Outline

- In this lesson, we will:
  - Describe an algorithm for searching an array
  - Discuss the use of the `const` keyword
  - Consider the drawbacks of our implementation
    - Determine a better implementation

Searching an array

- Suppose we want to determine if an array contains a specific value
  - We would have to compare each entry with the specified value
    - If we find it, return `true`, otherwise, return `false`
- The function declaration will be

```
bool find( double value, double array[], std::size_t capacity );
```

- Recall previously, we used an `unsigned int` for array indices;
  however, `std::size_t` is an unsigned integer type guaranteed to be the appropriate size for an index into an array
  - On a 32-bit computer, `std::size_t` is 4 bytes
  - On a 64-bit computer, `std::size_t` is 8 bytes

Searching an array

- Question: should we use `const`?
  - Is there any reason that this function should:
    - Change any of the entries of the array?
    - Change any of the other parameters?
- A better function declaration would be:

```
bool find( double const array[], std::size_t const capacity,
           double const value );
```

Searching an array

- Implement this algorithm, we would need a for loop that runs from zero to the array capacity minus one

```
for ( std::size_t k{0}; k < capacity; ++k ) {



}
```

  - This is called a *linear search*
    - We are searching the entries in a linear order

Searching an array

- We will compare the entry of the array at index k with the value that is being searched for

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

    }
}
```

Searching an array

- If we find an entry containing that value, we will return `true`

```
for ( std::size_t k{0}; k < capacity; ++k ) {
    if ( array[k] == value ) {
        return true;
    }
}
```

Searching an array

- If we have finished searching the entire array and not found it, we will return `false`

```
for ( std::size_t k{0}; k < capacity; ++k ) {
    if ( array[k] == value ) {
        return true;
    }
}

return false;
```

Searching an array

- Here is the function with its declaration and definition:

```
bool find( double const array[], std::size_t const capacity,
           double const value );

bool find( double const array[], std::size_t const capacity,
           double const value ) {
    for ( std::size_t k{0}; k < capacity; ++k ) {
        if ( array[k] == value ) {
            return true;
        }
    }

    return false;
}
```


Useful return values

- Question:
  - How useful is the response?
  - If the returned value is `false`, we know the value is not in the array
  - If the returned value is `true`,
    now we have to search the array again to find it!
- If we already have found the entry,
  would it not be better to let the user know where it was found?

Useful return values

- Here is one possible improvement:
  - Pass a local variable by reference, and if the value is found,
    set that parameter to be the index where it is found

```
bool find( double const array[], std::size_t const capacity,
           double const value, std::size_t &index );

bool find( double const array[], std::size_t const capacity,
           double const value, std::size_t &index ) {
    for ( std::size_t k{0}; k < capacity; ++k ) {
        if ( array[k] == value ) {
            index = k;
            return true;
        }
    }

    return false;
}
```


The standard solution

- Here is a better solution:
  - If the value is found, return an index between `0` and `capacity - 1`
  - If the value is not found, return a special value: `capacity`

```
std::size_t find( double const array[], std::size_t const capacity,
                  double const value );

std::size_t find( double const array[], std::size_t const capacity,
                  double const value ) {
    for ( std::size_t k{0}; k < capacity; ++k ) {
        if ( array[k] == value ) {
            return k;
        }
    }

    return capacity;
}
```

Sample usage

- This is how we could use this function:
  - This is the approach in the standard template library (STL)

```
int main() {
    std::size_t CAPACITY{100};
    double data[CAPACITY];

    // Fill the array...

    std::size_t index{ find( data, CAPACITY, 13.0 ) };

    if ( index == CAPACITY ) {
        std::cout << "The array does not contain 13" << std::endl;
    } else {
        std::cout << "The array contain 13 at index " << index << std::endl;
    }

    // Finish...

    return 0;
}
```


Why const?

- Why declare parameters to be const?
  - Consider this error:

```
std::size_t find( double array[], std::size_t capacity,
                  double value ) {
    for ( std::size_t k{0}; k < capacity; ++k ) {
        if ( array[k] = value ) {
            return k;
        }
    }

    return capacity;
}
```

  - If value is `0`, this will assign all the entries to 0 and return `capacity`
  - Otherwise, the first entry will be assigned `value` and `0` will be returned

Summary

- Following this presentation, you now:
  - Know how to search an array
  - Understand that, were possible, parameters should be declared `const`
  - Understand that there are different ways of implementing the same functionality
  - Are aware of the approach used in the standard template library

References

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

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

Acknowledgments

Proof read by Dr. Thomas McConkey and Charlie Liu.

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