Recursion
C++ Recursive Functions

In this week

- ➢ Recursion in Mathematics
- ➢ Stopping Criteria and Recursive Steps
- ➢ Recursive C++ functions
- ➢ Identifying Recursive Problems
  - ➢ How and When to think recursively
- ➢ Searching: Sequential and Binary Searches
- ➢ Stacks for Recursion

Recursion

- In mathematics and generally in computing, we say a solution to a problem is **recursive** if the solution is defined in terms of itself and satisfies the following two conditions:
  - **Stopping Criteria**: At certain stage the solution is defined explicitly
  - **Down Sizing**: Each subsequent self definition of the solution approaches the stopping criteria. **Self definition step is also known as recursive step**
- **Recursion**: The process of solving problems recursively

Recursion: Example

- In mathematics, the most common recursive solution of a problem might be the solution to the factorial problem, given by

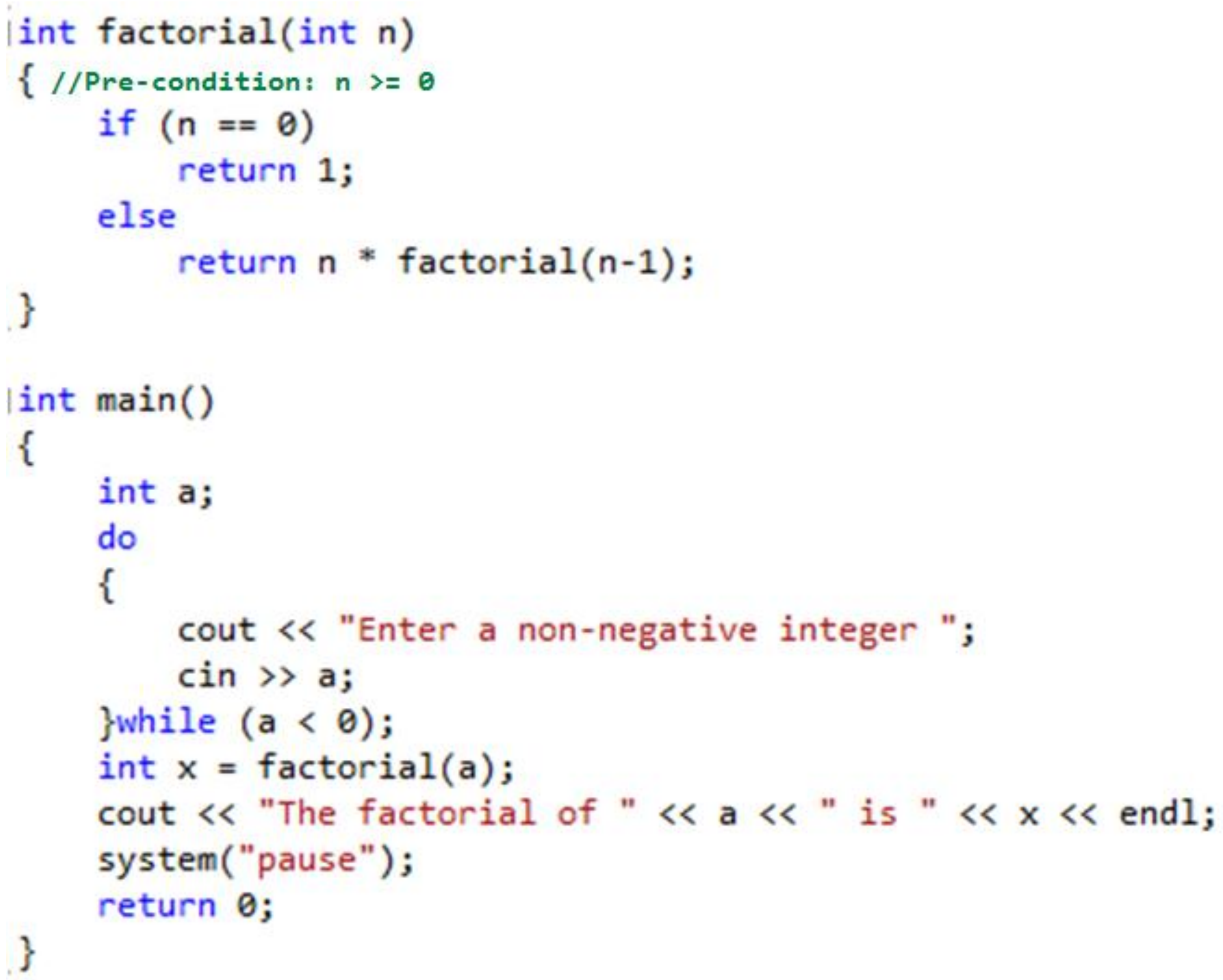
$$f(n) = \begin{cases} 1 & if \quad n = 0 \\ n * f(n-1) & if \quad n > 0 \end{cases}$$

- As can be seen, the solution to a factorial problem is **defined in terms of the solution**
- However, the solution has a **stopping criteria which happens when n=0**. For **n > 0**, the solution is defined in terms of itself but closer to the stopping criteria

Recursion in C++

- In C++, recursion is possible because **C++ allows a function to call itself**
- As an example, we may write a function named **factorial** in order to calculate the factorial of a non-negative integer
- Looking back at the mathematical recursive definition of factorial function, we therefore see that the following C++ function implementation does the job

