Computer Programming (b)

E1124

Lecture 4

Applications of Arrays (Searching and Sorting)

INSTRUCTOR

DR / AYMAN SOLIMAN

➢ Contents

- ➢ Objectives
- ➢ List Processing
- ➢ Sequential Search
- ➢ Sorting a List: Bubble Sort
- ➢ Sorting a List: Selection Sort
- ➢ Sorting a List: Insertion Sort

➢ Objectives

- ➢ Learn how to implement the sequential search algorithm
- ➢ Explore how to sort an array using the bubble sort, selection sort, and insertion sort algorithms

➢ List Processing

- ➢ **List**: a set of values of the same type
  - ❑ Basic list operations:
    - a) Search for a given item
    - b) Sort the list
    - c) Insert an item in the list
    - d) Delete an item from the list

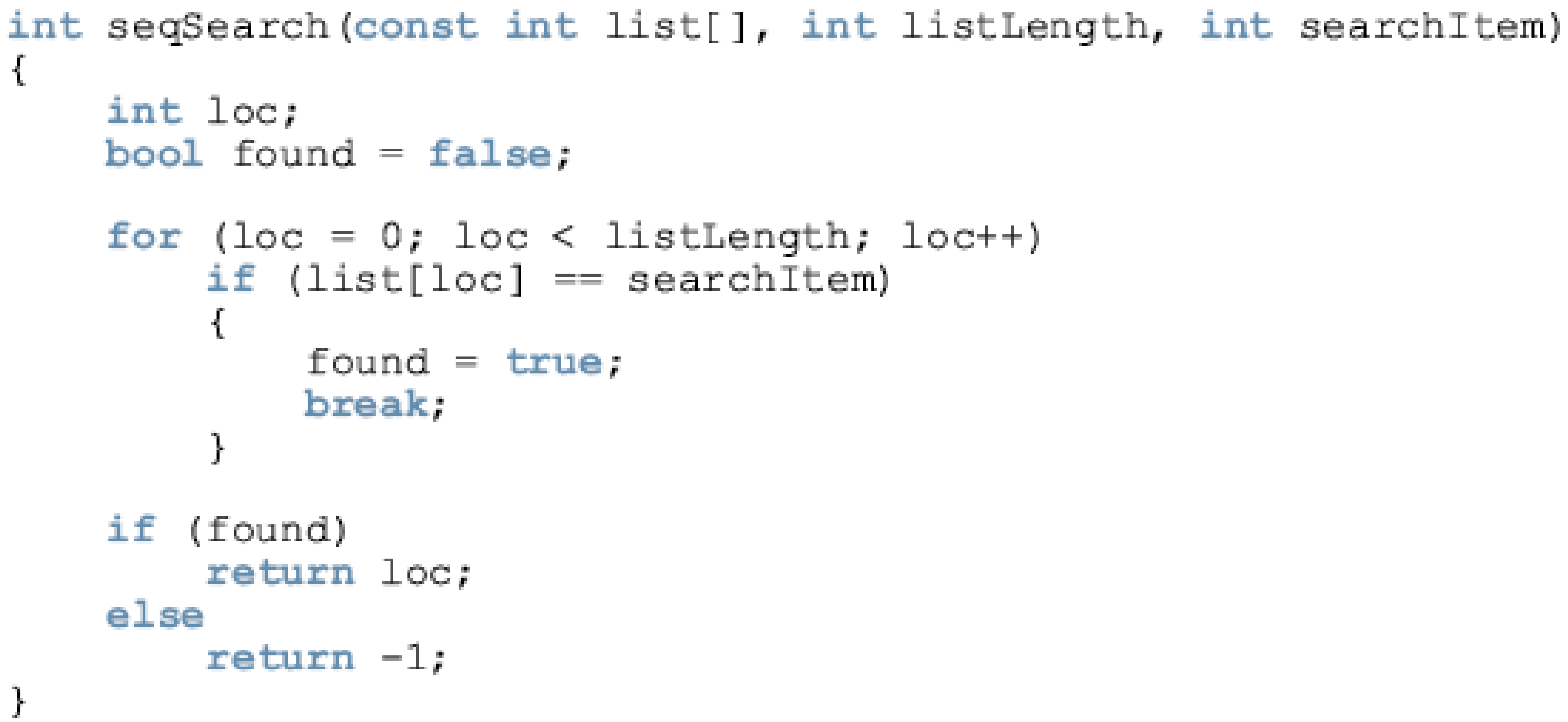
➢ Searching

- ➢ To search a list, you need
  - a) The list (array) containing the list
  - b) List length
  - c) Item to be found
- ➢ After the search is completed
  - d) If found,
    - ✓ Report “success”
    - ✓ Location where the item was found
  - e) If not found, report “failure”

Sequential Search

- Sequential search: search a list for an item
- Compare search item with other elements until either
  - Item is found
  - List has no more elements left
- Average number of comparisons made by the sequential search equals half the list size
- Good only for very short lists

➢ Sequential Search (cont.)

