COSC 220: Computer Science II

Module 1

Instructor:

Dr. Xiaohong (Sophie) Wang
(xswang@salisbury.edu)

Department of Mathematics & Computer Science

Salisbury University

Spring 2021

Arrays

1. Array
    1.1 Array in C++
    1.2 Range-Based `for` loop
    1.3 Processing Array Contents
    1.4 Arrays as Function Arguments
    1.5 Two-Dimensional Arrays
2. Searching and Sorting Arrays
    2.1 Array Search Algorithms
    2.2 Array Sorting Algorithms

Salisbury UNIVERSITY

1.1 Array in C++

- Array: **variable** that can store **multiple** values of the **same type**
- Values are stored in **adjacent memory** locations
- Declared using **`[]`** operator:

```
int tests[5];
```

Data type Name Size

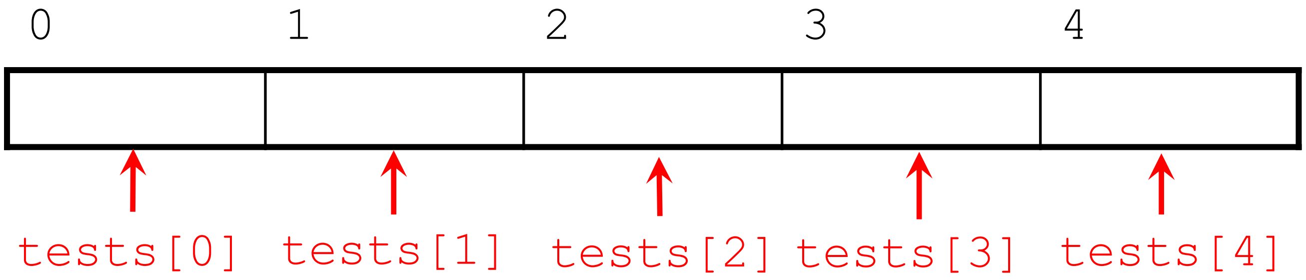
Salisbury UNIVERSITY

Accessing Array Elements

- Each element in an array is assigned a **unique** *subscript* from `0` to `n-1`
- Access an element in an array:

  `array_name[subscript]`

```
int tests[5];
```



Accessing Array Elements (Cont'd)

- Each array element can be used as a regular variable:

```
tests[0] = 79;
cout << tests[0];
cin >> tests[1];
tests[4] = tests[0] + tests[1];
```

- Arrays must be accessed via **individual** elements:

```
cout << tests; // not legal
```

Using a Loop to Step Through an Array

- Example – The following code defines an array, `numbers`, and assigns 99 to each element:

```
const int ARRAY_SIZE = 5;
int numbers[ARRAY_SIZE];

for (int count = 0; count < ARRAY_SIZE; count++)
     numbers[count] = 99;
```

The variable `count` starts at `0`, which is the first valid subscript value

The loop ends when the variable `count` reaches 5, which is the first invalid subscript value

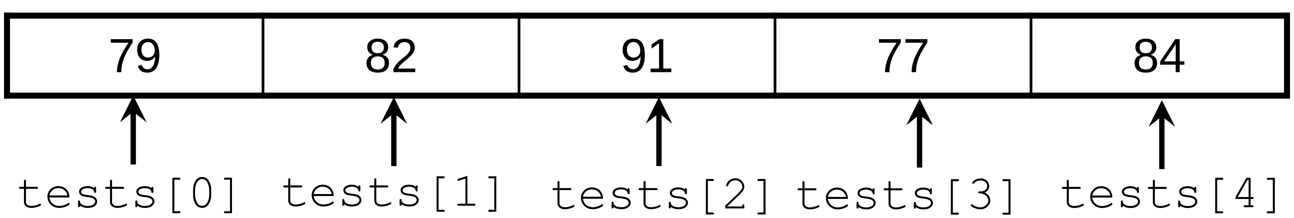
The variable `count` is incremented after each iteration

Array Initialization

- An array can be initialized with an <u>initialization list</u>:

```
const int SIZE = 5;
int tests[SIZE] = {79,82,91,77,84};
```

- The values are stored in the array in the order in which they appear in the list.



- The initialization list cannot exceed the array size.