Recursion in C++

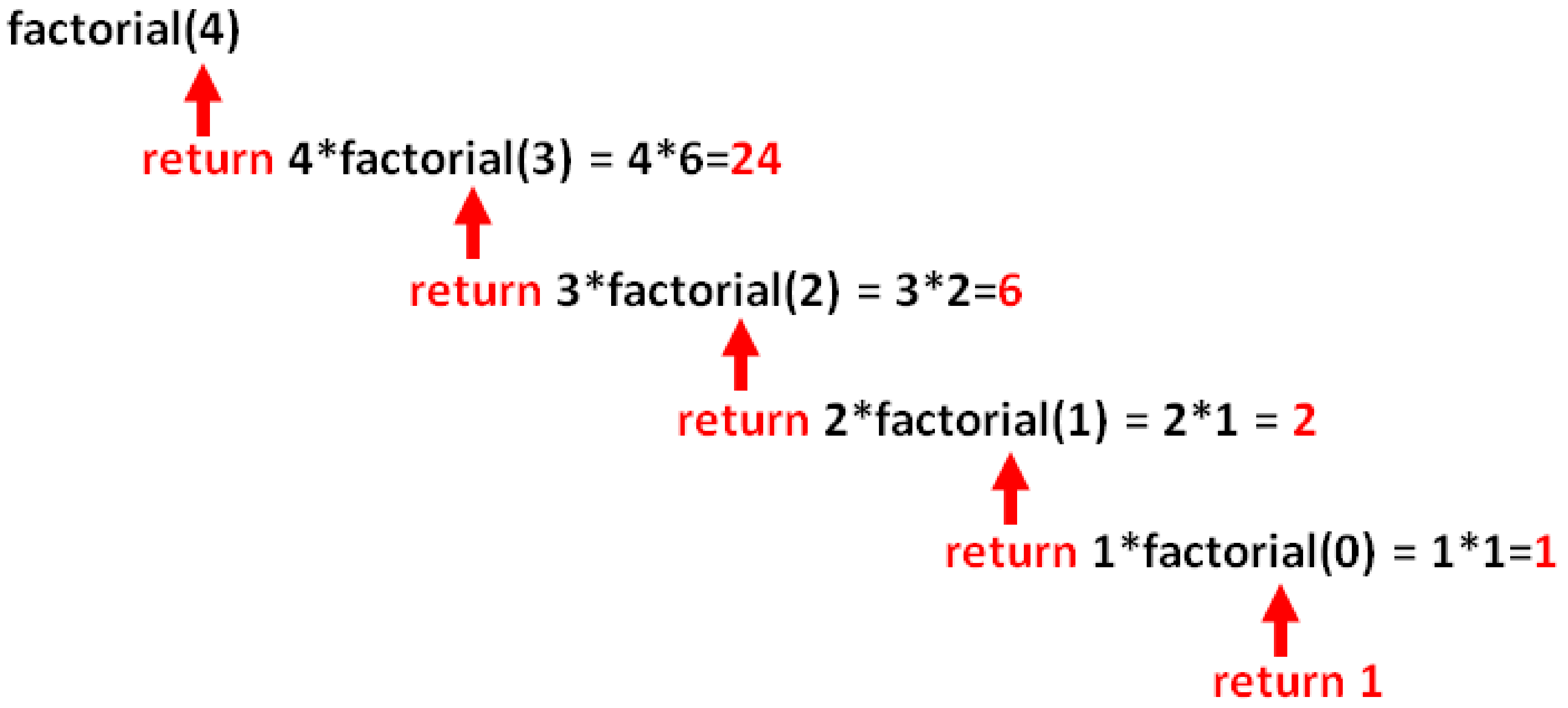


```
int factorial(int n)
{ //Pre-condition: n >= 0
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}

int main()
{
    int a;
    do
    {
        cout << "Enter a non-negative integer ";
        cin >> a;
    }while (a < 0);
    int x = factorial(a);
    cout << "The factorial of " << a << " is " << x << endl;
    system("pause");
    return 0;
}
```

Recursion in C++

- Pictorially, this will look like as follows for the function call factorial(4):



factorial(4)

return 4*factorial(3) = 4*6=24

return 3*factorial(2) = 3*2=6

return 2*factorial(1) = 2*1 = 2

return 1*factorial(0) = 1*1=1

return 1

Recursion in C++

- The heart of recursion in programming lies in the fact that a function, such as the **factorial** function shown above, was able to call itself
- That is to say each function call knows where it was called from and thus returns to the same place from where it was called from
- In C++ recursive function calls are managed using a special memory called the stack memory that keeps track of function calls in a very elegant way as described below

Stack Memory for Recursion

- C++ uses a stack memory to process function calls
- A stack memory is a data structure that allows addition of new item on top of it and only one item at a time can be extracted from the top of the stack
- Whenever a C++ function is called in a program, the function call is added on top of the stack
- Processing of a series of function calls is performed by adding the function calls on top of the stack in the order they were called and then processing the function on top of the stack
- Once processed a function is cleared from the stack and then the next function on top of the stack is processed
- When we first call a function from the main program, we start with an empty stack and whenever we return to the main program the stack memory becomes empty

Stack Memory for Recursion

- Considering the factorial(4) function call from the main program, we see that factorial(4) is placed onto the empty stack
- Then factorial(4) calls factorial(3) which places function call factorial(3) on top of the stack
- Next, factorial(3) calls factorial(2) which places function call factorial(2) on top of the stack
- Next , factorial(2) calls factorial(1) which places function call factorial(1) on top of the stack
- Next, factorial(1) calls factorial(0) which places function call factorial(0) on top of the stack
- Finally, factorial(0) returns 1 without adding anything on top of the stack **and is cleared from the stack**

