Integrating data

In many cases, an array may have data that you would like to integrate to determine the overall impact. For example, if you were recording your speed, integrating that speed from some initial time t_0 to the present will tell you exactly how far you moved since the initial time t_0.

Now, suppose you are using a system that periodically samples the speed of your vehicle at a rate of once per second. Now, 90 km/h is exactly 25 m/s, so if you were periodically sampling the speed of your vehicle at the given rate, you may get an array of entries that looks like:

0.0	3.0	6.3	9.5	12.7	15.5	18.0	19.8	20.9	21.7	22.1	22.3	22.2

From the data, you can see the car was initially accelerating from rest, and after approximately ten seconds, was driving at approximately 80 km/h. To estimate how far the car has moved in this time, we must replace the k^{th} entry of with the sum of the first *k* entries of this array.

Thus, we will write a function that is called accumulate and will take an array and its capacity as arguments, and when the function returns, the entries of the array will be summed up as described.

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

Because you are replacing the entries of the array, one must be careful as to how to do this. There are different solutions, and one is significantly better than most,

A second array

One solution may be to create a second array, and populate it with the sums, and then to copy that data back to the original array:

```
void accumulate( double array[], std::size_t const capacity ) {
    double accumulation_array[capacity];

    for ( std::size_t index{0}; index < capacity; ++index ) {
        accumulation_array[index] = 0.0;

        for ( std::size_t k{0}; k <= index; ++k ) {
            accumulation_array[index] += array[k];
        }
    }

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

Accumulating backward

Another solution would be to start from the end of the array and work your way back, adding to each entry all previous entries in the array:

```
void accumulate( double array[], std::size_t const capacity ) {
    for ( std::size_t index{capacity}; index > 0; --index ) {
        // Add to array[index - 1] each previous value
        for ( std::size_t k{0}; k < index - 1; ++k ) {
            array[index - 1] += array[k];
        }
    }
}
```

Now, this may look a little awkward, because the first loop starts at `capacity`, which is beyond the end of the array, but goes down until `index` equals `1`. At this point, it updates `array[index - 1]` inside the loop body. This is because the type of the loop variable is an unsigned integer, so consider what would happen if we had the loop run as follows:

```
for ( std::size_t index{capacity - 1}; index >= 0; --index ) {
    // Loop body
}
```

Here, when `index` is `0`, performing the operation `--index` results in `index` being assigned the largest possible value of `std::size_t`, so either $2^{32} - 1$ or $2^{64} - 1$. Well, if `index` is now assigned the largest possible value, then the condition `index >= 0` is still true, so this loop will run forever. Therefore, one way of having the loop run the appropriate number of iterations is as shown above.

Another solution, which looks a little more awkward, is to start at `capacity - 1`, but to change the condition to the following:

```
for ( std::size_t index{capacity - 1}; index < capacity; --index ) {
    // Loop body
}
```

Thus, the condition will be true until we subtract one from zero. Now the function could be implemented as follows, but we need a comment to inform any future reader of this code what we are doing:

```
void accumulate( double array[], std::size_t const capacity ) {
    // Iterate from 'capacity - 1' down to and including '0'
    for ( std::size_t index{capacity - 1}; index < capacity; --index ) {
        // Add to array[index] all previous entries in the array
        for ( std::size_t k{0}; k < index; ++k ) {
            array[index] += array[k];
        }
    }
}
```

The best solution

At this point, we have introduced two alternative solutions so as to introduce two programming techniques that under certain circumstances you may be forced to use anyways; however, for this specific problem, there is a solution that is actually faster and clean than either of these two other solutions:

```
void accumulate( double array[], std::size_t const capacity ) {
    for ( std::size_t index{1}; index < capacity; ++index ) {
        array[index] += array[index - 1];
    }
}
```

With the first iteration of this loop, we get that

```
array[1] = array[1] + array[0]
```

With the next iteration of the loop,

```
array[2] = array[2] + array[1]
```

But by this point, `array[1]` has already been updated to equal `array[1] + array[0]`, so in fact, array[2] is now assigned the sum of the first three entries of the original values in the array.

The same goes with the next array, which assigns

```
array[3] = array[3] + array[2]
```

But by this point, `array[2]` has already been updated with what is now `array[2] + array[1] + array[0]` from the original values of the array. Thus, `array[3]` is now assigned the sum of the first four entries of the original array.

Summary

In any case, any of these solutions will take the array with entries

0.0	3.0	6.3	9.5	12.7	15.5	18.0	19.8	20.9	21.7	22.1	22.3	22.2

and replace it with the array with entries

0.0	3.0	9.3	18.8	31.5	47.0	65.0	84.8	105.7	127.4	149.5	171.8	194.0

Thus, in these first 12 seconds, the car has moved approximately 194 metres. This is not the best approximation given this data, and in your course on numerical analysis, you will learn better ways of estimating the distance the car has moved under the assumption that the acceleration of the vehicle is continuous and reasonably smooth.