Optimized C++

Chapter 10

Wonjun Lee

July 15, 2017

Table of Contents

Intro

Overview the Standard Library

- Optimization 관점에서, 컨테이너들의 아래의 프로퍼티들이 중요
  - 각 메소드별 $\mathcal{O}(n)$ performance
  - Item을 appending 할 시 발생하는 상수 시간 비용
  - Dynamic memory allocation시 미세한 조절이 가능한가?

- 추가로 알아둬야할 점
  - 다른 컨테이너의 같은 이름의 메소드라도, $\mathcal{O}(n)$ performance 는 다를 수 있음
  - 같은 이름의 메소드라도, Semantic 이 다를 수 있음

Sequence Containers

In STL...

- std::string
- std::vector
- std::deque
- std::list
- std::forward_list

Features

- Front, Back Insertion
- Subscripting operator ([] operator)
- Array-like internal backbone

Sequence Containers

In STL...

- std::string
- std::vector
- std::deque
- std::list
- std::forward_list

Features

- Front, Back Insertion
- Subscripting operator ([] operator)
- Array-like internal backbone

Sequence Containers

In STL...

- std::string
- std::vector
- std::deque
- std::list
- std::forward_list

Features

- Front, Back Insertion
- Subscripting operator ([] operator)
- Array-like internal backbone

Constant time operation!

Sequence Containers

In STL...

- std::string
- std::vector
- std::deque
- std::list
- std::forward_list

Features

- Front, Back Insertion
- Subscripting operator ([] operator)
- Array-like internal backbone

아이템 엑세스를 다음과 같이 가능 *container[i]*

Sequence Containers

In STL...

- std::string
- std::vector
- std::deque
- std::list
- std::forward_list

Features

- Front, Back Insertion
- Subscripting operator ([] operator)
- Array-like internal backbone

아이템이 삽입될 때 *reallocation*이 발생 할 수 있음

Sequence Containers

In STL...

- std::string
- std::vector
- std::deque
- std::list
- std::forward_list

Features

- Front, Back Insertion
- Subscripting operator ([] operator)
- Array-like internal backbone

삭제되는 아이템의 *pointer, iterator*만 무효화. *Iterator* 무효화 없이 *splice, merge* 가능

Associative Containers

Ordered Associative Containers

- std::map
- std::set
- std::multimap
- std::multiset

Features

- Inserting 순서와 상관없이 소팅을 한 후 저장
- operator<() 를 필요로 함
- Insertion, Deletion 은 보통 $\mathcal{O}(\log_2 n)$

Associative Containers

Unordered Associative Containers

- std::unordered_map
- std::unordered_set
- std::unordered_multimap
- std::unordered_multiset

Features

- keys, item 간의 equality relationship만 정의되어 있으면 됨
- operator=() 를 필요로 함
- Hash-table로 내부구현이 되어 있음
- Insertion, Deletion 은 평균 $\mathcal{O}(1)$, worst-case 경우 $\mathcal{O}(n)$

std::vector Overview

std::vector Overview

- std::string 은 std::vector의 derived class
- Sequence container
- Insertion time from back $\mathcal{O}(1)$, else $\mathcal{O}(n)$
- Index time $\mathcal{O}(1)$
- Sort in $\mathcal{O}(n \log_2 n)$
- Search in $\mathcal{O}(\log_2 n)$ if sorted, else $\mathcal{O}(n)$
- Internal memory가 reallocation 되면 iterator, reference 를 사용할 수 없음.
- Random accessible iterator(동일 벡터 내의 iterator 사이의 거리 계산이 constant time임.)
- Dynamically Resizable Memory
- Flat, Continous memory structure
- Reasonable control over allocated capacity independent of size

std::vector Size & Capacity

- Size vs Capacity
  - Size == # of items in vector
  - Capacity == size of internal memory (보통 Size의 몇 배 크기로 자동 설정됨)
  - Reallocate memory when Capacity < Size