Stack Memory for Recursion

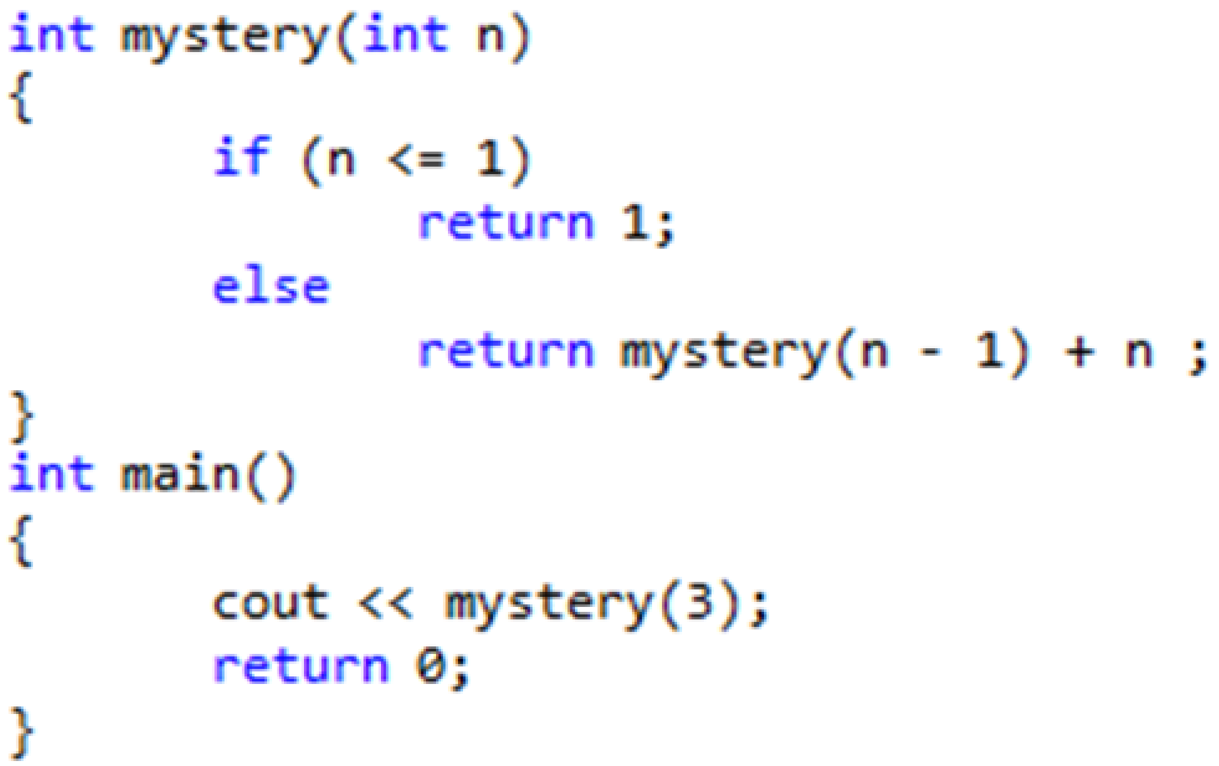
- Although, our aim is to get factorial(4); by its design, the stack will not allow factorial(4) to be accessed before factorial(3) is accessed; which won't be accessed before factorial(2) is accessed; which again won't be accessed before factorial(1) is accessed; which once again won't be accessed before factorial(0) is processed
- Since factorial(0) doesn't add a new function call to the stack, now factorial(1) will be processed using the result of factorial(0). Once processed, factorial(1) is cleared from the stack
- This leads for factorial(2) to be processed and cleared from the stack; which leads for factorial(3) to be processed and cleared from the stack and finally for factorial(4) to be processed and cleared from the stack and its result returned to the main program

Stack Memory for Recursion

- Since computer's stack has finite memory size, the number of recursive calls that can be made is limited by the memory size
- If during a series of recursive calls the stack becomes full, then a stack overflow runtime error occurs which crashes the execution of the program
- The most common cause of a stack overflow runtime is an infinite recursion which occurs when the down sizing step doesn’t approach the stopping criteria
- Extra care must be placed to avoid infinite recursion which will surely result to stack overflow run time error

How and When to think recursively

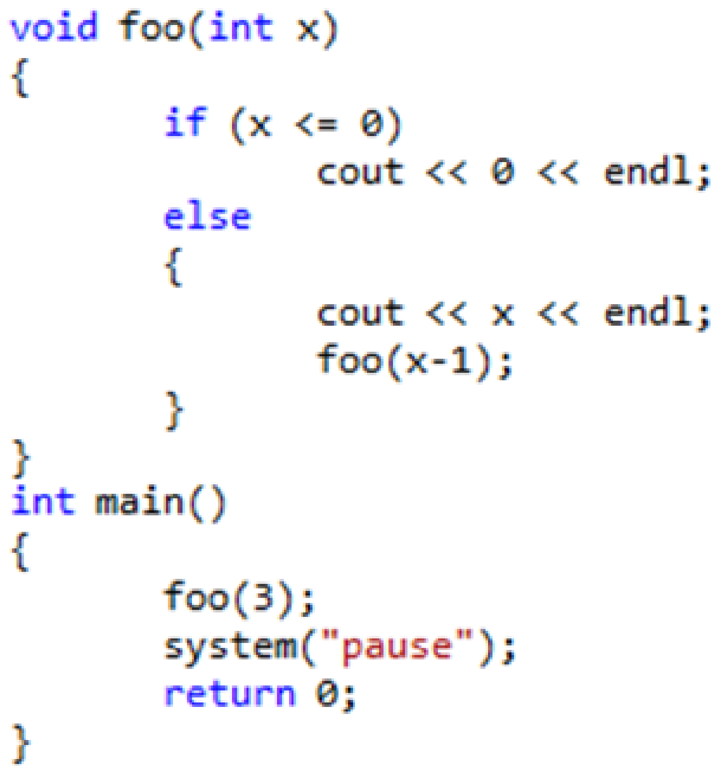
- Now we look at several examples and analyze these examples in order to reinforce our understanding of C++ recursive functions. Analyze the following recursive function and a test program and determine its output



```
int mystery(int n)
{
        if (n <= 1)
                return 1;
        else
                return mystery(n - 1) + n ;
}
int main()
{
        cout << mystery(3);
        return 0;
}
```

