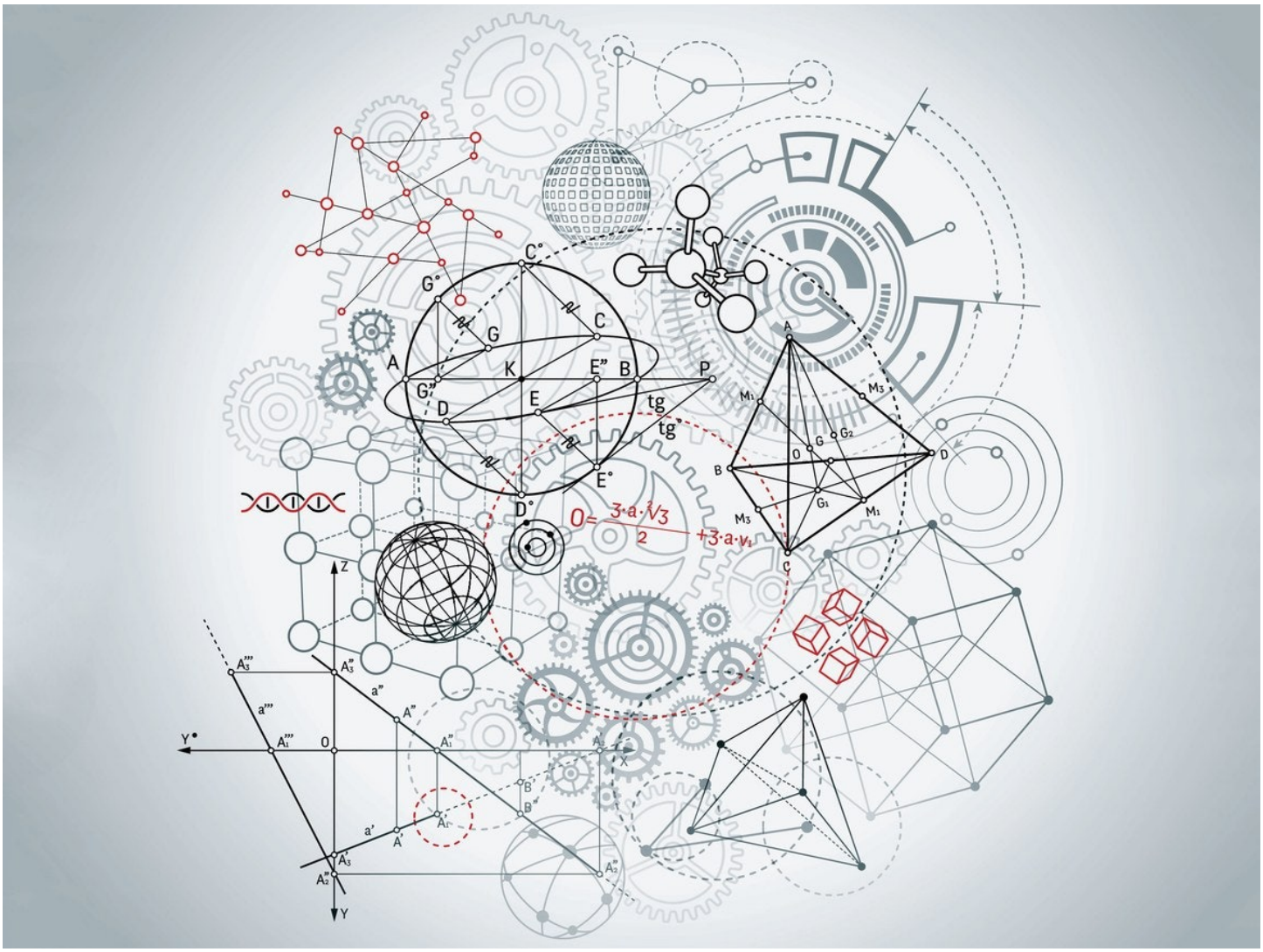
Lab 12. Vectors: A Song of Push and Pop



Introduction

Arrays are important and versatile data structures. They are very useful to hold a collection of similar items and have good synergy with loops. Many abstract data structures can be implemented with arrays. When we began to learn about arrays, we learned about static arrays, whose size must be

known at compile time. We then saw dynamic arrays, which can be created at runtime without a priori knowledge of their size. In this lab, we expand our repertoire of arrays with the introduction of the `vector` class from the standard C++ library. Vectors are essentially dynamic arrays that can resize automatically at runtime. In addition, vectors can be returned from a function!

