


## Outline

- In this lesson, we will:
  - Describe the algorithm of inserting into a sorted array
  - Consider the implementation
  - Ensure the implementation works with sufficient testing
  - Using this function to implement insertion sort



## Inserting into a sorted array

- Suppose you have an array of capacity 10
  and the first nine entries are sorted
  - For example:

| -3 | 1 | 5 | 8 | 12 | 13 | 17 | 20 | 23 | 6 |
|---|---|---|---|---|---|---|---|---|---|

- Question: How could you convert this into a sorted list?
  1. Assign the last entry to a temporary variable: 6
  2. Starting from the back,
     copy any entries greater than this value to the next entry
  3. Copy the temporary value into the now-available opening



## Inserting into a sorted array

- Can we implement this as a function?
  - The function declaration would be:

```
void insert( double array[], std::size_t capacity );
```

## Inserting into a sorted array

- What operations did we perform here?

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| -3 | 1 | 5 | 8 | 12 | 13 | 17 | 20 | 23 | 6 |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| -3 | 1 | 5 | 6 | 8 | 12 | 13 | 17 | 20 | 23 |

```
double value{array[9]}
```

| | Empty index | Condition | Assignment |
|---|---|---|---|
| capacity - 1 | 9 | array[8] > value | array[9] = array[8] |
| k | 8 | array[k - 1] > value | array[k] = array[k - 1] |
| | 7 | array[6] > value | array[7] = array[6] |
| | 6 | array[5] > value | array[6] = array[5] |
| | 5 | array[4] > value | array[5] = array[4] |
| | 4 | array[3] > value | array[4] = array[3] |
| | 3 | ~~array[2] > value~~ | array[k] = value |



## Inserting into a sorted array

- The implementation would be straight-forward:

```
void insert( double array[], std::size_t capacity ) {
    assert( is_sorted( array, capacity - 1 ) == (capacity - 1) );

    double value{ array[capacity - 1] };

    for ( std::size_t k{capacity - 1}; array[k - 1] > value; --k ) {
        array[k] = array[k - 1];
    }

    array[k] = value;
}
```



## Inserting into a sorted array

- To test this, we will have the following program:

```
#include <iostream>
#include <cassert>
#include <string>

// Function declarations
int main();
std::size_t is_sorted( double array[], std::size_t capacity );
void insert( double array[], std::size_t capacity );

// Function definitions
int main() {
    std::size_t CAPACITY{ 10 };
    double data[CAPACITY]{ -3, 1, 5, 8, 12, 13, 17, 20, 23, 6 };

    insert( data, CAPACITY );

    std::cout << is_sorted( data, CAPACITY ) << std::endl;

    return 0;
}
// Other function definitions...
```



## Inserting into a sorted array

- Let us compile the programing containing this function:

```
void insert( double array[], std::size_t capacity ) {
    assert( is_sorted( array, capacity - 1 ) == (capacity - 1) );

    double value{ array[capacity - 1] };

    for ( std::size_t k{capacity - 1}; array[k - 1] > value; --k ) {
        array[k] = array[k - 1];
    }

    array[k] = value;
}
```

```
example.cpp: In function 'void insert(double*, std::size_t)':
example.cpp:23:11: error: 'k' was not declared in this scope
     array[k] = value;
           ^
```

Problem: The scope of k is restricted to the for loop

## Inserting into a sorted array

- Thus, we can declare k immediately before the for loop

```
void insert( double array[], std::size_t capacity ) {
    assert( is_sorted( array, capacity - 1 ) == (capacity - 1) );

    double value{ array[capacity - 1] };

    std::size_t k{};

    for ( k = capacity - 1; array[k - 1] > value; --k ) {
        array[k] = array[k - 1];
    }

    array[k] = value;
}
```



## Inserting into a sorted array

- We could have done the following:

```
void insert( double array[], std::size_t capacity ) {
    assert( is_sorted( array, capacity - 1 ) == (capacity - 1) );

    double value{ array[capacity - 1] };

    std::size_t k{ capacity - 1 };

    for ( ; array[k - 1] > value; --k ) {
        array[k] = array[k - 1];
    }

    array[k] = value;
}
```



## Inserting into a sorted array

- Now, when we test our program, the output is as desired

```
#include <iostream>
#include <cassert>
#include <string>

// Function declarations
int main();
std::size_t is_sorted( double array[], std::size_t capacity );
void insert( double array[], std::size_t capacity );

// Function definitions
int main() {
    std::size_t CAPACITY{ 10 };
    double data[CAPACITY]{ -3, 1, 5, 8, 12, 13, 17, 20, 23, 6 };

    insert( data, CAPACITY );

    std::cout << is_sorted( data, CAPACITY ) << std::endl;

    return 0;
}
// Other function definitions...
```

Output:
```
10
```



## Inserting into a sorted array

- We should also test it when the next entry is greater than all others:

```
#include <iostream>
#include <cassert>
#include <string>

// Function declarations
int main();
std::size_t is_sorted( double array[], std::size_t capacity );
void insert( double array[], std::size_t capacity );

// Function definitions
int main() {
    std::size_t CAPACITY{ 10 };
    double data[CAPACITY]{ -3, 1, 5, 8, 12, 13, 17, 20, 23, 32 };

    insert( data, CAPACITY );

    std::cout << is_sorted( data, CAPACITY ) << std::endl;

    return 0;
}
// Other function definitions...
```