How and When to think recursively

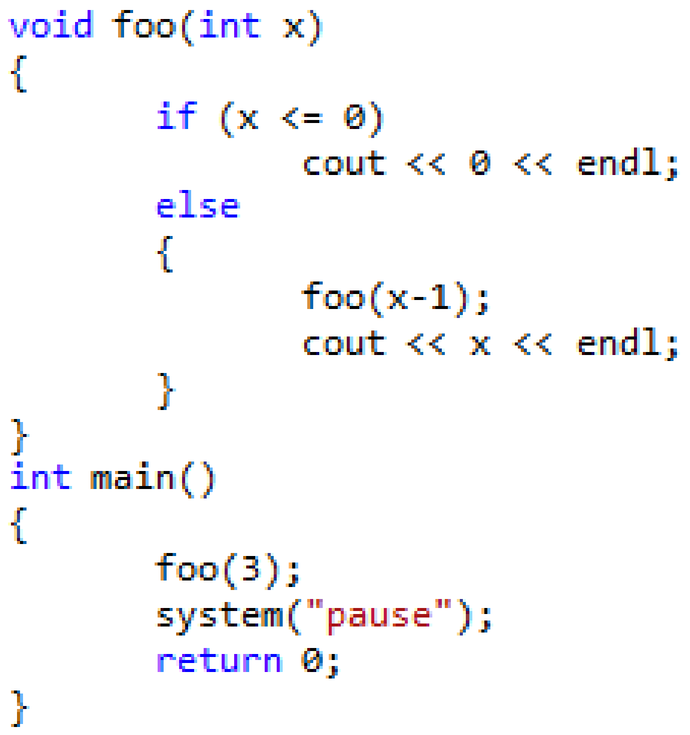
- Analyze the following recursive function and a test program and determine its output



```
void foo(int x)
{
        if (x <= 0)
                cout << 0 << endl;
        else
        {
                cout << x << endl;
                foo(x-1);
        }
}
int main()
{
        foo(3);
        system("pause");
        return 0;
}
```

How and When to think recursively

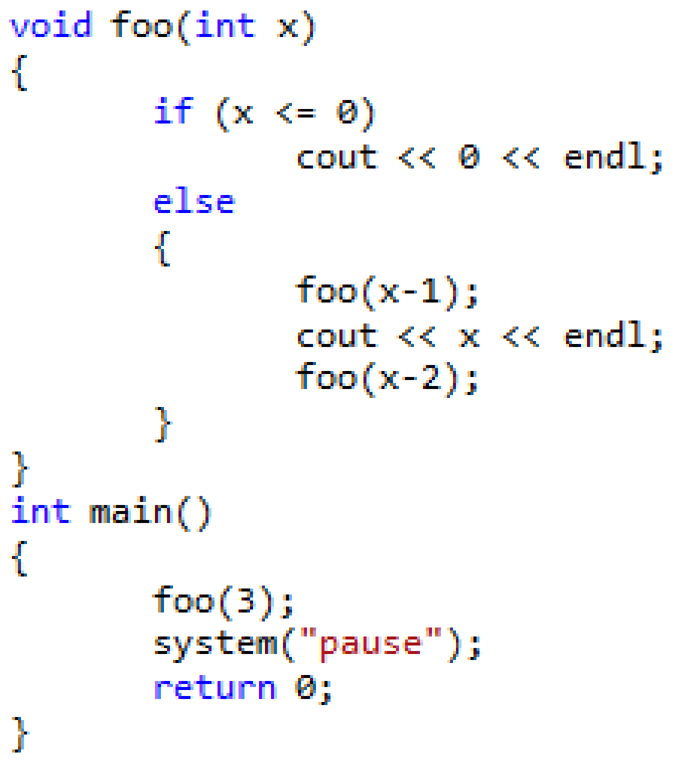
- Analyze the following recursive function and a test program and determine its output



```
void foo(int x)
{
        if (x <= 0)
                cout << 0 << endl;
        else
        {
                foo(x-1);
                cout << x << endl;
        }
}
int main()
{
        foo(3);
        system("pause");
        return 0;
}
```

How and When to think recursively

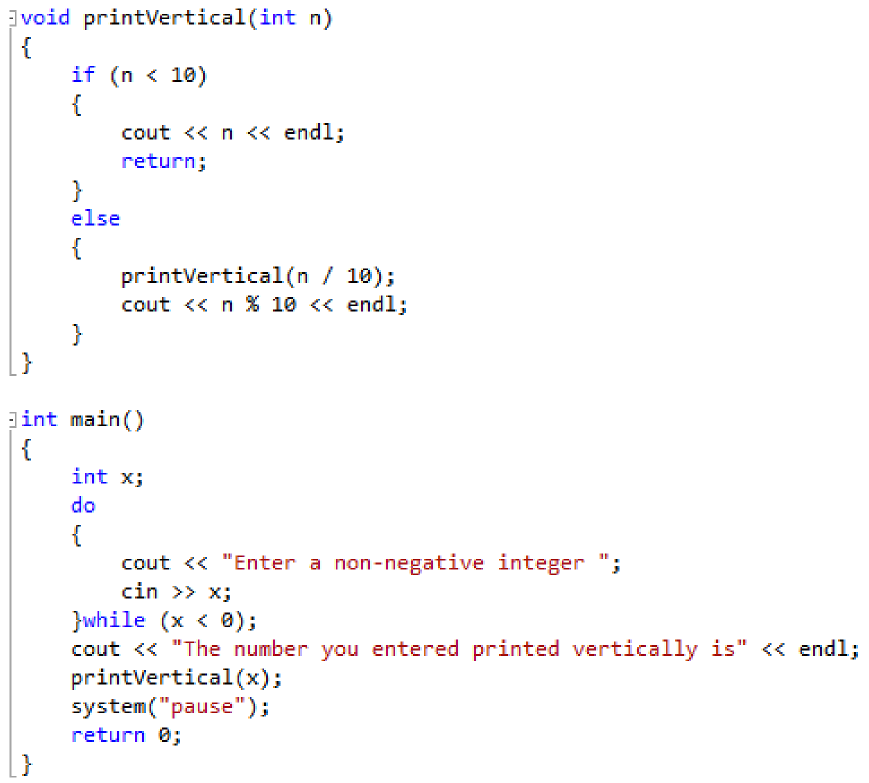
- Analyze the following recursive function and a test program and determine its output



```
void foo(int x)
{
        if (x <= 0)
                cout << 0 << endl;
        else
        {
                foo(x-1);
                cout << x << endl;
                foo(x-2);
        }
}
int main()
{
        foo(3);
        system("pause");
        return 0;
}
```

