IMAM ABDULRAHMAN BIN FAISAL UNIVERSITY
College of Computer Science & IT
Department of CS

Welcome to CS 221:Fundamentals of Programming

Weeks (7): Functions – Part 3

Chapters 9 and 10 (Zak Textbook)

Chapter 4,5 (Savitch Textbook)

Outline

- Array in functions
- Functions overloading
- Recursion

ARRAY IN FUNCTIONS

Arrays in Functions

- Indexed variables can be arguments to functions
  - Example: If a program contains these declarations:

```
int i, n, a[10];
void my_function(int n);
```

  - Variables a[0] through a[9] are of type int, making these calls legal:

```
my_function(a[0]);
my_function(a[3]);
my_function(a[i]);
```

Passing array elements

Indexed Variable as an Argument

```
//Illustrates the use of an indexed variable as an argument.
//Adds 5 to each employee's allowed number of vacation days.
#include <iostream>

const int NUMBER_OF_EMPLOYEES = 3;

int adjust_days(int old_days);
//Returns old_days plus 5.

int main( )
{
    using namespace std;
    int vacation[NUMBER_OF_EMPLOYEES], number;

    cout << "Enter allowed vacation days for employees 1"
         << " through " << NUMBER_OF_EMPLOYEES << ":\n";
    for (number = 1; number <= NUMBER_OF_EMPLOYEES; number++)
        cin >> vacation[number-1];

    for (number = 0; number < NUMBER_OF_EMPLOYEES; number++)
        vacation[number] = adjust_days(vacation[number]);

    cout << "The revised number of vacation days are:\n";
    for (number = 1; number <= NUMBER_OF_EMPLOYEES; number++)
        cout << "Employee number " << number
             << " vacation days = " << vacation[number-1] << endl;

    return 0;
}

int adjust_days(int old_days)
{
    return (old_days + 5);
}
```

Sample Dialogue

```
Enter allowed vacation days for employees 1 through 3:
10 20 5
The revised number of vacation days are:
Employee number 1 vacation days = 15
Employee number 2 vacation days = 25
Employee number 3 vacation days = 10
```

Arrays as Function Arguments

- A formal parameter can be for an entire array
  - Such a parameter is called an array parameter
    - It is not a call-by-value parameter
    - It is not a call-by-reference parameter
    - Array parameters **behave much like** call-by-reference parameters

Base Address of an Array and Array in Computer Memory

- The base address of an array is the address, or memory location of the first array component
- If `list` is a one-dimensional array, its base address is the address of `list[0]`
- When we pass an array as a parameter, the base address of the actual array is passed to the formal parameter

Base Address of an Array and Array in Computer Memory

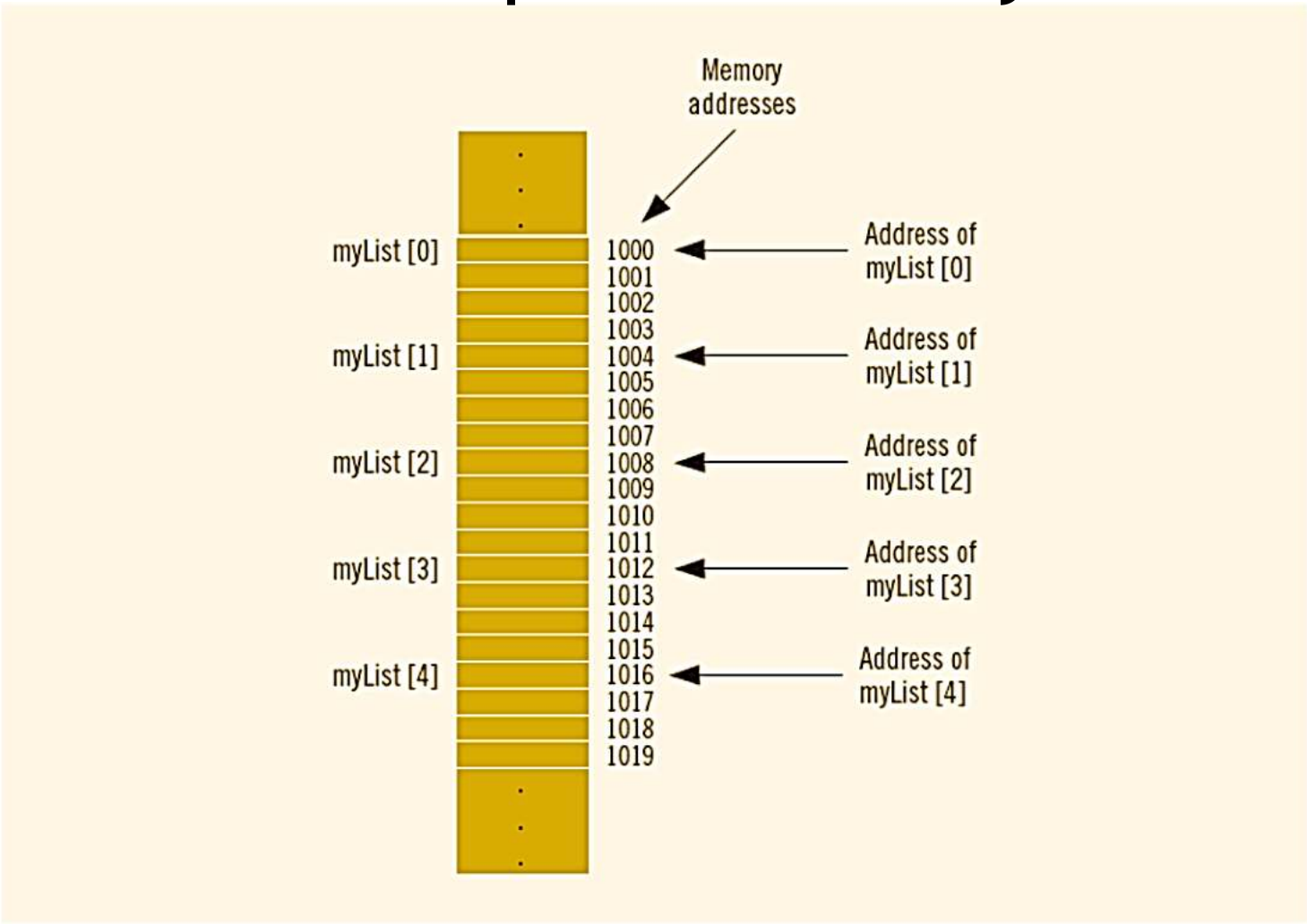


FIGURE 9-6 Array `myList` and the addresses of its components

Array Parameter Declaration

- An array parameter is indicated using empty brackets in the parameter list such as

```
void   fill_up(int a[ ],int size);
```