Usefull Vector Methods

Initialization

To use vector, the program must contain the `#include <vector>` preprocessor directive. The simplest way to initialize a vector is to declare an object of that type:

```
vector<int> v; // creates a vector of int called v
```

For all intents and purposes, think of a vector as an array that can resize itself. In practice, this means we do not need to specify a size when creating a vector. To add an element to a vector, use the `push_back(element)` method, where element is a value of the same type as the vector. Example:

```
vector<int> v;
v.push_back(10);
v.push_back(20);
v.push_back(30);

// v now contains elements [10, 20, 30]
```

Vectors can also be initialized with list initialization syntax. The same vector could have been created as follows:

```
vector<int> v{10, 20, 30};
// v now contains elements[10, 20, 30]
```

Element Access

- To access an element, use the same reference operator used with arrays `[ ]`. In the above example, `v[1]` would return the value 20 as an `int`.
- `at(i)` returns a reference to the element at position i (zero-indexed) in the vector.
- `front()` returns a reference to the first element in the vector.
- `back()` returns a reference to the last element in the vector.

Size vs Capacity

The **size** of a vector is the number of elements in the vector. The **capacity** of a vector is the storage space currently allocated to the vector. As such,

- `size()` returns the number of elements in the vector.
- `capacity()` returns the number of elements the vector can hold before more memory must be allocated.
- `empty()` returns true if the size is 0. To make this easier to understand, think of the vector as a container, say a bottle. A 1L bottle can hold one liter of water, this is its *capacity*. But if it is half full (or half empty for the goths) then its *size* is half a liter. This means that if we want to store more than one liter, we will need a bigger bottle (allocate more memory). To test this, create a few vectors, fill them with elements, and check the difference between their size and capacity as the number of elements increase.

Mutators (also known as setters or modifiers)

- `push_back(n)` adds element `n` at the back (end) of the vector.
- `pop_back()` removes the last element in the vector.
- `clear()` removes all elements from the vector.

Task A: The easy one

A big benefit of vectors is their ability to be returned from functions. For this task, program a function called `vector<int> makeVector(int n)` that returns a vector of n integers that range from 0 to n-1. Call your program `vectors.cpp`. Your function *must* be implemented outside the main

function and must return a vector.

Task B: A Happy Filter

Make a program called `optimism.cpp` that implements the function `vector<int> goodVibes(const & vector<int> v);` that, given a vector of integers, returns a vector with only the positive integers in the same order.

```
vector<int> v{1,2,-1,3,4,-1,6};

goodVibes(v); // returns [1,2,3,4,6]
```

Task C: It's over 9000!

Make a program called `fusion.cpp` that implements the function `void gogeta(vector<int> &goku, vector<int> &vegeta)` that appends elements of the second vector into the first and empties the second vector. For example:

```
vector<int> v1{1,2,3};
vector<int> v2{4,5};

gogeta(v1, v2); // v1 is now [1,2,3,4,5] and v2 is empty.
```

Task D: Pairwise sum

Write a program called `pairwise.cpp` that implements the function `vector<int> sumPairWise(const vector<int> &v1, const vector<int> &v2)` that returns a vector of integers whose elements are the pairwise sum of the elements from the two vectors passed as

arguments. If a vector has a smaller size than the other, consider extra entries from the shorter vectors as 0. Example:

```
vector<int> v1{1,2,3};
vector<int> v2{4,5};

sumPairWise(v1, v2); // returns [5, 7, 3]
```
