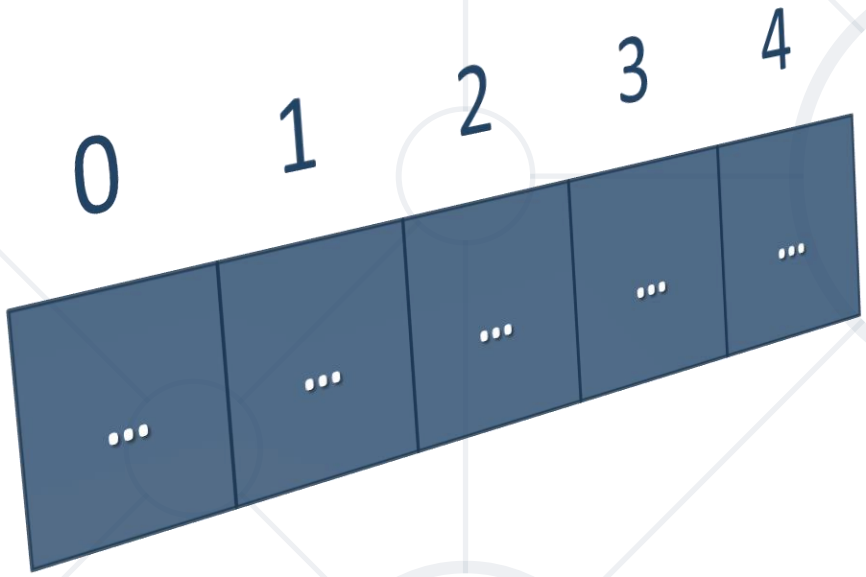
Vectors, Lists and Iterators



SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Have a Question?

Software University

sli.do

#cpp-fundamentals

Table of Contents

Data Structures and Complexity

Data Structures

- Data Structures **organize data for efficient access**
  - Different data structures **are efficient for different use-cases**
  - Essentially: **a data container + algorithms for access**
- Common data structures:
  - **Arrays** - fast access by index and **constant** or **dynamic** size
  - **Linked-list** - fast **add** or **remove** at any position and no index access
  - **Map** / **Dictionary** - contains **key** / **value** pairs and fast access by key used for searching

Complexity 101

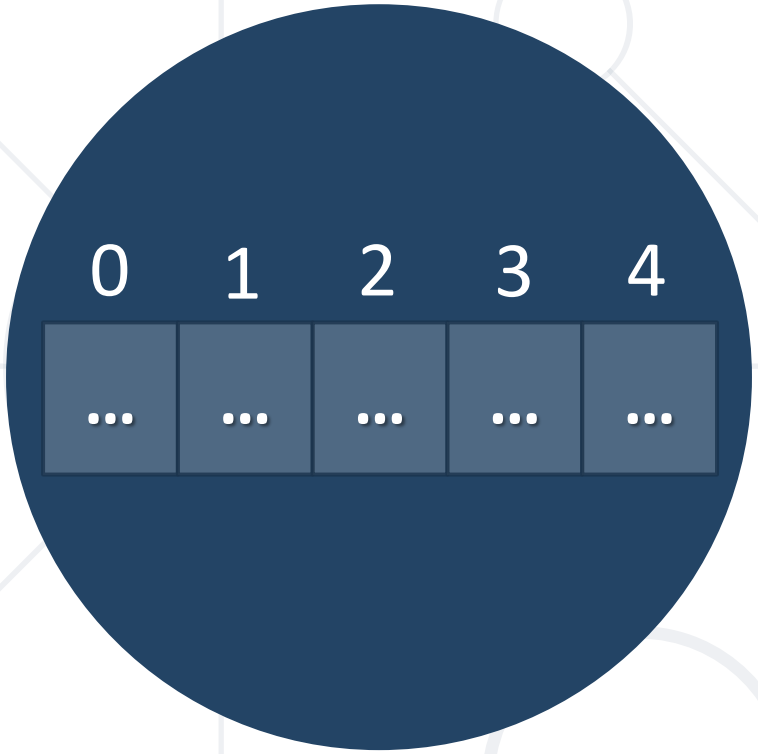
- We usually care about **X** orders of magnitude, not **+X** or ***X**
  - `O(N+3) == O(2N) == O(N)`
  - `O(1)` – **"constant" time / memory** – input size has no effect
  - `O(log(N))` – **logarithmic** – complexity grows as log(input) grows
  - `O(N)` – **linear** – complexity grows as input grows
  - $O(N^2)$, $O(N^3)$ – **quadratic, cubic** – complexity grows with square/cube of input size
  - $O(2^N)$, $O(3^N)$ – **exponential** – this is a monster