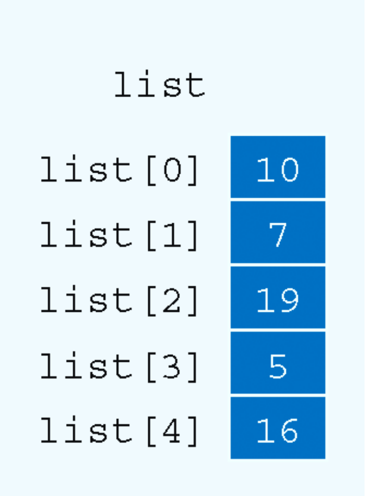


```
int seqSearch(const int list[], int listLength, int searchItem)
{
    int loc;
    bool found = false;

    for (loc = 0; loc < listLength; loc++)
        if (list[loc] == searchItem)
        {
            found = true;
            break;
        }

    if (found)
        return loc;
    else
        return -1;
}
```

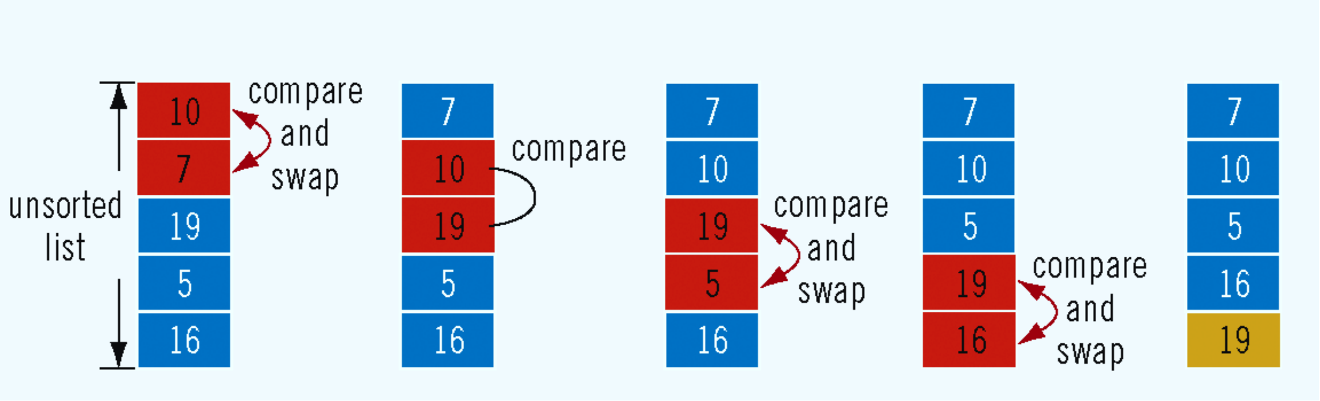
Sorting a List: Bubble Sort

- Suppose list[0]...list[n - 1] is a list of n elements, indexed 0 to n – 1
- Bubble sort algorithm:
  - ❑ In a series of n - 1 iterations, compare successive elements, list[index] and list[index + 1]
  - ❑ If list[index] is greater than list[index + 1], then swap them

