Data structures & algorithms

Tutorial 2

Lesson overview

- Recap of important topics from last week
- Introduction to `std::vector`
  - Main operations for `std::vector`
  - Advanced operations for `std::vector`
  - Looping through an `std::vector`
- Product of array except self problem
- A toy version of a vector

A quick recap of important things from last week

Data types and declaring and initializing variables

- Primitives:
  - `int integerNum;`
  - `double doubleNum;`
  - `float floatNum;`
  - `char character;`
  - `bool booleanValue;`
- Non-primitive
  - `std::string userName;`

  `// must #include <string>`

- Declaring and initializing:
  - `int num; // declaration`
  - `int num {}; // declaration`
  - `int num {24}; // initialization`
  - `int num = 24; // initialization`
- Pointer and reference
  - `int* numPtr {&num}; // Pointer initialization`
  - `int& numPtr {num}; // Reference initialization`
- Vector
  - `std::vector<int> myVector {}; // Vector initialization`

  `// must #include <vector>`

- Classes
  - `MyClass myClass {...}; // Class initialization`
  - `MyClass myClass = MyClass(...); // Static initialization`
  - `MyClass myClass = new MyClass(...); // Dynamic initialization`

Pass by ...

- `passByValue(int a)`
  - Pass-by-value functions create a copy of the original value. Any alterations made to the value within the function will not change the original
- `void passByReference(int &a)`
  - Pass-by-reference functions are passed a reference to the original value. Any alterations made to the value within the function will change the original
- `void passByPointer(int *a)`
  - Pass-by-pointer functions are passed a pointer to the original value's memory address. Any alterations made to the value within the function will change the original
- `void passByConstReference(const int &a)`
  - Pass-by-const-reference functions are similar to pass-by-reference, however, the value is immutable and can't be altered

Pointers and References

As initializers

- Reference, `int &value`
  - A reference assigns an alias to an existing variable, both allocated to the same memory address

- Pointer, `int *value`
  - A pointer initializes a variable which contains the memory address of the variable it is pointing at

As operators

- Address-of, `&value`
  - The address-of operator will return the memory address of a variable

- Dereference, `*value`
  - The dereference operator will dereference the pointer and return the value it is pointing to

Lab time 😎

std::vector

```
std::vector<type> myVector{};
```

- A vector is essentially C++'s version of an ArrayList from Java. Once again it is found in the standard library and can be included within your cpp file with #include <vector>
- For those unfamiliar with ArrayList's, they are a resizable array that have their elements stored in contiguous locations. Which allows element access via an index.
- Replacing type with a type of your choosing will only allow storage of that type only. I.e., std::vector<int> myVector{}; will only ever be able to store int's

std::vector

Main operations for std::vector

```
std::vector<int> myVector{4, 5, 9, 10}; // Initialize with values
```

- `myVector[n]`: accesses element at index n
  - `myVector[0] // returns 4 (element at the 0th index)`
- `myVector.at(n)`: accesses element at index n
  - `myVector.at(2) // returns 9 (element at the 2nd index)`
- `myVector.front()`: accesses element at the front of the vector
  - `myVector.front() // returns 4`
- `myVector.back()`: accesses element at the back of the vector
  - `myVector.back() // returns 10`
- `myVector.push_back(value)`: pushes value to the back of the vector
  - `myVector.push_back(13) // vector now contains {4, 5, 9, 10, 13}`
- `myVector.pop_back()`: removes element at the end of the vector
  - `myVector.pop_back() // vector now contains {4, 5, 9}`
- `myVector.size()`: returns the size of the vector
  - `myVector.size() // returns 9 (element at the 2nd index)`
- `myVector.empty()`: empties the vector
  - `myVector.empty() // vector now contains {}`

std::vector

`myVector[n] vs myVector.at(n)`

- Both `myVector[n]` and `myVector.at(n)` perform the same operation, which allows you to access elements of the vector via an index.
- The only minor difference is that `myVector.at(n)` performs a bound checking whilst `myVector[n]` doesn't
- For example, if you have a vector with 5 elements and try to access the 6th element. `myVector.at(n)` will perform a bound check and throw an std::out_of_range exception since the 6th element doesn't exist. Whilst `myVector[n]` won't throw an exception and will try and access an element that doesn't exist, which will result in undefined behaviour

std::vector

Advanced operations for `std::vector`

```
std::vector<int> myVector{4, 5, 9, 10}; // Initialize with values
```

- `myVector.begin()`: access the element at the beginning of the vector as an iterator
  - `*(myVector.begin()) // returns 4`
- `myVector.end()`: access the element one past the end of the vector as an iterator
  - `*(myVector.end()) // returns 0 since it is one past the last element, we can get the last element with *(myVector.end() - 1)`
- `myVector.rbegin()`: accesses reverse iterator to reverse beginning
  - Is essentially `myVector.end()` but makes iterating through the vector in reverse easier
- `myVector.rend()`: accesses reverse iterator to reverse end
  - Is essentially `myVector.begin()` but makes iterating through the vector in reverse easier

std::vector

Looping through an std::vector

For loop

```
for(int i = 0; i < myVector.size(); i++) {
     std::cout << myVector.at(i) << std::endl
}
```

Range based for loop

```
for(int i : myVector) {
     std::cout << i << std::endl
}
```

For loop with iterators

```
for(auto i = myVector.begin(); i != myVector.end(); i++) {
     std::cout << *i << std::endl
}
```

Give “playing with std::vector” a crack to see how vectors work

Product of array except self

nums

4	2	5	1	3

- The goal is to return a `std::vector` result of the same size as nums such that `results[n]` is the product of all the integers in `nums` *except* for the ith one.