Data Structure Performance 101

Software University

- Time complexity of operations, if **N** is the number of elements in the container (the `.size()`):

	vector	list	map, set	unordered_map, unordered_set
access i^{th}	O(1)	O(i)	O(i)	---
Find (V)	O(N)	O(N)	O(log(N))	O(1) (usually)
Insert (V)	O(1) at end (usually), O(N) otherwise	O(1)	O(log(N))	O(1) (usually)
Remove (V)	O(1) at end (usually), O(N) otherwise	O(1)	O(log(N))	O(1) (usually)
Getting a sorted sequence	O(N*log(N)) (using std::sort algorithm)	O(N + N*log(N)) (using .sort() method)	O(N) (by just iterating)	---



Vectors

Dynamically-Sized Arrays

STL Vector Basics

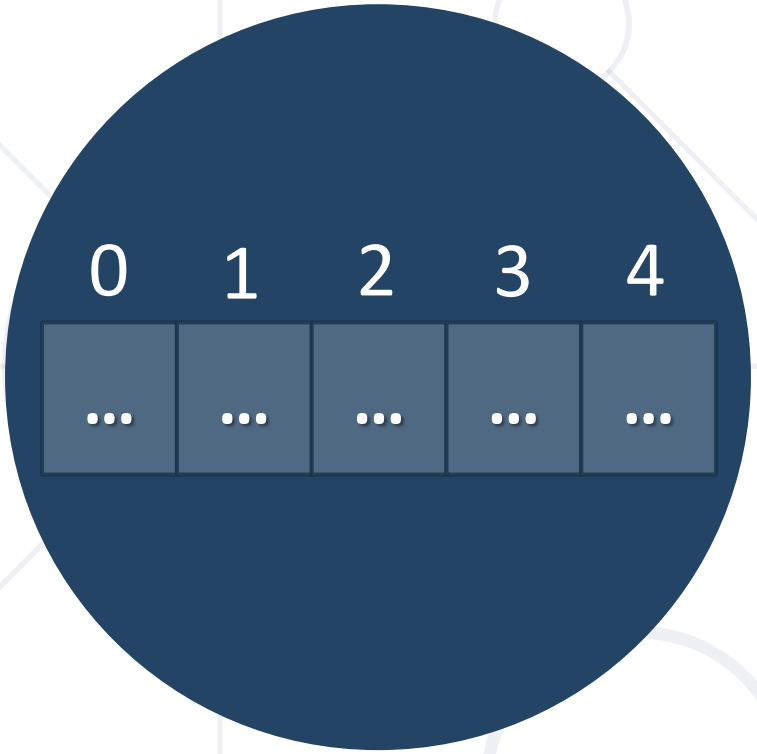
- `std::vector` class is a resizable array

  `#include<vector>`

  - Normal array-like access – `[]` operator
  - Size is known (`size()`)
  - Adding elements (`push_back(), emplace_back()`)
- Acts like a normal variable
  - Can be **assigned like a normal variable**
  - Can be **returned from a function**

std::vector

- Has **all array operations**
- **Changes size automatically when elements added**
- **`push_back()`**
  - Complexity is **amortized `O(1)`**
  - Use when it **has to reallocate**
  - Usually takes **`O(1)`** time, occasionally takes **`O(N)`** time
- **`reserve()`**
  - Use when you **know the size in advance**



Initializing STL Vectors

Initializing a Vector

- Declaration Syntax: `std::vector<T> name;`
- The vector is **initially empty** – items need to be added
  - Use `push_back(T element)` on the vector to add elements

```
std::vector<int> myVector;
myVector.reserve(100);
for (int i = 0; i < 100; i++)
{
    myVector.push_back(i);
}
```

