UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING
Department of Electrical & Computer Engineering

ECE 150 *Fundamentals of Programming*

# Partition

Douglas Wilhelm Harder, M.Math.
Prof. Hiren Patel, Ph.D.
hiren.patel@uwaterloo.ca dwharder@uwaterloo.ca

## Outline

- In this lesson, we will:
  - Describe a partition of an array
  - Look at three implementations
    - Each will be more successively more efficient
  - Consider how to generalize the ranges
  - Consider how to generalize the functions

## Introduction

- We have already discussed sorting an array
  - Suppose we simply want to partition an array
  - For example, reorder the entries so that:
    - All negative (less than zero) entries appear first in the array
    - All non-negative (zero or positive) entries appear at the end of the array

- For example, given

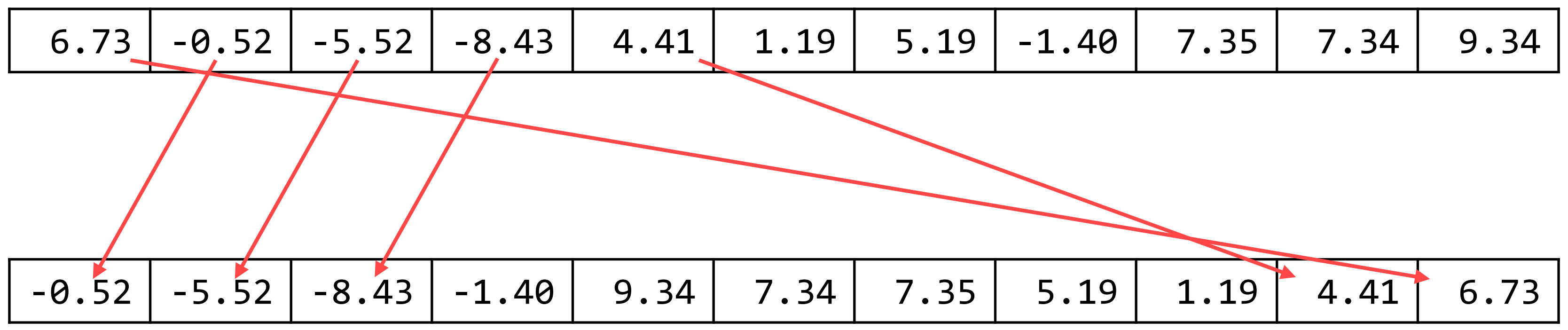
```
 6.73 -0.52 -5.52 -8.43  4.41  1.19  5.19 -1.40  7.35  7.34  9.34
```

a partition may be

```
-0.52 -5.52 -8.43 -1.40  6.73  4.41  1.19  5.19  7.35  7.34  9.34
```

## Initial implementation

- How could we implement this?
  - Create a second array
  - If the entry is negative, put it at the front of the array
  - Otherwise, put it at the end of the new array



## Initial implementation

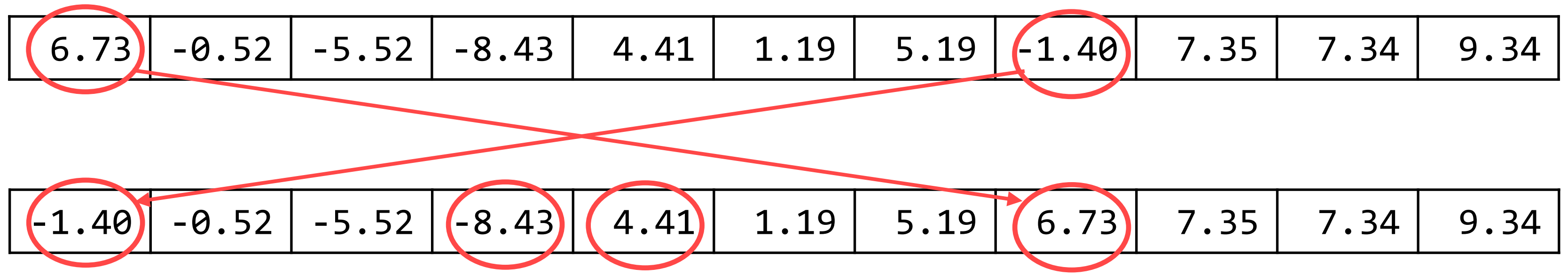
```
void partition( double array[],
                std::size_t capacity ) {
    double a_tmp[capacity];
    std::size_t front{ 0 };
    std::size_t back{ capacity - 1};

    for ( std::size_t k{0}; k < capacity; ++k ) {
        if ( array[k] < 0.0 ) {
            a_tmp[front] = array[k];
            ++front;
        } else {
            a_tmp[back] = array[k];
            --back;
        }
    }

    for ( std::size_t k{0}; k < capacity; ++k ) {
        array[k] = a_tmp[k];
    }
}
```

## Initial implementation

- Positive:
  - We have two loops that go through the array once
- Issue:
  - We must go through the array twice
  - We must allocate a second array…

## A second implementation

- Consider
  - Start from the front and find the first entry that is non-negative
  - Start from the back and find the last entry that is negative
  - Swap the two
  - Repeat