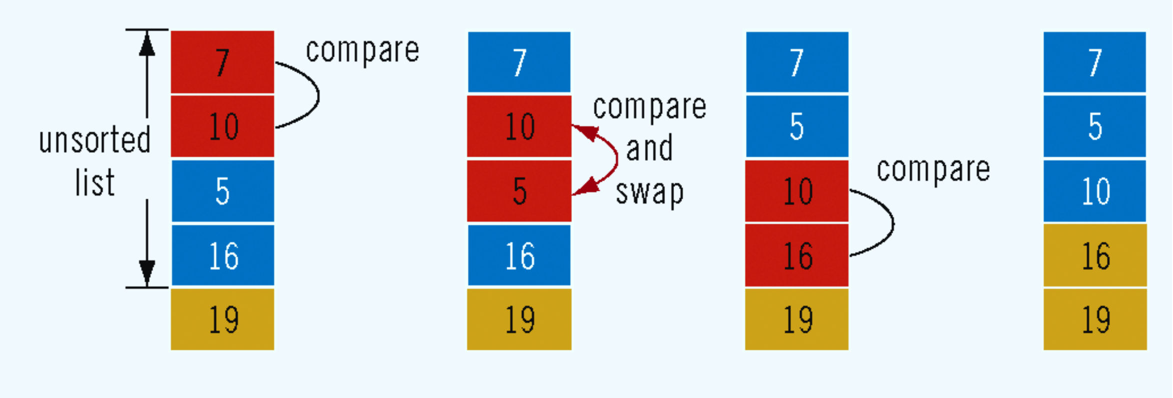
Example



List of five elements

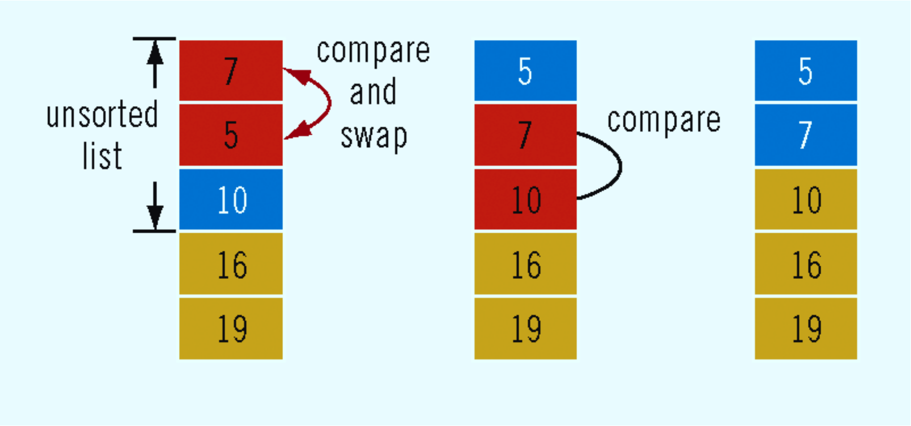


First iteration

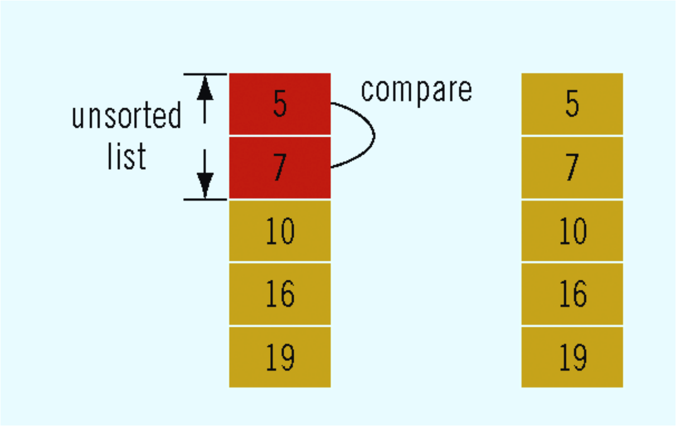


Second iteration

➢ Example (cont.)