Function Calls With Arrays

- If function fill_up is declared in this way:

```
void fill_up(int a[ ], int size);
```

- and array score is declared this way:

```
int score[5], number_of_scores;
```

- fill_up is called in this way:

```
fill_up(score, number_of_scores);
```

Example: Function Calls With Arrays

Function with an Array Parameter

Function Declaration

```
void fill_up(int a[], int size);
//Precondition: size is the declared size of the array a.
//The user will type in size integers.
//Postcondition: The array a is filled with size integers
//from the keyboard.
```

Function Definition

```
//Uses iostream:
void fill_up(int a[], int size)
{
    using namespace std;
    cout << "Enter " << size << " numbers:\n";
    for (int i = 0; i < size; i++)
        cin >> a[i];
    size--;
    cout << "The last array index used is " << size << endl;
}
```

Function Call Details

- A formal parameter is identified as an array parameter by the [ ]'s with no index expression

```
void fill_up(int a[],int size);
```

- An array argument does not use the [ ]'s

```
fill_up(score, number_of_scores);
```

Array Formal Parameters

- An array formal parameter is a placeholder for the argument
  - When an array is an argument in a function call, an action performed on the array parameter is performed on the array argument
  - The values of the indexed variables can be changed by the function

Array Argument Details

- What does the computer know about an array?
  - The base type
  - The address of the first indexed variable
  - The number of indexed variables
- What does a function know about an array argument?
  - The base type
  - The address of the first indexed variable

Array Parameter Considerations

- Because a function does not know the size of an array argument…
  - The programmer should include a formal parameter that specifies the size of the array
  - The function can process arrays of various sizes
    - Function `fill_up` from previous example can be used to fill an array of any size:

```
fill_up(score, 5);
fill_up(time, 10);
```

A Complete Example

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

double getAverage(int arr[], int
size);

 int main () {
/* an int array with 5 elements */

int balance[5] = {1000, 2, 3, 17,
50};
 double avg;

/* pass pointer to the array as an
argument */

avg = getAverage( balance, 5 ) ;
 /* output the returned value */
cout<< "Average value is:“ << avg ;
return 0;
}
```

```
double getAverage(int arr[], int
size) {
      int i;
     double avg = 0.0;
     double sum = 0.0 ;
        for (i = 0; i < size; ++i)
{
           sum += arr[i];
}
       avg = sum / size;
         return avg;

}
```

const Modifier

- Array parameters allow a function to change the values stored in the array argument
- If a function should not change the values of the array argument, use the modifier `const`
- An array parameter modified with `const` is a constant array parameter
  - Example:

```
void show_the_world(const int a[], int size);
```

Using const With Arrays

- If `const` is used to modify an array parameter:
  - `const` is used in both the function declaration(prototype) and definition to modify the array parameter
  - The compiler will issue an error if you write code that changes the values stored in the array parameter

Using const With Arrays

EXAMPLE 9-6

```
     //Function to initialize an int array to 0.
     //The array to be initialized and its size are passed
     //as parameters. The parameter listSize specifies the
     //number of elements to be initialized.
 void initializeArray(int list[], int listSize)
 {
     int index;

     for (index = 0; index < listSize; index++)
         list[index] = 0;
 }

    //Function to print the elements of an int array.
    //The array to be printed and the number of elements
    //are passed as parameters. The parameter listSize
    //specifies the number of elements to be printed.
void printArray(const int list[], int listSize)
{
    int index;

    for (index = 0; index < listSize; index++)
        cout << list[index] << " ";
}
```


19

Function Calls and const

- If a function with a constant array parameter calls another function using the const array parameter as an argument…
  - The called function must use a constant array parameter as a placeholder for the array
  - The compiler will issue an error if a function is called that does not have a const array parameter to accept the array argument

const Parameters Example

- ```
  double compute_average(int a[ ], int size);

  void show_difference(const int a[ ], int size)
  {
        double average = compute_average(a, size);
        …
  }
  ```
- `compute_average` has no constant array parameter
- This code generates an error message because `compute_average` could change the array parameter
  ```

Returning An Array

- Recall that functions can return a value of type int, double, char, …, or a class type
- **Functions cannot return arrays**
- We will learn later how to return a pointer to an array

Array in Functions Conclusion

- Can you
  - Write a function definition for a function called `one_more`, which has a formal parameter for an array of integers and increases the value of each array element by one.
  - Are other formal parameters needed?

```
void one_more( int a[], int size);
```