How and When to think recursively

- Analyze the following recursive function and a test program and determine its output. Assume x = 1782



```cpp
void printVertical(int n)
{
    if (n < 10)
    {
        cout << n << endl;
        return;
    }
    else
    {
        printVertical(n / 10);
        cout << n % 10 << endl;
    }
}

int main()
{
    int x;
    do
    {
        cout << "Enter a non-negative integer ";
        cin >> x;
    }while (x < 0);
    cout << "The number you entered printed vertically is" << endl;
    printVertical(x);
    system("pause");
    return 0;
}
```

Recursive Search Algorithms

- Searching is a very common process in computing
- While searching can easily be performed using non-recursive algorithms, it is a problem that has a natural recursive solution as well
- For the sake of exploring recursion and how to think recursively, we now look at sequential and binary recursive search algorithms

Sequential Search

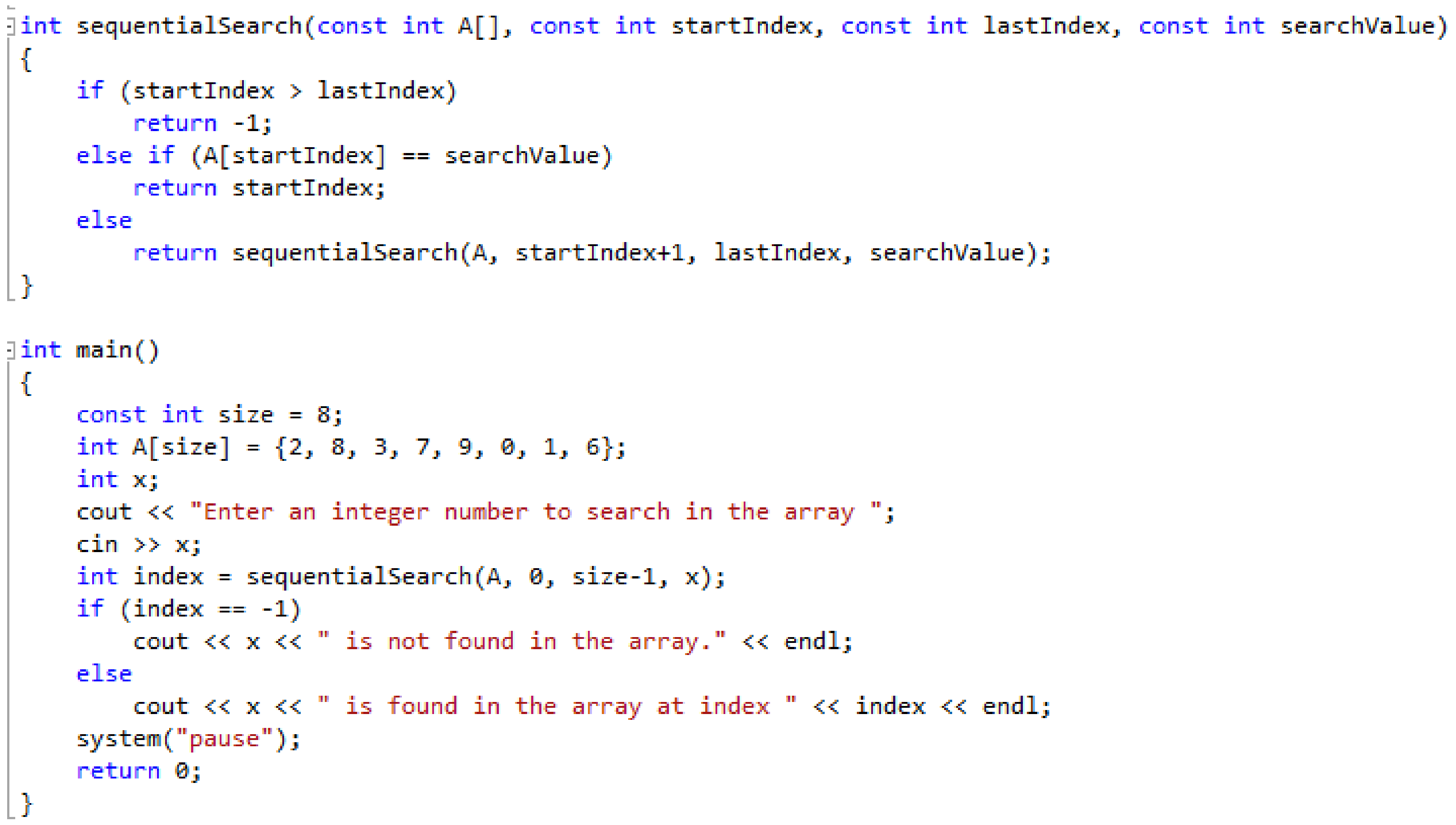
- **Problem Statement**: Given an array of **n** elements (say integers), we would like to search the array if a search value is found in the array. If the search value is found, we would like to return the index of the element matching the search value; otherwise we would like to return -1 to mean the search value is not found in the array.
- Let us name the array **A** and the value to be searched **searchValue**

