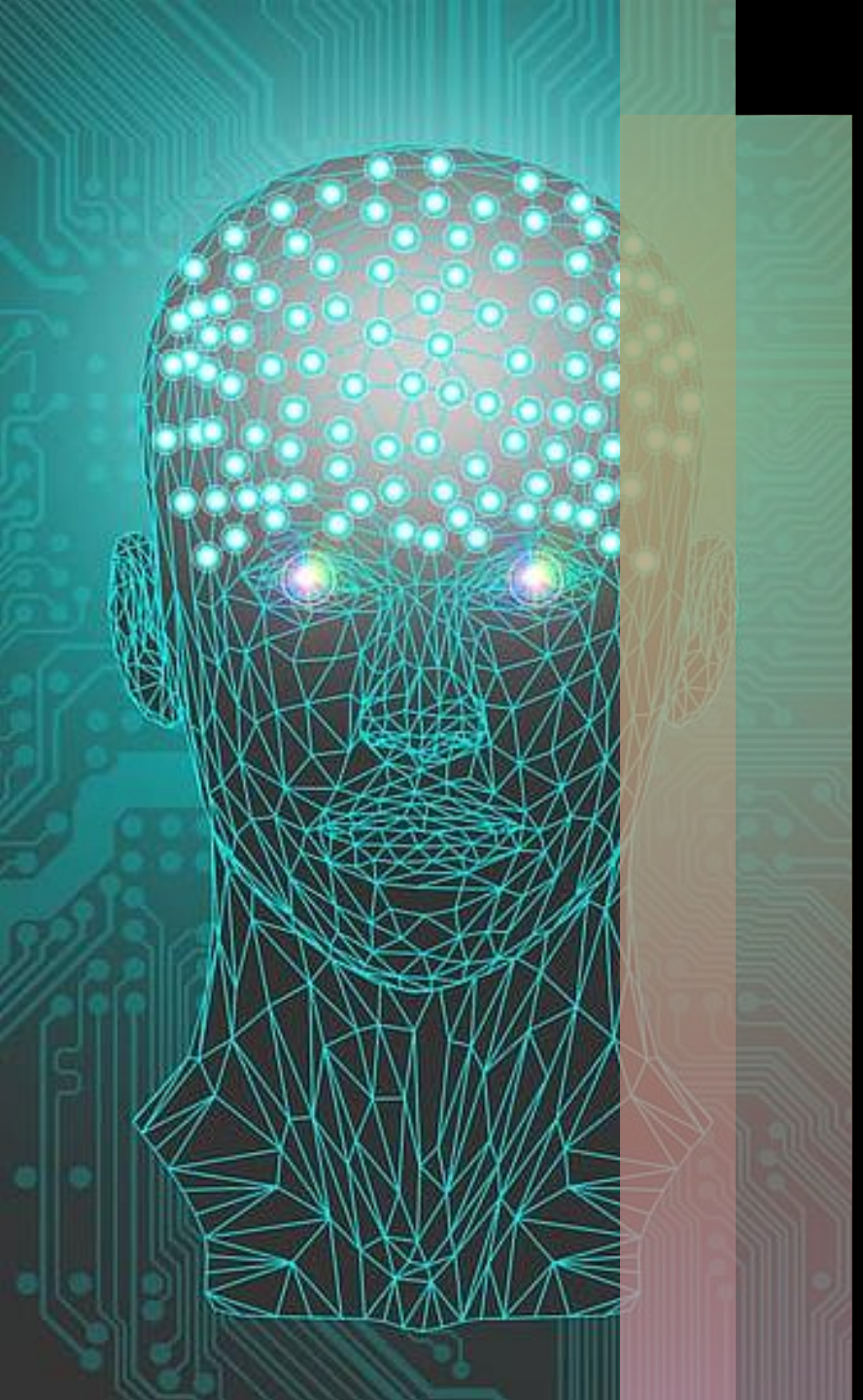


Return

- A function can send data back using the **return** statement.
- Specifies the value to be returned by the function.

```
int FunctionWithReturn()
{
    int number = 7;
    return number;
}
```