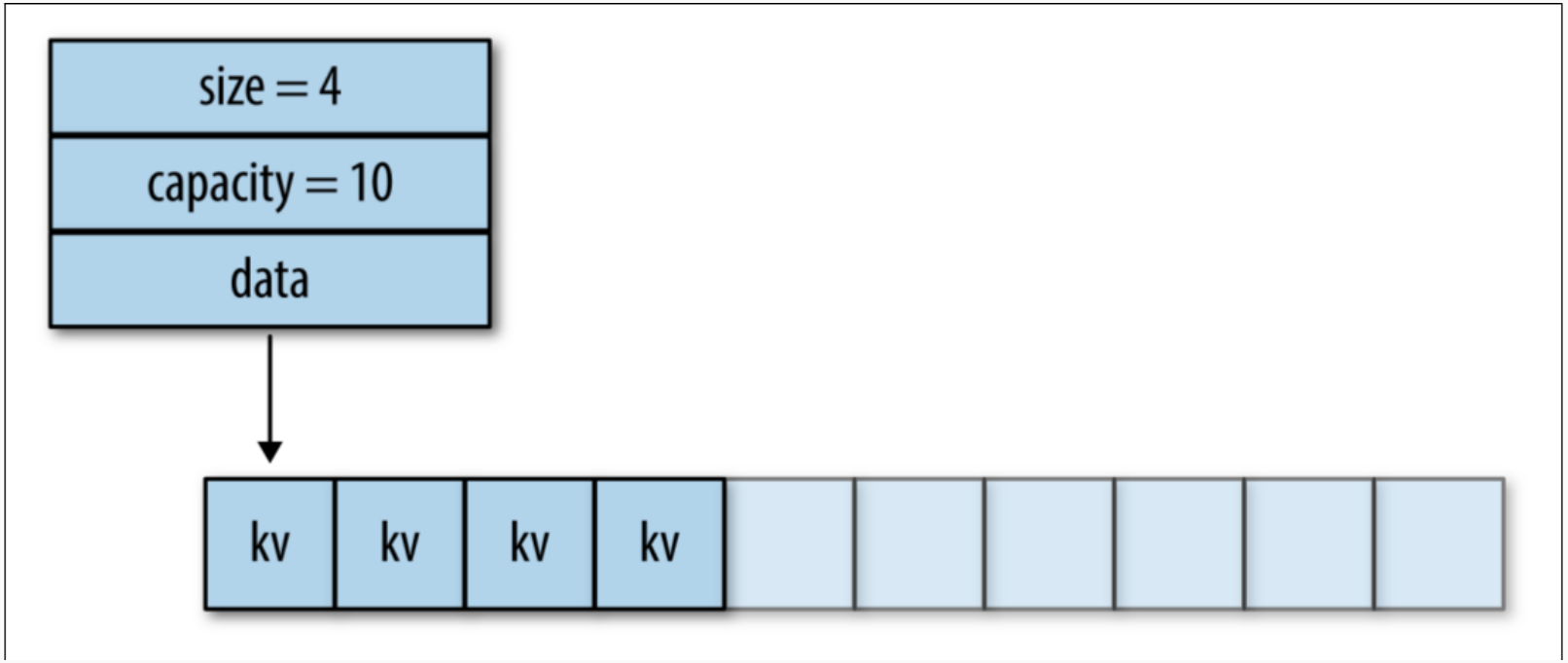
- Capacity는 `void reserve(size\_t)` 로 조절 가능

- std::vector 는 내부 아이템을 삭제하더라도 메모리를 반환하지 않음. (`void clear()` 사용 시에도 reallocation이 일어나지 않을 수 있음!)

- **TRICK**, 벡터 메모리 회수

```
std::vector<char> x;
...
vector<char>().swap(x);
```

std::vector Size& Capacity



std::vector Experiment

WHY Experiment?

- *big-O* 퍼포먼스 가 같더라도 특정 컨테이너가 실제 더 빠를 수 있으므로 실험 필요

- 똑같은 역할을 하는 서로 다른 방법 (iterator vs subscript) 중 가장 빠른 방법을 찾기 위해

Experiment Setup

```
struct kvstruct {
    char key[9];
    unsigned value; // could be anything at all
    kvstruct(unsigned k) : value(k)
    {
        if (strcpy_s(key, stringify(k)))
        DebugBreak();
    }
    bool operator<(kvstruct const& that) const {
        return strcmp(this->key, that.key) < 0;
    }
    bool operator==(kvstruct const& that) const {
        return strcmp(this->key, that.key) == 0;
    }
};
```

Experiment Setup

- 다양한 컨테이너에 100,000 개의 item을 넣고 method 별 performance 를 체크
  - Insertion
  - Deletion
  - Sorting
  - Visiting(Lookup)

Experiment Setup

Insertion cost : 컨테이너에 아이템을 집어 넣을 때 발생하는 코스트

Allocation stage cost : 메모리를 할당 할 때 발생하는 코스트.
할당하고자 하는 메모리의 크기에 의존적

Copy constructor cost : 할당된 메모리에 아이템들을 채울 때 발생하는 코스트, Copy constructor가 무거운 연산일 시 퍼포먼스에 지대한 영향을 미침