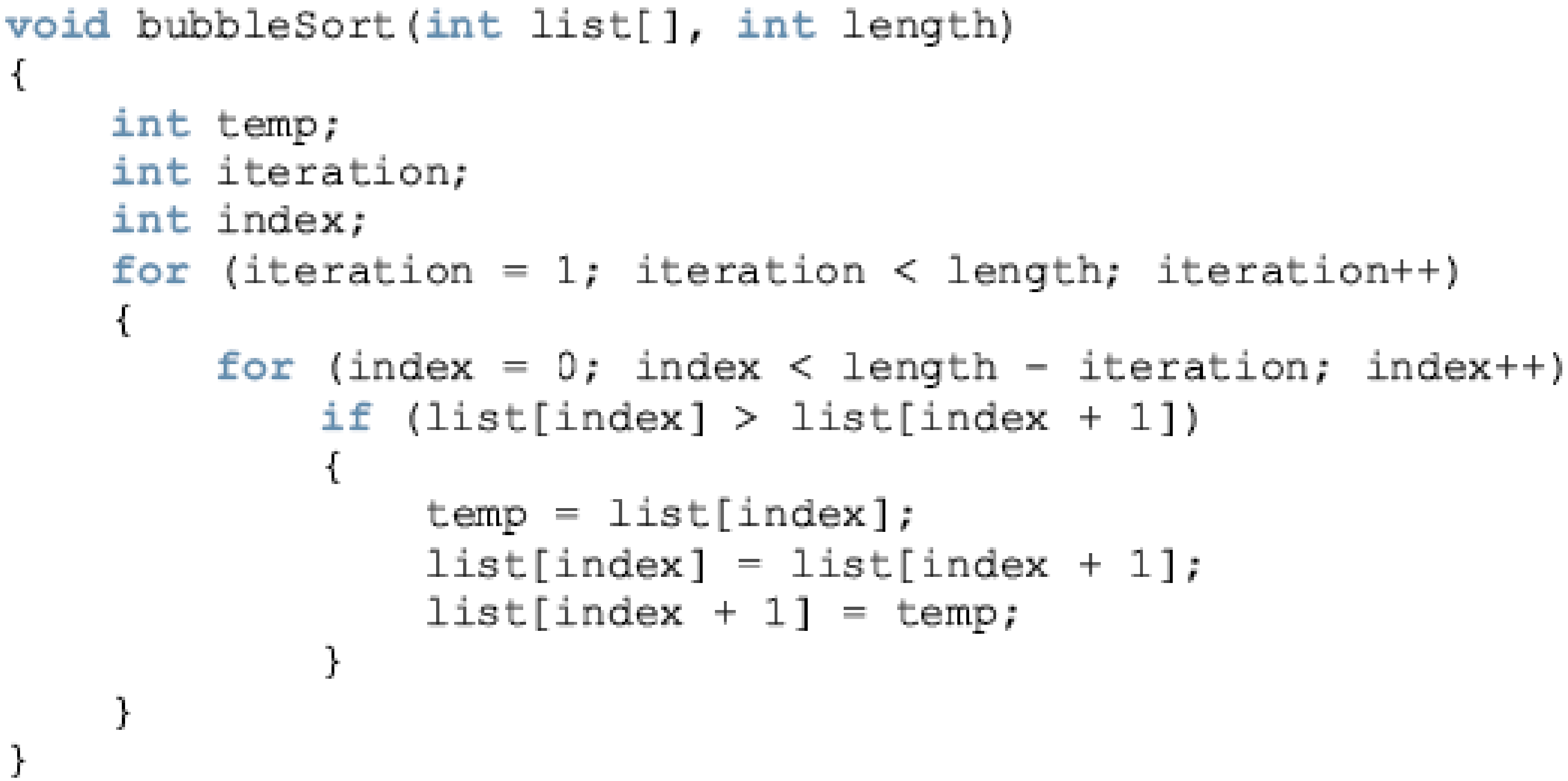


Third iteration



Fourth iteration

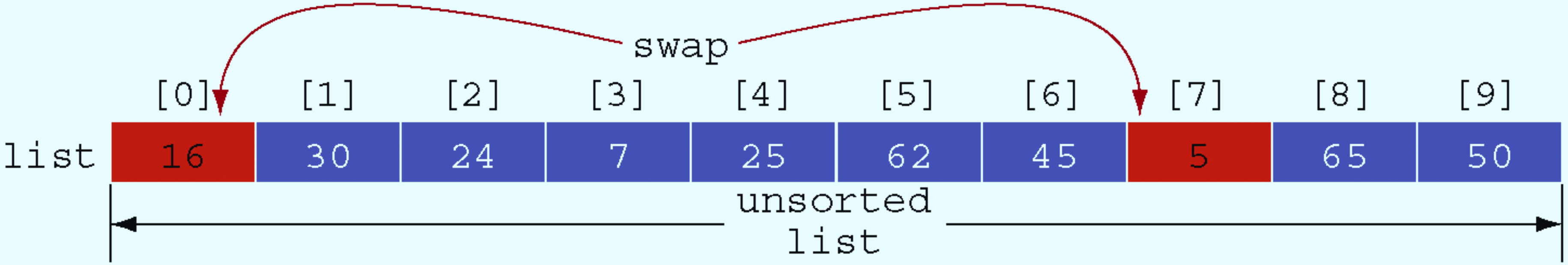
➢ Bubble Sort Code



```
void bubbleSort(int list[], int length)
{
    int temp;
    int iteration;
    int index;
    for (iteration = 1; iteration < length; iteration++)
    {
        for (index = 0; index < length - iteration; index++)
            if (list[index] > list[index + 1])
            {
                temp = list[index];
                list[index] = list[index + 1];
                list[index + 1] = temp;
            }
    }
}
```

Sorting a List: Selection Sort

- Selection sort: rearrange list by selecting an element and moving it to its proper position
- Find the smallest (or largest) element and move it to the beginning (end) of the list

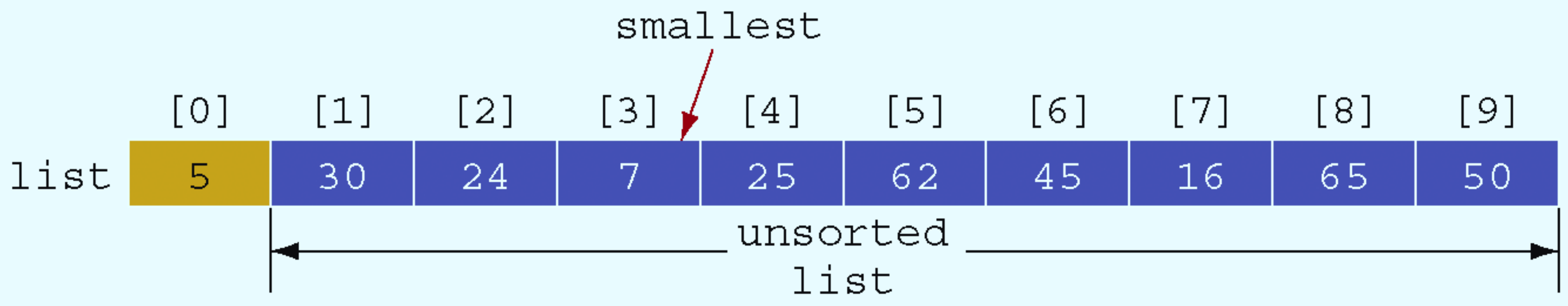


[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
16	30	24	7	25	62	45	5	65	50

Swap elements list[0] and list[7]

➢ Sorting a List: Selection Sort (cont.)

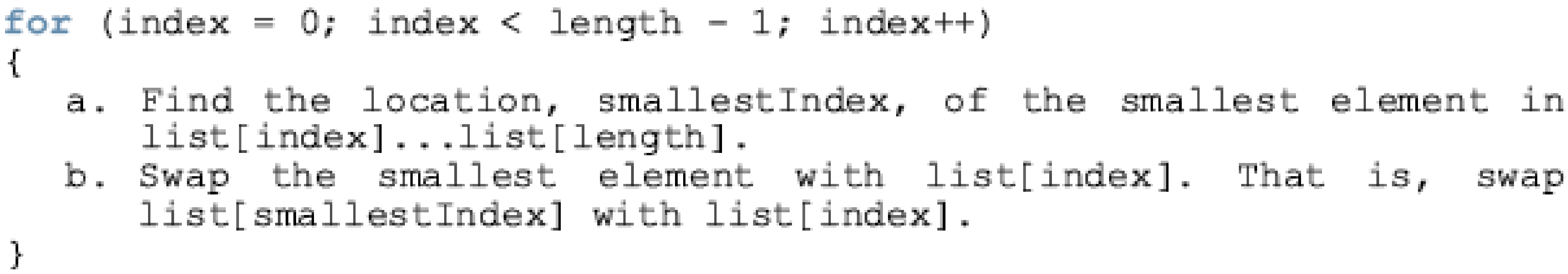
- ➢ On successive passes, locate the smallest item in the list starting from the next element