- Can be initialized directly with `{}` syntax

```
std::vector<int> numbers {13, 42, 69};
std::vector<int> numbers = {13, 42, 69};
```

Returning STL Vectors from Functions

Returning STL Vectors from Functions

```
void print(const vector<double> &numbers)
{
    for (int number : numbers)
    {
        cout << number << " "
    }
    cout << endl;
}
```

Vectors acts as normal variables when returned

Returning STL Vectors from Functions

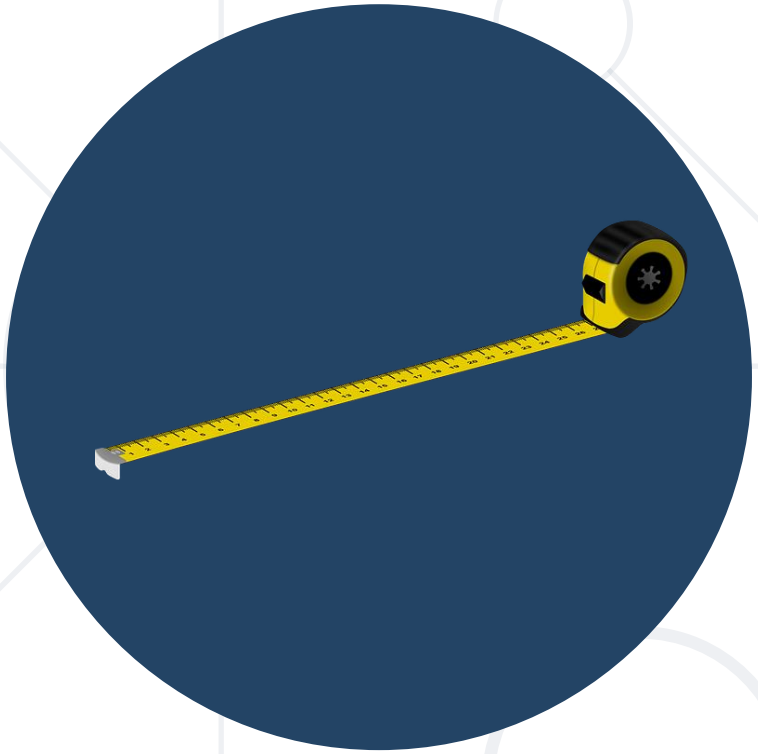
Software University

```
vector<double> getSquareRoots(int from, int to)
{
    vector<double> roots;
    roots.reserve(to-from);
    for (int i = from; i <= to; i++)
    {
        roots.push_back(sqrt(i));
    }
    return roots;
}

int main()
{
    print(getSquareRoots(4, 25));
    return 0;
}
```

