# C++ 시그 3주차

## SCSC 장필식

## 오늘 하게 될 것

간단한 C++ 코드를 작성할 수 있을 만한 거의 모든 것.

# 참고 자료

C++ Primer: Chap 9 (Sequential Containers)

시그 후에 읽어오는 것을 추천.

# 기본적인 자료구조

```
std::array<int, 5>
std::vector<int>
std::list<int>
std::map<std::string, int>
std::unordered_map<std::string, int>
std::set<int>
```

# Array

고정된 갯수의 값들을 순서대로 저장하는 자료구조.

<array> 헤더파일에 들어있음

```
// STL에 있는 어레이 (C++11 이상)
std::array<int, 5> arr = {1, 2, 3, 4, 5};
// C 어레이
int[5] arr;
```

사실 std::array도 안을 파고들면 C array로 구현이 되어있지만,
std::array는 좀 더 편리한 기능들을 많이 제공한다.

# 꺽쇠꺽쇠는 무엇인가요?

C++의 template라는 기능을 통해 여러 "종류"의 오브젝트를 만들 수 있다.

타입 뿐만 아니라 값도 집어넣을 수 있다!

```
std::array<int, 5> arr1; // 5개의 int가 들어있는 어레이
std::array<int, 10> arr2; // 10개의 int가 들어있는 어레이
std::array<std::string, 3> arr3; // 3개의 string이 들어있는 
```

추후에 템플릿을 선언하는 방법을 배울 것입니다.

# Array: 사용법

```cpp
#include <iostream>
#include <array>

using std::cout; using std::endl;

int main() {
    // initialize an array
    std::array<int, 5> arr = {1, 2, 3, 4, 5};

    // set element 0 to 3
    arr[0] = 3;

    // C-style for loop
    for (int i = 0; i < arr.size; ++i) {
        cout << arr[i] << " ";
    }
    cout << endl;

    // C++11-style foreach loop
    for (auto n : arr) {
        cout << n < " ";
    }
```

# initialization

어레이를 다음과 같이 생성시킬 수 있다.

default initialization: 기본값으로 모두 초기화

```
std::array<int, 3> arr1; // {0, 0, 0}
```

initializer list 사용

```
std::array<int, 3> arr2 = {2, 3, 5};
```

# foreach loop

- auto는 타입을 알아서 유추해주는 기능을 한다.
- 어레이의 원소들이 충분히 작을 때는 (int, float, double과 같은 primitive type): value로 받자
- 어레이의 원소들이 좀 클때는: reference 혹은 const reference로 받자

```
int main() {
    std::string result = "";
    std::array<std::string, 3> strings = { ... };
    for (const auto& str : strings) {
        result += str;
        result += " ";
    }
}
```

# 고차원 어레이

```
std::array<std::array<int, 3>, 3> mat = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
}

cout << mat[1][1] << endl;

for (auto& rows : mat) {
    for (auto n : rows) {
        cout << n << endl;
    }
}
```

# using declaration

좀 복잡한 타입은 using를 통해 간추릴 수 있다.

```
using Row = std::array<int, 3>;
using Matrix = std::array<Row, 3>;

int main() {
    Matrix mat = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };
}
```

# Vector

여러 개의 값을 순서대로 저장하는 자료구조. 크기를 마음대로 조정할 수 있다.
<vector> 헤더파일에 들어있음

```
#include <iostream>
#include <vector>

using std::cout; using std::endl;

int main() {
    std::vector<int> vec = {1, 2, 3};
    vec.push_back(5);
    vec.push_back(8);
    for (auto n : vec) {
        cout << n << endl;
    }
    cout << vec[2] << endl;

    int last = vec.pop_back();
    int secondToLast = vec.pop_back();
    cout << last << ", " << secondToLast << endl;
}
```

# List

Vector하고 같은 기능을 제공하지만, 내부 구현이 다르다.

```
int main() {
    std::list<int> lst = {1, 2, 3, 4, 5};
    l.push_front(3);
    l.push_back(6);
    for (int n : lst) {
        cout << n << endl;
    }
}
```

# std::vector vs. std::list

vector/list의 원소 개수가 n이라고 가정했을 때:

| | std::vector | std::list |
|---|---|---|
| Random access (v[i]) | $$O(1)$$ | $$O(n)$$ |
| Iteration (for(..)) | $$O(n)$$ | $$O(n)$$ (하지만 훨씬 느림) |
| Insert/Delete at end | $$O(1)$$ | $$O(1)$$ |
| Insert/Delete at middle | $$O(n)$$ | $$O(1)$$ |
| 메모리 사용 | 빈 자리가 많을 수 있음 | 꽉꽉 다 채우지만, 포인터에 의한 공간 소모 생김 |

# Map

각 Key에 대응되는 Value들을 나열해 놓은 자료구조.