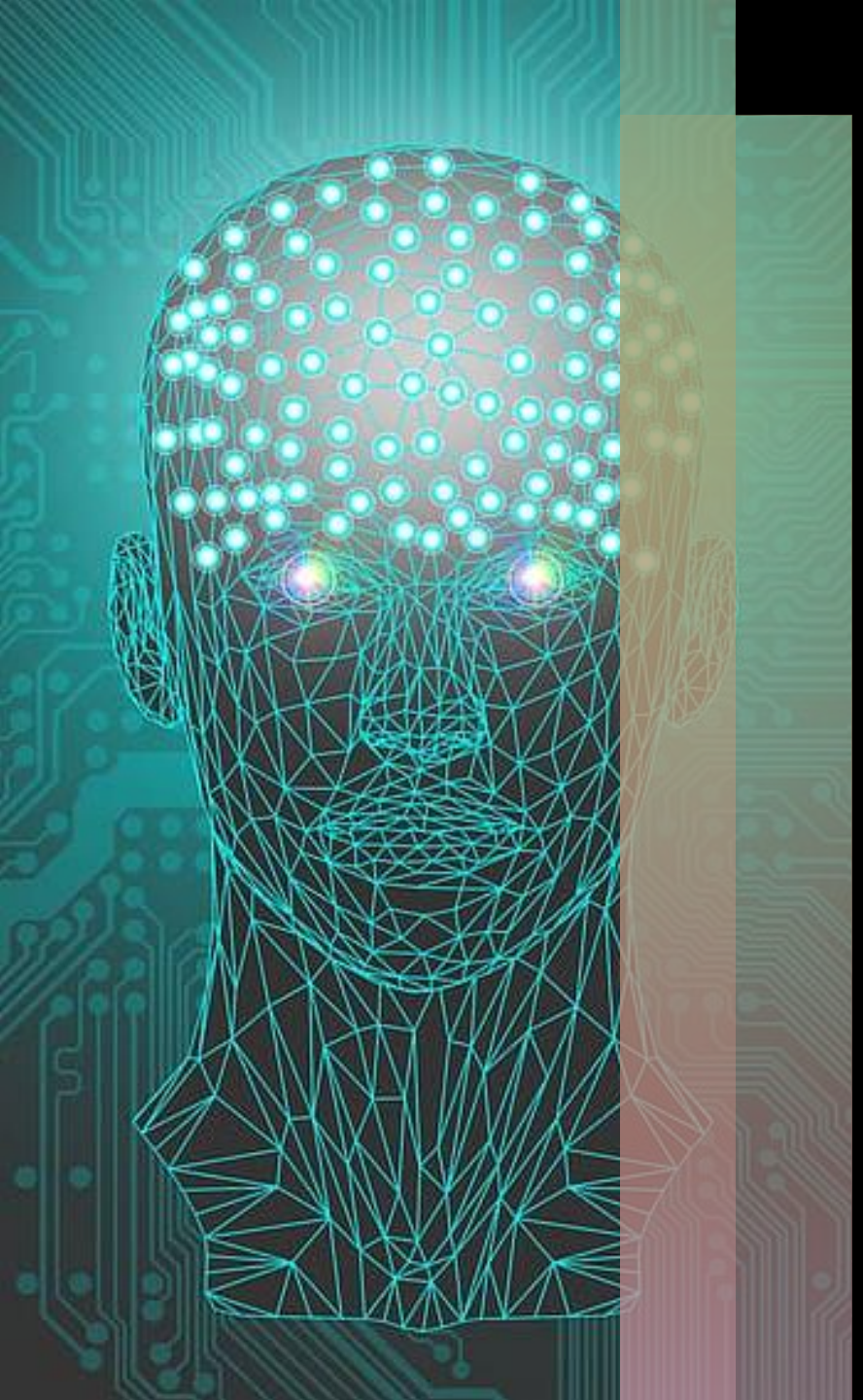
- However, return also has the useful characteristic that it causes a dynamic exit from a function.
- This can be used to simplify code.