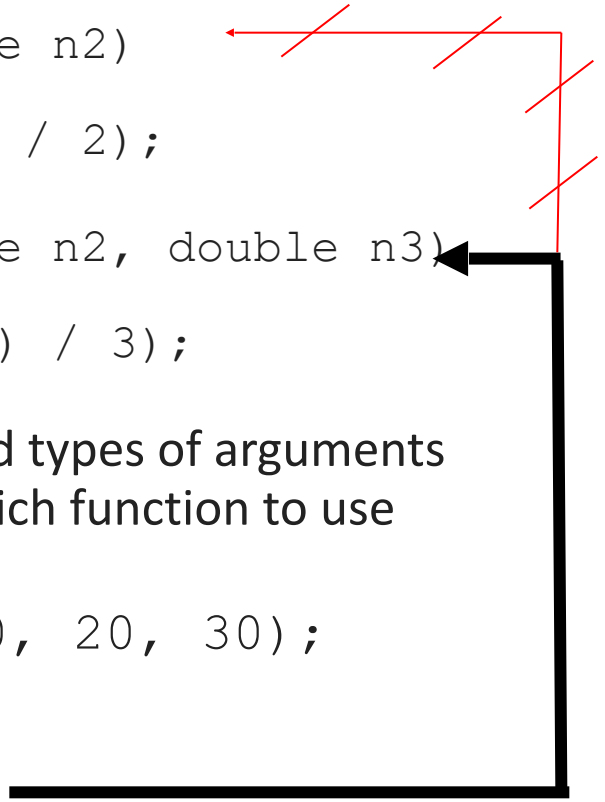
FUNCTION OVERLOADING

Overloading Function Names

- C++ allows more than one definition for the same function name
  - Very convenient for situations in which the “same” function is needed for different numbers or types of arguments
- Overloading a function name means providing more than one declaration and definition using the same function name

Overloading Examples

- ```
  double ave(double n1, double n2)
  {
            return ((n1 + n2) / 2);
  }
  ```
- ```
  double ave(double n1, double n2, double n3)
  {
       return (( n1 + n2 + n3) / 3);
  }
  ```
  - Compiler checks the number and types of arguments in the function call to decide which function to use



```
cout << ave( 10, 20, 30);
```

uses the second definition

Overloading Details

- Overloaded functions
  - Must have different numbers of formal parameters
    AND / OR
  - Must have at least one different type of parameter
  - Must return a value of the same type

Overloading Example

Overloading a Function Name

```
//Illustrates overloading the function name ave.
#include <iostream>

double ave(double n1, double n2);
//Returns the average of the two numbers n1 and n2.

double ave(double n1, double n2, double n3);
//Returns the average of the three numbers n1, n2, and n3.

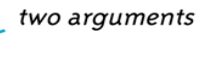
int main()
{
    using namespace std;
    cout << "The average of 2.0, 2.5, and 3.0 is "
         << ave(2.0, 2.5, 3.0) << endl;

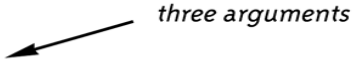
    cout << "The average of 4.5 and 5.5 is "
         << ave(4.5, 5.5) << endl;

    return 0;
}

double ave(double n1, double n2)
{
    return ((n1 + n2)/2.0);
}

double ave(double n1, double n2, double n3)
{
    return ((n1 + n2 + n3)/3.0);
}
```





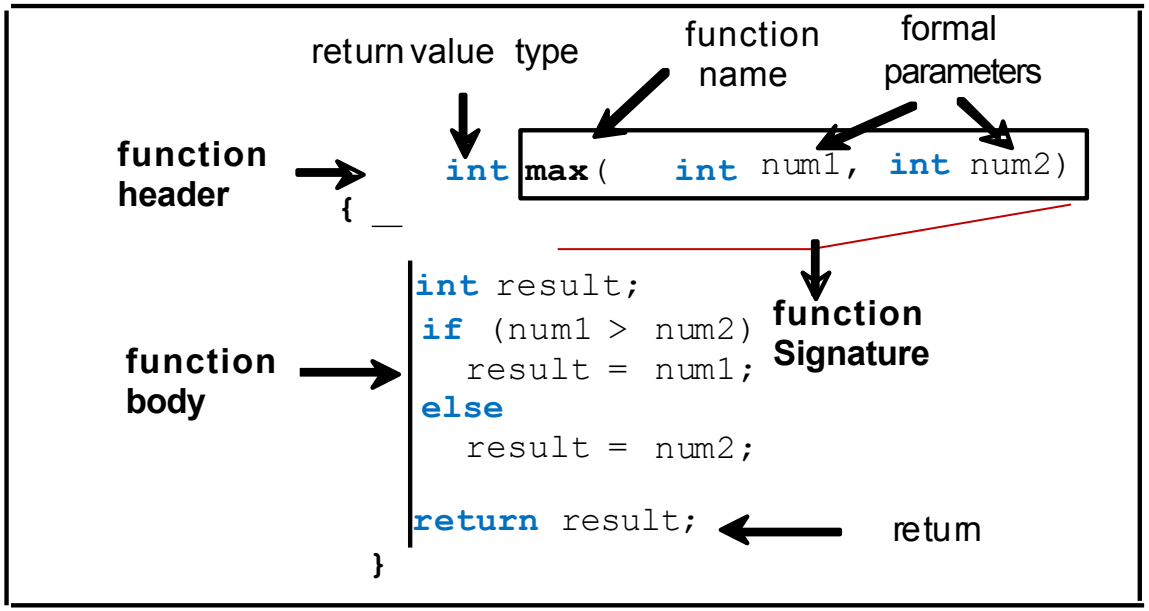
Output

```
The average of 2.0, 2.5, and 3.0 is 2.50000
The average of 4.5 and 5.5 is 5.00000
```

Function Overloading (continued)

- Function overloading: creating several functions with the same name
- The signature of a function consists of the function name and its formal parameter list
- Two functions have different signatures if they have
  - either different names or
  - different formal parameter lists
- Note that the signature of a function **does not include the return type** of the function

Define a function



Call a function

```
int z = max(i, j);
```

actual parameters (arguments)

Calling a function is equivalent to execut ing the function body

Declare a function (prototype)

```
int max(int num1, int num2);
```

Function Overloading (continued)

- Correct function overloading:

```
void functionXYZ()
void functionXYZ(int x, double y)
void functionXYZ(double one, int y)
void functionXYZ(int x, double y, char ch)
```

- Syntax error:

```
void functionABC(int x, double y)
int functionABC(int x, double y)
```

Automatic Type Conversion

- Given the definition

```
double mpg(double miles, double gallons)
{
    return (miles / gallons);
}
```

what will happen if `mpg` is called in this way?

```
cout << mpg(45, 2) << " miles per gallon";
```

- The values of the arguments will automatically be converted to type double (45.0 and 2.0)

