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

ECE 150 *Fundamentals of Programming*

Adjacent difference

ECE150

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

Outline

- In this lesson, we will:
  - Define an adjacent difference on an array
  - Look at various implementations and issues with them
  - Generalize the algorithm
  - Look at the implementation and examples

Introduction

- Given an array, certain numerical algorithms require you to calculate the difference between successive entries

```
            array[0]
            array[1] - array[0]
            array[2] - array[1]
            array[3] - array[2]
                     .
                     .
                     .
array[capacity - 1] - array[capacity - 2]
```

Implementation

- We will have an input and output arrays:

```
void adjacent_difference(
    double array1[], std::size_t capacity
    double array2[]
) {
    array2[0] = array1[0];

    for ( std::size_t k{1}; k < capacity; ++k ) {
        array2[k] = array1[k] - array1[k - 1];
    }
}
```

  – It is assumed there is sufficient entries in `array2`

Example

- For example:

```
int main() {
    std::size_t const N{ 5 };
    double input[N]{ 3.2, 2.9, 3.1, 3.2, 3.7 };
    double output[N];

    adjacent_difference( input, N, output );

    return 0;
}
```

  – The output array is now:

3.2	-0.3	0.2	0.1	0.5

  – Note that the first entry allows you to recreate the original array using a scan

Problem

- Issue: What if you want the adjacent difference in place?

```
int main() {
    std::size_t const N{ 5 };
    double input[N]{ 3.2, 2.9, 3.1, 3.2, 3.7 };

    adjacent_difference( input, N, input );

    return 0;
}
```

  – The values of the input array are now

3.2	-0.3	3.4	-0.2	3.9

A better implementation

- We must assume that the operations are in-place:

```
void adjacent_difference(
    double array1[], std::size_t capacity
    double array2[]
) {
    for ( std::size_t k{capacity - 1}; k > 0; ++k ) {
        array2[k] = array1[k] - array1[k - 1];
    }

    array2[0] = array1[0];
}
```

  – This now also works in-place
    however, unfortunately, some users require the operation in order

A better implementation

- To run in place and in order requires some finesse:

```
void adjacent_difference(
    double array1[], std::size_t capacity
    double array2[]
) {
    double x0{ array1[0] };
    array2[0] = x0;

    for ( std::size_t k1{1}, k2{1}; k1 < capacity; ++k1, ++k2 ) {
        double x1{ array1[k1] };
        array2[k2] = array1[k1] - x0;
        x0 = x1;
    }
}
```

Generalizing the range

- Have the algorithm work from

```
array1[begin1], ... , array1[end1 - 1]
array2[begin2], ...
```

- This is straight-forward:

```
void adjacent_difference(
    double array1[], std::size_t begin1, std::size_t end1,
    double array2[], std::size_t begin2
) {
    double x0{ array1[begin1] };
    array2[begin2] = x0;

    for ( std::size_t k1{begin1 + 1}, k2{begin2 + 1};
                               k1 < end1; ++k1, ++k2 ) {
        double x1{ array1[k1] };
        array2[k2] = array1[k1] - x0;
        x0 = x1;
    }
}
```

Generalizing the operation

- Our operation is calculating the difference:
  - As before, should we not allow the user to specify the operation?
- This, too, is straight-forward:

```
void adjacent_difference(
    double array1[], std::size_t begin1, std::size_t end1,
    double array2[], std::size_t begin2,
    std::function<double( double, double )> difference
) {
 double x0{ array1[begin1] };
    array2[begin2] = x0;

    for ( std::size_t k1{begin1 + 1}, k2{begin2 + 1};
          k1 < end1; ++k1, ++k2 ) {
        double x1{ array1[k1] };
        array2[k2] = difference( array1[k1], x0 );
        x0 = x1;
    }
}
```

Example 1

- What does this code do?

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

    std::cout << adjacent_difference( data, 0, N,
                                      data, 0, difference ) << std::endl;

    return 0;
}

double difference( double x, double y ) {
    return x - y;
}
```

Example 2

- An interesting example from cppreference.com

```
int main() {
    std::size_t N{ 10 };
    double data[N]{ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 };

    std::cout << adjacent_difference( data, 0, N - 1,
                                      data, 1, sum ) << std::endl;

    return 0;
}

double sum( double x, double y ) {
    return x + y;
}
```

The standard library

- In the standard library, there is a

  `std::divided_difference(…)`

  in the header

  `#include <numeric>`

  - Rather than passing an array pointer and indices, you pass the addresses of `array[begin]` and `array[end]`

Summary

- Following this lesson, you now:
  - Understand what the adjacent difference is
  - Know how to implement it
  - Understand it may be implemented in place
  - Looked at generalizations and some examples

References

[1] https://en.cppreference.com/w/cpp/algorithm/adjacent_difference

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

These slides are provided for the ECE 150 *Fundamentals of Programming* course taught at the University of Waterloo. The material in it reflects the authors' best judgment in light of the information available to them at the time of preparation. Any reliance on these course slides by any party for any other purpose are the responsibility of such parties. The authors accept no responsibility for damages, if any, suffered by any party as a result of decisions made or actions based on these course slides for any other purpose than that for which it was intended.