Searching Function

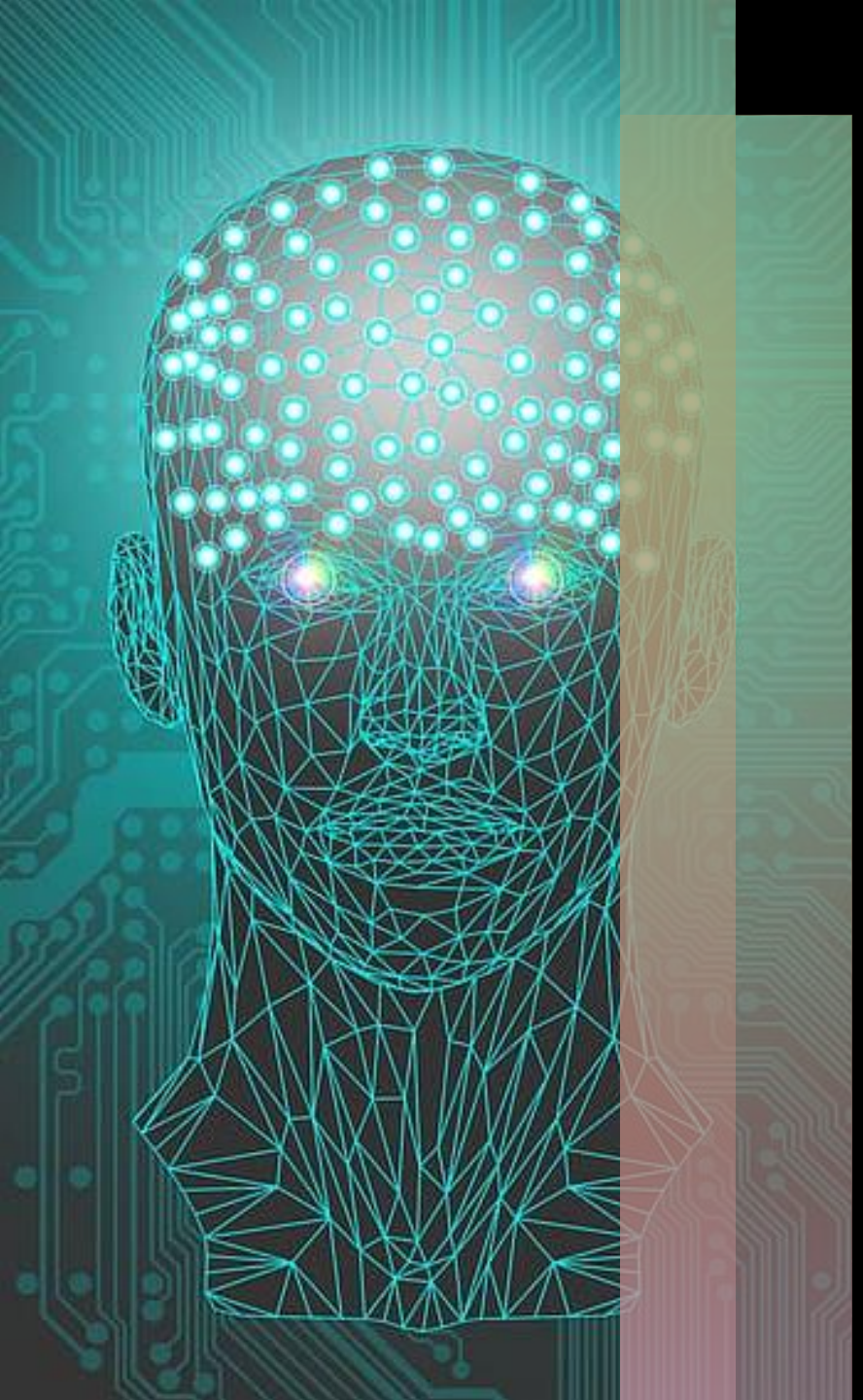
```
bool ExistsInArray(int myArray[SIZE], int searchTerm)
{
    for( int i = 0; i < SIZE; i++ )
    {
        if ( myArray[i] == searchTerm )
        {
            cout << searchTerm << " found" << endl;
            return true;
        }
    }
    cout << searchTerm << " not found" << endl;
    return false;
}
```