Type Conversion Problem

- Given the previous mpg definition and the following definition in the same program

```
int mpg (int goals, int misses)
    // returns the Measure of Perfect Goals
{
    return (goals – misses);
}
```

what happens if mpg is called this way now?

```
cout << mpg(45, 2) << " miles per gallon";
```

  - The compiler chooses the function that matches parameter types, so the Measure of Perfect Goals will be calculated

Do not use the same function name for unrelated functions

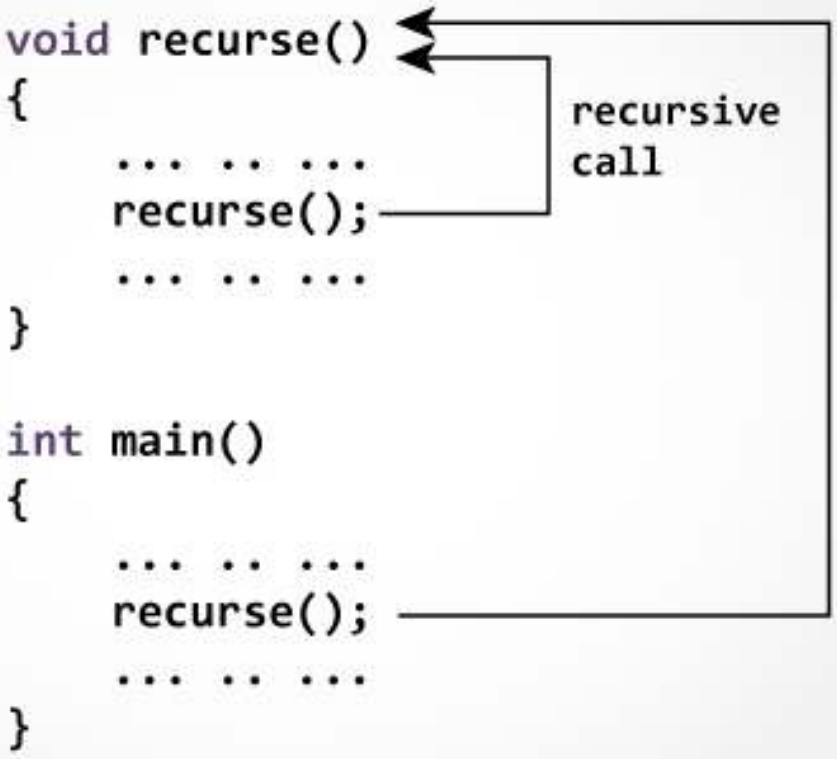
RECURSION AND RECURSIVE FUNCTION

Recursive Functions for Tasks

- A recursive function contains a call to itself
- When breaking a task into subtasks, it may be that the subtask is a smaller example of the same task
  - Searching an array could be divided into searching the first and second halves of the array
  - Searching each half is a smaller version of searching the whole array
  - Tasks like this can be solved with recursive functions

Recursive Functions for Tasks

How does recursion work?



Case Study: Vertical Numbers

- Problem Definition:

  - `void write_vertical(int n);`
    //Precondition: `n >= 0`
    //Postcondition: n is written to the screen vertically
    // with each digit on a separate line

Case Study: Vertical Numbers

- Algorithm design:
  - Simplest case:

    If n is one digit long, write the number
  - Typical case:

    1) Output all but the last digit vertically

    2) Write the last digit

    - Step 1 is a smaller version of the original task
    - Step 2 is the simplest case

Case Study: Vertical Numbers (cont.)

- The `write_vertical` algorithm:

```
if (n < 10)
  {
        cout << n << endl;
  }
 else  // n is two or more digits long
  {
  write_vertical(n with the last digit removed);
  cout << the last digit of n << endl;
  }
```

Case Study: Vertical Numbers (cont.)

- Translating the pseudocode into C++
  - `n/10` returns n with the last digit removed
    - `124/10=12`
  - `n%10` returns the last digit of `n`
    - `124%10=4`
- Removing the first digit would be just as valid for defining a recursive solution
  - It would be more difficult to translate into C++

Case Study: Vertical Numbers (cont.)

A Recursive Output Function (*part 1 of 2*)

```
//Program to demonstrate the recursive function write_vertical.
#include <iostream>
using namespace std;

void write_vertical(int n);
//Precondition: n >= 0.
//Postcondition: The number n is written to the screen vertically
//with each digit on a separate line.

int main( )
{
    cout << "write_vertical(3):" << endl;
    write_vertical(3);

    cout << "write_vertical(12):" << endl;
    write_vertical(12);

    cout << "write_vertical(123):" << endl;
    write_vertical(123);

    return 0;
}

//uses iostream:
void write_vertical(int n)
{
    if (n < 10)
    {
        cout << n << endl;
    }
    else //n is two or more digits long:
    {
        write_vertical(n/10);
        cout << (n%10) << endl;
    }
}
```

Case Study: Vertical Numbers (cont.)

A Recursive Output Function (*part 2 of 2*)

Sample Dialogue

```
write_vertical(3):
3
write_vertical(12):
1
2
write_vertical(123):
1
2
3
```