Vectors acts as normal variables when returned

Function returns a copy

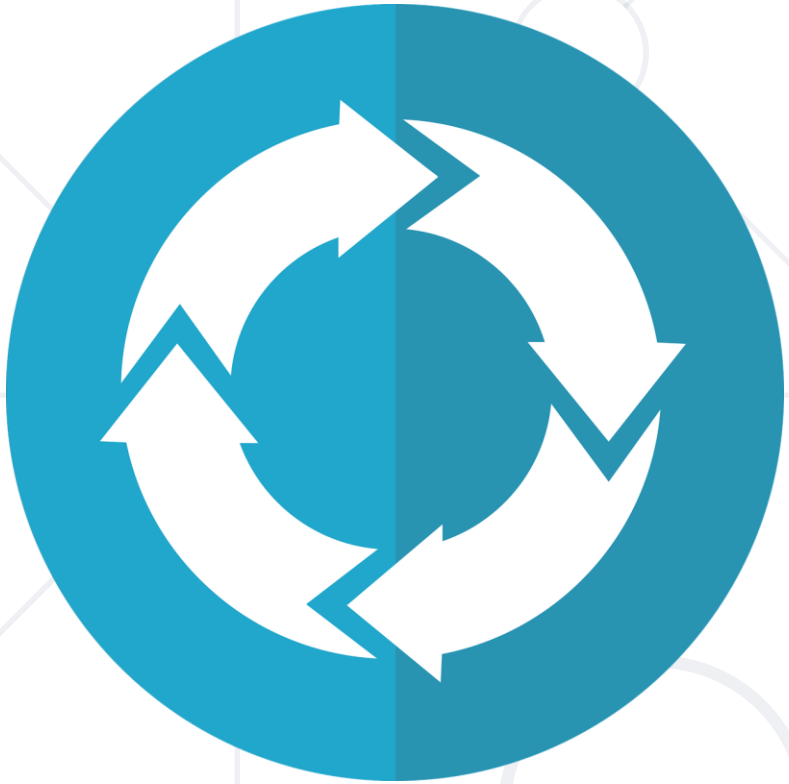


size_t and size_type

size_t and size_type

- Alias of one of the integer types
  - `unsigned long int` or `unsigned long long int`
  - Able to represent the **size of any object in bytes**
  - `sizeof()` returns `size_t`
- Each STL container offers a similar `::size_type`

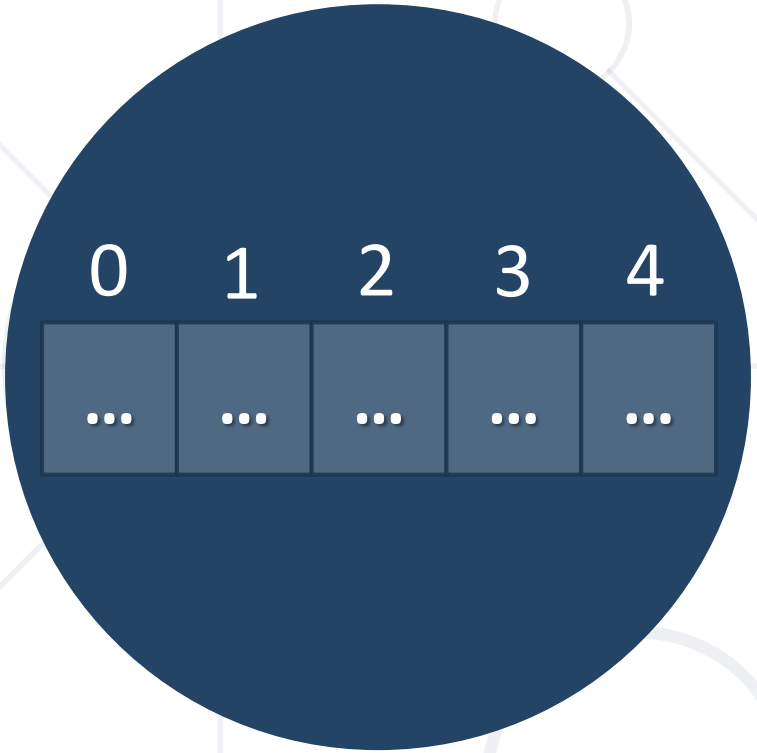
```
for (vector<int>::size_type i = 0; i < nums.size(); i++)
{
  cout << nums[i] << endl;
}
```



Iterators

Container Iterators

- STL Iterators are things that know how to **traverse a container**
  - `operator++` - moves iterator to the **next element**
  - `operator*` - accesses the **element**
  - `operator->` - same as **`operator.`** on the element
- Each container has **an iterator** (`std::vector<T>::iterator`)
- Each container has `begin()` and `end()` iterators
  - `begin()` points to **first** element and `end()` to **after** last
  - Range-based `for`-loop uses them to work on **any** container



Using Iterators with Vectors

Using Iterators with Vectors

- Using iterators on **vectors** is almost the same as using indexes
- To go through a vector:
  - Start from **begin()**, move with **++** until you reach **end()**

```
vector<int> nums {42, 13, 69};
for (vector<int>::iterator i = nums.begin(); i != nums.end(); i++)
{
  cout << *i << endl;
}
```

```
for (vector<int>::size_type i = 0; i < nums.size(); i++)
{
  cout << nums[i] << endl;
}
```