- Shorter more elegant code, return is dynamically exiting from the function, which ends the fucntions execution.
- Note that all paths must have a return in C++ (two returns)



Arrays of Structs

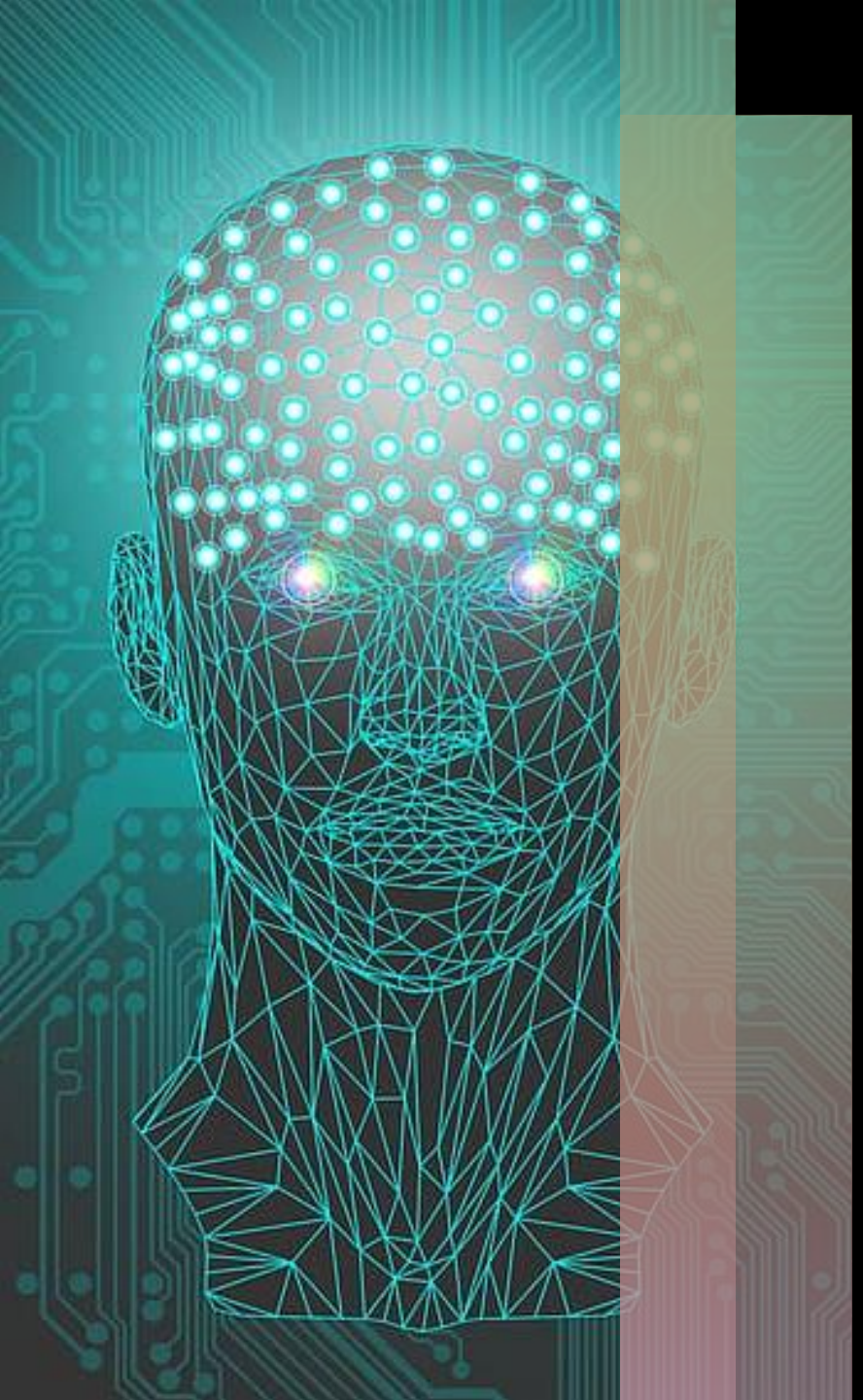
```
struct SRecord
{
    int mId;
    string mName;
};

int main()
{
    SRecord table[SIZE] =
    {
        { 1, "Oliver"  },
        { 3, "Arefeen" },
        { 7, "Noya"    },
        { 2, "Amar"    }
    };
}
```