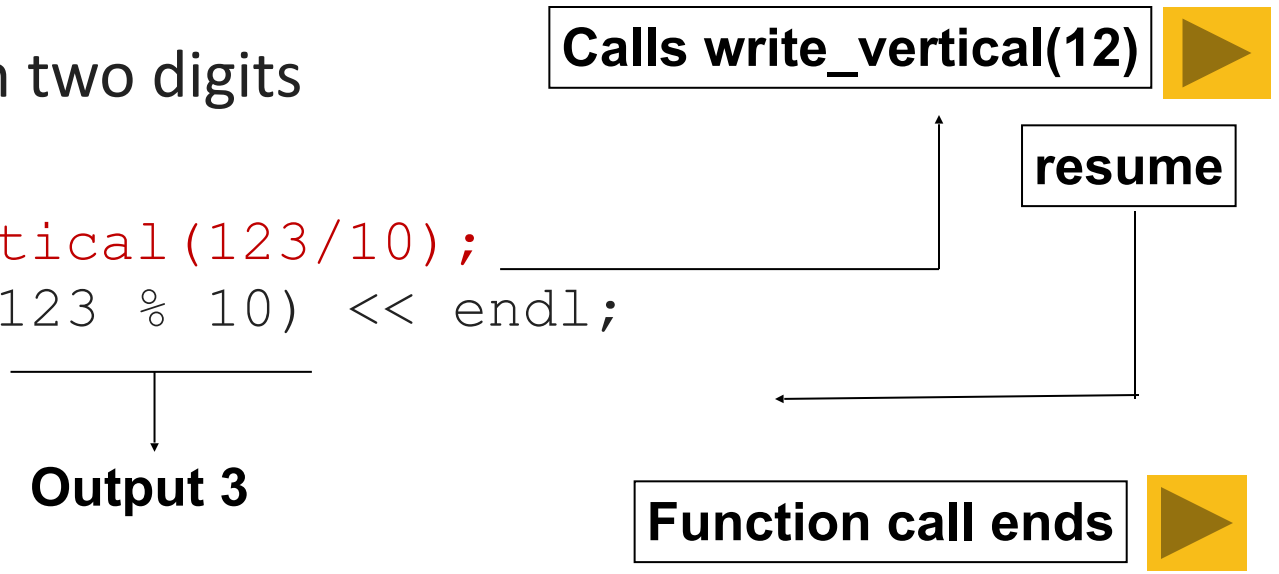
Tracing a Recursive Call

- write_vertical(123)

```
if (123 < 10)
    { cout << 123 << endl;
    }

 else
// n is more than two digits
   {
    write_vertical(123/10);
    cout << (123 % 10) << endl;

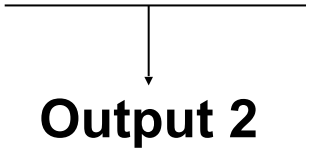
    }
```



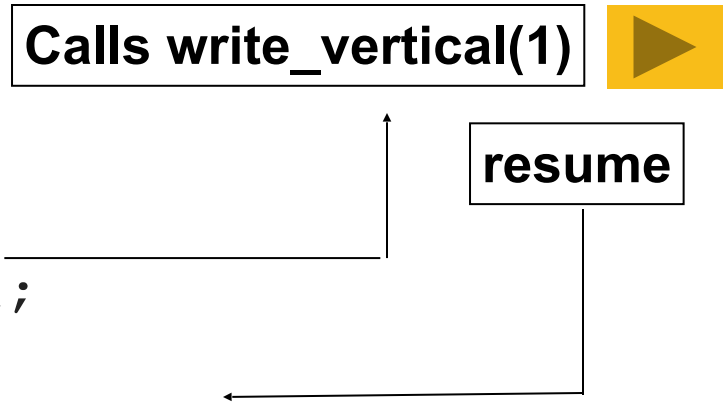
Tracing write_vertical(12)

- ```
  write_vertical(12)
  if (12 < 10)
      { cout << 12 << endl;
      }
   else
  ```
  // n is more than two digits
  ```
      {

    write_vertical(12/10);
    cout << (12 % 10) << endl;
    }
  ```



Output 2




  ```

Tracing write_vertical(1)

- write_vertical(1)
  if (1 < 10) → **Simplest case is now true**
  { cout << 1 << endl;
  }

Output 1

Function call ends

```
else
// n is more than two digits
  {
        write_vertical(1/10);
         cout << (1 % 10) << endl;
    }
```

A Closer Look at Recursion

- `write_vertical` uses recursion
  - Used no new keywords or anything "new"
  - It simply called itself with a different argument
- Recursive calls are tracked by
  - Temporarily stopping execution at the recursive call
    - The result of the call is needed before proceeding
  - Saving information to continue execution later
  - Evaluating the recursive call
  - Resuming the stopped execution

How Recursion Ends

- Eventually one of the recursive calls must not depend on another recursive call
- Recursive functions are defined as
  - One or more cases where the task is accomplished by using recursive calls to do a smaller version of the task
  - One or more cases where the task is accomplished without the use of any recursive calls
    - These are called **base cases** or **stopping cases.**

"Infinite" Recursion

- A function that never reaches a base case, in theory, will run forever
  - In practice, the computer will often run out of resources and the program will terminate abnormally

Example: Infinite Recursion

- Function write_vertical, without the base case