No Bounds Checking in C++

- When you use an array subscript, C++ does not check whether it is a *valid* subscript or not
  - You can use subscripts that are beyond the bounds of the array

```
int values[3] = {5, 8, 10};

// Syntax correct, but may corrupt other memory
// locations, crash program, or cause elusive bugs
values[3] = 12;
```

- A common mistake: ***off-by-one error***
  - Subscripts are between `0` and `n-1`, not `1` and `n`

```
int numbers[10];
for (int count = 1; count <= 10; count++)
    numbers[count] = 0;
```

1.2 The Range-Based `for` Loop

- The range-based `for` loop is a loop that iterates **once for each element** in an array

- Each time the loop iterates, it copies an element from the array to a built-in variable, known as the **`range variable`**

- The range-based `for` loop automatically knows the number of elements in an array

The Range-Based `for` Loop

- General format of the range-based `for` loop:

```
for (dataType rangeVariable : array)
     statement;
```

- ***dataType*** is the data type of the range variable.
- ***rangeVariable*** is the name of the range variable. This variable will receive the value of a different array element during each loop iteration.
- ***array*** is the name of an array.
- ***statement*** is a statement that executes during a loop iteration.

Example

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

int main() {
    // Define an array of integers
    int numbers[] = {10, 20, 30, 40, 50};

    // Display the values in the array
    for (int val : numbers) {
        cout << val << endl;
    }

    return 0;
}
```

Output:
10
20
30
40
50

1.3 Processing Array Contents

- Array elements can be treated as ordinary variables of the same type as the array
  - Each element is a variable
  - Processing an element is no different than processing other variables

- When using `++`, `--` operators, don't confuse the element with the subscript:

```
tests[i]++; // add 1 to tests[i]
tests[i++]; // increment i, no
            // effect on tests
```

Array Assignment

To copy one array to another,

- Don't try to assign one array to the other:

```
newTests = tests;  // Won't work
```

- Instead, assign element-by-element:

```
for (i = 0; i < ARRAY_SIZE; i++)
    newTests[i] = tests[i];
```

Note: Anytime the name of an array is used without brackets and a subscript, it is seen as the array's **beginning memory address** (not a variable).

In-class practice

- Take 5 integers from user and store these numbers in an array
- Use a `for` loop to find the largest element of this array
- Display this element
- Test you code

Question: How to implement this practice using range-based `for` loop?

 Reference code: LargestInteger.cpp

1.4 Arrays as Function Arguments

- To pass an array to a function, use the array name:

```
int tests[5] = {79,82,91,77,84};
showScores(tests);
```

- To define a function that takes an array parameter, use empty `[]` for array argument:

```
// function prototype
void showScores(int []);

// function header
void showScores(int scores[])
```

No size declarator inside the brackets

Note: When an entire array is passed to a function, it is not passed by value, but passed by reference (only the starting memory address is passed).

Arrays as Function Arguments

- When passing an array to a function, it is common to pass **array size** so that function knows how many elements to process:

```
showScores(tests, ARRAY_SIZE);
```

# of elements

- Array size must also be reflected in prototype, header:

```
// function prototype
void showScores(int [], int);

// function header
void showScores(int scores[], int size)
```

Example

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

void showValues(int [], int);  // Function prototype

int main() {
    const int ARRAY_SIZE = 8;
    int numbers[ARRAY_SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};

    showValues(numbers, ARRAY_SIZE);
    return 0;
}

void showValues(int nums[], int size) {
    for (int index = 0; index < size; index++)
        cout << nums[index] << " ";
    cout << endl;
}
```

Output:
5 10 15 20 25 30 35 40

In-class practice: Array Rotation

- Write a function `Rotate` that rotates an array of size `n` by `d` elements to the left
- Use array as argument
- In the `main` function, call the function `Rotate` and show the rotated array
- Test your code

```
For example:
Input: [1 2 3 4 5 6 7], n = 7, d = 2
Output: [3 4 5 6 7 1 2]
```


Reference code: ArrayRotation.cpp https://www.geeksforgeeks.org/array-rotation/

1.5 Two-Dimensional Arrays

- A 2-D array is an array of 1-D arrays
- Use two size declarators in definition:
  - First declarator is number of rows; second is number of columns

```
const int ROWS = 4, COLS = 3;
int exams[ROWS][COLS];
```

columns

rows

r	exams[0][0]	exams[0][1]	exams[0][2]
o	exams[1][0]	exams[1][1]	exams[1][2]
w	exams[2][0]	exams[2][1]	exams[2][2]
s	exams[3][0]	exams[3][1]	exams[3][2]

- Use two subscripts to access element:

```
exams[2][2] = 86;
```

2D Array Initialization

- Two-dimensional arrays are initialized row-by-row:

```
const int ROWS = 2, COLS = 2;
int exams[ROWS][COLS] = {{84, 78}, {92, 97}};
```

84	78
92	97

- Some array elements without initial values will be set to `0` or `NULL`

```
int exams[ROWS][COLS] = {{84}, {92, 97}};
```

exams[0][1] is automatically set to 0

Passing Two-Dimensional Array to Function

- When a 2-D array is passed to a function, the parameter type must contain a size declarator for the columns
  - The size declarator for rows is optional (use empty [ ])

```
const int COLS = 2;
// Prototype
void getExams(int [][COLS], int);

// Header
void getExams(int exams[][COLS], int rows)
```

Here COLS is a global constant

- Use array name as argument in function call:

```
getExams(exams, 2);
```

Use Nested Loop to Step through 2D Array

- What is the output of the following program?