Recursion

- **Recursion** is a programming concept where a function **calls itself** during its execution
- The task is broken down into smaller tasks, and each smaller task is resolved using the same technique
- The function will keep calling itself until the **base case** condition is met
  - The **base case** defines the condition that prevents the recursive function from calling itself
  - Without a base case, the function would keep making recursive calls – which would be bad!

- Recursion is a powerful and elegant tool
  - It does require finesse to implement recursive functions
  - The programmer is required to understand all aspects of their code – or at the very least how the problem gets smaller with each call



Recursion

- Code can be viewed as instructions to the computer.
- A function is a set of instructions. It is reasonable that a function can be called multiple times in a program.
- It is also reasonable that a function can call itself.
- It is called "recursion" when a function calls itself.
- This does not have to be a member function (method) of a class. It works with any function.

Normal Function

- A factorial function to discern the factorial (!) of a whole number (integer)
- !5 = 1 * 2 * 3 * 4 * 5 = 120

```
int Factorial(int n)
{
    int factorial = 1;
    for (int i = 1; i <= n; i++)
    {
      factorial *= i;
    }
    return factorial;
}
```

Recursive Function

- A recursive version of the factorial function to discern the factorial (!) of a whole number (integer)
- !5 = 1 * 2 * 3 * 4 * 5 = 120

```
int FactorialRecursive(int n)
{
    if (n == 0 || n == 1)
    {
        return 1; // Base case always comes first
    }
    else
    {
        return n * FactorialRecursive(n - 1);
    }
}
```

- Notice how we start with 5 and decrement this time around

Factorial Recursion Breakdown

```
FactorialRecursive(5) [Push]
    |
    |--> FactorialRecursive(4) [Push]
    |     |
    |     |--> FactorialRecursive(3) [Push]
    |     |     |
    |     |     |--> FactorialRecursive(2) [Push]
    |     |     |     |
    |     |     |     |--> FactorialRecursive(1) [Push]
    |     |     |     |     |
    |     |     |     |     |--> FactorialRecursive(0) [Push]
    |     |     |     |     |     |
    |     |     |     |     |     |--> Returns 1 [Pop]
    |     |     |     |     |<-- Returns 1 * 1 = 1 [Pop]
    |     |     |     |
    |     |     |     |<-- Returns 2 * 1 = 2 [Pop]
    |     |     |
    |     |     |<-- Returns 3 * 2 = 6 [Pop]
    |     |
    |     |--> Returns 4 * 6 = 24 [Pop]
    |
    |<-- Returns 5 * 24 = 120 [Pop]
```

- Recursive calls in FactorialRecursive add new frames to the call stack
- Each frame stores local variables and the execution point for a specific function call
- The last function call made is the first to return, following the Last In, First Out (LIFO) principle
    - FactorialRecursive(0) is the first to be removed
- The base case (n == 0 or n == 1) stops the recursion, allowing the stack to unwind
    - Stack unwinding is a critical process in recursive functions
    - It's the mechanism through which the results are calculated as the function calls return
    - Note: that the base case return value is used in the calcualtions.
- The stack diagram visually represents recursive calls and returns in a stacked format
- [Push] adds frames during a recursive call, and [Pop] removes frames when a function returns

Fibonacci Sequence

- A Fibonacci sequence function that prints out the Fibonacci sequence.
- The Fibonacci sequence is a series of numbers in which each number is the sum of the two preceding ones
- Usually starts with 0 and 1 as the first elements

```
int Fibbonachi(int n)
{
    int n0 = 0;
    int n1 = 1;

    for (int i = 0; i < n; i++)
    {
        cout << n0 << " ";
        n1 = n0 + n1;
        n0 = n1 - n0;
    }
}
```