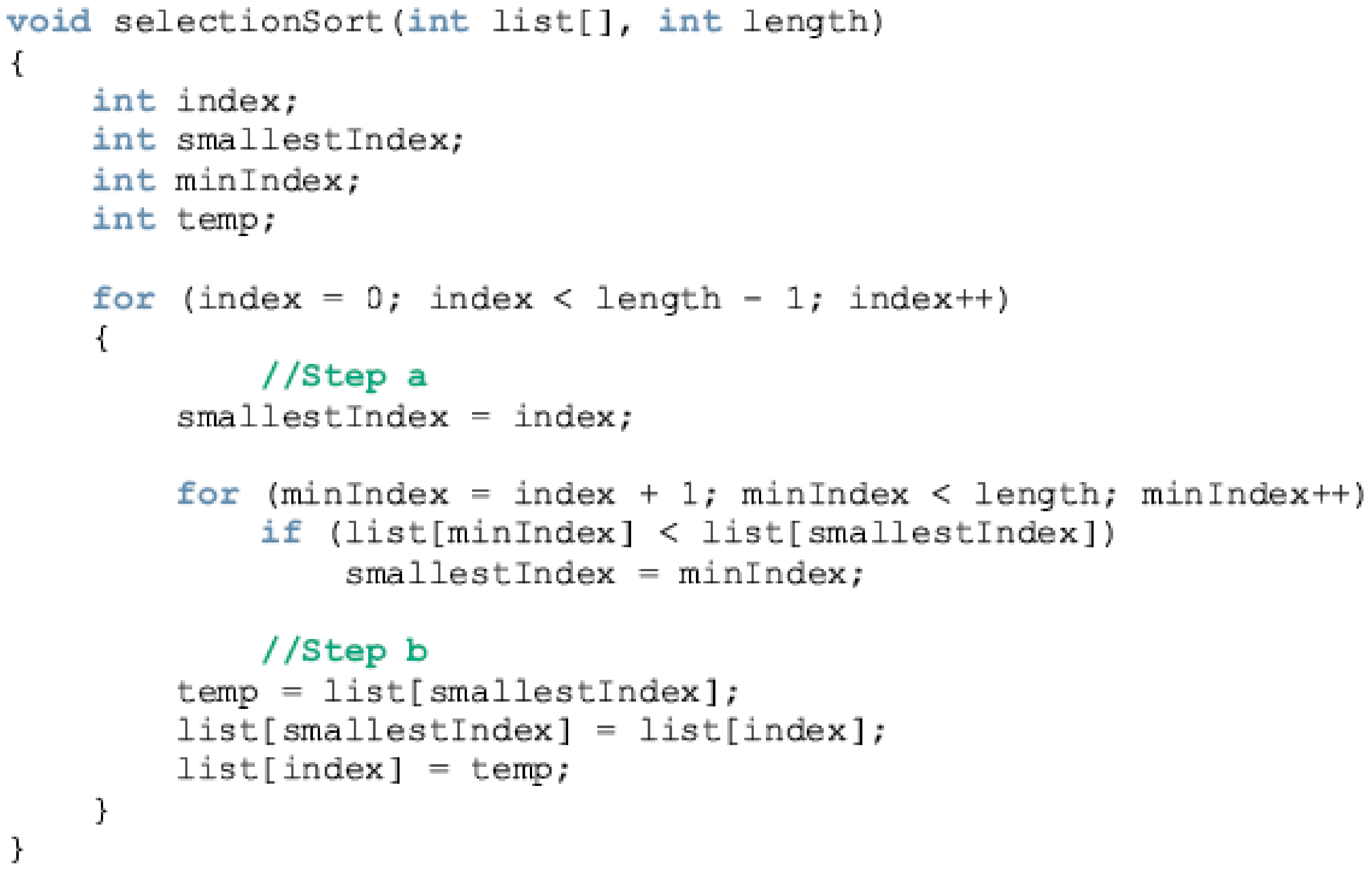
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
list	5	30	24	7	25	62	45	16	65	50

Smallest element in unsorted portion of list

Selection Sort Code



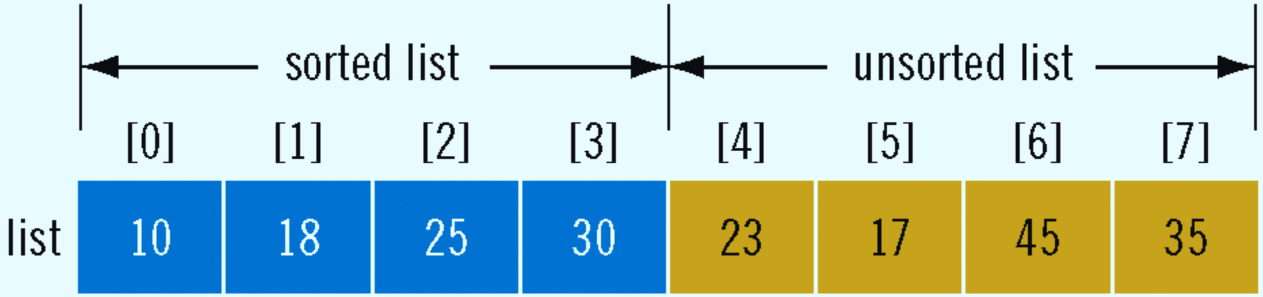
```
for (index = 0; index < length - 1; index++)
{
   a. Find the location, smallestIndex, of the smallest element in
      list[index]...list[length].
   b. Swap the smallest element with list[index]. That is, swap
      list[smallestIndex] with list[index].
}
```



```
void selectionSort(int list[], int length)
{
    int index;
    int smallestIndex;
    int minIndex;
    int temp;

    for (index = 0; index < length - 1; index++)
    {
            //Step a
        smallestIndex = index;

        for (minIndex = index + 1; minIndex < length; minIndex++)
            if (list[minIndex] < list[smallestIndex])
                smallestIndex = minIndex;

            //Step b
        temp = list[smallestIndex];
        list[smallestIndex] = list[index];
        list[index] = temp;
    }
}
```