Sequential Search

- **Solution**: A recursive solution can be given as follows: Check if the **searchValue** is equal to the first element of the array. If yes, we return the index of the first element; otherwise we search in the remaining elements of the array. We stop the search when there are no more elements to search in the array
- **Programming**: In order to keep track of the first element of the array we need to have the knowledge of the **start index** at each step. Moreover, in order not to exceed the last element we will also need to have the knowledge of the **last index** at each step

Sequential Search

- Thus the following function does the job



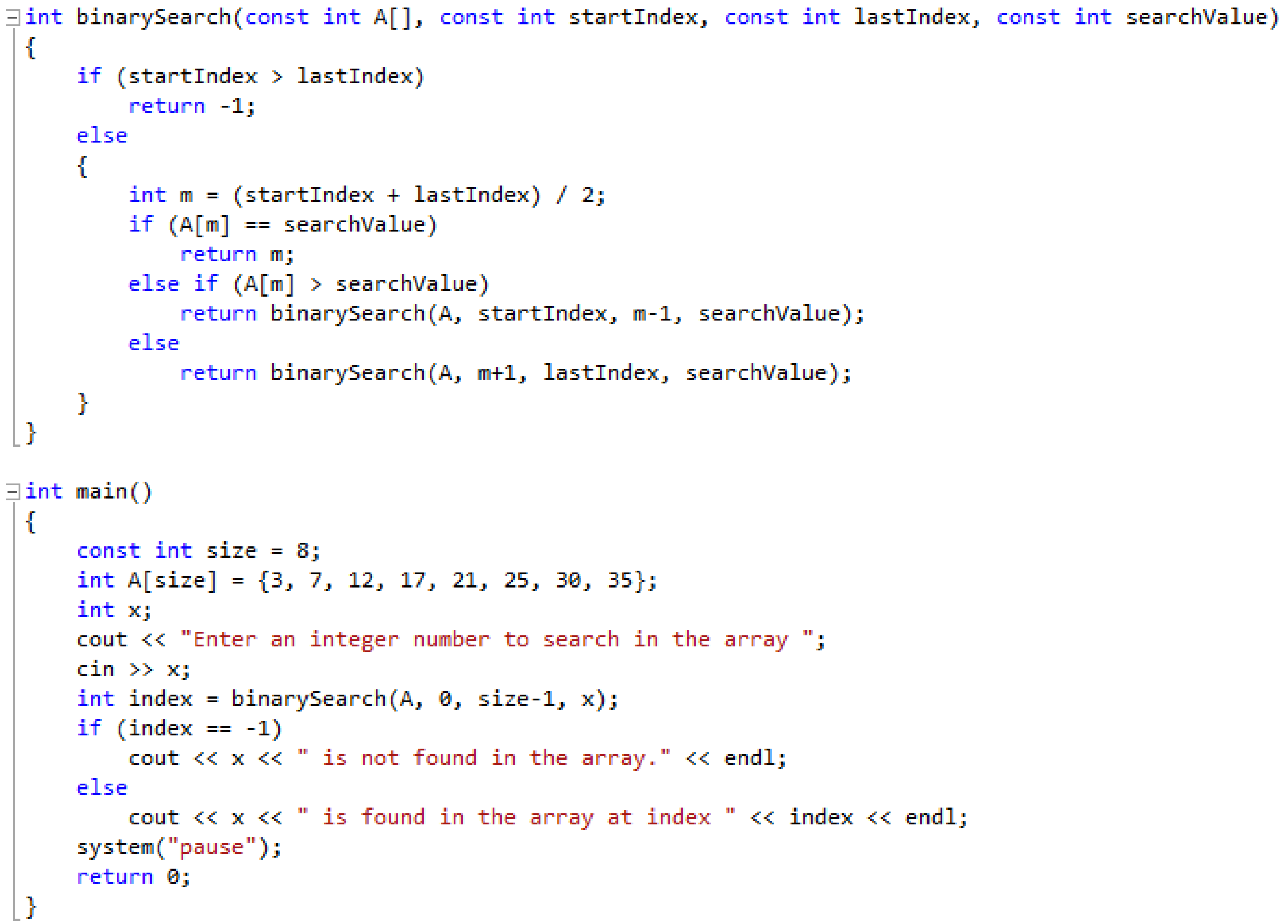
```
int sequentialSearch(const int A[], const int startIndex, const int lastIndex, const int searchValue)
{
    if (startIndex > lastIndex)
        return -1;
    else if (A[startIndex] == searchValue)
        return startIndex;
    else
        return sequentialSearch(A, startIndex+1, lastIndex, searchValue);
}

int main()
{
    const int size = 8;
    int A[size] = {2, 8, 3, 7, 9, 0, 1, 6};
    int x;
    cout << "Enter an integer number to search in the array ";
    cin >> x;
    int index = sequentialSearch(A, 0, size-1, x);
    if (index == -1)
        cout << x << " is not found in the array." << endl;
    else
        cout << x << " is found in the array at index " << index << endl;
    system("pause");
    return 0;
}
```