Fibonacci Sequence

- A recursive Fibonacci sequence function that prints out the Fibonacci sequence.
- Again, we're working backwards from n.

```
int Fibonacci(int n)
{
    if (n <= 1)
    {
        return n;
    }
    else
    {
        return Fibonacci(n - 1) + Fibonacci(n - 2);
    }
}
```

Factorial Recursion Breakdown

```
Fibonacci(5) [Push]
|
|  --> Fibonacci(4) [Push]
|  |
|  | --> Fibonacci(3) [Push]
|  |  |
|  |  |--> Fibonacci(2) [Push]
|  |  |  |
|  |  |  | --> Fibonacci(1) [Push]
|  |  |  |  |
|  |  |  |  |--> Returns 0 [Pop]
|  |  |  |
|  |  |  |<-- Returns 1 [Pop]
|  |  |
|  |  |<-- Returns 0 + 1 = 1 [Pop]
|  |
|  | --> Fibonacci(2) [Push]
|  |  |
|  |  | --> Fibonacci(1) [Push]
|  |  |  |
|  |  |  |--> Returns 1 [Pop]
|  |  |  |
|  |  |  |<-- Returns 1 [Pop]
|  |  |
|  |  |<-- Returns 1 + 1 = 2 [Pop]
|  |
|  |<-- Returns 1 + 2 = 3 [Pop]
|
|<-- Returns 2 + 3 = 5 [Pop]
```

- Like the factorial breakdown, Fibonacci recursion involves multiple calls to smaller Fibonacci problems until the base cases are reached
- The Fibonacci sequence is calculated by summing up the results of two smaller Fibonacci problems
- The end result of using this function is 0 1 1 2 3 5
- The base case is crucial here again as they return 0 and 1 respectively

Tree Recursion

- Tree recursion occurs when a function makes more than one recursive call
- It often leads to multiple branches of recursive calls, creating a tree-like structure

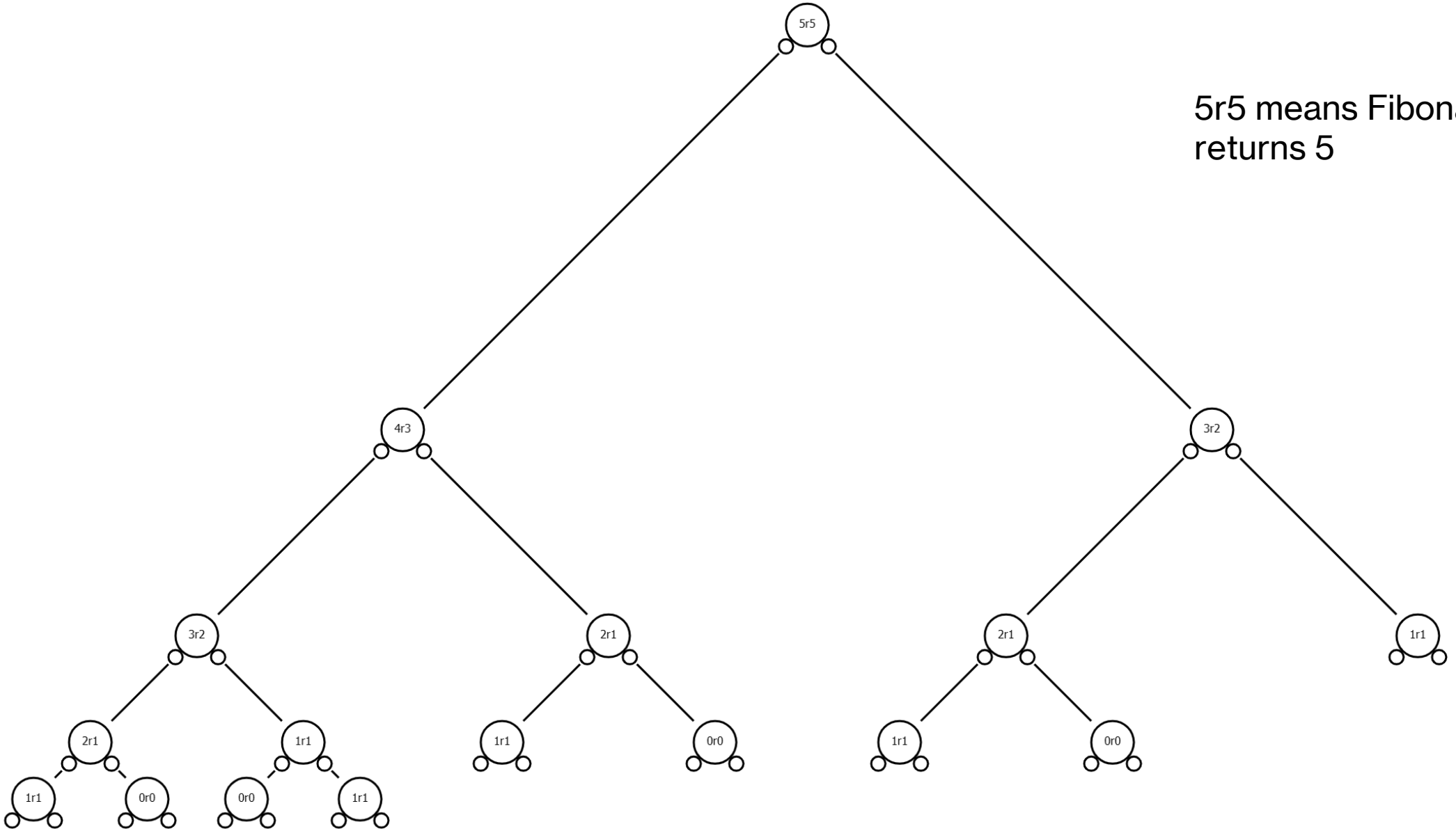
- You just saw an example of tree recursion

```
return Fibonacci(n - 1) + Fibonacci(n - 2);
```