Product of array except self

nums	4	2	5	1	3
results	30	60	24	120	40

- The goal is to return a `std::vector` result of the same size as nums such that `results[n]` is the product of all the integers in `nums` *except* for the ith one.

Product of array except self

nums				
4	2	5	1	3

~~4~~ x 2 x 5 x 1 x 3 = 30

results				
30	60	24	120	40

- The goal is to return a `std::vector` result of the same size as nums such that `results[n]` is the product of all the integers in `nums` *except* for the ith one.

Product of array except self

nums					
nums	4	2	5	1	3

4 x ~~2~~ x 5 x 1 x 3 = 60

results					
results	30	60	24	120	40

- The goal is to return a `std::vector` result of the same size as nums such that `results[n]` is the product of all the integers in `nums` *except* for the ith one.

Product of array except self

nums				
4	2	5	1	3

$$4 \times 2 \times \not{5} \times 1 \times 3 = 24$$

results				
30	60	24	120	40

- The goal is to return a `std::vector` result of the same size as nums such that `results[n]` is the product of all the integers in `nums` *except* for the ith one.

Product of array except self

nums

4	2	5	1	3

4 x 2 x 5 x ~~1~~ x 3 = 120

results

30	60	24	120	40

- The goal is to return a `std::vector` result of the same size as nums such that `results[n]` is the product of all the integers in `nums` *except* for the ith one.

Product of array except self

nums	4	2	5	1	3

4 x 2 x 5 x 1 x ~~3~~ = 40

results	30	60	24	120	40

- The goal is to return a `std::vector` result of the same size as nums such that `results[n]` is the product of all the integers in `nums` *except* for the ith one.

Product of array except self: brute-force

```
std::vector<int> productExceptSelf(const std::vector<int>& nums) {
    std::vector<int> results{};
    for(unsigned int i = 0; i < nums.size(); i++) {
        int sum {1};
        for(unsigned int j = 0; j < nums.size(); j++) {
            if(i == j) continue;
            sum *= nums.at(j);
        }
        results.push_back(sum);
    }
    return results;
}
```

- This is the most general brute-force approach for this problem. A brute-force approach is usually slower than a more optimized one, but it is usually easy to think of and confirms that the problem is solvable

Product of array except self: brute-force

```
std::vector<int> productExceptSelf(const std::vector<int>& nums) {
    std::vector<int> results{};
    for(unsigned int i = 0; i < nums.size(); i++) {
        int sum {1};
        for(unsigned int j = 0; j < nums.size(); j++) {
            if(i == j) continue;
            sum *= nums.at(j);
        }
        results.push_back(sum);
    }
    return results;
}
```

- This is the most general brute-force approach for this problem. A brute-force approach is usually slower than a more optimized one, but it is usually easy to think of and confirms that the problem is solvable
- Due to the nested for-loops, the problem has a time complexity of O(n^2). This means if you use an array with 5 elements in it, it will take on average 5^2 iterations

Product of array except self: brute-force

```
std::vector<int> productExceptSelf(const std::vector<int>& nums) {
    std::vector<int> results{};
    for(unsigned int i = 0; i < nums.size(); i++) {
        int sum {1};
        for(unsigned int j = 0; j < nums.size(); j++) {
            if(i == j) continue;
            sum *= nums.at(j);
        }
        results.push_back(sum);
    }
    return results;
}
```

- This is the most general brute-force approach for this problem. A brute-force approach is usually slower than a more optimized one, but it is usually easy to think of and confirms that the problem is solvable
- Due to the nested for-loops, the problem has a time complexity of O(n^2). This means if you use an array with 5 elements in it, it will take on average 5^2 iterations to solve
- Before I let you try and implement a better solution, here's a hint. What if the array has no zeros, one zero, or two zeros

Give “product of array except self” a go and see what solutions you come up with

A toy version of vector

I'm going to let you try and write out the member functions for our MyVector class, I'd suggest creating the functions in this order:

1. Constructor, one to make an n-sized array, and another to create one with a pre-existing list
2. Size, capacity, empty, and back (very simple, only one line of code!)
3. Operator function which allows you to get and set the ith element
4. Destructor to the dynamically allocated memory in MyVector
5. Pop_back to remove the last element in the vector
6. Push_back, this will be the trickiest one to implement. This function will allow you to insert a new element to the end of the vector, and once the vector is full you will need to implement the array doubling scheme you saw in the lecture

Access to google drive

- I will upload slides to the Google Drive after every class
- https://drive.google.com/drive/folders/1H5psebndM_YVyoJE-BJ_ODNJOfgq9-ul

Contact: Thomas.golding@uts.edu.au