Output:
```
10
```

## Inserting into a sorted array

- We should also test our program when it is smaller than all entries:

```
#include <iostream>
#include <cassert>
#include <string>

// Function declarations
int main();
std::size_t is_sorted( double array[], std::size_t capacity );
void insert( double array[], std::size_t capacity );

// Function definitions
int main() {
    std::size_t CAPACITY{ 10 };
    double data[CAPACITY]{ -3, 1, 5, 8, 12, 13, 17, 20, 23, -9 };

    insert( data, CAPACITY );

    std::cout << is_sorted( data, CAPACITY ) << std::endl;

    return 0;
}
// Other function definitions...
```

Segmentation fault (core dumped)



## Inserting into a sorted array

- What might be going on here?

```
void insert( double array[], std::size_t capacity ) {
    assert( is_sorted( array, capacity - 1 ) == (capacity - 1) );

    double value{ array[capacity - 1] };

    std::size_t k{};

    for ( k = capacity - 1; array[k - 1] > value; --k ) {
        array[k] = array[k - 1];
    }

    array[k] = value;
}
```

When `k == 1`, `array[0] > value`
`array[1] = array[0]`

Now `k == 0`, `array[0 - 1] > value`

Thus, we should stop looping if `k == 0`



## Inserting into a sorted array

- If we are to halt if `k == 0`, we keep iterating the for loop if `k > 0`

```
void insert( double array[], std::size_t capacity ) {
    assert( is_sorted( array, capacity - 1 ) == (capacity - 1) );

    double value{ array[capacity - 1] };

    std::size_t k{};

    for ( k = capacity - 1; (k > 0) && (array[k - 1] > value); --k ) {
        array[k] = array[k - 1];
    }

    array[k] = value;
}
```

We rely on short-circuit evaluation of logical operators:
Given *condition-1* `&&` *condition-2*,
if *condition-1* evaluates to `false`,
second condition is not even tested
Recall: `false &&` *anything* is `false`



## Insertion sort

- We will now use this function in our implementation of the insertion sort algorithm...

## Sorting

- How can we use this function to sort an array of a given capacity?

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | |
|---|---|---|---|---|---|---|---|---|---|---|
| -16 | -35 | 15 | 9 | 1 | -8 | -10 | 30 | 0 | 13 | |
| -16 | -35 | 15 | 9 | 1 | -8 | -10 | 30 | 0 | 13 | `insert( array, 2 )` |
| -35 | -16 | 15 | 9 | 1 | -8 | -10 | 30 | 0 | 13 | `insert( array, 3 )` |
| -35 | -16 | 15 | 9 | 1 | -8 | -10 | 30 | 0 | 13 | `insert( array, 4 )` |
| -35 | -16 | 9 | 15 | 1 | -8 | -10 | 30 | 0 | 13 | |



## Sorting

- Thus, we have a straight-forward implementation:

```
// Function declaration
void insertion_sort( double array[], std::size_t capacity );

// Function definition
void insertion_sort( double array[], std::size_t capacity ) {
    for ( std::size_t k{2}; k <= capacity; ++k ) {
        insert( array, k );
    }

    assert( is_sorted( array, capacity ) == capacity );
}
```



## Weakness in our testing?

- Note that we are simply testing in the assertion that the array is sorted
  - Is this sufficient?
- What if there is an error in our implementation and the array

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| -16 | -35 | 15 | 9 | 1 | -8 | -10 | 30 | 0 | 13 |

  becomes this array?

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| -16 | -16 | -16 | -16 | -16 | -16 | -16 | -16 | -16 | -16 |

  - Thus, our testing should actually compare the resulting array with the expected sorted array ☺



## Usefulness of the `insert(…)` function

- Online, you may find that most implementations of insertion sort accomplish everything in one function:
  - Combining our two functions, we a few modifications, we have:

```
void insertion_sort( double array[], std::size_t capacity ) {
    for ( std::size_t k{1}; k < capacity; ++k ) {
        double value{ array[k] };
        std::size_t j{};

        for ( j = k; (j > 0) && (array[j - 1] > value); --j ) {
            array[j] = array[j - 1];
        }

        array[j] = value;
    }

    assert( is_sorted( array, capacity ) == capacity );
}
```

## Summary

- Following this presentation, you now:
  - Know how to implement an insertion into a sorted array
    - We reviewed the concept of the scope of loop variables
  - Understand how to test such a function and ensure it works
    - We reviewed the concept of short-circuit evaluation
  - Know the insertion sort algorithm
  - Have seen how to implement this algorithm with `insert(…)`
  - Understand that more rigorous testing may be necessary
  - Appreciate that breaking a larger problem into simpler problems can make solving larger problems much more easily
    - We looked at the standard insertion sort implementation



## References

[1] Wikipedia, https://en.wikipedia.org/wiki/Insertion_sort

[2] Dictionary of Algorithms and Data Structures (DADS) https://xlinux.nist.gov/dads/HTML/insertionSort.html



## Acknowledgments

None so far.



## Colophon

These slides were prepared using the Georgia typeface. Mathematical equations use Times New Roman, and source code is presented using `Consolas`.

The photographs of lilacs in bloom appearing on the title slide and accenting the top of each other slide were taken at the Royal Botanical Gardens on May 27, 2018 by Douglas Wilhelm Harder. Please see

https://www.rbg.ca/

for more information.



## Disclaimer