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

int sumOfArray(int n[][2], int row) {
    int total = 0;
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < 2; j++) {
            total += n[i][j];
        }
    }
    return total;
}

int main() {
    int num[3][2] = {{3, 4}, {9, 5}, {7, 1}};
    cout << "The sum is: " << sumOfArray(num, 3);
    return 0;
}
```

Output:
The sum is: 29

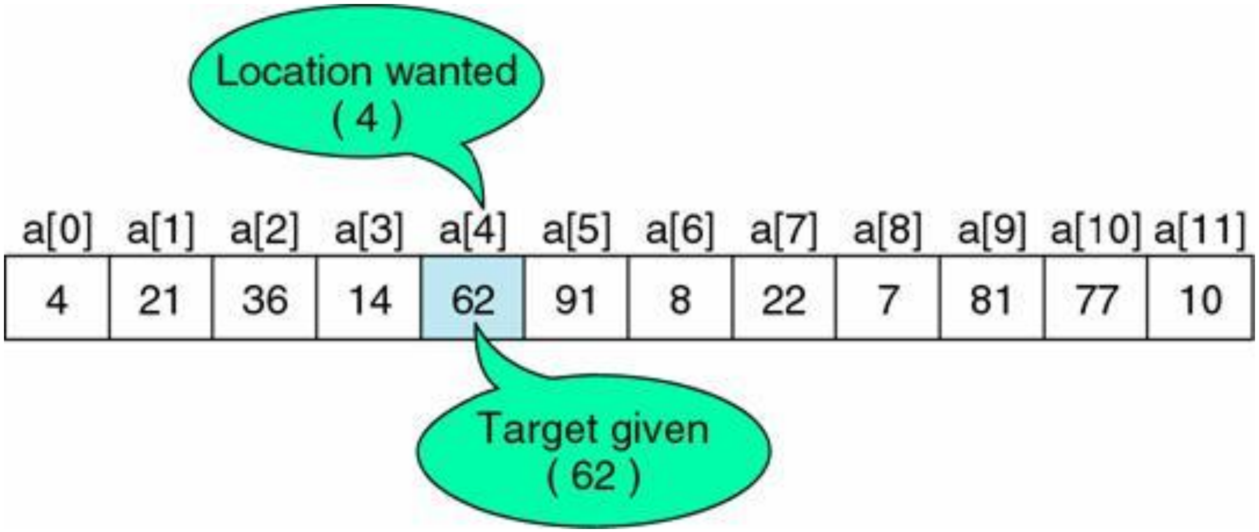
2. Searching and Sorting Arrays

- 2.1 Array Search Algorithms

- 2.2 Array Sorting Algorithms

2.1 Array Search Algorithms

- Search: locate an item in a list of data

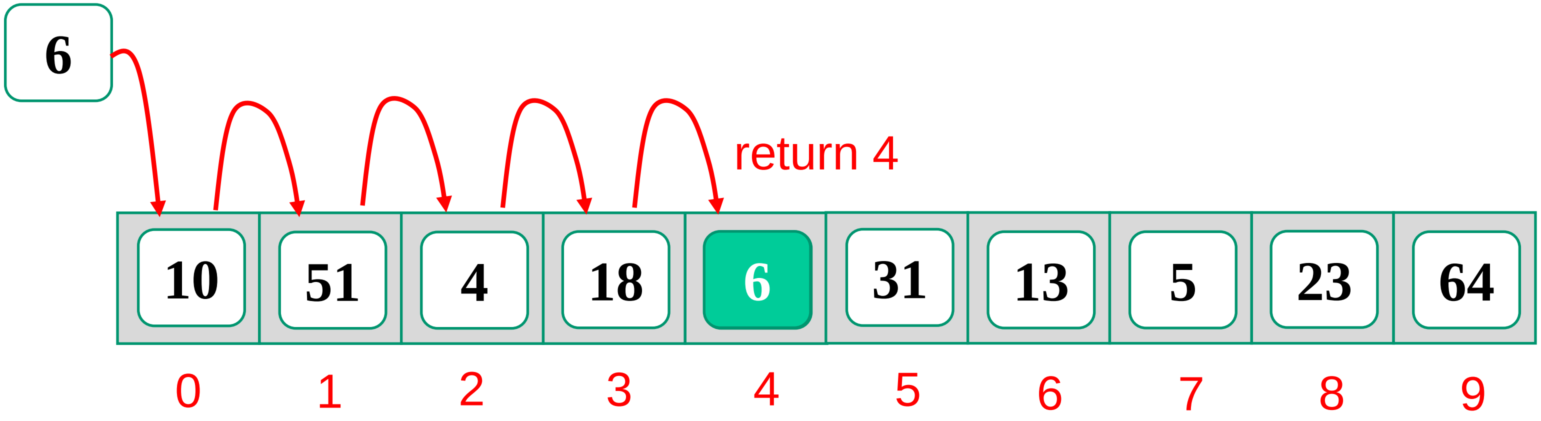


- Two algorithms we will examine:
  - Linear search
  - Binary search

 https://web.ics.purdue.edu/~cs154/lectures/lecture011.htm

Linear search

- Process
  - Compare target **x** with each element in an array in turn
  - If **x** matches with an element, return the index of this element
  - If **x** does not match with any elements, return -1



C++ implementation