<map> 헤더파이에 들어있다.

```
std::map<string, int> scores;
scores["Dongsu"] = 100;
scores["Yeongjae"] = 99;
scores["SeongWoo"] = 98;

cout << scores["Dongsu"] << endl;

// Type of entry: std::pair<string, int>
for (const auto& entry : scores) {
    cout << entry.first << ": " << entry.second << endl;
}
```

# Map initialization

```
std::map<string, int> scores = {
    {"Dongsu", 100},
    {"Yeongjae", 99},
    {"SeongWoo", 99}
};
```

# Map 정렬

키값을 기준으로 정렬이 되어 있다.

```
std::map<char, int> charCount = {
    {'b', 10},
    {'a', 15},
    {'c', 20}
}
for (const auto& entry : charCount) {
    cout << entry.first << ": " << entry.second << endl;
}
```

# **Map**에 원하는 키가 들어있는지 조사

```
// Increment Dongsu's score if found
if (scores.count("Dongsu") != 0) {
    scores["Dongsu"] += 10;
}
```

# std::unordered_map

std::map과 기능은 거의 같지만, 키값을 기준으로 정렬해주지 않는다.

만약 키 값 순서대로 프린트를 해야 한다면 우리가 따로 정렬해 주어야 한다.

대신 대부분의 경우 std::map보다 빠르다.

# std::map vs std::unordered_map

map/unordered_map의 원소 개수가 n이라고 가정했을 때:

| | **std::unordered_map** | **std::map** |
|---|---|---|
| Ordered | X | O |
| Insert/Delete entry | 보통 $$O(1)$$ | $$O(\log(n))$$ |
| Get value from key (m[k]) | 보통 $$O(1)$$ | $$O(\log(n))$$ |
| 메모리 사용 | 빈 자리가 많을 수 있음 | 꽉꽉 다 채움 |

# Set

여러개의 값들을 순서 없이 모아놓은 자료구조. (중복 허용 X)

<set> 헤더파일에 들어있다.

가끔씩 쓰게 될 수도 있으므로 소개.

```
std::set<int> numbers = {1, 5, 23, 7, 3, 2};
numbers.insert(4);
auto foundNum = numbers.find(3);
if (foundNum != numbers.end()) {
    cout << "found!" << endl;
}
```

find 함수 사용법은 iterator를 배운 후에.

# 이외의 컨테이너들

자세한 내용은 http://en.cppreference.com/w/cpp/container 참조.

```
std::stack<T>
std::queue<T>
std::priority_queue<T>
std::dequeue<T>
std::multimap<K, V>
std::unordered_multimap<K, V>
등등...
```

# 컨테이너의 내용물 탐색하기

예: `std::vector<int>` 에 있는 항목들의 합을 구하기
(단, 첫번째 항목과 마지막 항목은 제외)

```
std::vector<int> vec = {1, 1, 2, 3, 5, 8, 13, 21, 34}

int sum = 0;
for (int i = 1; i < vec.size() - 1; ++i) {
    sum += vec[i];
}
```

# int index의 문제점

만약에 다른 종류의 컨테이너가 들어온다면?

```
std::list<int> lst = {1, 1, 2, 3, 5, 8, 13, 21, 34}
int sum = 0;
for (int i = 1; i < lst.size() - 1; ++i) {
    sum += lst[i]; // error: cannot perform random access
}
```

# 해결법: iterator

컨테이너에 있는 item의 위치를 가르키는 "포인터"같은 존재.

모든 STL 컨테이너에 iterator 타입이 달려있다.

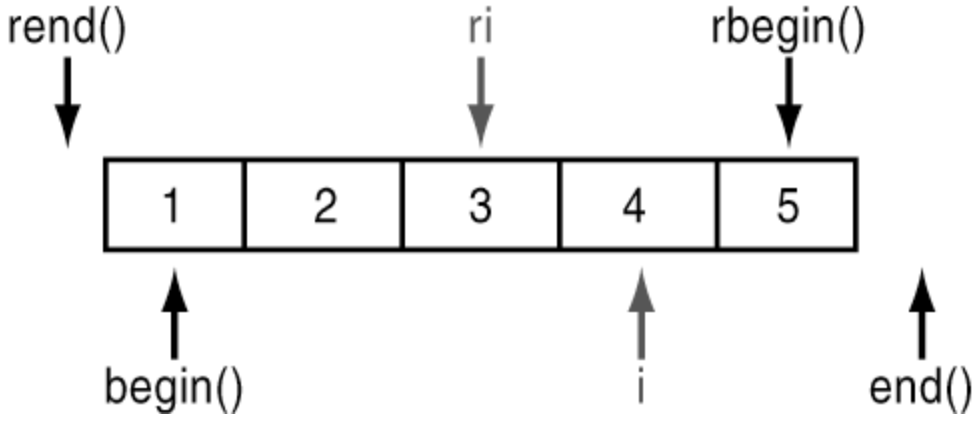
```
std::list<int> lst = {1, 1, 2, 3, 5, 8, 13, 21, 34}
int sum = 0;
for (std::list<int>::iterator it = lst.begin() + 1; it !=
    sum += *it;
}
```