Using Iterators

- **Example**: Change each element in the vector by dividing it by 2

```
vector<int> numbers {42, 13, 69};
for (vector<int>::iterator i = numbers.begin(); i != numbers.end(); i++)
{
  *i /= 2;
}
```

```
for (int i = 0; i < numbers.size(); i++)
{
  numbers[i] /= 2;
}
```

Using Iterators

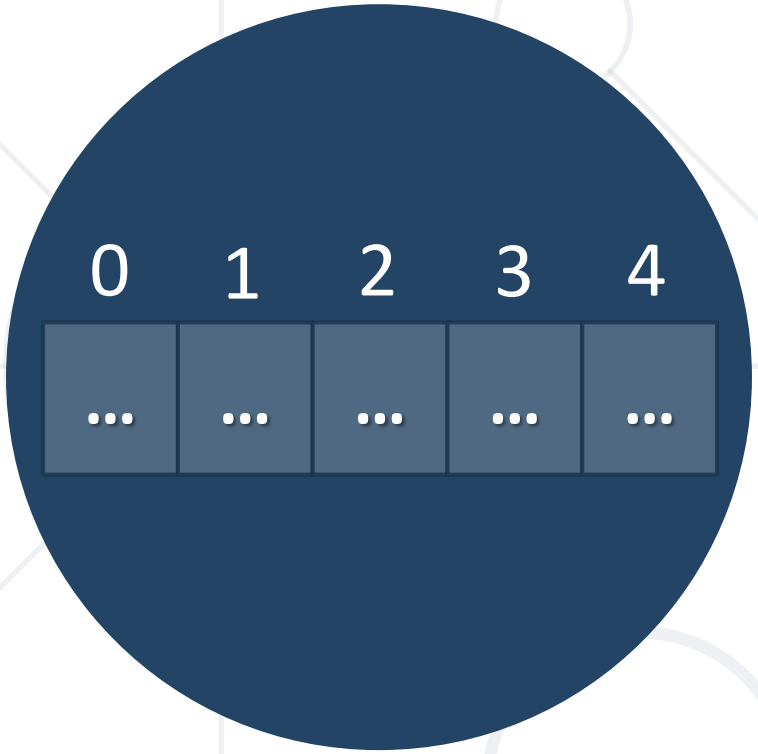
- **Example**: Print each string element and its length

```
vector<string> words {"the", "quick", "purple", "fox"};
for (vector<string>::iterator i = words.begin(); i != words.end(); i++)
{
  cout << *i << ": " << i->length() << endl;
}
```

```
for (int i = 0; i < words.size(); i++)
{
  cout << words[i] << ": " << words[i].length() << endl;
}
```

Why Use Iterators?

- Vectors may not need iterators, because **they have indexes**
  - They have sequential elements accessible by **`operator[]`**
- Not **all containers have indexes**
  - Only **`std::array`**, **`std::vector`** and **`std::deque`** have indexes
  - The other containers **don't offer access by index**
- Iterators **work on all containers**, abstract-away container details
  - No matter what container you iterate, **code is the same**



Lists

std::list

- Represents elements **connected to each other in a sequence**

```
std::list<int> values;
std::list<string> names;
```

- Each element **connects to the previous and next element**
- All element access is done with **iterators**
- Can add or remove elements anywhere in `O(1)` time
- Requires iterator to where an element should be **added** or **removed**
- `push_back()`, `push_front()`, `insert()`, `size()`

Summary

- We usually measure performance based on input
  - We care how quickly much performance degrades based on input size
  - We use Big-O notation to denote that
- STL Vectors
- Iterators
- Lists

SoftUni Diamond Partners

Software University

SmartIT

Coca-Cola HBC Bulgaria

SUPER HOSTING .BG

createX

Postbank

top EMPLOYER България Bulgaria 2024 FOR A BETTER WORLD OF WORK

DRAFT KINGS

THE CROWN IS YOURS

VIVACOM

INDEAVR

Serving the high achievers

AMBITIONED

PHAR VISION

Questions?

SoftUni

Software University | SoftUni AI | SoftUni Digital | SoftUni Creative | SoftUni Foundation | СофтУни БУДИТЕЛ | Finance Academy

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, about.softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity

License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – https://about.softuni.bg/
- © Software University – https://softuni.bg