➢ Sorting a List: Insertion Sort

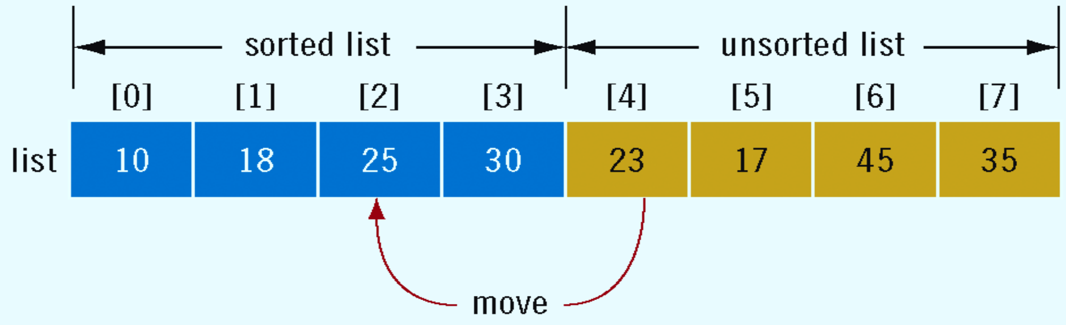
- ➢ The insertion sort algorithm sorts the list by moving each element to its proper place.



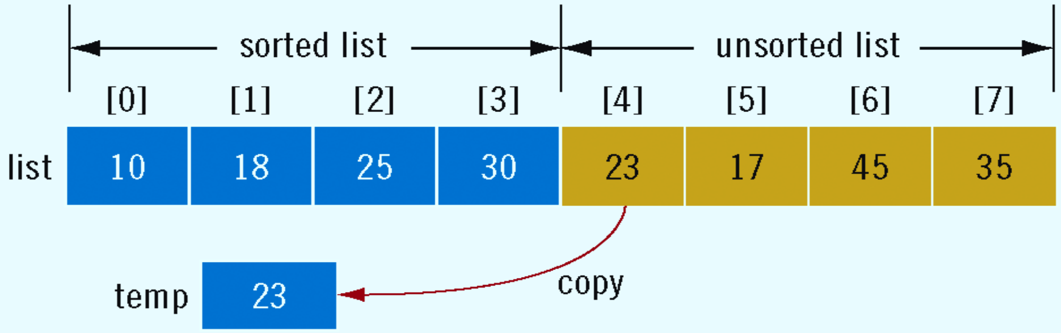
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
list	10	18	25	30	23	17	45	35

Sorted and unsorted portion of `list`

➢ Sorting a List: Insertion Sort (cont.)

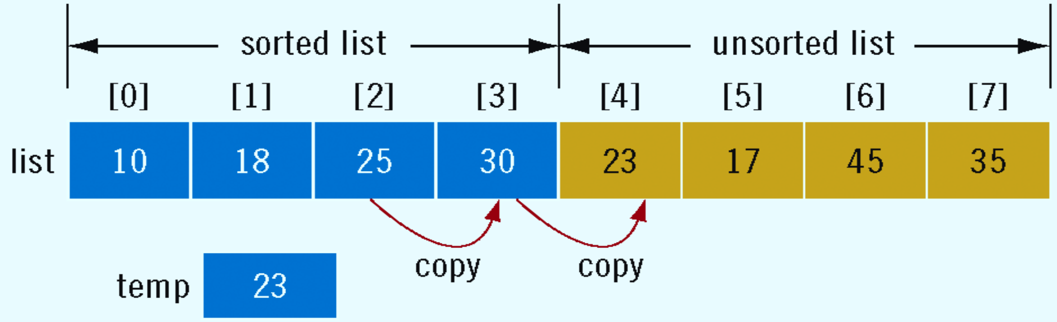


Move list[4] into list[2]

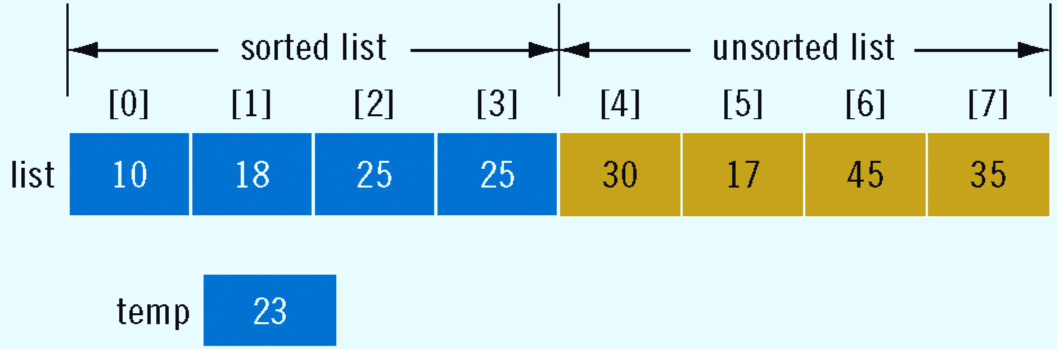


Copy list[4] into temp

➢ Sorting a List: Insertion Sort (cont.)