# 해결법: iterator

좀 더 간결하게 표현하자면...

```
std::list<int> lst = {1, 1, 2, 3, 5, 8, 13, 21, 34}
int sum = 0;
for (auto it = lst.begin() + 1; it != lst.end() - 1; ++it)
    sum += *it;
}
```

# iterators



## **iterator**를 생성하는 방법

- begin(): 컨테이너의 첫 번째 값에 있는 iterator
- end(): 컨테이너의 마지막 값 다음에 있는 iterator
- cbegin(), cend(): const iterator를 생성함
- rbegin(), rend(): 거꾸로 가는 iterator를 생성함

# iterator range

left inclusive, right exclusive!

(예: [0, 10) => 0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

# **iterator**에 가할 수 있는 연산들

- it++: iterator를 바로 다음 값으로 이동
- it--: iterator를 바로 전 값으로 이동
- it + n: n 만큼 다음으로 이동한 iterator를 반환
- it2 - it1: 두 iterator의 거리를 계산 (difference_type를 반환)

# reverse iterators

거꾸로 iterate하고 싶은 경우:

```
std::list<int> lst = {1, 1, 2, 3, 5, 8, 13, 21, 34}
for (auto it = lst.rbegin(); it != lst.rend(); ++it) {
    cout << *it << endl;
}
```

# difference_type

두 iterator과의 거리를 계산할 수 있음

```
std::vector<int> vec = {1, 2, 3, 4, 5};
std::vector<int>::iterator firstIt = vec.begin();
std::vector<int>::iterator lastIt = vec.end();
std::vector<int>::difference_type size = lastIt - firstIt;
```

# difference_type

두 iterator과의 거리를 계산할 수 있음

```
std::vector<int> vec = {1, 2, 3, 4, 5};
auto firstIt = vec.begin();
auto lastIt = vec.end();
auto size = lastIt - firstIt;
```

# iterator의 사용 예

binary search로 vector에서 특정 숫자를 찾기

```
std::vector<int> vec = {...};

auto beg = vec.begin(), end = vec.end();
auto mid = vec.begin() + (end - beg)/2;
int sought = 100;

while (mid != end && *mid != sought) {
    if (sought < *mid) {
        end = mid;
    }
    else {
        beg = mid + 1;
    }
    mid = beg + (end - beg)/2;
}

cout << *mid << endl;
```

# iterator와 알고리즘

<algorithm> 헤더에 매우 유용한 알고리즘 함수들이 있다.

```
std::vector<int> vec = {1, 4, 2, 3, 6}
std::vector<int>::iterator it = std::find(vec.begin(), vec
```

# 실습

단어들의 목록이 fruit.txt에 저장되어 있다. 이것을 vector으로 가져온다.

```
std::vector<string> words = loadWords();
```

`loadWords()` 는 스켈레톤 코드에 정의되어 있으므로 걱정하지 않아도 된다.

# 실습

1. 다음과 같이 단어들을 길이별로 정렬해서 출력하라.

```
Words by length:
3 (1 word): fig
4 (8 words): date goji lime pear plum star ugli yuzu
5 (22 words): apple chico fruit apple berry grape guava be
6 (18 words): banana cherry damson durian feijoa raisin ja
7 (12 words): apricot avocado currant coconut custard juni
...
```

# 실습

2. 다음과 같이 단어들에서 나오는 문자들의 빈도를 계산해서 출력하라.

```
Character frequency:
a: 10.5615%
b: 4.14438%
c: 4.01069%
...
```

# 과제

다음주에 추석 연휴이므로....

# Poor man's machine learning

머신 러닝 (혹은 그 비스무리한 것)을 해보자!
코드 100줄 내외로 완성 가능.

# 가장 심플한 모델: **Markov chain**

입력 데이터: "Dongsu is slave of SCSC club."

이 문장을 학습하자.

# Markov chain

글자마다 다음에 오게 되는 글자들을 저장하자.

```
'a' => 'v'
'b' => '.'
'c' => 'l'
's' => 'u', 'l'
'l' => 'a', 'u'
'v' => 'e'
'e' => ' '
' ' => 'i', 's', 'o', 'S', 'c'
...
```

# Markov chain

주어진 확률 분포를 가지고 문장을 생성하자!

"ave slave cl...."

# Chain length

n개의 글자마다 다음에 오게 되는 글자들을 저장하자.

```
'Don' => 'g'
'ong' => 's'
'ngs' => 'u'
...
```

# 끄으읕

즐거운 추석 보내세용