```
void new_write_vertical(int n)
    {
        new_write_vertical (n /10);
        cout << n % 10 << endl;
    }
```

will eventually call **`write_vertical(0)`**, which will call **`write_vertical(0)`**, which will call **`write_vertical(0)`**, which will call **`write_vertical(0)`**, which will call **`write_vertical(0)`**, which will call **`write_vertical(0)`**, ……

Simple Example

With Loop

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

With Recursion

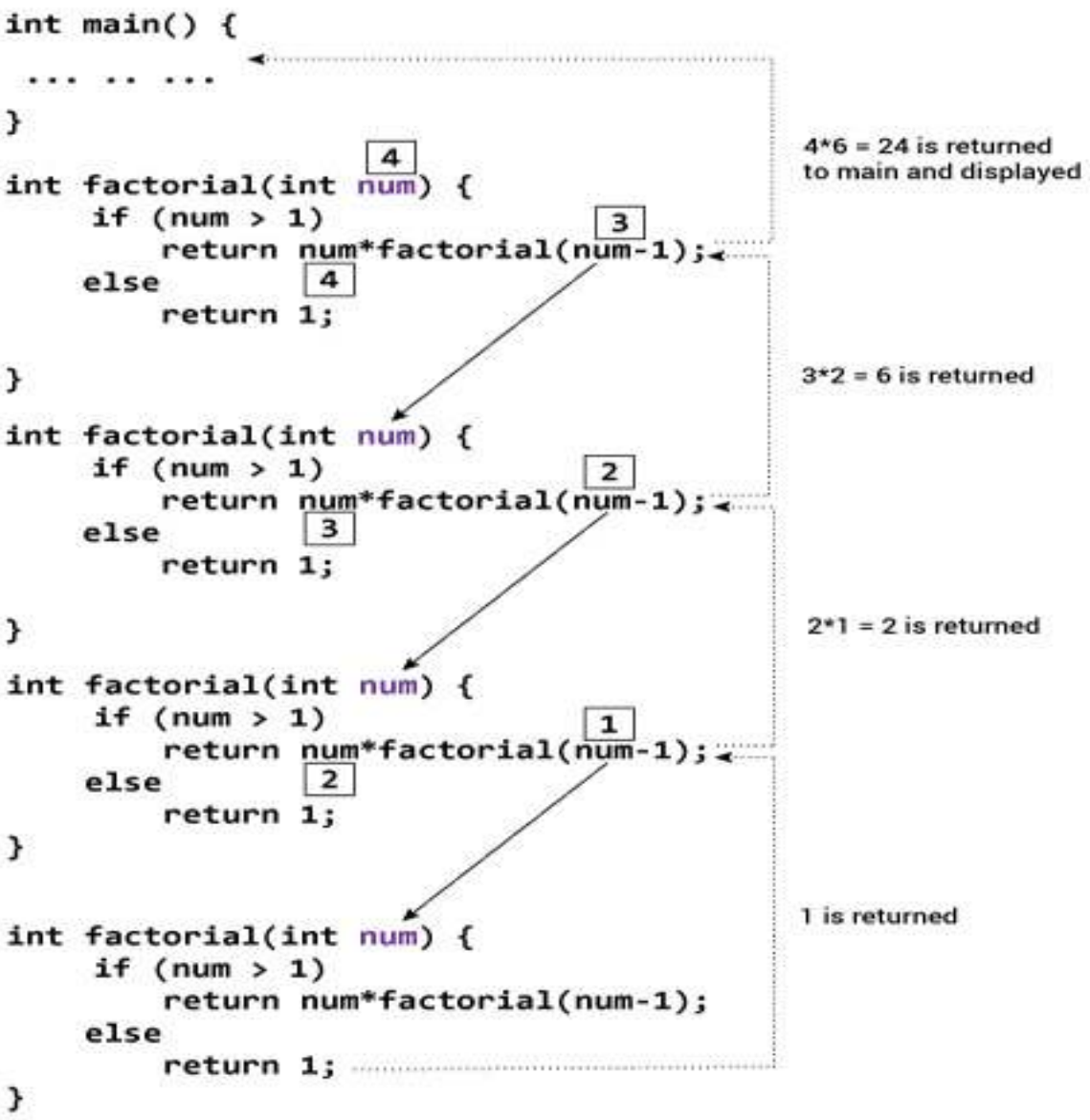
```
int factorial (int n)
{
  if (n == 1)
  return 1;
  else
  return n*factorial(n-1);
}
```

Factorial of 4

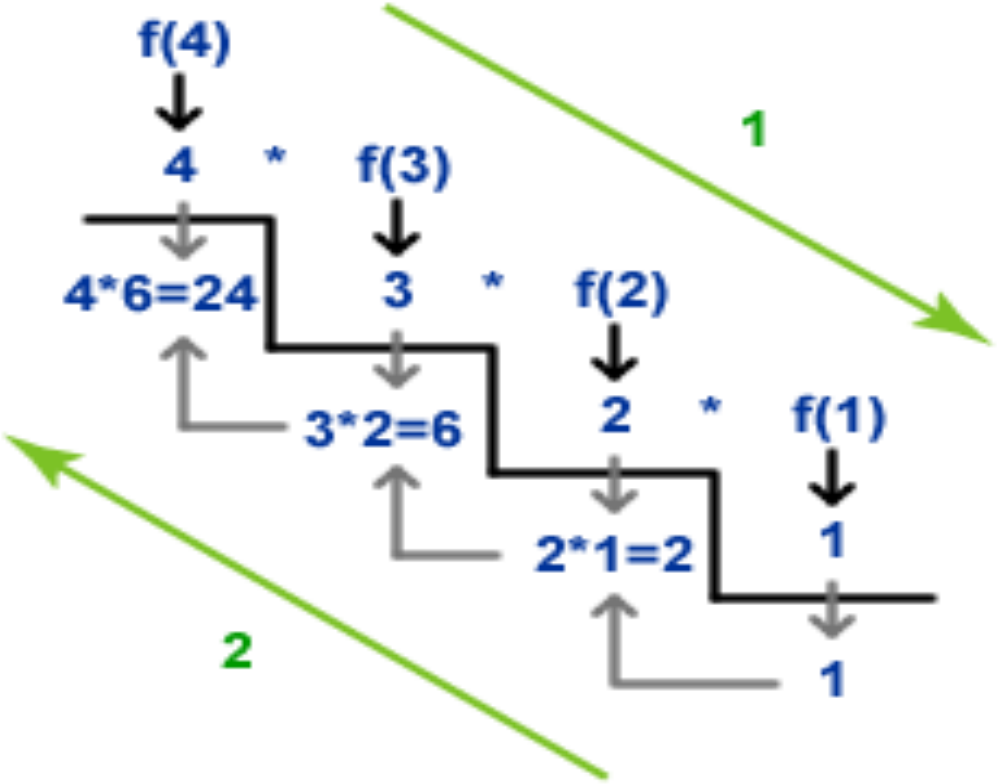
```
// Factorial of n = 1*2*3*...*n
#include <iostream>
using namespace std;
int factorial(int);
int main()
{
    int n = 4;
    cout << "Factorial of " << n <<" = " << factorial(n);
    return 0;
}

int factorial(int n)
{
    if (n > 1)
    {
        return n*factorial(n-1);
    }
    else
    {return 1;}
}
```

Factorial of 4



Factorial of 4



Recursion

- What does the following function Do? Suppose value of begin is 5

```
void printnum ( int begin )
 {
    cout<< begin;
    if (begin< 9)
     // The base case is when begin is greater than 9
    printnum ( begin + 1 );
    cout<< begin;
}
```

SORT AND SEARCH WITH FUNCTIONS AND RECURSION

Rewrite search and sort algorithms using functions

Pseudocode for Binary Search

Display 11.5 **Pseudocode for Binary Search**

ALGORITHM TO SEARCH a[first] THROUGH a[last]