Insertion Cost = Allocation stage cost + Copy constructor cost

Insertion Experiment - 1

Insert item using `operator=()`

Inserting 100,000 kvstruct items

```
std::vector<kvstruct> test_container, random_vector;
...
test_container = random_vector;
```

0.445 ms

FASTEST

Insertion Experiment - 2

Insert item using `insert()`

Inserting 100,000 kvstruct items

```
std::vector<kvstruct> test_container, random_vector;
...
test_container.insert(
    test_container.end(),
    random_vector.begin(),
    random_vector.end());
```

0.696 ms

Insertion Experiment - 3.1

Insert item using push_back() + iterator

Inserting 100,000 kvstruct items

```
std::vector<kvstruct> test_container, random_vector;
...
for (auto it=random_vector.begin(); it!=random_vector.end(); ++it)
    test_container.push_back(*it);
```

2.26 ms

TOO SLOW

Insertion Experiment - 3.2

Insert item using push_back() + at()

Inserting 100,000 kvstruct items

```
std::vector<kvstruct> test_container, random_vector;
...
for (unsigned i = 0; i < nelts; ++i)
    test_container.push_back(random_vector.at(i));
```

2.05 ms

TOO SLOW

Insertion Experiment - 3.3

Insert item using push_back() + subscriptor

Inserting 100,000 kvstruct items

```
std::vector<kvstruct> test_container, random_vector;
...
for (unsigned i = 0; i < nelts; ++i)
    test_container.push_back(random_vector[i]);
```

1.99 ms

TOO SLOW

Problem Observation

WHY `operator=()` »» `push_back()`

- vector가 insert될 item의 개수를 모름
- Capacity < Size가 되는 상황이 자주 발생(reallocation 발생)

Problem Observation

WHY `operator=()` »» `push_back()`

- vector가 insert될 item의 개수를 모름
- Capacity < Size가 되는 상황이 자주 발생(reallocation 발생)

Preallocation으로 해결하자!

Insertion Experiment - 3.1*

Insert item using push_back() + iterator + Prealloc

Inserting 100,000 kvstruct items

```
std::vector<kvstruct> test_container, random_vector;
...
test_container.reserve(nelts);
for (auto it=random_vector.begin(); it!=random_vector.end(); ++it)
    test_container.push_back(*it);
```

2.26 ms → 0.674 ms

NOW HAPPY :)

Insertion Experiment - 4

Insert item using `insert()` + iterator

Inserting 100,000 kvstruct items
Push item at the front of memory

```
std::vector<kvstruct> test_container, random_vector;
...
for (auto it=random_vector.begin(); it!=random_vector.end(); ++it)
    test_container.insert(test_container.begin(), *it);
```

8,064 ms

x3000 SLOWER!

Iterating Experiment

Subscript vs Iterator vs at()

```
std::vector<kvstruct> test_container;
...
// Iterator, 0.236ms
unsigned sum = 0;
for (auto it=test_container.begin(); it!=test_container.end(); ++it
    sum += it->value;
...
// at(), 0.230ms
unsigned sum = 0;
for (unsigned i = 0; i < nelts; ++i)
    sum += test_container.at(i).value;
...
// Subscript, 0.129ms(83% faster in VS)
unsigned sum = 0;
for (unsigned i = 0; i < nelts; ++i)
    sum += test_container[i].value;
```

Sorting Experiment

std::sort() vs std::stable_sort()

```
std::vector<kvstruct> sorted_container, random_vector;
...
sorted_container = random_vector;
...
// 18.61ms (3.77ms)
std::sort(sorted_container.begin(), sorted_container.end());
...
// 16.08ms (5.01ms)
std::stable_sort(sorted_container.begin(), sorted_container.end());
```

Both time complexity : $\mathcal{O}(n \log_2 n)$

Lookup Experiment

Lookup for every key in random_vector, item in sorted_vector

Inserting 100,000 kvstruct items

```
std::vector<kvstruct> sorted_container, // items
                      random_vector;    // keys
...
for (auto it=random_vector.begin(); it!=random_vector.end(); ++it)
    kp = std::lower_bound(
          sorted_container.begin(),
          sorted_container.end(),
          *it);
    if (kp != sorted_container.end() && *it < *kp)
        kp = sorted_container.end();
}
```

28.92ms

Thank you!

Questions?