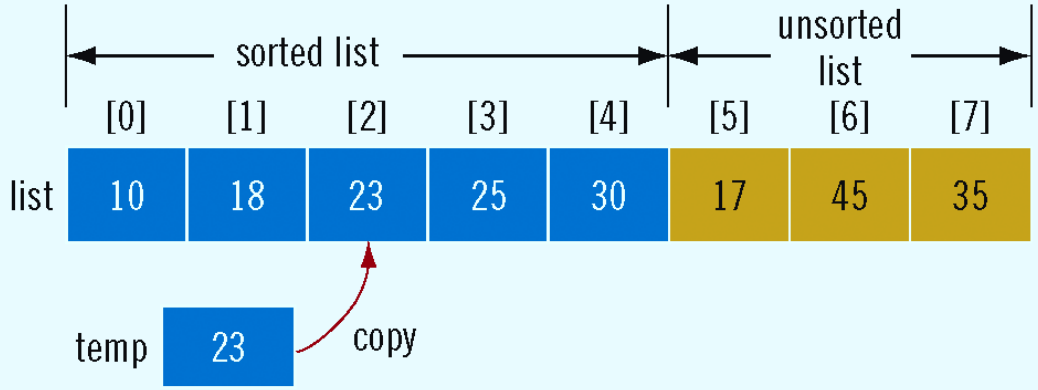


list before copying list[3] into list[4] and then list[2] into list[3]



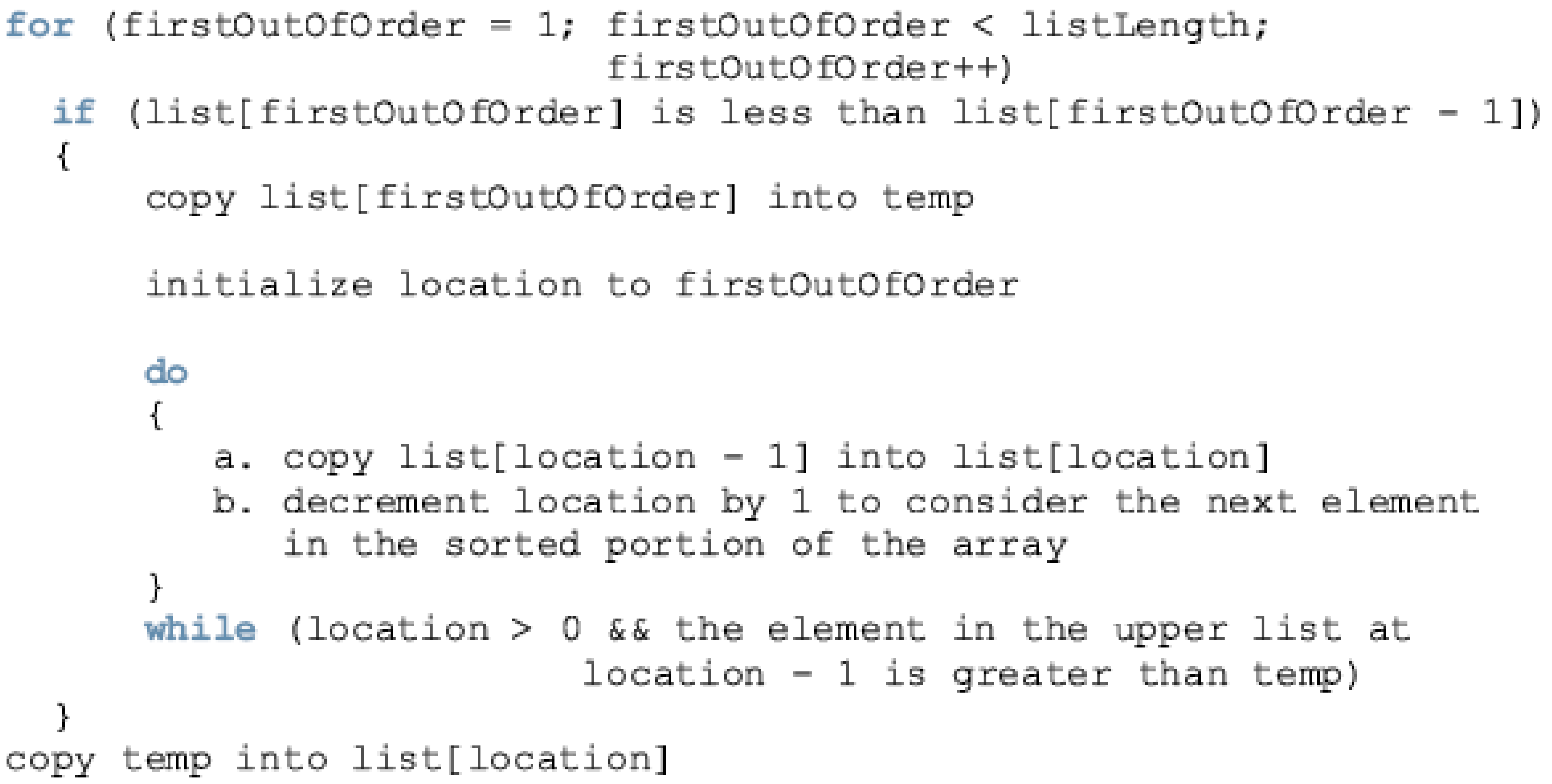
list after copying list[3] into list[4] and then list[2] into list[3]

➢ Sorting a List: Insertion Sort (cont.)