```
/**
 Precondition:
 a[first]<= a[first + 1] <= a[first + 2] <=... <= a[last]
*/
```

TO LOCATE THE VALUE key:

```
if (first > last) //A stopping case
    return -1;
else
{
    mid = approximate midpoint between first and last;
    if (key == a[mid]) //A stopping case
        return mid;
    else if key < a[mid] //A case with recursion
        return the result of searching a[first] through a[mid - 1];
    else if key > a[mid] //A case with recursion
        return  the result of searching  a[mid + 1] through a[last];
 }
```

Recursive Method for Binary Search

Display 11.6 **Recursive Method for Binary Search**

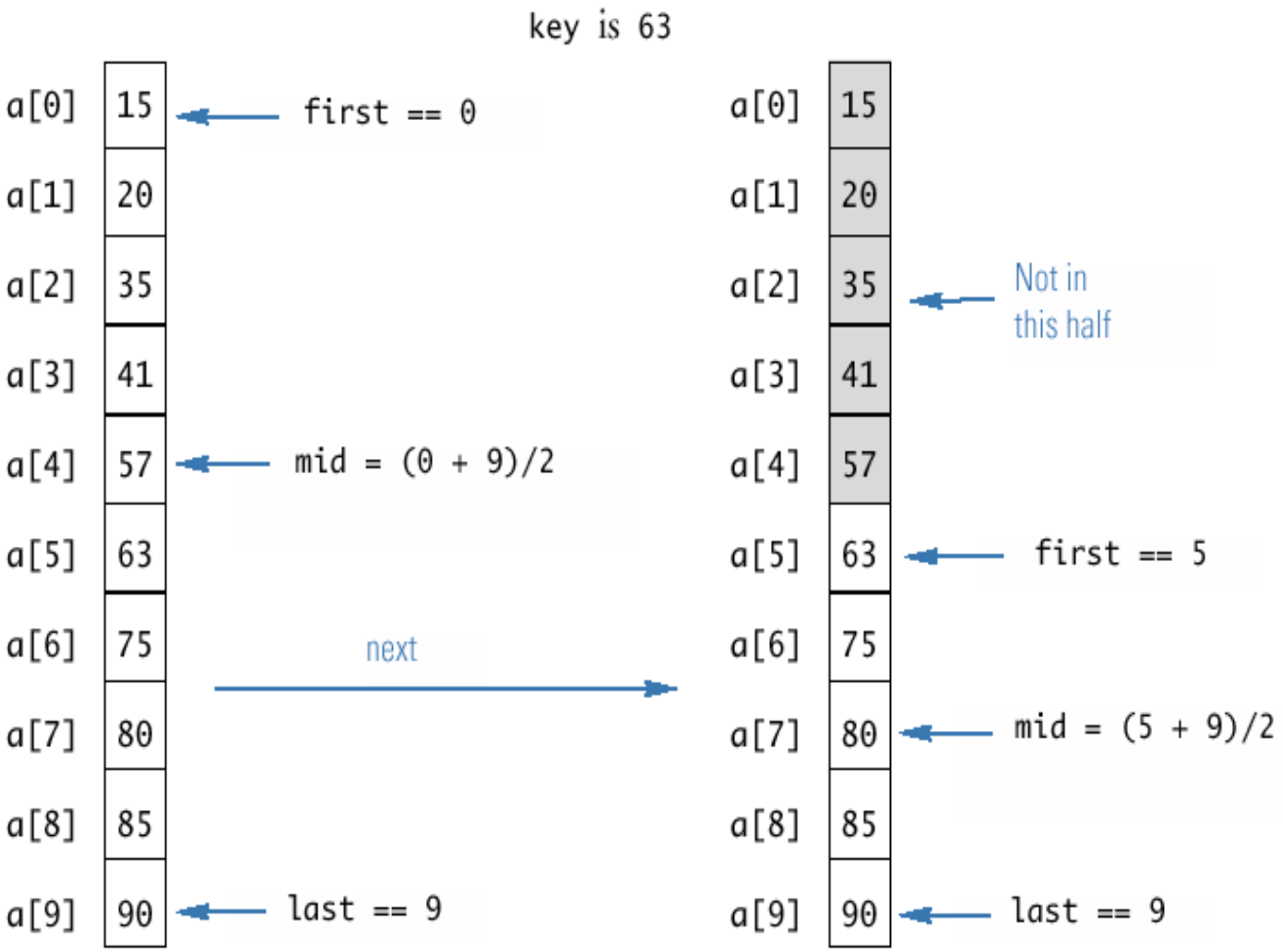
```
1  public class BinarySearch
2  {
3      /**
4       Searches the array a for key. If key is not in the array segment, then -1 is
5       returned. Otherwise returns an index in the segment such that key == a[index].
6       Precondition: a[first] <= a[first + 1]<= ... <= a[last]
7      */
8      public static int search(int[] a, int first, int last, int key)
9      {
10         int result = 0; //to keep the compiler happy.

11         if (first > last)
12             result = -1;
13         else
14         {
15             int mid = (first + last)/2;