```
int linearSearch(int arr[], int size, int value)
{
   int index = 0;        // Search index
   int position = -1;    // Location of the value
   bool found = false;   // Search flag

   while (index < size && !found)
   {
      if (arr[index] == value) // Value is found
      {
         found = true;           // Set the flag
         position = index;       // Record the location
      }
      index++;                   // Search the next
   }
return position;                 // Return the position
}
```


Salisbury UNIVERSITY

Linear Search - Tradeoffs

- Benefits:
  - Easy algorithm to understand
  - Array can be in any order

- Disadvantages:
  - Inefficient (slow): for array of `N` elements, examines `N/2` elements on average for value in array, `N` elements for value not in array

Binary search (Example)

Value = 6

0	1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9	10

6 > 5
Take 2nd half

L= 0	1	2	3	**M = 4**	5	6	7	8	**H = 9**
1	2	3	4	5	6	7	8	9	10

6 < 8
Take 1st half

0	1	2	3	4	**L = 5**	6	**M = 7**	8	**H = 9**
1	2	3	4	5	6	7	8	9	10

Found 6
Return index 5

0	1	2	3	4	**L = 5** **M = 5**	**H = 6**	7	8	9
1	2	3	4	5	6	7	8	9	10

Process of binary search

- Step 1: find the middle element, *middle*
- Step 2: compare *middle* with *value*
  - If *value* < *middle*, drop the second half
  - If *value* > *middle*, drop the first half
  - If *value* == *middle*, the search is finished
- Repeat above steps. If no element left, *value* is not in the array

C++ implementation

```
int binarySearch(int array[], int size, int value)
{
   int first = 0,               // First array element
       last = size - 1,         // Last array element
       middle,                  // Mid point of search
       position = -1;           // Position of search value
   bool found = false;          // Flag

