4. Vectors and STL

Ján Dugáček

October 10, 2018

Table of Contents

1. Motivation
2. Vector
   - Exercise
3. Files
   - Working with files
   - The easy way
   - Exercise
4. Other containers
5. Homework

Motivation

- Can you write a program that gets 10 numbers and sorts them from the smallest up to the largest?

Vector

```
#include <vector>
//...
        std::vector<int> numbers;
        for (int i = 0; i < 10; i++) {
                int read;
                std::cin >> read;
                numbers.push_back(read);
        }
        std::cout << "At 3 is: " << numbers[3] << std::endl;
```

- `std::vector` is a data structure that stores an indefinite number of elements
- It can store only one type of element, as defined in the < > brackets
- Elements are inserted using `push_back`

Vector #2

```
for (unsigned int i = 0; i < numbers.size(); i++) {
        std::cout << "At index " << i << " is " <<
                numbers[i] << std::endl;
}
```

- Elements can be accessed through the square brackets
- Number of elements can be read using the `length()` method
- **The first element is at index zero**
- Accessing elements at negative indexes or after the last one causes *undefined behaviour*, which can mean random overwriting of variables or crashes

Vector #3

```
for (int& num : numbers) {
        std::cout << "We have " << num << std::endl;
}
```

- The `for` can be shortened if we don't need to know the index

```
numbers.erase(numbers.begin() + 5);
```

- Erases element 5

Vector #4

```
#include "easy.h"
//...
        easy::vector<float> vec;
```

- Accessing elements at negative indexes or after the last one causes *undefined behaviour*, which can mean random overwriting of variables or crashes
- Use my `easy::vector` instead to receive warnings instead (at the cost of execution speed)

Exercise

1. Have the user supply 5 numbers and output them afterwards
2. Have the user supply 5 numbers and output them in some other ordering
3. Have the user supply 5 numbers and then 1 number to set if he wants to get their arithmetic average, geometric average or harmonic average

Working with files

```
#include <fstream>
/...
std::ofstream out("output.dat");
out << 42 << std::endl;
```

- The changes are written to disk when the variable ceases to exist
- Works much like `std::cout`, but is faster

```
std::ifstream in("input.dat");
int num;
in >> num;
```

- Works much like `std::cin`

The easy way

```
#include "easy.h"
//...
easy::vector<float> numbers("numbers.dat", '\n');
        for (float& num : numbers) {
                std::cout << "We have " << num << std::endl;
        }
```

- I have created this comfy library for you that does it easily
- The first argument is the file name, the second argument is the separator

The easy way #2

```
#include "easy.h"
//...
easy::vector<easy::vector<float>> numbers
        ("numbers.dat", '\n', '\t');
std::cout << numbers[3][4] << std::endl;
```

- It can also parse tables, the second argument is line separator, the third is the column separator

The easy way #3

```
#include "easy.h"
//...
        easy::vector<float> numbers;
        //...
        easy::write_file(numbers, "numbers.dat", '\n');
        // or
        numbers.write_file("numbers.dat", '\n');
```

- So that you could also write files easily

Exercise

1. Have the user supply numbers in a file, output them in order from smallest to greatest
2. Have the number supply numbers in a file and perform a linear fit, i. e. `f(line_number)` = `a` · `line_number`, you may assume the slope will be between 0.01 and 1000

Other containers

1. `std::map` allows indexing using nearly anything, but is slower (access time depends on the logarithm of size)
2. `std::unordered_map` is not so slow, but elements are read in a strange order
3. Accessing an unused location will create an uninitialised variable there

```
#include <map>
//...
std::map<std::string, int> array;
array["hi"] = 3;
array["zaphod"] = 4;
array["ford"] = array["zaphod"] - 1;
for (std::pair<std::string, int>& it : array)
    std::cout << it.first << "="<< it.second << std::endl;
```

Other containers

1. `std::array` is faster than vector, but it has a fixed size
2. Use `easy::array` to get boundary checking, it suffers from *undefined behaviour* like vector

```
#include <map>
//...
std::array<int, 4> array; // Has space for 4 elements
array[3] = 3;
array[2] = 4;
std::cout << "At 1 is " << array[1] << std::endl;
```

Homework

- Read numbers from a file and print them ordered, making sure the execution time is does not depend quadratically on the size of input
- The numbers will be between 0 and 1000000, ordered evenly over the range
- You have two weeks to do it