Binary Search

- In binary search algorithm, we assume that the array is already sorted (say in increasing order) and we don't have to search sequentially
- Instead, we check the middle element of the array if it is equal to the **searchValue**; if yes we return the middle index; otherwise we search one half of the array depending which side the **searchValue** lies
- The following function implements the recursive binary search algorithm

Binary Search

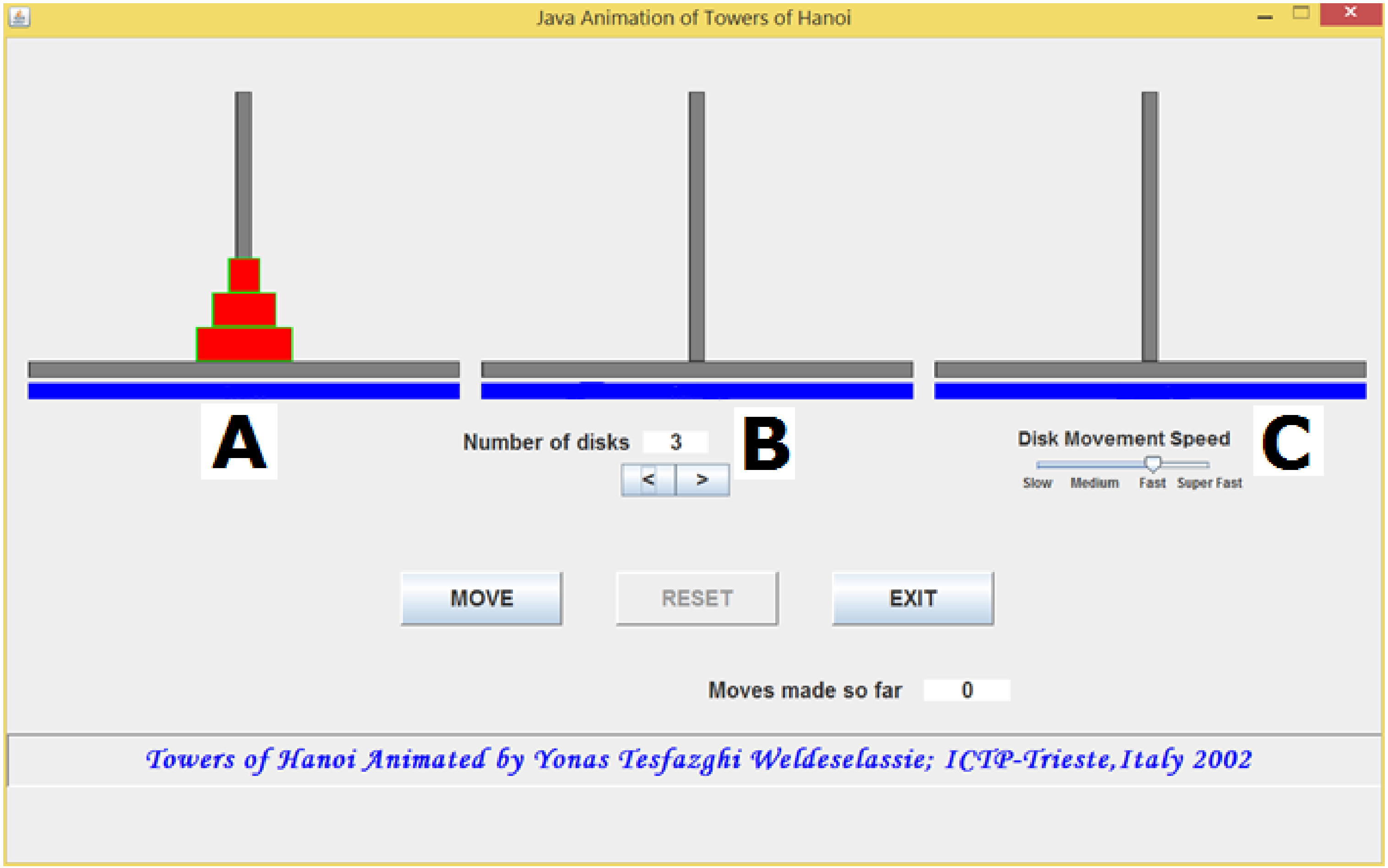


```cpp
int binarySearch(const int A[], const int startIndex, const int lastIndex, const int searchValue)
{
    if (startIndex > lastIndex)
        return -1;
    else
    {
        int m = (startIndex + lastIndex) / 2;
        if (A[m] == searchValue)
            return m;
        else if (A[m] > searchValue)
            return binarySearch(A, startIndex, m-1, searchValue);
        else
            return binarySearch(A, m+1, lastIndex, searchValue);
    }
}

int main()
{
    const int size = 8;
    int A[size] = {3, 7, 12, 17, 21, 25, 30, 35};
    int x;
    cout << "Enter an integer number to search in the array ";
    cin >> x;
    int index = binarySearch(A, 0, size-1, x);
    if (index == -1)
        cout << x << " is not found in the array." << endl;
    else
        cout << x << " is found in the array at index " << index << endl;
    system("pause");
    return 0;
}
```

Towers of Hanoi

- As another example, consider the famous Tower's of Hanoi problem


Java Animation of Towers of Hanoi
A
B
C
Number of disks 3
< >
Disk Movement Speed
Slow
Medium
Fast
Super Fast
MOVE
RESET
EXIT
Moves made so far 0
Towers of Hanoi Animated by Yonas Tesfazghi Weldeselassie; ICTP-Trieste, Italy 2002

Towers of Hanoi

- In this problem, you are given three pegs named **A**, **B** and **C**
- There are **n** discs in peg **A**
- The discs are arranged in increasing size top to bottom
- The aim is to **move the discs from peg A to peg C** such that
  - **Only one TOP disc can be moved at a time** and
  - **No bigger disc is allowed to sit on top of a smaller disc**
- You may use the peg **B** as temporary disc holder