   while (!found && first <= last)
   {
      middle = (first + last) / 2;     // Middle point
      if (array[middle] == value)      // If value = middle
      {
         found = true;
         position = middle;
      }
      else if (array[middle] > value)  // If value < middle
         last = middle - 1;            // Search lower half
      else
         first = middle + 1;           // If value > middle
   }                                   // Search upper half
   return position;
}
```

Binary Search - Tradeoffs

- Benefits:
  - Much more efficient than linear search. For array of N elements, performs at most $\log_2 N$ comparisons

1	3	5	8	10	15	18	20

After 1st search:

10	15	18	20

$8 \times \frac{1}{2} = 4$

After 2nd search:

18	20

$8 \times (\frac{1}{2})^2 = 2$

After 3rd search:

18

$8 \times (\frac{1}{2})^3 = 1$

- Disadvantages:
  - Requires that array elements be sorted

Linear search VS binary search

Linear search	Binary search
+ No need to sort elements	- Need to sort elements first
Only *equality* comparisons	*Equality* & *ordering* comparisons
Sequential access to the data	Random access to the data
- Search is inefficient (slow)	+ Search is efficient (fast)

In-class practice

- **Search Insert Position**
  - Given a sorted array in ascending order and a target value
  - Use binary search algorithm to return the index if the target is found. If not, return the index where it would be if it is inserted in order
  - You may assume no duplicates in the array

Example :

Input: [1,3,5,6], 5
Output: 2

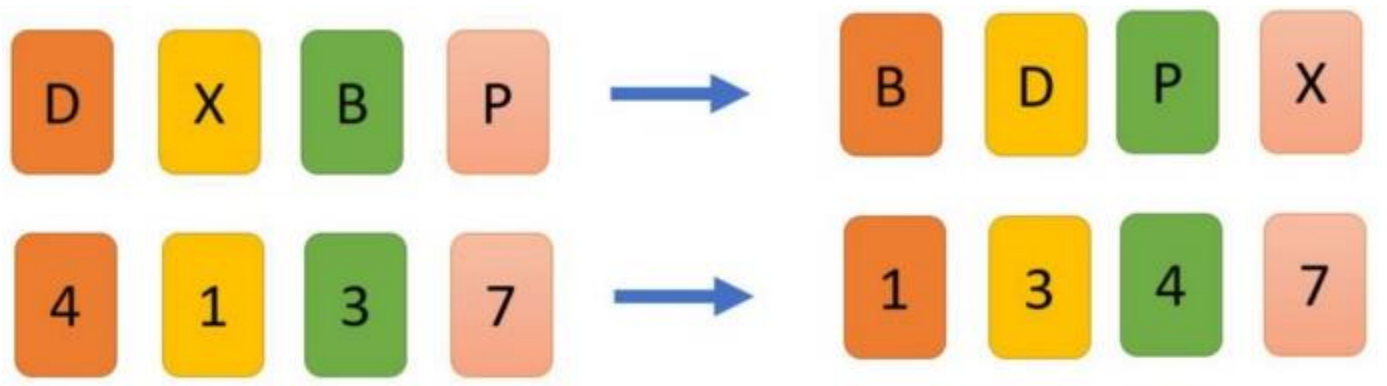
Input: [1,3,5,6], 2
Output: 1

 Reference code: InsertPosition.cpp

https://gist.github.com/hyfrey/3803948

2.2 Array Sorting Algorithms

- Sort: arrange values into an order
  - Alphabetical
  - Ascending numeric
  - Descending numeric



- Two algorithms considered here:
  - Bubble sort
  - Selection sort

Salisbury University

Bubble Sort (Example)

- Sort an array in ascending order

First pass:

7 > 2, swap

7	2	3	8	9	1

7 > 3, swap

2	7	3	8	9	1

… …

9 > 1, swap

2	3	7	8	9	1

2	3	7	8	1	**9**

The largest value 9 is in the correct position

Second pass:

2 < 3, no swap

2	3	7	8	1	**9**

… …

8 > 1, swap

2	3	7	8	1	**9**

2	3	7	1	**8**	**9**

The 2nd largest value 8 is in the correct position

Note: the 2nd pass will not involve the last element bcz 9 is the largest

Bubble Sort

Third pass:

2 < 3, no swap

2	3	7	1	**8**	**9**

…. ….

7 > 1, swap

2	3	7	1	**8**	**9**

2	3	1	**7**	**8**	**9**

The 3rd largest value 7 is in the correct position

Note: the 3rd pass will not involve the last two elements bcz they are sorted

Fourth pass:

2 < 3, no swap

2	3	1	**7**	**8**	**9**

3 > 1, swap

2	3	1	**7**	**8**	**9**

2	1	**3**	**7**	**8**	**9**

Bubble Sort

Fifth pass:

2 > 1, swap

2	1	**3**	**7**	**8**	**9**

1	**2**	**3**	**7**	**8**	**9**

- There are `(n-1)` passes. `n` is the number of elements in the array

Pass	**1**	**2**	**…**	**n-2**	**n-1**
# of compares	n-1	n-2	…	2	1

In total `(n-1)+(n-2)+ ... +(2)+(1) = n(n-1)/2` comparisons.

Bubble Sort

Process:

- Compare 1st and 2nd elements
  - If out of order, exchange them to put in order
- Move down one element, compare 2nd and 3rd elements, exchange if necessary. Continue until end of array
- Pass through array (*one element less*) again, exchanging as necessary
- Repeat until the last pass

C++ Implementation

```
void bubbleSort(int array[], int size) {
    int maxElement;
    int index;

    for (maxElement = size - 1; maxElement > 0; maxElement--) {
        for (index = 0; index < maxElement; index++) {
            if (array[index] > array[index + 1]) {
                swap(array[index], array[index + 1]);
            }
        }
    }
}

void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}
```

Hold the subscript of the last element to be compared

Used as an array subscript in one of the loops

Reference parameters

Bubble Sort - Tradeoffs

- Benefit:
  - Easy to understand and implement

- Disadvantage:
  - Inefficient: slow for large arrays
    - Too much unnecessary swaps

Question: How many swaps for bubble sort in the worst case?

`n(n-1)/2` when the array is reversely sorted

Selection Sort

- Concept for sort in ascending order:
  - Locate smallest element in array.
    Exchange it with element in position 0
  - Locate next smallest element in array.
    Exchange it with element in position 1.
  - Continue until all elements are arranged in order

Selection Sort - Example

Array `numlist` contains:

11	2	29	3

1. Smallest element is `2`. Exchange `2` with element in 1st position in array:

11	2	29	3

⇨

2	11	29	3

Example (Continued)

2. Next smallest element is `3`. Exchange `3` with element in 2nd position in array:

2	11	29	3

⇨

2	**3**	29	11

3. Next smallest element is `11`. Exchange `11` with element in 3rd position in array:

2	**3**	29	11

⇨

2	**3**	**11**	**29**

C++ Implementation