- Each call of the function spawns two more recursive calls, creating a binary tree structure.

- Understanding and **visualising the recursion tree** is crucial for efficient problem-solving

Tree Recursion



5r5 means Fibonacci(5) returns 5

Recursion Advantages

- Certain concepts are difficult to express without using recursion
  - The concept of ancestor can be described as "father or mother of..." and then applied recursively, e.g., the father or mother of my mother. This is a concept that naturally extends recursively, delving into the father or mother of one's mother, and so on.
- Recursion provides a natural way to represent problems that exhibit self-similar or fractal-like structures
  - Recursive algorithms naturally follow the divide-and-conquer paradigm
- This innate alignment with problem characteristics often leads to more intuitive and readable code
- A natural way to express certain concepts
- Can produce very elegant solutions
- Used extensively in certain areas such as AI

Recursion Disadvantages

- It can be difficult to estimate the precise number of calls in a recursively defined algorithm
- Control is passed over to the recursive process
  - his may prove problematic in certain time-constrained situations
- May not execute very efficiently
- Can takes up a lot of stack space
- Debugging using trace can be difficult
- You must design your algorithms well to mitigate these issues.

Common Mistakes

- Forgetting to include a base case or defining it incorrectly can lead to infinite recursion
  - This is not a problem inside of our IDEs since they're virtual environments (self-contained)
  - However, in a real-world system it can cause catastrophic failures
- Making mistakes in the recursive call, such as passing the wrong parameters, can cause unexpected behavior
  - You need to understand the stack to be able to design recursive algorithms well
  - You work backwards, and then unwind the stack to get the answers
- Recursive functions may lead to a stack overflow if there are too many nested calls
- Some problems are better solved iteratively, and forcing a recursive solution may lead to inefficiency

Tail Recursion

- Tail recursion is a special form of recursion where the recursive call is the last operation in the function
- Helps to avoid stack overflow as the current stack is reused
- Helps with debugging too
- The Fibonacci example from earlier can be optimised to use tail recursion

```
// Default values provided, but can be overridden
int FibonacciTail(int n, int a = 0, int b = 1)
{
    if (n == 0)
    {
        return a;
    }
    else
    {
        return FibonacciTail(n - 1, b, a + b);
    }
}
```

Summary

We've covered quite a few topics today:

- Introducing recursion

Any questions?