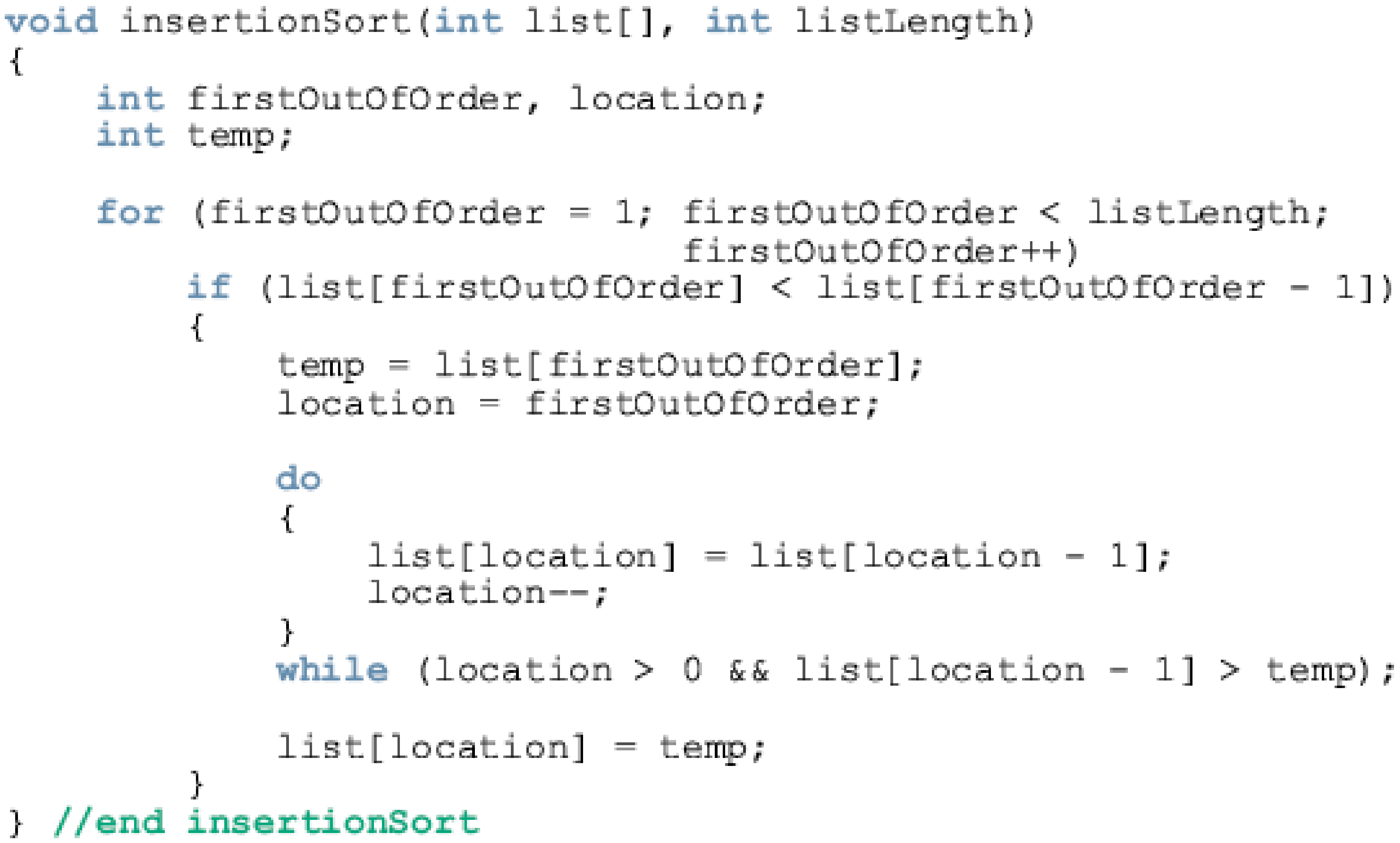
list after copying temp into list[2]

➢ Insertion Sort Code



```
for (firstOutOfOrder = 1; firstOutOfOrder < listLength;
                          firstOutOfOrder++)
  if (list[firstOutOfOrder] is less than list[firstOutOfOrder - 1])
  {
      copy list[firstOutOfOrder] into temp

      initialize location to firstOutOfOrder

      do
      {
         a. copy list[location - 1] into list[location]
         b. decrement location by 1 to consider the next element
            in the sorted portion of the array
      }
      while (location > 0 && the element in the upper list at
                         location - 1 is greater than temp)
  }
copy temp into list[location]
```

➢ Insertion Sort Code (cont.)



```
void insertionSort(int list[], int listLength)
{
    int firstOutOfOrder, location;
    int temp;

    for (firstOutOfOrder = 1; firstOutOfOrder < listLength;
                              firstOutOfOrder++)
        if (list[firstOutOfOrder] < list[firstOutOfOrder - 1])
        {
            temp = list[firstOutOfOrder];
            location = firstOutOfOrder;

            do
            {
                list[location] = list[location - 1];
                location--;
            }
            while (location > 0 && list[location - 1] > temp);

            list[location] = temp;
        }
} //end insertionSort
```

Thank
you