```
void selectionSort(int array[], int size) {
    int minIndex, minValue;

    for (int start = 0; start < (size - 1); start++) {
        minIndex = start;
        minValue = array[start];
        for (int index = start + 1; index < size; index++) {
            if (array[index] < minValue) {
                minValue = array[index];
                minIndex = index;
            }
        }
        swap(array[minIndex], array[start]);
    }
}
```

Selection Sort - Tradeoffs

- Benefit:
  - More efficient than Bubble Sort, since fewer exchanges/swaps

- Disadvantage:
  - May not be as easy as Bubble Sort to understand

Question: How many comparisons for selection sort?

In total `(n-1)+(n-2)+ ... +(2)+(1) = n(n-1)/2` comparisons.

Question: How many swaps for selection sort in the worst case?

`(n-1)` when the array is reversely sorted

In-class practice

- For an array with `n` elements, the bubble sort needs `n-1` passes. However, if the array elements are in order in the midway, there is no need to execute the subsequent passes.
- Write code to implement the above optimized bubble sort algorithm.
- Test you code.

Reference code: OptBubbleSort.cpp

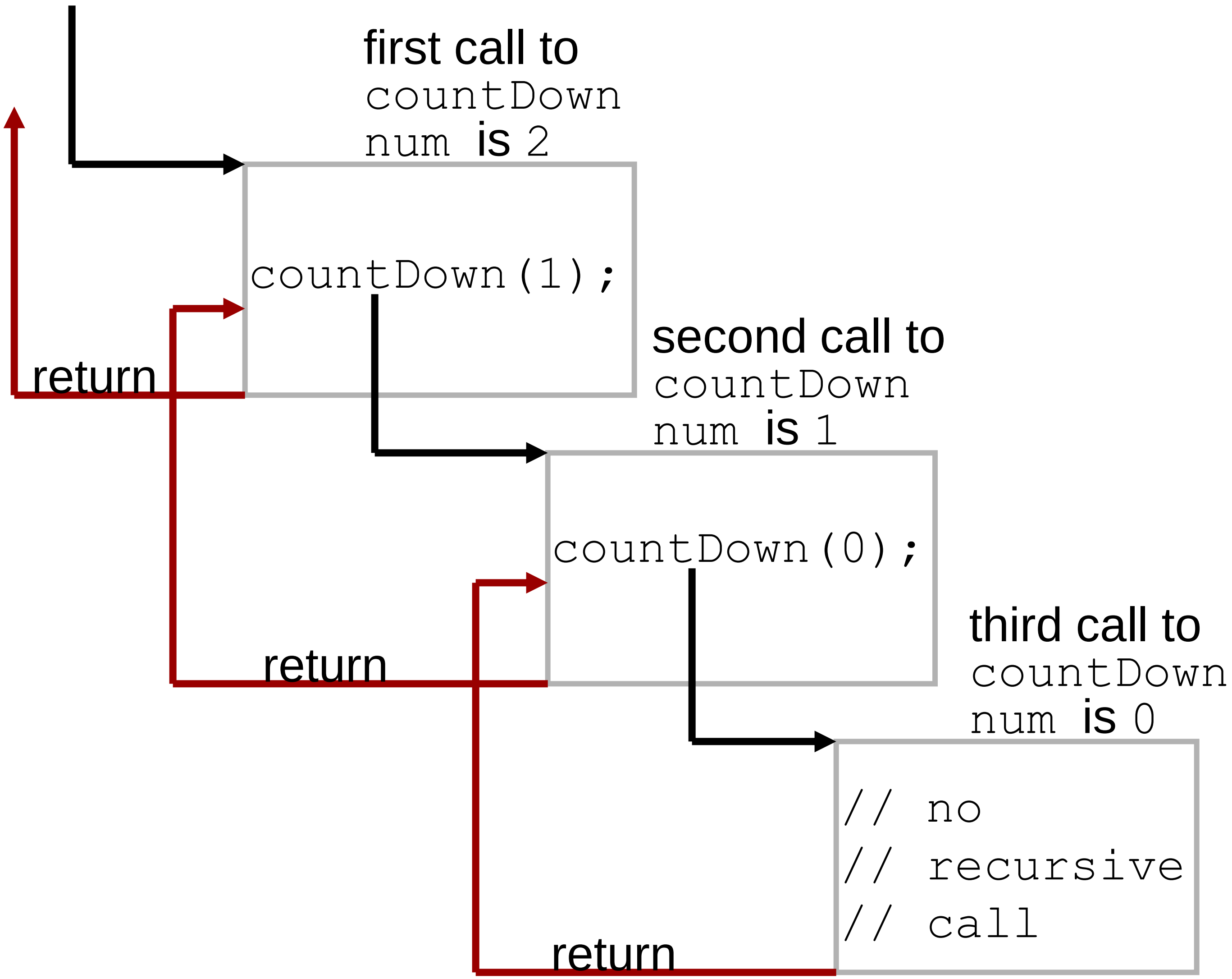
Recursion Function

- A recursive function is one that **calls itself**