## A second implementation

```
void partition( double array[],
                std::size_t capacity ) {
    std::size_t front{ 0 };
    std::size_t back{ capacity - 1};

    while ( true ) {
        // Find the first entry that is non-negative
        while ( front < back ) {
            if ( array[front] >= 0 ) {
                break;
            } else {
                ++front;
            }
        }
```

## A second implementation

```
        // Find the last entry that is negative
        while ( front < back ) {
            if ( array[back] < 0 ) {
                break;
            } else {
                --back;
            }
        }

        // If the two are out of place, swap them
        if ( front < back ) {
            double tmp{ array[front] };
            array[front] = array[back];
            array[back] = tmp;
            ++front;
            --back;
        } else {
            break;
        }
    }
```

## A second implementation

- Positive:
  - We only go through the loop once
- Issue:
  - We must make three assignments for a swap…

## An optimal implementation

- Consider the following:
  - Copy the last entry to a temporary variable
    - This creates a "hole" at the end of the array
  - Find the first non-negative entry and copy it to this hole
    - This creates a "hole" at the front of the array
  - Find the last negative entry and copy it to this hole
    - This creates a "hole" at the back of the array
  - Repeat until an entry is moved in the wrong direction
    - The "hole" is now between the negative and non-negative entries
    - Fill the hole with what was the last entry
      - It doesn't matter if the last entry is negative or not…

## An optimal implementation

```
void partition( double array[],
                std::size_t capacity ) {
    double tmp{ array[capcity - 1] };
    std::size_t left{ 0 };
    std::size_t right{ capacity - 1 };   // The 'hole'
```

## An optimal implementation

```
while ( true ) {
    while ( left <= right ) {
        if ( array[left] >= 0.0 ) {
            array[right] = array[left];
            --right;
            break;
        }

        ++left;
    }

    if ( left > right ) {
        array[right] = tmp;
        return;
    }
```

## An optimal implementation

```
        while ( left <= right ) {
            if ( array[right] < 0.0 ) {
                array[left] = array[right];
                ++left;
                break;
            }

            --right;
        }

        if ( left > right ) {
            array[left] = tmp;
            return;
        }
    }
}
```

## A reasonable return value?

- Could we return something?
  - How about the first position that is non-negative?
  - If this is assigned the variable `part`:

    `array[0], . . . , array[part - 1]` are all negative

    `array[part], ..., array[capacity - 1]` are all nonnegative

## Generalizing the range

- Have the algorithm work from

```
array[begin], ... , array[end - 1]
```

- This is rather easy, too:

```
void partition( double      array[],
                std::size_t begin,
                std::size_t end ) {
    double tmp{ array[end - 1] };
    std::size_t left{ begin };
    std::size_t right{ end - 1 };   // The 'hole'

    // The rest of the code is identical!
}
```

## Generalizing the condition

- What if we want a different condition?
  - Do we write a brand-new function?
  - Wouldn't 99% of the content be identical
    - In fact, only the conditions would differ…
- This is rather easy, too:
  - Allow the user to pass the name of a Boolean-valued function that takes a double as an argument
  - A Boolean-valued function is often referred to as a *predicate*
    - All entries for which the predicate returns `true` are at the front
    - All other entries are at the back

```
void partition( double      array[],
                std::size_t begin,
                std::size_t end,
                std::function<bool(double)> predicate );
```

## Example 1

- What does this code do?

```
int main() {
    std::size_t N{ 10 };
    double data{ 3.2, -5.4,  1.9,  8.6,  0.7,
                 6.5,  2.0,  7.1, -4.3, -9.8 };

    std::cout << partition( data, 0, N, is_negative ) << std::endl;

    return 0;
}

double is_negative( double x ) {
    return (x < 0.0);
}
```

## Example 2

- What does this code do?

```
int main() {
    std::size_t N{ 10 };
    double data{ 3.2, -5.4,  1.9,  8.6,  0.7,
                 6.5,  2.0,  7.1, -4.3, -9.8 };

    std::cout << partition( data, 0, N, is_negative ) << std::endl;

    return 0;
}

double is_in_n1_1( double x ) {
    return (x >= 1.0) && (x <= 1.0);
}
```

## Example 3

- What does this code do?

```
int main() {
    std::size_t N{ 10 };
    double data{ 3.2, -5.4,  1.9,  8.6,  0.7,
                 6.5,  2.0,  7.1, -4.3, -9.8 };

    std::cout << partition( data, 0, N, is_negative ) << std::endl;

    return 0;
}

// Does it round to a valid unsigned short?
double is_unsigned_short( double x ) {
    return (x >= -0.5) && (x <= 65535.5);
}
```

## The standard library

- In the standard library, there is a
  `std::partition(…)`
  in the header
  `#include <algorithm>`
  - Again, despite it appearing there are many function evaluations, a good compiler will eliminate these and simply inline these operations
  - Rather than passing an array pointer and indices, you pass the addresses of `array[begin]` and `array[end]`

## Summary

- Following this lesson, you now:
  - know what a partition is
  - Have seen three implementations,
    with a final optimal implementation
  - Know how to generalizing the ranges
  - Understand how to generalize the test or *predicate* so that you don't have to rewrite your code over umpteen times…

## References

[1] https://www.cplusplus.com/reference/algorithm/partition/
[2] https://en.cppreference.com/w/cpp/algorithm/partition

## 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