16             if (key == a[mid])
17                 result = mid;
18             else if (key < a[mid])
19                 result = search(a, first, mid - 1, key);
20             else if (key > a[mid])
21                 result = search(a, mid + 1, last, key);
22         }
23         return result;
24     }
25 }
```

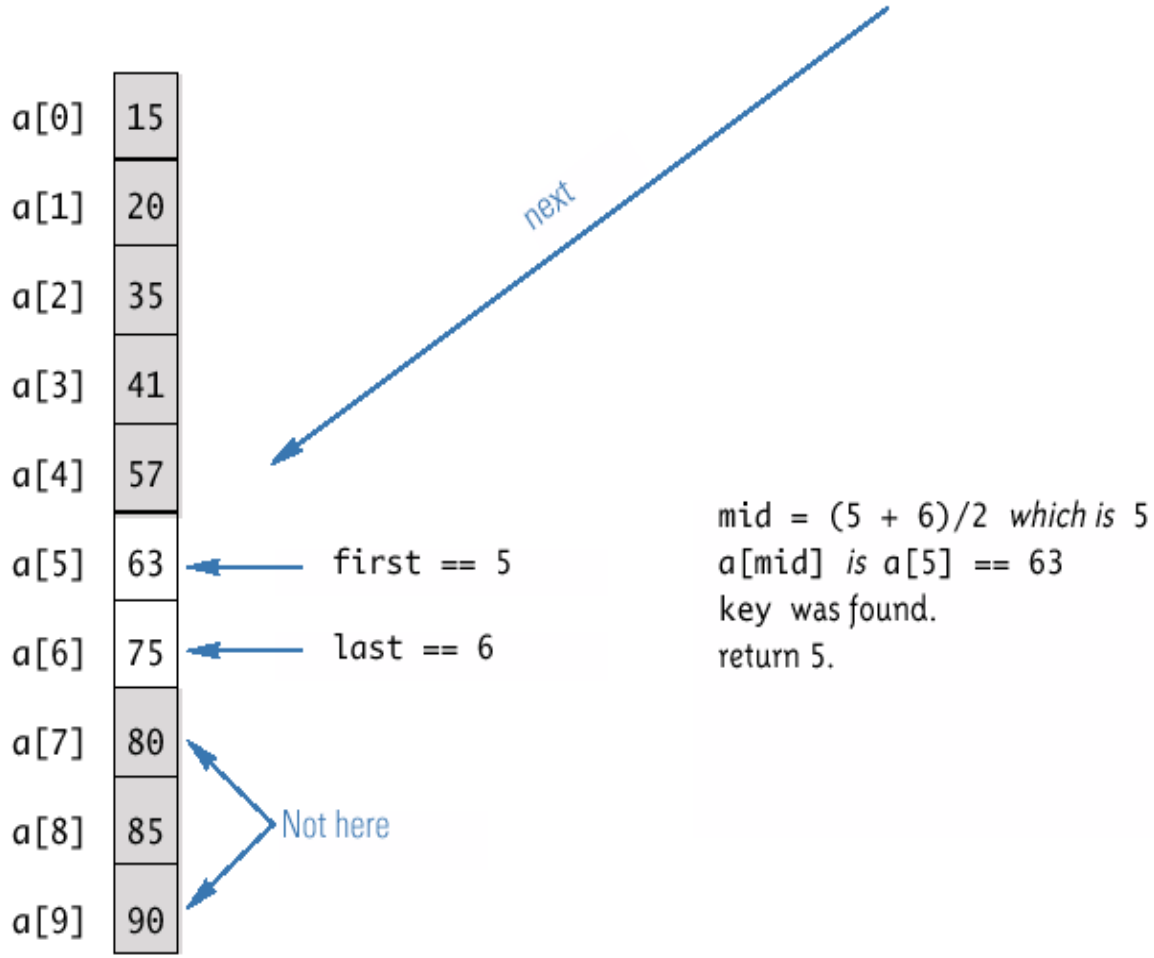
Execution of the Method `search` (Part 1 of 2)

Display 11.7 **Execution of the Method search**



Execution of the Method `search` (Part 1 of 2)

Display 11.7 **Execution of the Method search** (continued)



Checking the `search` Method

1. There is no infinite recursion
   - On each recursive call, the value of **`first`** is increased, or the value of **`last`** is decreased
   - If the chain of recursive calls does not end in some other way, then eventually the method will be called with **`first`** larger than **`last`**

Checking the `search` Method

2. Each stopping case performs the correct action for that case
   - If `first > last`, there are no array elements between `a[first]` and `a[last]`, so `key` is not in this segment of the array, and `result` is correctly set to `-1`
   - If `key == a[mid]`, `result` is correctly set to `mid`

Checking the `search` Method

3. For each of the cases that involve recursion, *if* all recursive calls perform their actions correctly, *then* the entire case performs correctly
   - If `key < a[mid]`, then `key` must be one of the elements `a[first]` through `a[mid-1]`, or it is not in the array
   - The method should then search only those elements, which it does
   - The recursive call is correct, therefore the entire action is correct

Checking the `search` Method

- If `key > a[mid]`, then `key` must be one of the elements `a[mid+1]` through `a[last]`, or it is not in the array
- The method should then search only those elements, which it does
- The recursive call is correct, therefore the entire action is correct

The method `search` passes all three tests:

Therefore, it is a good recursive method definition

Iterative Version of Binary Search (Part 1 of 2)

Display 11.9 Iterative Version of Binary Search

```
1   /**
2    Searches the array a for key. If key is not in the array segment, then -1 is
3    returned. Otherwise returns an index in the segment such that key == a[index].
4    Precondition: a[lowEnd] <= a[lowEnd + 1]<= ... <= a[highEnd]
5   */
6   public static int search(int[] a, int lowEnd, int highEnd, int key)
7   {
8       int first = lowEnd;
9       int last = highEnd;
10      int mid;

11      boolean found = false; //so far
12      int result = 0; //to keep compiler happy

13      while ( (first <= last) && !(found) )
14      {
15          mid = (first + last)/2;
```

Iterative Version of Binary Search (Part 2 of 2)

Display 11.9 **Iterative Version of Binary Search** ✣ (continued)

```
16          if (key == a[mid])
17          {
18              found = true;
19              result = mid;
20          }
21          else if (key < a[mid])
22          {
23              last = mid - 1;
24          }
25          else if (key > a[mid])
26          {
27              first = mid + 1;
28          }
29      }

30      if (first > last)
31          result = -1;

32      return result;
33  }
```

References

- https://www.youtube.com/watch?v=czrbhZjp4JA
- https://www.youtube.com/watch?v=IAMzWp3kS_k
- https://www.youtube.com/watch?v=NvVYd08NUXI
- https://www.programiz.com/cpp-programming/recursion

Any Questions

- Thanks for Listening ☺️