```
void countDown(int num){
    if (num == 0)                    // stop condition
        cout << "Blastoff!";
    else{
        cout << num << "...\n";
        countDown(num-1);            // recursive call
    }
}
```

- Assume the input argument is 2:
  - `countDown(2)` outputs `2...`, then it calls `countDown(1)`
  - `countDown(1)` outputs `1...`, then it calls `countDown(0)`
  - `countDown(0)` outputs `Blastoff!`, then returns to `countDown(1)`
  - `countDown(1)` returns to `countDown(2)`
  - `countDown(2)` returns to the calling function

What Happens When Called?



output:

2...

1...

Blastoff!

Solving Problems with Recursion

- Two important steps:
  - Define the recursive function
  - Define the stop condition

- Example: factorial calculation

  `n! = n*(n-1)*(n-2)*...*3*2*1` if `n > 0`

  `n! = 1` if `n = 0`

  - Define the recursive function:

    `n! = n * (n-1)!`

  - Define the stop condition:

    `0! = 1` (base case)

Recursive Factorial Function

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

int factorial(int);    // Function prototype

int main(){
    int number;
    cout << "Enter an integer value to display its factorial: ";
    cin >> number;

    cout << "The factorial of " << number << " is " << factorial(number);
    return 0;
}

int factorial(int n){
    if (n == 0)
        return 1;                              // Base case
    else
        return n * factorial(n - 1); // Recursive case
}
```

Enter an integer value to display its factorial: 5
The factorial of 5 is 120

Example code: Factorial.cpp

In-class practice

- The Fibonacci numbers are the numbers in the following integer sequence

  `0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …`

- In mathematical terms, the sequence `Fn` of Fibonacci numbers is defined as

  `Fn = Fn-1 + Fn-2`

  where:

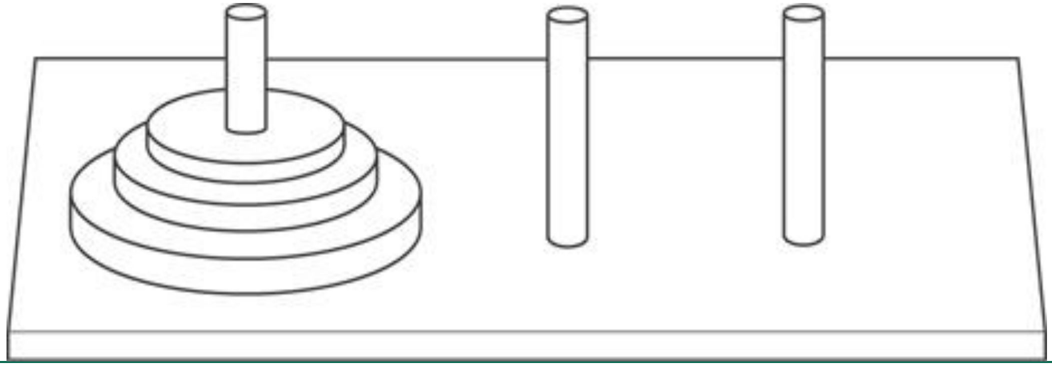
  `F0 = 0` and `F1 = 1`

- Use recursive function to calculate and display the first `10` Fibonacci numbers
- Test your code