Towers of Hanoi

- Attempting to come up with a direct solution to this problem is next to impossible especially at the CMPT130 level. Try it!
- However, one can easily see the solution can be stated as follows:
- Consider the case **n = 1**. That is there is only one disc in the peg **A**
- Then the solution is **trivial**: **move the disc from the peg A directly to peg C**. **This is a stopping criteria!**

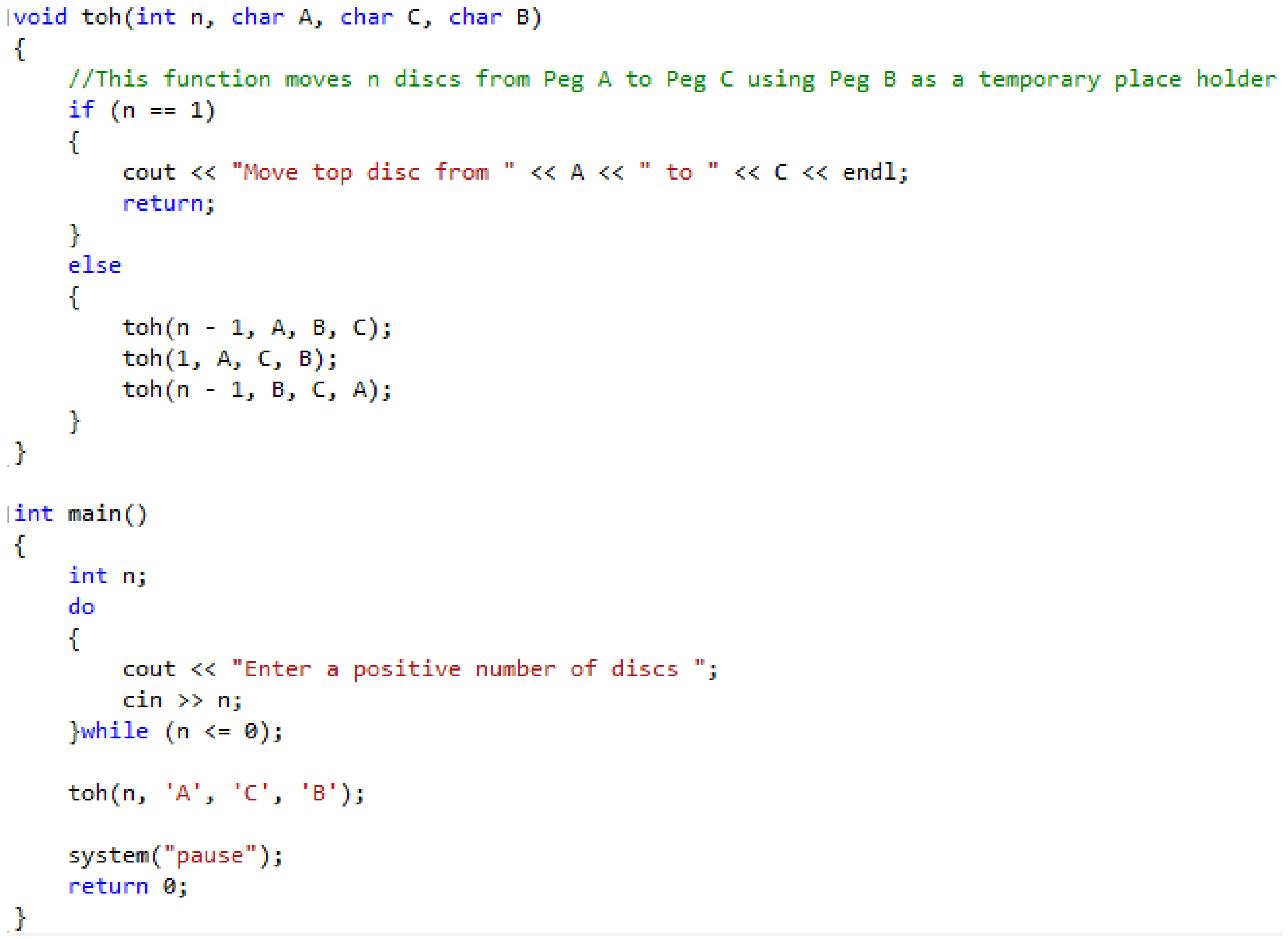
Towers of Hanoi

- How about when there are **n > 1** discs on peg **A**?
- It is easy to see that we need to first move the top **n-1** discs from peg **A** to peg **B**.... using peg **C** as temporary place
- Next we move the bottom disc from peg **A** to peg **C** directly
- Finally, we move the **n-1** discs in peg **B** to peg **C**.... this time using peg **A** as temporary place

Towers of Hanoi

- But how do we move the top **n-1** discs from peg **A** to peg **B**?
- Well, it is a sub-problem of the original problem that requires the exact same solution; hence recursion!
- Importantly, **moving n-1 discs is closer to the stopping criteria than the original problem of moving n discs**
- Hence we clearly have recursion and a recursive solution is the way to go and the following function implementation does the job

Towers of Hanoi



```
void toh(int n, char A, char C, char B)
{
    //This function moves n discs from Peg A to Peg C using Peg B as a temporary place holder
    if (n == 1)
    {
        cout << "Move top disc from " << A << " to " << C << endl;
        return;
    }
    else
    {
        toh(n - 1, A, B, C);
        toh(1, A, C, B);
        toh(n - 1, B, C, A);
    }
}

int main()
{
    int n;
    do
    {
        cout << "Enter a positive number of discs ";
        cin >> n;
    }while (n <= 0);

    toh(n, 'A', 'C', 'B');

    system("pause");
    return 0;
}
```