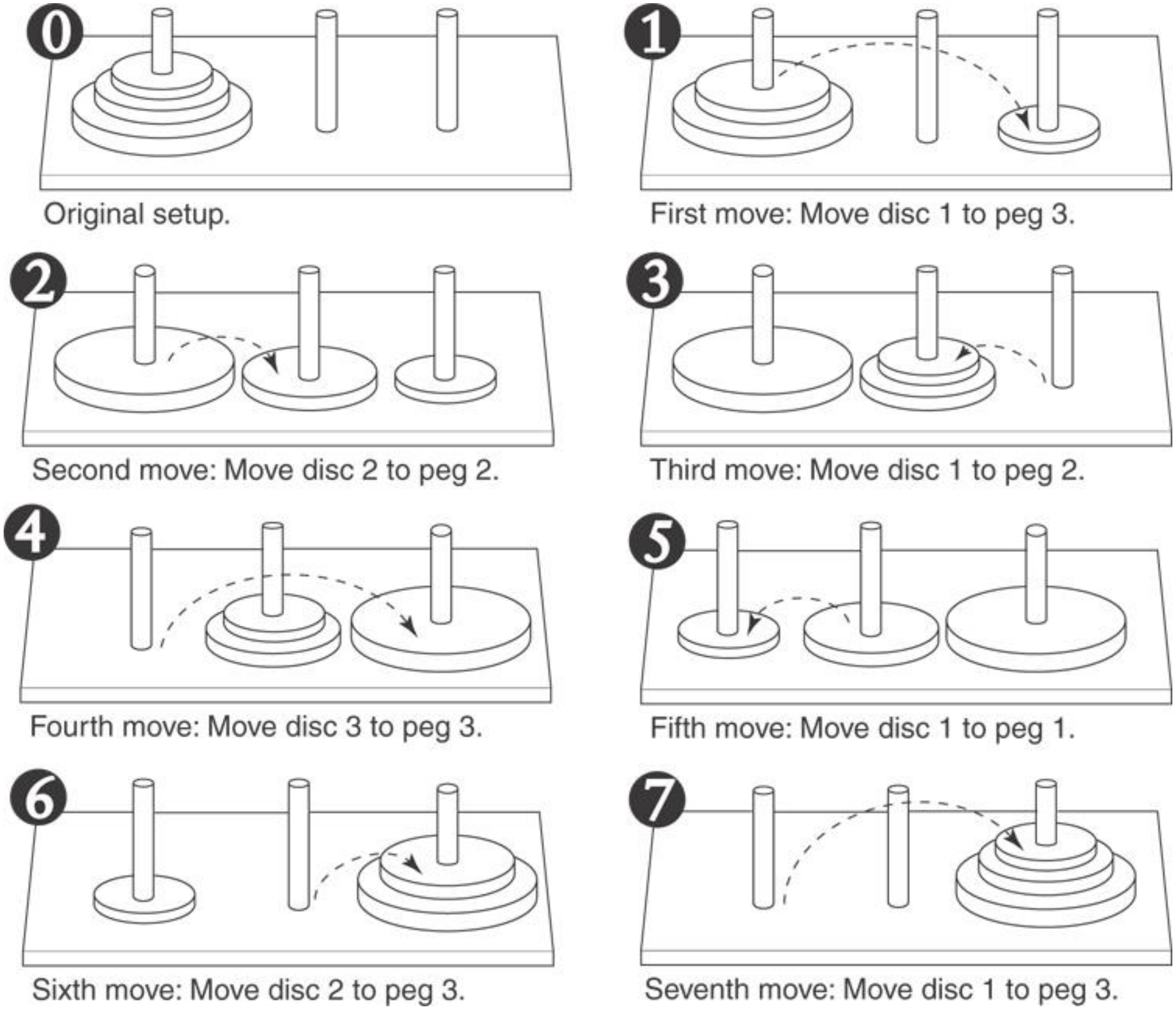
Reference code: Pr20-6.cpp

Application: The Towers of Hanoi

- The game uses three pegs and a set of discs, stacked on one of the pegs
- The object of this game is to move the discs from the first peg to the third peg
- Here are the rules:
  - Only one disc may be moved at a time
  - A disc cannot be placed on top of a smaller disc
  - All discs must be stored on a peg except while being moved



Moving Three Discs



Original setup.

First move: Move disc 1 to peg 3.

Second move: Move disc 2 to peg 2.

Third move: Move disc 1 to peg 2.

Fourth move: Move disc 3 to peg 3.

Fifth move: Move disc 1 to peg 1.

Sixth move: Move disc 2 to peg 3.

Seventh move: Move disc 1 to peg 3.

The Towers of Hanoi

- Algorithm
  - *To move* `n` *discs from peg A to peg C, using peg B as a temporary peg:*
    If `n > 0` *Then*
      Move `n - 1` *discs from peg A to peg B, using peg C as a temporary peg.*

      Move the remaining disc from the peg A to peg C.

      Move `n - 1` *discs from peg B to peg C, using peg A as a temporary peg.*

    End If

The Towers of Hanoi

- C++ Implementation
  - Refer to “Pr20-10.cpp”

Reference

- The teaching materials of this course refer to:
  - Professor Xiaohong (Sophie) Wang. COSC 120 teaching materials
    - Salisbury University
  - Textbook:
    - Starting Out with C++: From Control Structures through Objects, by Tony Gaddis, Pearson (9th Edition)
    - Instructor materials of the above textbook (All rights reserved)